MIPSpro™ Fortran 77 Programmer's Guide

Introduction

This manual provides information on implementing Fortran 77 programs using the MIPSpro™ Fortran 77 compiler on IRIX™ and the IRIS® 4D™ series workstation. This implementation of Fortran 77 contains full American National Standards Institute (ANSI) Programming Language Fortran (X3.9-1978). Extensions provide full VMS Fortran compatibility to the extent possible without the VMS operating system or VAX data representation. This implementation of Fortran 77 also contains extensions that provide partial compatibility with programs written in SVS Fortran.

Organization

This manual contains the following chapters and appendix:

- **Chapter 1, "Compiling, Linking, and Running Programs,"** gives an overview of components of the compiler system, and describes how to compile, link, and execute a Fortran program. It also describes special considerations for programs running on IRIX systems, such as file format and error handling.
- **Chapter 2, "Storage Mapping,"** describes how the Fortran compiler implements size and value ranges for various data types and how they are mapped to storage. It also describes how to access misaligned data.
- **Chapter 3, "Fortran Program Interfaces,"** provides reference and guide information on writing programs in Fortran and C that can communicate with each other. It also describes the process of generating wrappers for C routines called by Fortran.
- **Chapter 4, "System Functions and Subroutines,"** describes functions and subroutines that can be used with a program to communicate with the IRIX operating system.
- **Chapter 5, "Scalar Optimizations,"** describes the scalar optimizations you can enable from the command line.
- **Chapter 6, "Inlining and Interprocedural Analysis,"** explains how to perform inlining and interprocedural analysis by specifying options to the compiler.
- **Chapter 7, "Fortran Enhancements for Multiprocessors,"** describes programming directives for running Fortran programs in a multiprocessor mode.
- **Chapter 8, "Compiling and Debugging Parallel Fortran,"** describes and illustrates compilation and debugging techniques for running Fortran programs in a multiprocessor mode.
- **Chapter 9, "Fine–Tuning Program Execution,"** describes how to fine–tune program exection by specifying assertions and directives in your source program.
- **Appendix A, "Run–Time Error Messages,"** lists the error messages that can be generated during program execution.

Additional Reading

Refer to the *MIPSpro Fortran 77 Language Reference Manual* for a description of the Fortran language as implemented by the Silicon Graphics® IRIS 4D series workstation.

Refer to the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for information on the following topics:

- an overview of the compiler system
- improving program performance by using the profiling and optimization facilities of the compiler system
- the dump utilities, archiver, debugger, and other tools used to maintain Fortran programs

Refer to the *MIPSpro Porting and Transition Guide* for information on:

- an overview of the 64–bit compiler system
- language implementation differences
- porting source code to the 64–bit system
- compilation and run–time issues

- performance tuning

For information on interfaces to programs written in assembly language, refer to the *Assembly Language Programmer's Guide*.

Refer to the *CASEVision™/WorkShop Pro MPF User's Guide* for information about using WorkShop Pro MPF.

Typographical Conventions

The following conventions and symbols are used in the text to describe the form of Fortran statements:

Bold	Indicates literal command line options, filenames, keywords, function/subroutine names, pathnames, and directory names.
Italics	Represents user–defined values. Replace the item in italics with a legal value. Italics are also used for command names, manual page names, and manual titles.
`Courier`	Indicates command syntax, program listings, computer output, and error messages.
`Courier bold`	Indicates user input.
[ ]	Enclose optional command arguments.
()	Surround arguments or are empty if the function has no arguments following function/subroutine names. Surround manual page section in which the command is described following IRIX commands.
{}	Enclose two or more items from which you must specify exactly one.
|	Separates two or more optional items.
...	Indicates that the preceding optional items can appear more than once in succession.
#	IRIX shell prompt for the superuser.
%	IRIX shell prompt for users other than the superuser.

Here are two examples illustrating the syntax conventions.

`DIMENSION a(d) [,a(d)] ...`

indicates that the Fortran keyword **DIMENSION** must be written as shown, that the user–defined entity *a*(*d*) is required, and that one or more of *a*(*d*) can be optionally specified. Note that the pair of parentheses ( ) enclosing *d* is required.

`{STATIC | AUTOMATIC} v [,v] ...`

indicates that either the **STATIC** or **AUTOMATIC** keyword must be written as shown, that the user–defined entity *v* is required, and that one or more of *v* items can be optionally specified.

Chapter 1
Compiling, Linking, and Running Programs

This chapter contains the following major sections:

- **"Compiling and Linking"** describes the compilation environment and how to compile and link Fortran programs. This section also contains examples that show how to create separate linkable objects written in Fortran, C, or other languages supported by the compiler system and how to link them into an executable object program.
- **"Driver Options"** gives an overview of debugging, profiling, optimizing, and other options provided with the Fortran *f77* driver.
- **"Object File Tools"** briefly summarizes the capabilities of the *dump*, *dis*, *nm*, *file*, *size* and *strip* programs that provide listing and other information on object files.
- **"Archiver"** summarizes the functions of the *ar* program that maintains archive libraries.
- **"Run–Time Considerations"** describes how to invoke a Fortran program, how the operating system treats files, and how to handle run–time errors.

Also refer to the *Fortran Release Notes* for a list of compiler enhancements, possible compiler errors, and instructions on how to circumvent them.

Compiling and Linking

Drivers

Programs called *drivers* invoke the major components of the compiler system: the C preprocessor, the Fortran compiler, the optimizing code generator, and the linker. The *f77* command runs the driver that causes your programs to be compiled, optimized, assembled, and linked.

The format of the *f77* driver command is as follows:

```
f77 [option] ... filename [option]
```

where

f77	invokes the various processing phases that compile, optimize, assemble, and link the program.
option	represents the driver options through which you provide instructions to the processing phases. They can be anywhere in the command line. These options are discussed later in this chapter.
filename	is the name of the file that contains the Fortran source statements. The filename must always have the suffix **.f**, **.F**, **.for**, **.FOR**, or **.i**. For example, **myprog.f**.

Compilation

The driver command *f77* can both compile and link a source module. **Figure 1–1** shows the primary drivers phases. It also shows their principal inputs and outputs for the source modules **more.f**.

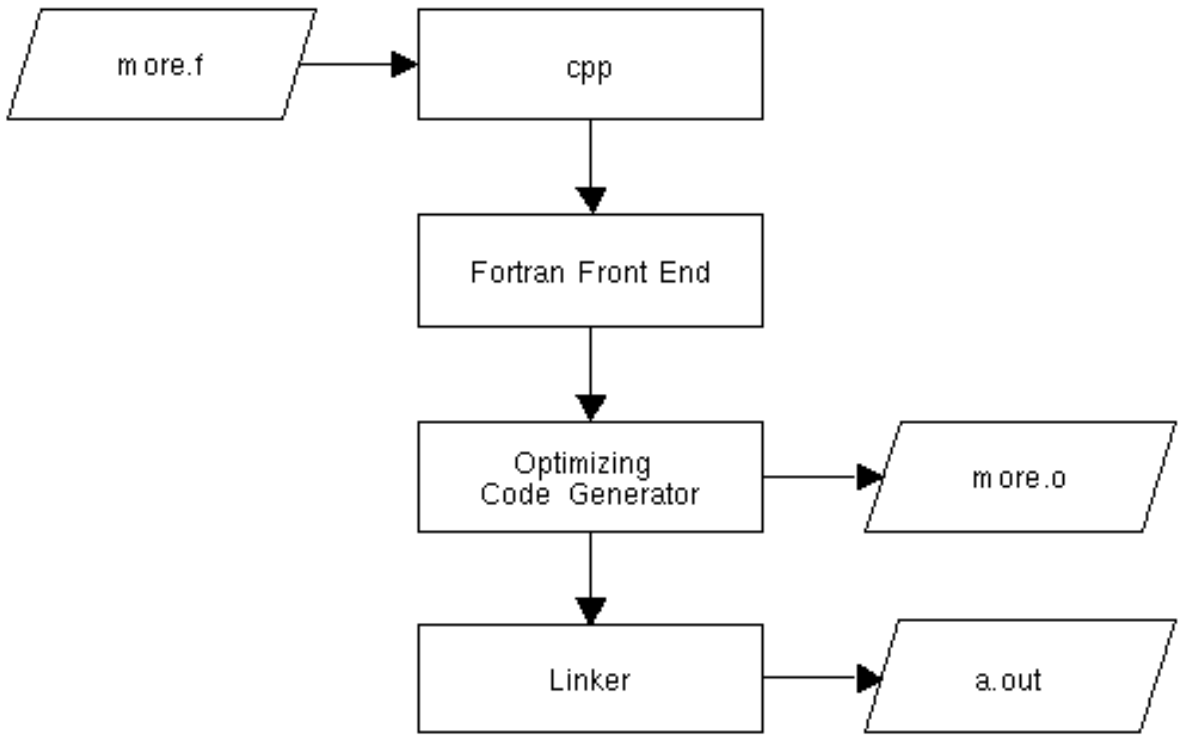

Figure 1–1 Compilation Process

Note the following:

- The source file ends with the required suffixes **.f**, **.F**, **.for**, **.FOR**, or **.i**.
- The source file is passed through the C preprocessor, *cpp*, by default. *cpp* does not recognize

Hollerith strings and may interpret a character sequence in a Holleritch string that looks like a C–style comment or a macro as a C–style comment or macro. The **-nocpp** option prevents this misinterpretation. (See the **-nocpp** option in **"Driver Options"** for details.) In the example

```
% f77 myprog.f –nocpp
```

the file **myprog.f** will not be preprocessed by *cpp*.

- The driver produces a linkable object file when you specify the **-c** driver option. This file has the same name as the source file, except with the suffix **.o**. For example, the command line

```
% f77 more.f –c
```

produces the **more.o** file in the above example.

- The default name of the executable object file is **a.out**. For example, the command line

```
%   f77 myprog.f
```

produces the executable object **a.out**.

- You can specify a name other than **a.out** for the executable object by using the driver option **-o** *name*, where *name* is the name of the executable object. For example, the command line

```
%   f77 myprog.o –o myprog
```

links the object module **myprog.o** and produces an executable object named **myprog**.

- The command line

```
% f77 myprog.f –o myprog
```

compiles and links the source module **myprog.f** and produces an executable object named **myprog**.

Compiling Multilanguage Programs

The compiler system provides drivers for other languages, including C and C++. If one of these drivers is installed in your system, you can compile and link your Fortran programs to the language supported by the driver. (See the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for a list of available drivers and the commands that invoke them; refer to **Chapter 3, "Fortran Program Interfaces,"** in this manual for conventions you must follow when writing Fortran program interfaces to C programs.)

When your application has two or more source programs written in different languages, you should compile each program module separately with the appropriate driver and then link them in a separate step. Create objects suitable for linking by specifying the **-c** option, which stops the driver immediately after the assembler phase. For example,

```
%   cc –c main.c
%   f77 –c rest.f
```

The two command lines shown above produce linkable objects named **main.o** and **rest.o**, as illustrated in **Figure 1–2** .

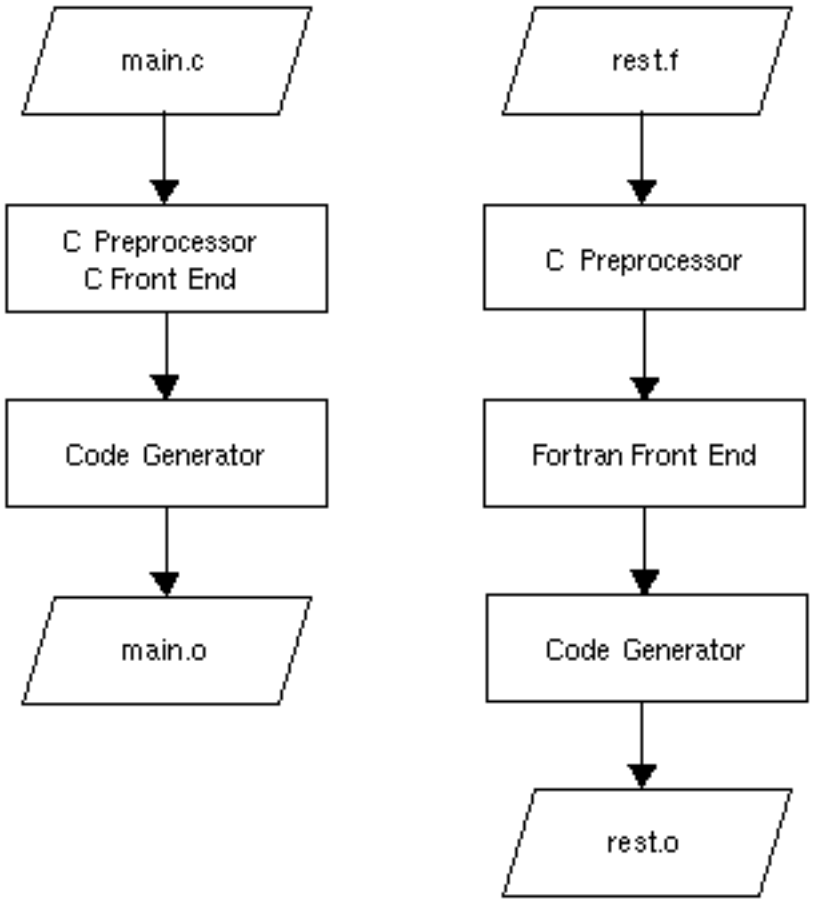

Figure 1–2 Compiling Multilanguage Programs

Linking Objects

You can use the *f77* driver command to link separate objects into one executable program when any one of the objects is compiled from a Fortran source. The driver recognizes the **.o** suffix as the name of a file containing object code suitable for linking and immediately invokes the linker. The following command links the object created in the last example:

```
% f77 -o myprog main.o rest.o
```

You can also use the *cc* driver command, as shown below:

```
% cc -o myprog main.o rest.o -lftn -lm
```

Figure 1–3 shows the flow of control for this link.

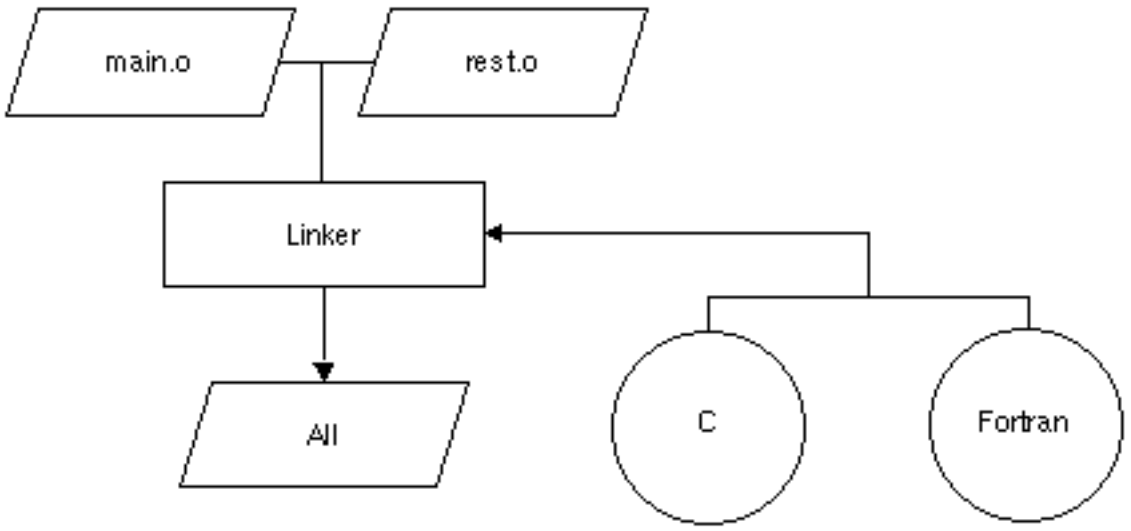

Figure 1–3 Linking

Both *f77* and *cc* use the C link library by default. However, the *cc* driver command does not know the names of the link libraries required by the Fortran objects; therefore, you must specify them explicitly to the linker using the **-l** option as shown in the example. The characters following **-l** are shorthand for link library files, as shown in **Table 1–1** .

Table 1–1 Link Libraries

-l	**Link Library**	**Contents**
ftn	/usr/lib64/nonshared/libftn.a	Intrinsic function, I/O, multiprocessing, IRIX interface, and indexed sequential access method library for nonshared linking and compiling
ftn	/usr/lib64/libftn.so	Same as above, except for shared linking and compiling (this is the default library)
m	/usr/lib64/libm.so	Mathematics library

See the section called "FILES" in the *f77*(1) manual page for a complete list of the files used by the Fortran driver. Also refer to the *ld*(1) manual page for information on specifying the **-l** option.

Specifying Link Libraries

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

```
% f77 main.o more.o rest.o -lbsd
```

To specify a library created with the archiver, type in the pathname of the library as shown below.

```
% f77 main.o more.o rest.o libfft.a
```

Note: The linker searches libraries in the order you specify. Therefore, if you have a library (for example, *libfft.a*) that uses data or procedures from **-lm**, you *must* specify *libfft.a* first.

Driver Options

This section contains a summary of the Fortran-specific driver options. See the *f77*(1) manual page for a complete description of the compiler options; see the *ld*(1) manual page for a description of the linker options. For a list and description of general driver options refer to the *MIPSpro Compiling, Debugging,*

and Performance Tuning Guide.

When used at compile time, the -**align***n* options generate various degrees of misaligned data in common blocks. They then generate the code to deal with the misalignment.

Note: When specified, the -**align***n* options can degrade program performance; **-align8** causes the greatest degree of degradation, and **-align32** causes the least.

-**align8** Aligns objects larger than 8 bits on 8−bit boundaries. This option has the largest impact on performance.

-**align16** Aligns objects larger than 16 bits on 16−bit boundaries; 16−bit objects must still be aligned on 16−bit boundaries (MC68000−like alignment rules).

-**align32** Aligns objects larger than 32 bits on 32−bit boundaries; 16−bit objects must still be aligned on 16−bit boundaries, and 32−bit objects must still be aligned on 32−bit boundaries.

-**align64** Aligns objects larger than 64 bits on 64−bit boundaries. Objects with size 64 bits or smaller must still be aligned on the corresponding boundaries.
The default alignment for 32−bit objects is 32 bits; for 64−bit objects it is 64 bits. However, the default alignment for **COMMON** blocks is always 32 bits.
You must specify the appropriate alignment option in the compilation of all modules that reference or define common blocks with misaligned data. Failure to do so could cause core dumps (if the trap handler is not used) or mismatched common blocks.
To load the system libraries capable of handling misaligned data, use the **-L/usr/lib/align** switch at load time. The trap handler may be needed to handle misaligned data passed to system libraries that are not included in the **/usr/lib/align** directory (see the *fixade*(3f) and *unalign*(3x) manual pages).

-ansi Compiles in strict accordance with the Fortran 77 ANSI standard. Rejects syntax not allowed by the standard.

-backslash Allows the backslash character to be used as a normal Fortran character instead of the beginning of an escape sequence.

-bytereclen Interprets the record length specifier for a direct unformatted file as a number of bytes instead of a number of words.

-c Suppresses the loading phase of compilation and forces the compiler to produce an object file even if only one program is compiled.

-chunk=*integer*
Has the same effect as putting a **C$CHUNK=***integer* directive at the beginning of the file. See **Chapter 7, "Fortran Enhancements for Multiprocessors,"** and **Chapter 8, "Compiling and Debugging Parallel Fortran,"** for details.

-col72 Sets the source statement format as described in **Table 1−2**.

Table 1−2 Source Statement Settings for −col72 Option

Column	**Contents**
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

If the source statement contains fewer than 72 characters, no blank padding occurs; the TAB−format facility is disabled.
This option provides the SVS Fortran 72−column option mode.

-col120 Sets the source statement format as described in **Table 1−3**.

Table 1−3 Source Statement Settings for −col120 Option

Column	**Contents**
1−5	Statement label
6	Continuation indicator
7−120	Statement body
121−end	Ignored

If the source statement contains fewer than 120 characters, no blank padding occurs; the TAB−format facility is disabled.

This option provides the SVS Fortran default mode.

-cpp Runs the C macro preprocessor *cpp* on all source files before compilation. (This option is enabled by default.)

-d16 Uses **REAL*16** and **COMPLEX*32** as the defaults for variables that are declared as **DOUBLE** and DOUBLE PRECISION. See also **-r8**.

-d_lines Causes any lines with a D in column 1 to be compiled. By default, the compiler treats all lines with a character in column 1 as comment lines.

-extend_source

Sets the source statement format as described in **Table 1–4** .

Table 1–4 Source Statement Settings for –extend_source Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-132	Statement body
133-end	Warning message issued

If the source statement contains fewer than 132 characters, blanks are assumed at the end; the ability of TAB–formatted lines to extend past column 132 is disabled.
This option provides VMS Fortran 132–column mode, except that a warning, instead of a fatal error message, is generated when text extends beyond column 132.

-E Runs only the C macro preprocessor on the files and sends the results to standard output.

-i2 All small integer constants become **INTEGER*2**. All variables and functions implicitly or explicitly declared type **INTEGER** or **LOGICAL** (without a size designator, that is, *2, *4, and so on) are **INTEGER *2** or **LOGICAL *2**, respectively.

-i4 Same as the **-i2** option, except the type becomes **INTEGER*4** or **LOGICAL *4**. This option is in effect by default.

-i8 Same as the **-i2** option, except the type becomes **INTEGER*8** or **LOGICAL *8**.

-listing Produces the source listing file with **.L** suffix containing line numbers, error messages, symbol table information, and cross references.

-mp Enable the multiprocessing directives. For further options affecting multiprocessing compilation, see the *f77*(1) manual page, and **Chapter 7, "Fortran Enhancements for Multiprocessors,"** and **Chapter 8, "Compiling and Debugging Parallel Fortran."**

-mp_schedtype=*type*

Has the same effect as putting a **C$MP_SCHEDTYPE=** *type* directive at the beginning of the file. Valid values for *type* are **simple**, **interleave**, **dynamic**, **gss**, and **runtime**. See **Chapter 7, "Fortran Enhancements for Multiprocessors,"** and **Chapter 8, "Compiling and Debugging Parallel Fortran,"** of this manual for more details.

-nocpp Suppresses execution of the C preprocessor on the source files. Specifying this option allows you to specify C–style comments inside Hollerith strings. Use this option when you want your program to strictly conform to the Fortran 77 standard.

-noexpopt Excludes floating point constant exponent optimization to achieve the same precision as releases prior to 4D1–4.0.

-noextend_source

Sets the source statement format as described in **Table 1–5** .

Table 1–5 Source Statement Settings for –noextend_source Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

If the source statement contains fewer than 72 characters, blanks are assumed at the end; the ability of TAB–formatted lines to extend past Column 72 is disabled.
This option provides VMS Fortran default mode.

-noi4 Same as **-i2** option.

-old_rl This option has been replaced with the -bytereclen option.

-P Runs only the C macro preprocessor and puts the result of each source file into a

corresponding **.i** file. The **.i** file cannot contain # lines.

-p Produces pc sampling information.

-pfa Runs Power Fortran to automatically discover parallelism in the source code. The compiler will recognize this option only if you have purchased and installed the separate product MIPSpro Power Fortran 77.

-r8 Uses **REAL*8** and **COMPLEX*16** as the defaults for real and complex variables that are not explicitly declared with a type size.

-static Local variables are saved in a static location, initialized to zeros, and retain values between calls. This option overrides the default **-automatic** option.

-u Disables Fortran default data typing and any data typing explicitly specified in an **IMPLICIT** statement. Forces the explicit declaration of all data types.

-vms_cc Uses VMS Fortran carriage control interpretation on unit 6.

-vms_endfile
Causes a VMS endfile record to be written when an **ENDFILE** statement is executed, allows records to be written after an endfile record and then subsequent reading from an input file after an endfile record is encountered.

-vms_library
Treats subprograms starting with **LIB$**, **OTS$**, and **SMG$** as VMS run–time routines that accept a variable number of arguments.

-vms_stdin Allows rereading from *stdin* after **EOF** has been encountered.

-WK,*option*[,*option*]...
Specifies an optimization option to the compiler, where *option* is any of the options described in **Chapter 5, "Scalar Optimizations."**

-w Suppresses warning messages.

Debugging

The compiler system provides a source–level, interactive debugger called *dbx* that you can use to debug programs as they execute. With *dbx* you can control program execution to set breakpoints, monitor what is happening, modify values, and evaluate results. *dbx* keeps track of variables, subprograms, subroutines, and data types in terms of the symbols used in the source language. You can use this debugger to access the source text of the program, to identify and reference program entities, and to detect errors in the logic of the program.

You can also use the WorkShop Pro MPF debugger along with MIPSpro Power Fortran 77 to debug your Fortran code. These products are sold separately from MIPSpro Fortran 77.

Reference Information

The **-g***n* options produce various levels of symbol table information for symbolic debugging. For a complete list of **-g***n* driver options, see the *f77*(1) manual page. See the *dbx*(1) manual page for information on the debugger. For a complete description see the *MIPSpro Compiling, Debugging, and Performance Tuning Guide*.

Profiling

You can use the compiler system to generate profiled programs that, when executed, provide operational statistics. This is done through driver option **-p** (which provides pc sampling information) and the *pixie* and *prof* programs.

A variety of options and methods of profiling are available. To learn more about them, read Chapter 4 of the *MIPSpro Compiling, Debugging, and Performance Tuning Guide*, which describes the advantages and methods of profiling. It also gives examples of the various options and commands. See the *prof*(1) manual page for detailed reference information.

Optimizing

The compiler performs general optimizations when you specify the **-O***n* option. This option enables the back–end optimizations described in the *MIPSpro Compiling, Debugging, and Performance Tuning Guide*. It also enables the equivalent of various combinations of the **-WK** option with the **-scalaropt** (or **-so**), **-roundoff** (or **-r**), and **-optimize** (or **-o**) options (depending on the value you specify for *n*). **Table**

1–6 summarizes these optimizations.

Table 1–6 Fortran Front End Optimization Options

Option	Result
-O3	Same as -O2 but also performs all optimizations including expression simplification, sum reduction, loop interchange, and memory management transformations.
-O2	Same as -O1 but also performs global optimization including simplification of expressions and intrinsic functions, strength reduction, loop interchange around serial arithmetic reductions, loop rerolling. Enables all scalar optimizations including memory management transformations.
-O1	Performs limited optimizations including simplification of expressions and intrinsic functions, strength reduction, loop interchange around serial arithmetic reducitons, loop rerolling, all scalar optimizations (but not memory optimizations), and induction variable recognition.
-O0	Suppresses optimization. This option is in effect by default.

Refer to **Chapter 5, "Scalar Optimizations,"** for complete descriptions of the **-r**, **-so**, and **-o** options.

The **-sopt** compiler option allows you to enable all scalar optimizations while disabling the roundoff transformations. This option has the same effect as specifying **-WK,-r=0,-so=3,-o=5**. Do not confuse this option with the **-WK,-scalaropt** option.

For combinations of the **-O***n* option with **-sopt**, the compiler performs the greater of the implied options. For example, specfying **-O1** (which is the same as **-WK,-r=0, -so=2,-o=1**) and **-sopt** (which is the same as **-WK,-r=0,-so=3,-o=5**) has the same effect as **-WK,-r=0,-so=3,-o=5**.

Refer to the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for details on the general optimization techniques used by the compiler and tips on writing optimal code for optimizer processing. Refer to **Chapter 5, "Scalar Optimizations,"** for details on scalar optimizations.

Performance

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

- Two linker options**, -G** 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) manual 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 not requiring extremely large blocks of virtual memory. See *ld*(1) for more information.

Object File Tools

The following tools provide information on object files as indicated:

dump	Lists headers, tables, and other selected parts of an ELF–format object or archive file.
dis	Disassembles object files into machine instructions.
nm	Prints symbol table information for object and archive files.
file	Lists the properties of program source, text, object, and other files. This tool often erroneously recognizes command files as C programs. It does not recognize Pascal or LISP programs.
size	Prints information about the **text**, **rdata**, **data**, **sdata**, **bss**, and **sbss** sections of the specified object or archive files. See the *a.out*(4) manual page for a description of the contents and format of section data.
strip	Removes symbol table and relocation bits.

For more information on these tools, see the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* and the *dis*(1), *dump*(1), *file*(1), *nm*(1), *size*(1), and *strip*(1) manual pages.

Archiver

An archive library is a file that contains one or more routines in object (**.o**) file format. The term *object* as used in this chapter refers to an **.o** file that is part of an archive library file. When a program calls an object not explicitly included in the program, the link editor *ld* looks for that object in an archive library. The link editor then loads only that object (not the whole library) and links it with the calling program. The archiver (*ar*) creates and maintains archive libraries and has the following main functions:

- copying new objects into the library
- replacing existing objects in the library
- moving objects about the library
- copying individual objects from the library into individual object files

See the *Compiling, Debugging, and Performance Tuning Guide* and the *ar*(1) manual page for additional information on the archiver.

Run–Time Considerations

Invoking a Program

To run a Fortran program, invoke the executable object module produced by the *f77* command by entering the name of the module as a command. By default, the name of the executable module is **a.out**. If you included the **-o** *filename* option on the *ld* (or *f77*) command line, the executable object module has the name that you specified.

File Formats

Fortran supports five kinds of external files:

- sequential formatted
- sequential unformatted
- direct formatted
- direct unformatted
- key indexed file

The operating system implements other files as ordinary files and makes no assumptions about their internal structure.

Fortran I/O is based on records. When a program opens a direct file or key indexed file, the length of the records must be given. The Fortran I/O system uses the length to make the file appear to be made up of records of the given length. When the record length of a direct file is 1 byte, the system treats the file as ordinary system files (as byte strings, in which each byte is addressable). A **READ** or **WRITE** request on such files consumes bytes until satisfied, rather than restricting itself to a single record.

Because of special requirements, sequential unformatted files will probably be read or written only by Fortran I/O statements. Each record is preceded and followed by an integer containing the length of the record in bytes.

During a **READ**, Fortran I/O breaks sequential formatted files into records by using each new line indicator as a record separator. The Fortran 77 standard does not define the required result after reading past the end of a record; the I/O system treats the record as being extended by blanks. On output, the I/O system writes a new line indicator at the end of each record. If a user program also writes a new line indicator, the I/O system treats it as a separate record.

Preconnected Files

Table 1–7 shows the standard preconnected files at program start.

Table 1–7 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. The default connection is for sequentially formatted I/O.

File Positions

The Fortran 77 standard does not specify where **OPEN** should initially position a file explicitly opened for sequential I/O. The I/O system positions the file to 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.

Unknown File Status

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 the beginning of the file.

Quad−Precision Operations

When running 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 details.:

Run−Time Error Handling

When the Fortran run−time system detects an error, 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 Messages,"** for a list of the error messages.
- A core file is produced if the **f77_dump_flag** environment variable is set, as described in **Appendix A, "Run−Time Error Messages."** You can use *dbx* to inspect this file and determine the state of the program at termination. For more information, see the *dbx Reference Manual*.

  To invoke *dbx* using the core file, enter the following:

  ```
  %   dbx binary-file   core
  ```

  where *binary−file* is the name of the object file output (the default is **a.out**). For more information on *dbx*, see **"Debugging"** .

Trap Handling

The library *libfpe.a* provides two methods for handling floating point exceptions: 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) manual page for more information.

Chapter 2
Storage Mapping

This chapter contains two sections:

- **"Alignment, Size, and Value Ranges"** describes how the Fortran compiler implements size and value ranges for various data types as well as how data alignment occurs under normal conditions.
- **"Access of Misaligned Data"** describes two methods of accessing misaligned data.

Alignment, Size, and Value Ranges

Table 2–1 contains information about various Fortran data types.

Table 2–1 Size, Alignment, and Value Ranges of Data Types

Type	Synonym	Size	Alignment	Value Range
BYTE	INTEGER*1	8 bits	Byte	-128…127
INTEGER*2		16 bits	Half word[a]	-32,768… 32,767
INTEGER	INTEGER*4[b]	32 bits	Word[c]	-2^{31}…2^{31} -1
INTEGER*8		64 bits	Double word	-2^{63}…2^{63} -1
LOGICAL*1		8 bits	Byte	0…1
LOGICAL*2		16 bits	Half word[a]	0…1
LOGICAL	LOGICAL*4[d]	32 bits	Word[c]	0…1
LOGICAL*8		64 bits	Double word	0...1
REAL	REAL*4[e]	32 bits	Word[c]	See **Table 2–2**
DOUBLE PRECISION	REAL*8[f]	64 bits	Double word[g]	See **Table 2–2**
REAL*16		128 bits	Double word	See **Table 2–3**
COMPLEX	COMPLEX*8[h]	64 bits	Double word[c]	See the fourth bullet item below
DOUBLE COMPLEX	COMPLEX*16[i]	128 bits	Double word[g]	See the fourth bullet item below
COMPLEX*32		256 bits	Double word	See the fourth bullet item below
CHARACTER		8 bits	Byte	-128…127

[a]Byte boundary divisible by two.

[b]When the -i2 option is used, type INTEGER is equivalent to INTEGER*2; when the -i8 option is used, INTEGER is equivalent to INTEGER*8.

[c]Byte boundary divisible by four.

[d]When the -i2 option is used, type LOGICAL is equivalent to LOGICAL*2; when the -i8 op tion is used, type LOGICAL is equivalent to LOGICAL*8.

[e]When the -r8 option is used, type REAL is equivalent to REAL*8.

[f]When the -d16 option is used, type DOUBLE PRECISION is equivalent to REAL*16.

[g]Byte boundary divisible by eight.

[h]When the -r8 option is used, type COMPLEX is equivalent to COMPLEX*16.

[i]When the -d16 option is used, type DOUBLE COMPLEX is equivalent to COMPLEX*32.

The following notes provide details on some of the items in **Table 2–1** .

- **Table 2–2** lists the approximate valid ranges for **REAL*4** and **REAL*8**.

Table 2–2 Valid Ranges for REAL*4 and REAL*8 Data Types

Range	REAL*4	REAL*8
Maximum	$3.40282356 * 10^{38}$	$1.7976931348623158 * 10^{308}$
Minimum normalized	$1.17549424 * 10^{-38}$	$2.2250738585072012 * 10^{-308}$
Minimum denormalized	$1.40129846 * 10^{-46}$	$1.1125369292536006 * 10^{-308}$

- **REAL*16** constants have the same form as **DOUBLE PRECISION** constants, except the exponent indicator is **Q** instead of **D**. **Table 2–3** lists the approximate valid range for **REAL*16**. **REAL*16** values have an 11–bit exponent and a 107–bit mantissa; they are represented internally as the sum or difference of two doubles. So, for **REAL*16** "normal" means that both high and low parts are normals.

Table 2–3 Valid Ranges for REAL*16 Data Type

Range	Precise Exception Mode w/FS Bit Clear	Fast Mode or Precise Exception Mode w/FS Bit Set
Maximum	$1.797693134862315807937289714053023 * 10^{308}$	1.7976931348623157807937289714053023* 10 308
Minimum normalized	$2.004168360008973000503493902070300 4 * 10^{-292}$	2.0041683600089730005034939020703004* 10 –292
Minimum denormalized	$4.940656458412465441765687928682214 * 10^{-324}$	2.2250738585072013830902327173324040 4* 10 –308

- **Table 2–1** states that **REAL*8** (that is, **DOUBLE PRECISION**) variables always align on a double–word boundary. However, Fortran permits these variables to align on a word boundary if a **COMMON** statement or equivalencing requires it.
- Forcing **INTEGER**, **LOGICAL**, **REAL**, and **COMPLEX** variables to align on a halfword boundary is not allowed, except as permitted by the **-align8**, **-align16**, and **-align32** command line options. See **Chapter 1, "Compiling, Linking, and Running Programs."**
- A **COMPLEX** data item is an ordered pair of **REAL*4** numbers; a **DOUBLE COMPLEX** data item is an ordered pair of **REAL*8** numbers; a **COMPLEX*32** data item is an ordered pair of **REAL*16** numbers. In each case, the first number represents the real part and the second represents the imaginary part. Therefore, refer to **Table 2–2** and **Table 2–3** for valid ranges.
- **LOGICAL** data items denote only the logical values TRUE and FALSE (written as **.TRUE.** or **.FALSE.**). However, to provide VMS compatibility, **LOGICAL** variables can be assigned all integral values of the same size.
- You must explicitly declare an array in a **DIMENSION** declaration or in a data type declaration. To support **DIMENSION**, the compiler
  - allows up to seven dimensions
  - assigns a default of 1 to the lower bound if a lower bound is not explicitly declared in the **DIMENSION** statement
  - creates an array the size of its element type times the number of elements
  - stores arrays in column–major mode
- The following rules apply to shared blocks of data set up by the **COMMON** statements:
  - The compiler assigns data items in the same sequence as they appear in the common statements defining the block. Data items are padded according to the alignment compiler options or the compiler defaults. See **"Access of Misaligned Data"** for more information.
  - You can allocate both character and noncharacter data in the same common block.
  - When a common block appears in multiple program units, the compiler allocates the same size for that block in each unit, even though the size required may differ (due to varying element names, types, and ordering sequences) from unit to unit. The size allocated corresponds to the maximum size required by the block among all the program units except when a common block is defined by using **DATA** statements, which initialize one or more of the common block variables. In this case the common block is allocated the same size as when it is defined.

Access of Misaligned Data

The Fortran compiler allows misalignment of data if specified by the use of special options.

As discussed in the previous section, the architecture of the IRIS–4D series assumes a particular alignment of data. ANSI standard Fortran 77 cannot violate the rules governing this alignment. Many opportunities for misalignment can arise when using common extensions to the dialect. This is particularly true for small integer types, which

- allow intermixing of character and non−character data in **COMMON** and **EQUIVALENCE** statements
- allow mismatching the types of formal and actual parameters across a subroutine interface
- provide many opportunities for misalignment to occur

Code using the extensions that compiled and executed correctly on other systems with less stringent alignment requirements may fail during compilation or execution on the IRIS−4D. This section describes a set of options to the Fortran compilation system that allow the compilation and execution of programs whose data may be misaligned. Be forewarned that the execution of programs that use these options is significantly slower than the execution of a program with aligned data.

This section describes the two methods that can be used to create an executable object file that accesses misaligned data.

Accessing Small Amounts of Misaligned Data

Use the first method if the number of instances of misaligned data access is small or to provide information on the occurrence of such accesses so that misalignment problems can be corrected at the source level.

This method catches and corrects bus errors due to misaligned accesses. This ties the extent of program degradation to the frequency of these accesses. This method also includes capabilities for producing a report of these accesses to enable their correction.

To use this method, keep the Fortran front end from padding data to force alignment by compiling your program with one of two options to *f77*.

- Use the **-align8** option if your program expects no restrictions on alignment.
- Use the -**align16** option if your program expects to be run on a machine that requires half−word alignment.

You must also use the misalignment trap handler. This requires minor source code changes to initialize the handler and the addition of the handler binary to the link step (see the *fixade*(3f) manual page).

Accessing Misaligned Data Without Modifying Source

Use the second method for programs with widespread misalignment or whose source may not be modified.

In this method, a set of special instructions is substituted by the IRIS−4D assembler for data accesses whose alignment cannot be guaranteed. The generation of these more forgiving instructions may be opted for each source file independently.

You can invoke this method by specifying of one of the alignment options (**-align8**, **-align16**) to *f77* when compiling any source file that references misaligned data (see the *f77*(1) manual page). If your program passes misaligned data to system libraries, you might also need to link it with the trap handler. See the *fixade*(3f) manual page for more information.

Chapter 3
Fortran Program Interfaces

This chapter contains the following major sections:

- **"Fortran/C Interface"** describes the interface between Fortran routines and routines written in C. It contains rules and gives examples for making calls and passing arguments between the two languages.
- **"Fortran/C Wrapper Interface"** describes the process of generating wrappers for C routines called by Fortran.

You may need to refer to other sources of information as you read this chapter.

- For information on storage mapping (how the variables of the various languages appear in storage), refer to **Chapter 1** for Fortran and to Chapter 2 in the appropriate language programmer's guide for other languages.
- For information on the standard linkage conventions used by the compiler in generating code, see Chapter 7 of the *Assembly Language Programmer's Guide*.

For information on built–in functions that provide access to non–Fortran system functions and library routines, see **Chapter 4** of this manual.

Fortran/C Interface

When writing Fortran programs that call C functions, consider procedure and function declaration conventions for both languages. Also, consider the rules for argument passing, array handling, and accessing common blocks of data.

Procedure and Function Declarations

This section discusses names and invocations to consider before calling C functions from Fortran.

Names

When calling a Fortran subprogram from C, the C program must append an underscore (_) to the name of the Fortran subprogram. For example, if the name of the subprogram is **matrix**, then call it by the name **matrix_**. When Fortran is calling a C function, the name of the C function must also end with an underscore.

The compiler will not append an underscore if a name ends in **$**, as in **foo$**. This behavior is useful for calling C functions that do not have a trailing underscore.

The Fortran compiler changes all its subprogram names to lowercase. Thus, all of the following subprograms refer to the same function **matrix** when interfacing with C:

```
subroutine MATRIX
subroutine Matrix
subroutine matrix
```

Note that only one main routine is allowed per program. The main routine can be written in either C or Fortran. **Table 3–1** contains an example of a C and a Fortran main routine.

Table 3–1 Main Routines

C	Fortran
main () {	write (6,10)
printf("hi!\n");	10 format ('hi!')
}	end

Invocations

Invoke a Fortran subprogram as if it were an integer–valued function whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the subprogram but cause an indexed branch in the calling subprogram. If the subprogram is *not* a function and has no entry points with alternate return arguments, the returned value is undefined. The Fortran statement

```
call nret (*1,*2Ex,*3)
```

is treated exactly as if it were the computed **goto**

```
goto (1,2,3), nret()
```

A C function that calls a Fortran subprogram can usually ignore the return value of a Fortran subroutine; however, the C function should not ignore the return value of a Fortran function. **Table 3–2** shows equivalent function and subprogram declarations in C and Fortran programs.

Table 3–2 Equivalent C and Fortran Function Declarations

C Function Declaration	Fortran Function Declaration
double dfort()	double precision function dfort()
float rfort()	real function rfort()
int ifort()	integer function ifort()
int lfort()	logical function lfort()

Note the following:

- Avoid calling Fortran functions of type **REAL**, **COMPLEX**, and **CHARACTER** from C.
- You cannot write a C function so that it will return a **COMPLEX** value to Fortran.
- A character–valued Fortran subprogram is equivalent to a C language routine with two extra initial arguments: a data address and a length. However, if the length is one, no extra argument is needed and the single character result is returned as in a normal numeric function.

  Thus

  ```
  character*15 function g(...)
  ```

  is equivalent to

  ```
  char result [15];
  long int length;
  g_(result, length, ...)
  ...
  ```

  and could be invoked in C by

  ```
  char chars[15]
  g_(chars, 15, ...);
  ```

  and

  ```
  character function h(...)
  ```

  could be invoked in C by

  ```
  char c, h();
  c=h_(...);
  ```

Arguments

The following rules apply to arguments passed between Fortran and C:

- All explicit arguments must be passed by reference. All routines must specify an *address* rather than a value. Thus, to pass constants or expressions to Fortran, the C routine must first store their values into variables and then pass the address of the variable. (The only exception occurs when passing the length of a string from C to a Fortran subroutine with a parameter of type **CHARACTER**.)
- When passing the address of a variable, the data representations of the variable in the calling and called routines must correspond, as shown in **Table 3–3** .

Table 3–3 Equivalent Fortran and C Data Types

Fortran	**C**
byte x	signed char x;
integer*1 x	signed char x;
integer*2 x	short int x;
integer x	int x;
integer*8	long long x;
logical x	int x;
real x	float x;
double precision x	double x;
real*16 x	long double x;
complex x	struct{float real, imag;} x;
double complex x	struct{double dreal,dimag;} x;
character*6 x	char x[6] [a]

[a]The array length must also be passed, as discussed in the next section.

- In Fortran, **INTEGER** and **LOGICAL** variables occupy 32 bits of memory by default, but you can change this by using the **-i2** or **-i8** options.
- The Fortran compiler may add items not explicitly specified in the source code to the argument list. The compiler adds the following items under the conditions specified:
  - destination address for character functions, when called

- length of a character variable, when an argument is the address of a character variable

When a C function calls a Fortran routine, the C function must explicitly specify these items in its argument list *in the following order:*

1. If the Fortran routine is a function that returns a character variable of length greater than 1, specify the address and length of the resultant character variable.
2. Specify normal arguments (addresses of arguments or functions).
3. Specify the length of each normal character parameter in the order it appeared in the argument list. The length must be specified as a constant value or **INTEGER** variable (that is, not an address).

The examples on the following pages illustrate these rules.

Example 1

This example shows how a C routine specifies the destination address of a Fortran function (which is only implied in a Fortran program).

Fortran

```
C  Fortran call to SAM, a routine written
C  in Fortran
     EXTERNAL F
     CHARACTER*7 S
     INTEGER B(3)
     ...
     CALL SAM (F, B(2), S)
```

C

```
/* C call to SAM, a routine written in Fortran */
/* We pass in the function pointer for the     */
/* Fortran SUBROUTINE F                        */
char s[7];
int  b[3];
extern void sam_(void (*)(), int *, char*);
                          /* Fortran subroutine SAM          */
extern void f_();       /* Fortran subroutine F            */
...
sam_(F, &B[1], S);      /* We pass in pointer to Fortran F */
                          /* for Fortran call-by-reference   */
```

Example 2

This example shows how a C routine must specify the length of a character string (which is only implied in a Fortran call).

Fortran

```
C  Fortran call to F, a function written
C  in Fortran
     EXTERNAL F
     CHARACTER*10 F, G
     G = F()
```

C

```
/* C call to F, a routine written in Fortran */
/* which returns a string.                   */
CHAR S[10];
. . .
f_(S, 10);
```

The function F, written in Fortran

```
C  function F, written in Fortran
     CHARACTER*10 FUNCTION F()
     F = '0123456789'
     RETURN
     END
```

Array Handling

Fortran stores arrays in column−major order with the leftmost subscript varying the fastest. C, however, stores arrays in the opposite arrangement (row−major order), with the rightmost subscripts varying the fastest.

Here is how the layout of the Fortran array looks:

```
integer t (2,3)
t(1,1), t(2,1), t(1,2), t(2,2), t(1,3), t(2,3)
```

Here is how the layout of the C array looks:

```
int t [2] [3]
t[0][0], t[0][1], t[0][2], t[1][0], t[1][0], t[1][1],t[1][2]
```

Note that the default for the lower bound of an array in Fortran is 1, where the default in C is 0.

When a C routine uses an array passed by a Fortran subprogram, the dimensions of the array and the use of the subscripts must be interchanged, as shown in **Figure 3−1** .

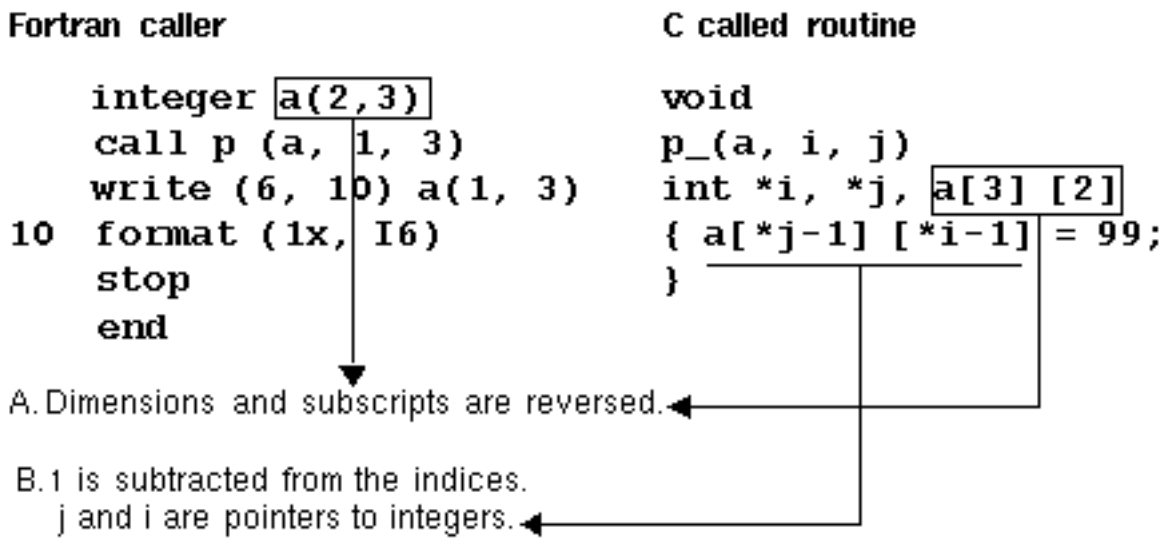

Figure 3−1 Array Subscripts

The Fortran caller prints out the value 99. Note the following:

- Because arrays are stored in column−major order in Fortran and row− major order in C, the dimension and subscript specifications are reversed.
- Because the lower−bound default is 1 for Fortran and 0 for C, 1 must be subtracted from the indexes in the C routine. Also, because Fortran passes parameters by reference, **j** and **i** are pointers in the C routine.

Accessing Common Blocks of Data

The following rules apply to accessing common blocks of data:

- Fortran common blocks must be declared by common statements; C can use any global variable. Note that the common block name in C (**r_**) must end with an underscore.
- Data types in Fortran and C programs must match unless you want equivalencing. If so, you must adhere to the alignment restrictions for the data types described in **Chapter 2** .
- If the same common block is of unequal length, the largest size is used to allocate space.
- Unnamed common blocks are given the name **_BLNK__**.

The following examples show C and Fortran routines that access common blocks of data.

Fortran

```
subroutine sam()
common /r/ i, r
i = 786
r = 3.2
return
end
```

C

```
struct S {int i; float j;}r_;
main () {
sam_() ;
printf("%d %f\n",r_.i,r_.j);
}
```

The C routine prints out the numbers 786 and 3.2.

Fortran/C Wrapper Interface

This section describes the process of generating wrappers for C routines called by Fortran. If you want to call existing C routines (which use value parameters rather than reference parameters) from Fortran, these *wrappers* convert the parameters during the call.

The program *mkf2c* provides an alternate interface for C routines called by Fortran.

Fortran routines called by C must use the method described in **"Fortran/C Wrapper Interface"** .

The Wrapper Generator mkf2c

The *mkf2c* program uses C data−type declarations for parameters to generate the correct assembly language interface. In generating a Fortran−callable entry point for an existing C−callable function, the C function is passed through *mkf2c*, and *mkf2c* adds additional entry points. Native language entry points are not altered.

Use these rules with *mkf2c*:

- Each function given to *mkf2c* must have the standard C function syntax.
- The function body must exist but can be empty. Function names are transformed as necessary in the output.

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

```
func()
{}
```

Here, the function **func** has no parameters. If *mkf2c* is used to produce a Fortran−to−C wrapper, the Fortran entry is **func_**. The wrapper **func_** simply calls the C routine **func***()*.

Here is another example:

```
simplefunc (a)
int a;
{}
```

In this example, the function **simplefunc** has one argument, **a**. The argument is of type **int**. For this function, **mkf2c** produces three items: a Fortran entry, **simple**, and two pieces of code. The first piece of code dereferences the address of **a**, which was passed by Fortran. The second passes the resulting **int** to C. It then calls the C routine **simplefunc()**.

Using Fortran Character Variables as Parameters

You can specify the length of a character variable passed as a parameter to Fortran either at compilation or at run time. The length is determined by the declaration of the parameter in the Fortran routine. If the declaration contains a length, the passed length must match the declaration. For example, in the following declaration, the length of the string is declared to be 10 characters:

```
character*10 string
```

The passed length must be 10 in order to match the declaration.

When this next declaration is used, the passed length is taken for operations performed on the variable inside the routine:

```
character*(*) string
```

The length can be retrieved by use of the Fortran intrinsic function **LEN**. Substring operations may cause Fortran run−time errors if they do not check this passed length.

Arrays of character variables are treated by Fortran as simple byte arrays, with no alignment of elements. The length of the individual elements is determined by the length passed at run time. For instance, the array **sarray()** can be declared in this manner:

```
character*(*) sarray()
```

This length is necessary to compute the indexes of the array elements. The program *mkf2c* has special constructs for dealing with the lengths of Fortran character variables.

Reduction of Parameters

The program *mkf2c* reduces each parameter to one of seven simple objects. The following list explains each object.

64−bit value The quantity is loaded from the passed address, and the result is passed to C. This applies to parameters with the C type **double**. When you specify the **-64bit** option, parameters with the C type **long** are also treated as 64−bit values and correspond to **INTEGER*8** parameters (refer to the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for details about the **-64bit** option).

32−bit value *mkf2c* uses the passed address to retrieve a 32−bit data value, which is then passed to C. Any parameter with an unspecified type is assumed to be **int**. If you specify the **-f** option, parameters with the C type **float** retain their 32−bit values. Otherwise, the 32−bit Fortran **REAL** value is promoted to **double** before parsing to C.

16−bit value A 16−bit value is loaded using the passed address. The value is either extended (if type is **signed** in the function parameter list) or masked (if type is **unsigned**) and passed to C. Any parameter whose C type is **short** is reduced to a 16−bit value.

8−bit value The **char** type in C corresponds to the **CHARACTER*1** type in Fortran 77. (There is no mechanism to pass **INTEGER*1** variables to C. A pointer to the value can be passed by declaring the parameter as **int***.) By default the character value is loaded as an unsigned quantity and passed to C. If the **-signed** option has been specified when invoking *mkf2c*, the character value is sign extended before being passed to C.

character string A copy is made of the Fortran character variable, and it is null terminated, and passed as a character pointer to C. Any modifications that C makes to the string will not affect the corresponding character variable in the Fortran routine.

character array When using *mkf2c* to call C from Fortran, the address of the Fortran character variable is passed. This character array can be modified by C. It is not guaranteed to be null terminated. The length of the Fortran character variable is treated differently (as discussed in the next section).

pointer The value found on the stack is treated as a pointer and is passed without alteration. Any array or pointer that is not of type **char**, any parameter with multiple levels of indirection, or any indirect array is assumed to be of type **pointer**. If the type of a parameter is specified but is not one of the standard C types, *mkf2c* will pass it as a pointer. The **-64bit** option is required to deal with 64–bit pointers (refer to the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for details).

Below is an example of a C specification for a function:

```
test (i,s,c,ptr1,ar1,u,f,d,d1,str1,str2,str3)
short s;
unsigned char c;
int *ptr1;
char *ptr2[];
short ar1[];
sometype u;
float f;
double d, *d1;
char *str1;
char str2[],str3[30];
{
     /* The C function body CAN go here. Nothing
     except the opening and closing braces are
     necessary */
}
```

If this function were passed to *mkf2c*, the parameters would be transformed as follows:

- **PTR1**, **PTR2**, **AR1**, **D1**, and **U** would be passed as simple pointers.
- *mkf2c* would complain about not understanding the type **SOMETYPE** but, by default, would assume it to be of type **POINTER**.
- **S**, **C**, and **D** would be passed as values of length 16 bits, 8 bits, and 64 bits, respectively. **F** would be converted to a 64–bit **DOUBLE** before being passed, unless the **-f** option had been specified. If the **-f** option had been specified, **F** would be passed as a 32–bit value. Because the type of **I** is not specified, it would be assumed to be **INT** and would also be passed as a 32–bit value. Storing values in any of these parameters would not have any effect on the original Fortran data.

Fortran Character Array Lengths

When the wrapper generator is used, a character variable that is specified as **char*** in the C parameter list is *copied* and null terminated. C may thus determine the length of the string by the use of the standard C function **strlen**.

If a character variable is specified as a character array in the C parameter list, the address of the character variable is passed, making it impossible for C to determine its length, as it is not generated to be null terminated. When the call occurs, the wrapper code receives this length from Fortran. For those C functions needing this information, the wrapper passes it by extending the C parameter list.

For example, if the C function header is specified as follows

```
func1 (carr1,i,str,j,carr2)
char carr1[],*str,carr2[];
int i, j;
{}
```

mkf2c passes a total of seven parameters to C. The sixth parameter is the length of the Fortran character variable corresponding to **carr1**, and the seventh is the length of **carr2**. The C function **func1()** must 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 and its counterpart **va_alist** appearing at the end of the parameter type list.

In the case above, use of these macros produces the function header

```
#include "varargs.h"
```

```
func1 (carr1,i,str,j,carr2,va_alist)
char carr1[], *str, carr2[];
int i, j;
va_dcl
{}
```

The C routine could retrieve the lengths of **carr1** and **carr2**, placing them in the local variables **carr1_len** and **carr2_len** by the following code fragment:

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

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 cannot understand constructs such as typedefs, external function declarations, or C preprocessor directives.

To ensure that only those 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*.

To illustrate the use of *extcentry*, the C file **foo.c** is shown below. It contains the function **foo**, which is to be made Fortran callable.

```
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** on your workstation.

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, an executable object file is

created from the files **main.f** (a Fortran main program) and **callc.c**:

```
test:  main.o callc.o
     f77 -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 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**.

Chapter 4

System Functions and Subroutines

This chapter describes extensions to Fortran 77 that are related to the IRIX compiler and operating system.

- **"Library Functions"** summarizes the Fortran run–time library functions.
- **"Intrinsic Subroutine Extensions"** describes the extensions to the Fortran intrinsic subroutines.
- **"Function Extensions"** describes the extensions to the Fortran functions.

Library Functions

The Fortran library functions provide an interface from Fortran programs to the system in the same way that the C library provides for C programs. The compiler automatically loads an interface routine when it processes the associated call.

Table 4–1 summarizes the functions in the Fortran run–time library.

Table 4–1 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
brk	change data segment space allocation
chdir	change default directory
chmod	change mode of a file
chown	change owner
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
ftell	reposition a file on a logical unit
ftell64	reposition a file on a logical unit for 64−bit architecture
gerror	get system error messages
getarg	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 command line arguments
idate	return date or time in numerical form
ierrno	get system error messages
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
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	get system error messages
pipe	create an interprocess channel
plock	lock process, test, or data in memory
prctl	control processes
profil	execution−time profile
ptrace	process trace
putc	write a character to a Fortran logical unit
putenv	set environment variable
qsort	quick sort
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
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
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[a]	return system time

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 file system
unblockproc	unblock processes
unlink	remove a directory entry
uscalloc	shared memory allocator
uscalloc64	shared memory allocator in 64–bit environment
uscas	compare and swap operator
usclosepollsema	detach file descriptor from a pollable semaphore
usconfig	semaphore and lock configuration operations
uscpsema	acquire a semaphore
uscsetlock	unconditionally set lock
usctlsema	semaphore control operations
usdumplock	dump lock information
usdumpsema	dump semaphore information
usfree	user shared memory allocation
usfreelock	free a lock
usfreepollsema	free a pollable semaphore
usfreesema	free a semaphore
usgetinfo	exchange information through an arena
usinit	semaphore and lock initialize routine
usinitlock	initialize a lock
usinitsema	initialize a semaphore
usmalloc	allocate shared memory
usmalloc64	allocate shared memory in 64–bit environment
usmallopt	control allocation algorithm
usnewlock	allocate and initialize a lock
usnewpollsema	allocate and initialize a pollable semaphore
usnewsema	allocate and initialize a semaphore
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
ussetlock	set lock
ustest lock	test lock
ustestsema	return value of semaphore
ustrace	trace
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

[a]The library function *time* can be invoked only if it is declared in an external statement. Otherwise, it will be misinterpreted as the VMS–compatible intrinsic subroutine *time.*

You can display information on a *function* with the *man* command:

```
% man function
```

Intrinsic Subroutine Extensions

This section describes the intrinsic subroutines that are extensions to Fortran 77. The rules for using the intrinsic subroutines are given below.

- The subroutine names are specially recognized by the compiler. A user−written subroutine with the same name as a system subroutine must be declared in an **EXTERNAL** statement in the calling subprogram.
- Using a user−written subroutine with the same name as a system subroutine in one subprogram does not preclude using the actual system subroutine in a different subprogram.
- To pass the name of a system subroutine as an argument to another subprogram, the name of the system subroutine must be declared in an **INTRINSIC** statement in the calling subprogram.
- When a system subroutine name is passed as an argument to another subprogram, the call to the system subroutine via the formal parameter name in the receiving subprogram must use the primary calling sequence for the subprogram (when there is more than one possible calling sequence).

Table 4−2 gives an overview of the system subroutines and their function; they are described in detail in the sections following the table.

Table 4−2 Overview of System Subroutines

Subroutine	Information Returned
DATE	Current date as nine−byte string in ASCII representation
IDATE	Current month, day, and year, each represented by a separate integer
ERRSNS	Description of the most recent error
EXIT	Terminates program execution
TIME	Current time in hours, minutes, and seconds as an eight−byte string in ASCII representation
MVBITS	Moves a bit field to a different storage location

DATE

The **DATE** routine returns the current date as set by the system; the format is as follows:

```
CALL DATE (buf)
```

where *buf* is a variable, array, array element, or character substring nine bytes long. After the call, *buf* contains an ASCII variable in the format *dd−mmm−yy*, where *dd* is the date in digits, *mmm* is the month in alphabetic characters, and *yy* is the year in digits.

IDATE

The **IDATE** routine returns the current date as three integer values representing the month, date, and year; the format is as follows:

```
CALL IDATE (m, d, y)
```

where *m*, *d*, and *y* are either **INTEGER*4** or **INTEGER*2** values representing the current month, day and year. For example, the values of *m*, *d*, and *y* on August 10, 1989, are

```
m = 8
d = 10
y = 89
```

ERRSNS

The **ERRSNS** routine returns information about the most recent program error; the format is as follows:

```
CALL ERRSNS (arg1, arg2, arg3, arg4, arg5)
```

The arguments (*arg1*, *arg2*, and so on) can be either **INTEGER*4** or **INTEGER*2** variables. On return from **ERRSNS**, the arguments contain the information shown in **Table 4−3** .

Table 4−3 Information Returned by ERRSNS

Argument	Contents
arg1	IRIX global variable *errno*, which is then reset to zero after the call
arg2	Zero
arg3	Zero
arg4	Logical unit number of the file that was being processed when the error occurred

arg5 Zero

Although only *arg1* and *arg4* return relevant information, *arg2*, *arg3*, and *arg5* are always required.

EXIT

The **EXIT** routine causes normal program termination and optionally returns an exit–status code; the format is as follows:

```
CALL EXIT (status)
```

where *status* is an **INTEGER*4** or **INTEGER*2** argument containing a status code.

TIME

The **TIME** routine returns the current time in hours, minutes, and seconds; the format is as follows:

```
CALL TIME (clock)
```

where *clock* is a variable, array, array element, or character substring; it must be eight bytes long. After execution, *clock* contains the time in the format *hh:mm:ss*, where *hh*, *mm*, and *ss* are numerical values representing the hour, the minute, and the second.

MVBITS

The **MVBITS** routine transfers a bit field from one storage location to another; the format is as follows:

```
CALL MVBITS (source,sbit,length,destination,dbit)
```

Table 4–4 defines the arguments. Arguments can be declared as **INTEGER*2**, **INTEGER*4**, or **INTEGER*8**.

Table 4–4 Arguments to MVBITS

Argument	Type	Contents
source	Integer variable or array element	Source location of bit field to be transferred.
sbit	Integer expression	First bit position in the field to be transferred from source.
length	Integer expression	Length of the field to be transferred from source.
destination	Integer variable or array element	Destination location of the bit field
dbit	Integer expression	First bit in destination to which the field is transferred.

Function Extensions

Table 4–5 gives an overview of the functions added as extensions of Fortran 77.

Table 4–5 Function Extensions

Function	Information Returned
SECNDS	Elapsed time as a floating point value in seconds. This is an intrinsic routine.
RAN	The next number from a sequence of pseudo–random numbers. This is not an intrinsic routine.

These functions are described in detail in the following sections.

SECNDS

SECNDS is an intrinsic routine that returns the number of seconds since midnight, minus the value of the passed argument; the format is as follows:

```
s = SECNDS(n)
```

After execution, *s* contains the number of seconds past midnight less the value specified by *n*. Both *s* and *n* are single–precision, floating point values.

RAN

The **RAN** routine generates a random number; the format is as follows:

```
v = RAN(s)
```

The argument *s* is an **INTEGER*4** variable or array element; *s* serves as a seed in determining the next random number and should initially be set to a large, odd integer value. This permits the computation of multiple random number series by supplying different variable names as the seed argument to **RAN**.

Note: Because **RAN** modifies the argument *s*, calling the function with a constant can cause a core dump.

Chapter 5

Scalar Optimizations

This chapter contains the following sections:

- **"Overview"** provides an overview of the scalar optimization command line options.
- **"Performing General Optimizations"** describes the general scalar optimizations you can enable from the command line.
- **"Performing Advanced Optimizations"** describes the advanced scalar optimizations you can enable from the command line.

Overview

You can use the compiler to perform various scalar optimizations by specifying any of the options listed in **Table 5–1** from the command line. Specify the options in a comma–separated list following the **-WK** option without any intervening blanks, as follows:

```
% f77   f77options   -WK,option[,option] ... file
```

Note: These options specifically control optimizations performed by the Fortran front end. The defaults are usually sufficient. You should use these options when trying to improve the last bit of performance of your code.

You can also initiate many of these optimizations with compiler directives (see **Chapter 9, "Fine–Tuning Program Execution."**)

Table 5–1 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

The **-O***n* option directly initiates basic optimizations. Refer to **Chapter 1, "Compiling, Linking, and Running Programs"** for details.

Performing General Optimizations

This section discusses the general optimizations that you can enable.

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.

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 **"Controlling Global Assumptions" in Chapter 5**). The default is **-assume=cel**.

list can contain the following characters:

a	Allows procedure argument aliasing, which is when different subroutine or function parameters refer to the same object. This practice is forbidden by the Fortran 77 standard. This option provides a method of dealing with programs that use argument aliasing anyway.
b	Allows array subscripts to go outside the declared bounds.
c	Places constants used in subroutine or function calls in temporary variables.
e	Allows variables in **EQUIVALENCE** statements 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 77 standard, that is, **-assume=el**, and includes **-asssume=c** to simplify some analysis and inlining. You can disable the default values by specifying the **-noassume** option.

Example

The following command compiles the Fortran program **source.f**, and permits argument aliasing and subscripts out of bounds:

```
% f77 -WK,-assume=ab source.f
```

Setting Invariant IF Floating Limits

When a loop contains an **IF** statement whose condition does not change from one iteration to another (loop–invariant), 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=***integer* option (or **-eiifg=***integer*) 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.

Valid values for *integer* are from 0 to 100; the default is 20.

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.

For example, the following code:

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

becomes

```
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 slow a program down (through cache contention, extra paging, and so on) more than the reduced number of **IF** tests speed it up. The **-each_invariant_if_growth** option provides a maximum size (in number of lines of executable code) of sections 1 and 4, below which the compiler will try to float an invariant **IF** statement outside a loop.

This can be controlled on a loop–by–loop basis with the **C*$* EACH_INVARIANT_IF_GROWTH** (*integer*) directive within the source (see **"Setting Invariant IF Floating Limits" in Chapter 9**).

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.

The **-max_invariant_if_growth=***integer* option (or **-miifg=***integer*) specifies an upperbound 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 *integer* 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**.

This can be controlled on a loop−by−loop basis with the **C*$* MAX_INVARIANT_IF_GROWTH (***integer***)** directive within the source (see **"Setting Invariant IF Floating Limits" in Chapter 9**).

Setting the Optimization Level

The **-optimize=***integer* option (or **-o=***integer*) 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 the **C*$* OPTIMIZE**(*integer*) directive within the source (see **"Optimization Level" in Chapter 9**).

Valid values for *integer* are:

fe0	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 wrap−around variables.
4	Generates two versions of a loop, if necessary, to break a data dependence arc.
5	Enables array expansion and loop fusion.

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. Refer to **"Optimizing" in Chapter 1** for details.

Although higher optimization levels increase performance, they also increase compilation time.

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 **-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):

```
        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
```

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 Variations in Round Off

The **-roundoff=***integer* option (or **-r=***integer*) controls the amount of variation in round−off error produced by optimization. If an arithmetic reduction is accumulated in a different order than in the scalar program, the round−off 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 *integer* 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 *integer* are

0 Suppresses any transformations that change round−off 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).

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. Refer to **"Optimizing" in Chapter 1** for details.

Example

Consider the following code segment:

```
        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.

```
      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) the original code becomes:

```
              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. In this example,

the compiler performs forward substitution of the transformed induction variable **X**.

The following code:

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

becomes

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

Controlling Scalar Optimizations

The **-scalaropt=***integer* option (or **-so=***integer*) controls the level of scalar optimizations that the compiler performs. Valid values for *integer* 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 invariant **IF** statements out of loops, loop rerolling and unrolling (if **-roundoff** is greater than zero), array expansion, loop fusion, loop peeling, and induction variable recognition.
3	Enables memory management transformations if **-roundoff=3** (see **"Performing Memory Management Transformations"** for details about memory management transformations). Performs dead–code elimination during output conversion.

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. Refer to **"Optimizing" in Chapter 1** for details.

Unlike the **-scalaropt** command line option, the **C*$* SCALAR OPTIMIZE** directive sets the level of loop–based optimizations (for example, loop fusion) only, and not straight–code optimizations (for example, dead–code elimination). Refer to **"Controlling Scalar Optimizations" in Chapter 9** for details about the **C*$* SCALAR OPTIMIZE** directive.

Performing Advanced Optimizations

This section describes advanced optimization techniques you can use to obtain maximum performance.

Using Aggressive Optimization

The **-aggressive=***letter* option (or **-ag=***letter*) performs optimizations that are normally forbidden. When using 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.

```
% f77 -WK,-ag=a program.f
```

For example, on a machine with a 64–kilobyte direct–mapped cache, a **COMMON** definition such as:

```
COMMON /alpha/ a(128,128),b(128,128),c(128,128)
```

can degrade performance if your program contains the following statement:

```
a(i,j) = b(i,j) * c(i,j)
```

All three of the arrays **a**, **b**, and **c** have the same starting virtual address modulo the cache size, and so every access to the array elements causes a cache miss. It would be much better to add some padding between each of the arrays to force the virtual addresses to be different. The **-aggressive=a** option does exactly this. Unfortunately, this transformation is not always possible. Fortran allows different routines to have different definitions of **COMMON**. If some other routine contained the definition

```
COMMON /alpha/ scratch(49152)
```

the compiler could not arbitrarily add padding. Therefore, when using this option the entire program must be in a single source file, so the compiler can check for this sort of occurrence.

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: The number of data dependencies (and the time required to do the analysis) is potentially non–linear in the length of the loop. Very long loops (several hundred lines) may be impossible to analyze regardless of the value of **-arclimit**.

You can use the **-arclimit** option to increase the size of the data structure to enable the compiler to perform more optimizations. (Most users do *not* need to change this value.)

Performing Memory Management Transformations

Memory management transformations are advanced optimizations you can enable by specifying options along with the **-WK** option.

Memory Management Techniques

When both **-roundoff** and **-scalaropt** are set to 3, the compiler attempts to perform outer loop unrolling (to improve register utilization) and automatic loop blocking (to improve cache utilization).

Normal loop unrolling (enabled with the **-unroll** and **-unroll2** options) applies to the innermost loop in a nest of loops. In outer loop unrolling, one of the other loops (typically the next innermost) is unrolled. In certain situations, this technique (also called "unroll and jam") can greatly improve the register utilization.

Loop blocking is a transformation that can be applied when the loop nesting depth is greater than the dimensions of the data arrays being manipulated. For example, the simple matrix multiply uses a nest of three loops operating on two–dimensional arrays. The simple approach repeatedly sweeps across the entire arrays. A better approach is to break the arrays up into blocks, each block being small enough to fit into the cache, and then make repeated sweeps over each (in cache) block. (This technique is also sometimes called "tiles" or "tiling.") However, the code needed to implement a block style algorithm is often very complex and messy. This automatic transformation allows you to write the simpler method, and have the compiler transform it into the more complex and efficient block method.

Memory Management Options

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 an unrolling factor.
- **-dpregisters** ensures that registers do not overflow during loop unrolling.
- **-setassociativity** specifies which memory management transformation to use.

The **-cacheline=***integer* option (or **-chl=***integer*) specifies the width of the memory channel, in bytes, between the cache and main memory. The default value for *integer* is 4. Refer to **Table 5–2** for the recommended setting for your machine.

The **-cachesize=***integer* option (or **-chs**=*integer*) specifies the size of the data cache, in kilobytes, for which to optimize. The default value for *integer* is 256 kilobytes. Refer to **Table 5–2** for the recommended setting for your machine. You can obtain the cache size for a given machine with the *hinv* (1) command. This option is generally useful only in conjunction with the other memory management transformations.

Table 5–2 Recommended Cache Option Settings

Machine	Cacheline Value	Cache Size Value
POWER Series 4D/100	16	64
POWER Series 4D/200	64	64
R4000 (including Crimson™ and Indigo^{2}™)	16	8
CHALLENGE™ and POWER CHALLENGE™ Series	128	16

The **-setassociativity=***integer* option (or **-sasc=***integer*) provides information on the mapping of physical addresses in main memory to cache pages. The default value for *integer*, 1, says 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. SGI recommends you set this value to 1 for all machines, except the POWER CHALLENGE series, where you should set it to 4.

The **-dpregisters=***integer* option (or **-dpr=***integer*) specifies the number of **DOUBLE PRECISION** registers each processor has. The **-fpregisters** option (or **-fpr=***integer*) specifies the number of single precision (that is, ordinary floating point) registers each processor has.

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

Enabling Loop Unrolling

The **-unroll** and the **-unroll2** options control how the compiler unrolls scalar loops. When loops cannot be optimized for concurrent execution, loop execution is often more efficient when the loops are unrolled. (Fewer iterations with more work per iteration require less overhead overall.) You must also specify **-scalaropt= 2** when using these options.

The **-unroll=***integer* (or **-ur=***integer*) option directs the compiler to unroll inner loops. *integer* specifies the number of times to replicate the loop. The default value is 4.

0	Uses default values to unroll.
1	Disables unrolling.
2–*n*	Unrolls at most, this many iterations.

The **-unroll2=***weight* (or **-ur2=***weight*) option specifies an upper bound on the number of operations in a loop when unrolling it with the **-unroll** option. The default value for *weight* is 100. The compiler unrolls an inner loop until the number of operations (the amount of work) in the unrolled loop is close to this upper bound, or until the number of iterations specified in the **-unroll** option is reached, whichever occurs first.

For the **-unroll2** option the compiler analyzes a given loop by computing an estimate of the computational work that is inside the loop for *one* iteration. This rough estimate is obtained by adding the number of:

- assignments
- **IF** statements
- subscripts
- arithmetic operations

The following example uses the **C*$* UNROLL** directive (see **"Enabling Loop Unrolling" in Chapter 9**) to specify 8 for the maximum number of iterations to unroll and 100 for the maximum "work per unrolled

iteration." (This is equivalent to specifying **-WK,-unroll=8,-unroll2=100**.)

```
C*$*UNROLL(8,100)
        DO 10 I = 2,N
           A(I) = B(I)/A(I-1)
   10   CONTINUE
```

This example has:

1 assignment
0 **IF** statements
3 subscripts
2 arithmetic operators
————————————————————
6 is the weighted sum (the work for 1 iteration)

This weighted sum is then divided into 100 to give a potential unrolling factor of 16. However, the example has also specified 8 for the maximum number of unrolled iterations. The compiler takes the minimum of the two values (8) and unrolls that many iterations. (The maximum number of iterations the compiler unrolls is 100.)

In this case (an unknown number of iterations), the compiler generates two loops – the primary unrolled loop and a cleanup loop to ensure that the number of iterations in the main loop is a multiple of the unrolling factor. The result is the following:

```
        INTEGER I1
C*$*UNROLL(8,100)
        I1 = MOD (N - 1, 8)
        DO 2 I=2,I1+1
        A(I) = B(I) / A(I-1)
    2   CONTINUE
        DO 10 I=I1+2,N,8
          A(I) = B(I)/A(I-1)
          A(I+1) = B(I+1) / A(I)
          A(I+2) = B(I+2) / A(I+1)
          A(I+3) = B(I+3) / A(I+2)
          A(I+4) = B(I+4) / A(I+3)
          A(I+5) = B(I+5) / A(I+4)
          A(I+6) = B(I+6) / A(I+5)
          A(I+7) = B(I+7) / A(I+6)
    10  CONTINUE
```

Recognizing Directives

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

a	Accepts Silicon Graphics **C*$* ASSERT** assertions.
c	Accepts Cray **CDIR$** directives.
k	Accepts Silicon Graphics **C*$*** and **C$PAR** directives.
p	Accepts parallel programming directives.
s	Accepts Sequent **C$** directives.
v	Accepts VAST **CVD$** directives.

The default value for *list* is **ackpv**. 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. **Chapter 9, "Fine−Tuning Program Execution,"** describes the Silicon Graphics, Cray, Sequent, and VAST directives the compiler accepts.

Assertions are similar in form to directives, but they assert program characteristics that the compiler can use in its optimizations. In addition to specifying **a** in *list*, you can control whether the compiler accepts assertions using the **C*$* ASSERTIONS** and **C*$* NOASSERTIONS** directives (refer to **"Using Assertions" in Chapter 9**).

Specifying Recursion

The **-recursion** option (or **-rc**) allows the compiler to call subroutines and functions in the source program recursively (that is, a subroutine or function calls itself, or it calls another routine which calls it). Recursion affects storage allocation decisions.

This option is enabled by default. To disable it, specify **-norecursion** (or **-nrc**).

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

Chapter 6

Inlining and Interprocedural Analysis

This chapter contains the following sections:

- **"Overview"** describes inlining and interprocedural analysis.
- **"Using Command Line Options"** explains how to use command line options to perform inlining and interprocedural analysis (IPA).
- **"Conditions That Prevent Inlining and IPA"** lists several conditions that prevent inlining and interprocedural analysis.

Overview

Inlining is the process of replacing a function reference with the text of the function. This process eliminates the overhead of the function call and can assist other optimizations by making relationships between function arguments, returned values, and the surrounding code easier to find.

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

You can perform inlining and IPA from the command line and using directives in your source code.

Using Command Line Options

The compiler performs inlining and IPA when you specify the options listed in **Table 6–1** along with the **-WK** option using the following syntax:

```
% f77 [f77option ...] -WK,option[,option]... file
```

f77_option is any option you can specify directly to the compiler and *option* is any of the options listed in **Table 6–1** .

Table 6–1 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 Routines for Inlining or IPA

The **-inline**[**=***list*] option (or **-inl**[**=***list*]) provides a list of routines to be expanded inline; the **-ipa**[**=***list*] option provides a list of routines to be analyzed. The routine names in *list* must be separated by colons. If you do not specify a list of routines, the compiler expands all eligible routines. 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.

Example

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

```
% f77 -WK,-inline=saxpy:daxpy foo.f
```

Refer to **"Conditions That Prevent Inlining and IPA"** for information about conditions that prevent inlining and IPA.

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.

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.

integer is defined from the most deeply nested leaf of the call graph. 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. For example, if you specify 1 for *integer*, the compiler expands routines in only the most deeply nested loop. If you specify 2 for *integer*, the compiler expand routines in the deepest and second deepest nested loops, and so on. Specifying a large number for *integer* enables inlining/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.

Example

Consider the following code:

```
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 routines referenced in the most deeply−nested call sites are inlined (subroutine C in the above example). (If more than one routine is called at the same loop nest level, the compiler selects all of them when that level is inlined/analyzed.)

-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**, then the new **B** to be inlined 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.

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" in Chapter 9**).

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, investigate using IPA, which has little effect on code space and the number of temporary variables.)

Depth

The **-inline_depth=***integer* option (or **-ind=***integer*) restricts the number of times 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.
−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.

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). This option limits the number of times the compiler performs this recursive expansion. Note that the default setting is quite low; if you know inlining is useful for a particular program, increase this setting.

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

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

Manual Control

The **-inline_man** (or **-inm**) option enables recognition of the **C*$* INLINE** directive. This directive, described in **"Fine−Tuning Inlining and IPA" in Chapter 9** , allows you to select individual instances of routines to be inlined. The **-ipa_man** (or **-ipam**) option is the analogous option for the **C*$* IPA** directive.

Specifying Where to Search for Routines

The options listed in **Table 6–2** tell 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 6–2 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*

If one of the names in *list* is a directory, the compiler uses all appropriate files in that directory. You can specify multiple files and directories simultaneously using a colon–separated list.

For example

```
–WK,–inline_from_files=file1:file2:file3
```

The compiler recognizes the type of file from its extension, or lack of one, as described in **Table 6–3** .

Table 6–3 Filename Extensions

Extension	Type of File
.f, .F, .for, .FOR	Fortran source
.i	Fortran source run through cpp
.klib	Library created with -inline_create or -ipa_create option
Other	Directory

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

Example

The following command specifies inline expansion on the source file, **calc.f**:

```
% f77 –WK,–inline,–inline_from_files=–:input.f calc.f
```

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

If you specify a non–existent file or directory, the compiler issues an error.

If you specify multiple **-inline_from** or **-ipa_from** options, the compiler concatenates their lists to produce a bigger universe. The lists are searched in the order that they appear on the command line.

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.

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.

Creating Libraries

When performing inlining and IPA, the compiler analyzes the routines in the source program. Normally, inlining is done directly from a source file. However, when inlining the same set of routines in 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 routines (for later use with the **-inline_from_libraries** option). The compiler assigns *name* to the library file it creates; for maximum compatibility, use the file name extension **.klib**. For example: **samp.klib**.

The **-ipa_create**[**=***name*] option (or **-ipacr**[**=***name*]) is the analogous option for IPA.

You do not have to generate your inlining/IPA library from the same source that will actually be linked into the running program. This capability can cause errors, but it can also be quite 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, but still actually call the hand−optimized assembly routines.

The procedure for creating and using a library for inlining or IPA is given below.

1. Create a library using the **-inline_create** option (or the **-ipa_create** option for IPA). For example, the following command line creates a library called **prog.klib** for the source program **prog.f**:

   ```
   % f77 -WK,-inline_create=prog.klib prog.f
   ```

   When you specify this option the compiler creates only the library; it does not compile the source program or create a transformed version of the file.

2. Compile the program with inlining enabled and specify the new library:

   ```
   % f77 -WK,-inl,-inlf=prog.klib samp.f
   ```

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.

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.

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.

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 (he 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 7

Fortran Enhancements for Multiprocessors

This chapter contains these sections:

- **"Overview"** provides an overview of this chapter.
- **"Parallel Loops"** discusses the concept of parallel **DO** loops.
- **"Writing Parallel Fortran"** explains how to use compiler directives to generate code that can be run in parallel.
- **"Analyzing Data Dependencies for Multiprocessing"** describes how to analyze **DO** loops to determine whether they can be parallelized.
- **"Breaking Data Dependencies"** explains how to rewrite **DO** loops that contain data dependencies so that some or all of the loop can be run in parallel.
- **"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 the cache.
- **"Advanced Features"** describes features that override multiprocessing defaults and customize parallelism.
- **"DOACROSS Implementation"** discusses how multiprocessing is implemented in a **DOACROSS** routine.
- **"PCF Directives"** describes how the PCF directives implement a general model of parallelism.

Overview

The Silicon Graphics Fortran compiler allows you to apply the capabilities of a Silicon Graphics multiprocessor workstation to the execution of a single job. By coding a few simple directives, the compiler splits the job into concurrently executing pieces, thereby decreasing the wall–clock run time of the job.

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 Loops

The model of parallelism used focuses on the Fortran **DO** loop. The compiler executes different iterations of the **DO** loop in parallel on multiple processors. For example, suppose a **DO** loop consisting of 200 iterations will run on a machine with four processors using the **SIMPLE** scheduling method (described in **"CHUNK, MP_SCHEDTYPE"**). The first 50 iterations run on one processor, the next 50 on another, and so on. The multiprocessing code adjusts itself at run time to the number of processors actually present on the machine. Thus, if the above 200–iteration loop was moved to a machine with only two processors, it would be divided into two blocks of 100 iterations each, without any need to recompile or relink. In fact, multiprocessing code can even be run on single–processor machines. The above loop would be divided into one block of 200 iterations. This allows code to be developed on a single–processor Silicon Graphics workstation, and later run on an IRIS POWER Series multiprocessor.

The processes that participate in the parallel execution of a task are arranged in a master/slave organization. The original process is the master. It creates zero or more slaves to assist. When a parallel **DO** loop is encountered, the master asks the slaves for help. When the loop is complete, the slaves wait on the master, and the master resumes normal execution. The master process and each of the slave processes are called a *thread of execution* or simply a *thread*. By default, the number of threads is set equal to the number of processors on the particular machine (this number cannot exceed four). If you want, you can override the default and explicitly control the number of threads of execution used by a Fortran job.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of when any other iteration of the

loop is executed. Not all **DO** loops have this property, and loops without it cannot be correctly executed in parallel. However, many of the loops encountered in practice fit this model. Further, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially in parallel.

To provide compatibility for existing parallel programs, Silicon Graphics has chosen to adopt the syntax for parallelism used by Sequent Computer Corporation. This syntax takes the form of compiler directives embedded in comments. These fairly high−level directives provide a convenient method for you to describe a parallel loop, while leaving the details to the Fortran compiler. For advanced users the proposed Parallel Computing Forum (PCF) standard (ANSI−X3H5 91−0023−B Fortran language binding) is available (refer to **"PCF Directives"**). Additionally, there are a number of special routines that permit more direct control over the parallel execution (refer to **"Advanced Features"** for more information.)

Writing Parallel Fortran

The Fortran compiler accepts directives that cause it to generate code that can be run in parallel. The compiler directives look like Fortran comments: they begin with a **C** in column one. If multiprocessing is not turned on, these statements 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 directives are distinguished by having a **$** as the second character. There are six directives that are supported: **C$DOACROSS**, **C$&**, **C$**, **C$MP_SCHEDTYPE**, **C$CHUNK**, and **C$COPYIN**. The **C$COPYIN** directive is described in **"Local COMMON Blocks"** . This section describes the others.

C$DOACROSS

The essential compiler directive for multiprocessing is **C$DOACROSS**. This directive directs the compiler to generate special code to run iterations of a **DO** loop in parallel. The **C$DOACROSS** directive applies only to the next statement (which must be a **DO** loop).

The **C$DOACROSS** directive has the form

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

where valid values for the 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. This section discusses the meaning of each clause.

IF

The **IF** clause determines whether the loop is actually executed in parallel. If the logical expression 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 that there is enough work in the loop to amortize the overhead of parallel execution. Currently, the break−even point is about 4000 CPU clocks of work, which normally translates to about 1000 floating point operations.

LOCAL, SHARE, LASTLOCAL

The **LOCAL**, **SHARE**, and **LASTLOCAL** clauses specify lists of variables used within parallel loops. A variable can appear in only one of these lists. To make the task of writing these lists easier, there are several defaults. The loop−iteration variable is **LASTLOCAL** by default. All other variables are **SHARE** by default.

LOCAL Specifies variables that are local to each process. If a variable is declared as **LOCAL**, each iteration of the loop is given its own uninitialized copy of the variable. You can

each iteration of the loop is given its own uninitialized copy of the variable. You can declare a variable as **LOCAL** if its value does not depend on any other iteration of the loop and if its value is used only within a single iteration. In effect the **LOCAL** variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. The name **LOCAL** is preferred over **PRIVATE**.

SHARE Specifies variables that are shared across all processes. If a variable is declared as **SHARE**, all iterations of the loop use the same copy of the variable. You can declare a variable as **SHARE** if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. The name **SHARE** is preferred over **SHARED**.

LASTLOCAL Specifies variables that are local to each process.Unlike with the **LOCAL** clause, the compiler saves only the value of the logically last iteration of the loop when it exits. The name **LASTLOCAL** is preferred over **LAST LOCAL**.

LOCAL is a little faster than **LASTLOCAL**, so if you do not need the final value, it is good practice to put the **DO** loop index variable into the **LOCAL** list, although this is not required.

Only variables can appear in these lists. In particular, **COMMON** blocks cannot appear in a **LOCAL** list (but see the discussion of local **COMMON** blocks in **"Advanced Features"**). The **SHARE**, **LOCAL**, and **LASTLOCAL** lists give only the names of the variables. If any member of the list is an array, it is listed without any subscripts.

REDUCTION

The **REDUCTION** clause specifies variables involved in a reduction operation. In a reduction operation, the compiler keeps local copies of the variables and combines them when it exits the loop. For an example and details see **"Example 4: Sum Reduction"** of **"Breaking Data Dependencies."** An element of the **REDUCTION** list must be an individual variable (also called a scalar variable) and cannot be an array. However, it can be an individual element of an array. In a **REDUCTION** clause, it would appear in the list with the proper subscripts.

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*(). Note that *min*(*max*) reductions must use the *min*(*max*) intrinsic functions to be recognized correctly.

The compiler confirms that the reduction expression is legal by making some simple checks. The compiler does not, however, check all statements in the **DO** loop for illegal reductions. You must ensure that the reduction variable is used correctly in a reduction operation.

CHUNK, MP_SCHEDTYPE

The **CHUNK** and **MP_SCHEDTYPE** clauses affect the way the compiler schedules work among the participating tasks in a loop. These clauses do not affect the correctness of the loop. They are useful for tuning the performance of critical loops. See **"Load Balancing"** for more details.

For the **MP_SCHEDTYPE=***mode* clause, *mode* can be one of the following:

```
[SIMPLE | simple | STATIC | static]
[DYNAMIC | dynamic]
[INTERLEAVE | interleave | INTERLEAVED | interleaved]
[GUIDED | guided | GSS | gss]
[RUNTIME | runtime]
```

You can use any or all of these modes in a single program. The **CHUNK** clause is valid only with the **DYNAMIC** and **INTERLEAVE** modes. **SIMPLE**, **DYNAMIC**, **INTERLEAVE**, **GSS**, and **RUNTIME** are the preferred names for each mode.

The simple method (**MP_SCHEDTYPE=SIMPLE**) divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process.

In dynamic scheduling (**MP_SCHEDTYPE=DYNAMIC**) the iterations are broken into pieces the size of which is specified with the **CHUNK** clause. As each process finishes a piece, it enters a critical section to

grab the next available piece. This gives good load balancing at the price of higher overhead.

The interleave method (**MP_SCHEDTYPE=INTERLEAVE**) breaks the iterations into pieces of the size specified by the **CHUNK** option, and execution of those pieces is interleaved among the processes. Instead of the **CHUNK** option, you can specify the **-WK,-chunk** command line option (see **"Memory Management Options" in Chapter 5** for details). 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.

The fourth method is a variation of the guided self–scheduling algorithm (**MP_SCHEDTYPE=GSS**). Here, the piece size is varied depending on the number of iterations remaining. By parceling out relatively large pieces to start with and relatively small pieces toward the end, the system can achieve good load balancing while reducing the number of entries into the critical section.

In addition to these four methods, you can specify the scheduling method at run time (**MP_SCHEDTYPE=RUNTIME**). Here, the scheduling routine examines values in your run–time environment and uses that information to select one of the other four methods. See **"Advanced Features"** for more details.

If both the **MP_SCHEDTYPE** and **CHUNK** clauses are omitted, **SIMPLE** scheduling is assumed. If **MP_SCHEDTYPE** is set to **INTERLEAVE** or **DYNAMIC** and the **CHUNK** clause are omitted, **CHUNK=1** is assumed. If **MP_SCHEDTYPE** is set to one of the other values, **CHUNK** is ignored. If the **MP_SCHEDTYPE** clause is omitted, but **CHUNK** is set, then **MP_SCHEDTYPE=DYNAMIC** is assumed.

Example 1

The code fragment

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

could be multiprocessed with the directive

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

Here, the defaults are sufficient, provided **A** and **B** are mentioned in a nonparallel region or in another **SHARE** list. The following then works:

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

Example 2

Consider the following code fragment:

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

You can be fully explicit, as shown below:

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

You can also use the defaults:

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

See Example 5 in **"Analyzing Data Dependencies for Multiprocessing"** for more information on this example.

Example 3

Consider the following code fragment:

```
     DO 10 I = M, K, N
        X = D(I)**2
        Y = X + X
        DO 20 J = I, MAX
           A(I,J) = A(I,J) + B(I,J) * C(I,J) * X + Y
20 CONTINUE
10 CONTINUE
     PRINT*, I, X
```

Here, the final values of **I** and **X** are needed after the loop completes. A correct directive is shown below:

```
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
```

You can also use the defaults:

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(X)
     DO 10 I = M, K, N
         X = D(I)**2
         Y = X + X
         DO 20 J = I, MAX
             A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20 CONTINUE
10 CONTINUE
     PRINT*, I, X
```

I is a loop index variable for the **C$DOACROSS** loop, so it is **LASTLOCAL** by default. However, even though **J** is a loop index variable, it is not the loop index of the loop being multiprocessed and has no special status. If it is not declared, it is assigned the default value of **SHARE**, which produces an incorrect answer.

C$&

Occasionally, the clauses in the **C$DOACROSS** directive are longer than one line. Use the **C$&** directive to continue the directive onto multiple lines. For example:

```
C$DOACROSS share(ALPHA, BETA, GAMMA, DELTA,
C$&  EPSILON, OMEGA), LASTLOCAL(I, J, K, L, M, N),
C$&  LOCAL(XXX1, XXX2, XXX3, XXX4, XXX5, XXX6, XXX7,
```

```
C$&  XXX8, XXX9)
```

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** and **$** are treated as if they are blanks. They can be used to insert debugging statements, or an experienced user can use them to insert arbitrary code into the multiprocessed version.

The following code demonstrates the use of the **C$** directive:

```
C$    PRINT 10
C$ 10 FORMAT('BEGIN MULTIPROCESSED LOOP')
C$DOACROSS LOCAL(I), SHARE(A,B)
        DO I = 1, 100
           CALL COMPUTE(A, B, I)
        END DO
```

C$MP_SCHEDTYPE and C$CHUNK

The **C$MP_SCHEDTYPE=***mode* directive acts as an implicit **MP_SCHEDTYPE** clause for all **C$DOACROSS** directives in scope. *mode* is any of the modes listed in the section called **"CHUNK, MP_SCHEDTYPE"** . A **C$DOACROSS** directive that does not have an explicit **MP_SCHEDTYPE** clause is given the value specified in the last directive prior to the look, rather than the normal default. If the **C$DOACROSS** does have an explicit clause, then the explicit value is used.

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 directive(s) as the first lines in the file.

Nesting C$DOACROSS

The Fortran compiler does not support direct nesting of **C$DOACROSS** loops.

For example, the following is illegal and generates a compilation error:

```
C$DOACROSS LOCAL(I)
        DO I = 1, N
C$DOACROSS LOCAL(J)
           DO J = 1, N
              A(I,J) = B(I,J)
           END DO
        END DO
```

However, to simplify separate compilation, a different form of nesting is allowed. A routine that uses **C$DOACROSS** can be called from within a multiprocessed region. This can be useful if a single routine is called from several different places: sometimes from within a multiprocessed region, sometimes not. Nesting does not increase the parallelism. When the first **C$DOACROSS** loop is encountered, that loop is run in parallel. If while in the parallel loop a call is made to a routine that itself has a **C$DOACROSS**, this subsequent loop is executed serially.

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 and/or writes a memory location repeatedly as long as no others do; it is all right if many iterations read the same location, as long as none of them write to it. In a Fortran program, memory locations are represented by variable names. So, to determine if a particular loop can be run in parallel, examine the way variables are used in the loop. Because data dependence occurs only when memory locations are modified, pay particular attention to variables that appear on the left–hand side of assignment statements. If a variable is not modified or if it is passed to a function or subroutine, there is no data dependence associated with it.

The Fortran compiler supports four kinds of variable usage within a parallel loop: **SHARE**, **LOCAL**, **LASTLOCAL**, and **REDUCTION**. If a variable is declared as **SHARE**, all iterations of the loop use the same copy. If a variable is declared as **LOCAL**, each iteration is given its own uninitialized copy. A variable is declared **SHARE** if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. A variable can be **LOCAL** if its value does not depend on any other iteration and if its value is used only within a single iteration. In effect the **LOCAL** variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. As a special case, if only the very last value of a variable computed on the very last iteration is used outside the loop (but would otherwise qualify as a **LOCAL** variable), the loop can be multiprocessed by declaring the variable to be **LASTLOCAL**. **"REDUCTION"** describes the use of **REDUCTION** variables.

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 variables' uses 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"** for information on rewriting code in parallel form.)

An alternative to analyzing variable usage by hand is to use Power Fortran. This optional software package is a Fortran preprocessor that analyzes loops for data dependence. If Power Fortran determines that a loop is data–independent, it automatically inserts the required compiler directives (see **"Writing Parallel Fortran"**). If Power Fortran cannot determine whether the loop is independent, it produces a listing file detailing where the problems lie. You can use Power Fortran in conjunction with WorkShop Pro MPF to visualize these dependencies and make it easier to understand the obstacles to parallelization.

The rest of this section is devoted to analyzing sample loops, some parallel and some not parallel.

Example 1: Simple Independence

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

In this example, each iteration writes to a different location in **A**, and none of the variables appearing on the right–hand side is ever written to, only read from. 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 the last value of **I** is used later in the code.

Example 2: Data Dependence

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

This fragment contains **A(I)** on the left–hand side and **A(I–1)** on the right. This means that one iteration of the loop writes to a location in **A** and the next iteration reads from that same location. Because different iterations of the loop read and write the same memory location, this loop cannot be run in parallel.

Example 3: Stride Not 1

```
      DO 20 I = 2,N,2
```

```
   20    A(I) = B(I) - A(I-1)
```

This example looks like the previous example. The difference is that the stride of the **DO** loop is now two rather than one. Now **A(I)** references every other element of **A**, and **A(I–1)** references exactly those elements of **A** that are not referenced by **A(I)**. None of the data locations on the right–hand side is ever the same as any of the data locations written to on the left–hand side. The data are disjoint, so there is no dependence. The loop can be run in parallel. Arrays **A** and **B** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Example 4: Local Variable

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

In this loop, each iteration of the loop reads and writes the variable **X**. However, no loop iteration ever needs the value of **X** from any other iteration. **X** is used as a temporary variable; its value does not survive from one iteration to the next. This loop can be parallelized by declaring **X** to be a **LOCAL** variable within the loop. Note that **B(I)** is both read and written by the loop. This is not a problem because each iteration has a different value for **I**, so each iteration uses a different **B(I)**. The same **B(I)** is allowed to be read and written as long as it is done by the same iteration of the loop. The loop can be run in parallel. Arrays **A** and **B** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Example 5: Function Call

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

The value of **X** in any iteration of the loop is independent of the value of **X** in any other iteration, so **X** can be made a **LOCAL** variable. The loop can be run in parallel. Arrays **A**, **B**, **C**, and **D** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

The interesting feature of this loop is that it invokes an external routine, **SQRT**. It is possible to use functions and/or subroutines (intrinsic or user defined) within a parallel loop. However, make sure that the various parallel invocations of the routine do not interfere with one another. In particular, **SQRT** returns a value that depends only on its input argument, does not modify global data, and does not use static storage. We say that **SQRT** has no *side effects*.

All the Fortran intrinsic functions listed in Appendix A of the *MIPSpro Fortran 77 Language Reference Manual* have no side effects and can safely be part of a parallel loop. For the most part, the Fortran library functions and VMS intrinsic subroutine extensions (listed in **Chapter 4, "System Functions and Subroutines,"**) cannot safely be included in a parallel loop. In particular, **rand** is not safe for multiprocessing. For user–written routines, it is the responsibility of the user to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the **-static** option when compiling routines called within a parallel loop.

Example 6: Rewritable Data Dependence

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

Here, the value of **INDX** survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written. Making **INDX** a **LOCAL** variable does not work; you need the value of

INDX computed in the previous iteration. It is possible to rewrite this loop to make it parallel (see Example 1 in **"Breaking Data Dependencies"**).

Example 7: Exit Branch

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

This loop contains an exit branch; that is, under certain conditions the flow of control suddenly exits the loop. The Fortran compiler cannot parallelize loops containing exit branches.

Example 8: Complicated Independence

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

At first glance, this loop 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**, then **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. Elements **W**, **B**, and **K** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

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.

Example 9: Inconsequential Data Dependence

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

There is a data dependence in this loop 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. In this special case, you can simply ignore it. You know that when **I** and **INDEX** are equal, the value written into **A(I)** is exactly the same as the value that is already there. The fact that some iterations of the loop read the value before it is written and some after it is written is not important because they all get the same value. Therefore, this loop can be parallelized. Array **A** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Example 10: Local Array

```
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
```

In this fragment, each iteration of the loop uses the same locations in the **D** array. 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 Fortran 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.

Therefore, this loop can be parallelized. 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 statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. If this fails, try to pull the statements out of the loop and into a separate loop, allowing the remainder of the original loop to be run in parallel.

The first step is to analyze the loop to discover the data dependencies (see **"Writing Parallel Fortran"**). You can use WorkShop Pro MPF with MIPSpro Power Fortran 77 to identify the problem areas. Once you have identified these areas, you can use various techniques to rewrite the code to break the dependence. Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to discover a new parallel method of solving the problem. The rest of this section is devoted to a series of "cookbook" examples on how to deal with commonly occurring situations. These are by no means exhaustive but cover many situations that happen in practice.

Example 1: Loop Carried Value

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

This code segment is the same as in **"Example 6: Rewritable Data Dependence"** . **INDX** has its value carried from iteration to iteration. However, you can compute the appropriate value for **INDX** without making reference to any previous value.

For example, consider the following code:

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

In this loop, the value of **INDX** is computed without using any values computed on any other iteration. **INDX** can correctly be made a **LOCAL** variable, and the loop can now be multiprocessed.

Example 2: 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
```

It is the final statement that causes problems. The indexes **IXX** and **IYY** are computed in a complex way and depend on the values from the **IXOFFSET** and **IYOFFSET** arrays. We do not know if **TOTAL (IXX,IYY)** in one iteration of the loop will always be different from **TOTAL (IXX,IYY)** in every other iteration of the loop.

We can pull the statement out into its own separate loop by expanding **IXX** and **IYY** into arrays to hold intermediate values:

```
C$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.

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.

Example 3: Recurrence

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

This is an example of *recurrence*, which exists when a value computed in one iteration is immediately used by another iteration. There is no good way of running this loop in parallel. If this type of construct appears in a critical loop, try pulling the statement(s) out of the loop as in the previous example. Sometimes another loop encloses the recurrence; in that case, try to parallelize the outer loop.

Example 4: Sum Reduction

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

This operation is known as a *reduction*. Reductions occur when an array of values is combined and reduced into a single value. This example is a sum reduction because the combining operation is addition. Here, the value of **SUM** is carried from one loop iteration to the next, so this loop cannot be multiprocessed. However, because this loop simply sums the elements of **A(I)**, we can rewrite the loop to accumulate multiple, independent subtotals.

Then we can do much of the work in parallel:

```
      NUM_THREADS = MP_NUMTHREADS()
C
C  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
C
      IPIECE_SIZE = (N + (NUM_THREADS -1)) / NUM_THREADS
      DO K = 1, NUM_THREADS
```

```
        PARTIAL_SUM(K) = 0.0
C
C  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
C  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
C  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
C  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
C  HENCE THE "MIN" EXPRESSION.
C
     DO I =K*IPIECE_SIZE -IPIECE_SIZE +1, MIN(K*IPIECE_SIZE,N)
        PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
     END DO
     END DO
C
C  NOW ADD UP THE PARTIAL SUMS
     SUM = 0.0
     DO I = 1, NUM_THREADS
        SUM = SUM + PARTIAL_SUM(I)
     END DO
```

The outer **K** loop can be run in parallel. In this method, the array pieces for the partial sums are contiguous, resulting in good cache utilization and performance.

This is an important and common transformation, and so automatic support is provided by the **REDUCTION** clause:

```
     SUM = 0.0
C$DOACROSS LOCAL (I), REDUCTION (SUM)
     DO 10 I = 1, N
        SUM = SUM + A(I)
10 CONTINUE
```

The previous code has essentially the same meaning as the much longer and more confusing code above. It is an important example to study because the idea of adding an extra dimension to an array to permit parallel computation, and then combining the partial results, is an important technique for trying to break data dependencies. This idea occurs over and over in various contexts and disguises.

Note that reduction transformations such as this do not produce the same results as the original code. Because computer arithmetic has limited precision, when you sum the values together in a different order, as was done here, the round–off errors accumulate slightly differently. It is likely that the final answer will be slightly different from the original loop. Both answers are equally "correct." Most of the time the difference is irrelevant, but sometimes it can be significant, so some caution is in order. If the difference is significant, neither answer is really trustworthy.

This example is a *sum* reduction because the operator is plus (+). The Fortran compiler supports three other types of reduction operations:

1. sum: p = p+a(i)
2. product: p = p*a(i)
3. min: m = min(m,a(i))
4. max: m = max(m,a(i))

For example,

```
C$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
```

One further example of a reduction transformation is noteworthy. Consider the following code:

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

Initially, it might look as if the inner loop should be parallelized with a **REDUCTION** clause. However, look at the outer **I** loop. Although **TOTAL** cannot be made a **LOCAL** variable in the inner loop, it 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. Thus, you do not have to rewrite the loop; you can parallelize this reduction on the outer **I** loop, making **TOTAL** and **J** local variables.

Work Quantum

A certain amount of overhead is associated with multiprocessing a loop. If the work occurring in the loop is small, the loop can actually run slower by multiprocessing than by single processing. To avoid this, make the amount of work inside the multiprocessed region as large as possible.

Example 1: Loop Interchange

```
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
```

Here you have several choices: parallelize the **J** loop or the **I** loop. You cannot parallelize the **K** loop because different iterations of the **K** loop will all try to read and write the same values of **A(I,J)**. Try to parallelize the outermost **DO** loop possible, because it encloses the most work. In this example, that is the **I** loop. For this example, use the technique called *loop interchange*. Although the parallelizable loops are not the outermost ones, you can reorder the loops to make one of them outermost.

Thus, loop interchange would produce

```
C$DOACROSS LOCAL(I, J, K)
        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.

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.

Example 2: Conditional Parallelism

```
      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
```

Here you are using loop unrolling of order four to improve speed. For the first loop, the number of iterations is always fewer than four, so this loop does not do enough work to justify running it in parallel. The second loop is worthwhile to parallelize if **N** is big enough. To overcome the parallel loop overhead, **N** needs to be around 500.

An optimized version would use the **IF** clause on the **DOACROSS** directive:

```
      J = (N/4) * 4
      DO I = J+1, N
         A(I) = A(I) + X*B(I)
      END DO
C$DOACROSS IF (J.GE.500), LOCAL(I)
         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
      ENDIF
```

Cache Effects

It is good policy to write loops that take the effect of the cache into account, with or without parallelism. The technique for the best cache performance is also quite simple: make the loop step through the array in the same way that the array is laid out in memory. For Fortran, this means stepping through the array without any gaps and with the leftmost subscript varying the fastest.

Note that this optimization does not depend on multiprocessing, nor is it required in order for multiprocessing to work correctly. However, multiprocessing can affect how the cache is used, so it is worthwhile to understand.

Performing a Matrix Multiply

Consider the following code segment:

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

This is the same as Example 1 in **"Work Quantum"** (after interchange). To get the best cache performance, the **I** loop should be innermost. At the same time, to get the best multiprocessing performance, the outermost loop should be parallelized. For this example, you can interchange the **I** and **J** loops, and get the best of both optimizations:

```
C$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
```

Understanding Trade−Offs

Sometimes you must choose between the possible optimizations and their costs. Look at the following code segment:

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

This loop can be parallelized on **I** but not on **J**. You could interchange the loops to put **I** on the outside, thus getting a bigger work quantum.

```
C$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 you will step through the **C** array in the wrong direction; the leftmost subscript should be the one that varies the fastest. It is possible to parallelize the **I** loop where it stands:

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

However, **M** needs to be large for the work quantum to show any improvement. In this example, **A(I)** is used to do a sum reduction, and it is possible to use the reduction techniques shown in Example 4 of **"Breaking Data Dependencies"** to rewrite this in a parallel form. (Recall that there is no support for an entire array as a member of the **REDUCTION** clause on a **DOACROSS**.) However, that involves converting array **A** from a one−dimensional array to a two−dimensional array to hold the partial sums; this is analogous to the way we converted the scalar summation variable into an array of partial sums.

If **A** is large, however, the conversion can take more memory than you can spare. It can also take extra time to initialize the expanded array and increase the memory bandwidth requirements.

```
     NUM = MP_NUMTHREADS()
     IPIECE = (N + (NUM-1)) / NUM
C$DOACROSS LOCAL(K,J,I)
     DO K = 1, NUM
        DO J = K*IPIECE - IPIECE + 1, MIN(N, K*IPIECE)
           DO I = 1, M
              PARTIAL_A(I,K) = PARTIAL_A(I,K) + B(J)*C(I,J)
           END DO
        END DO
     END DO
C$DOACROSS LOCAL (I,K)
     DO I = 1, M
```

```
         DO K = 1, NUM
            A(I) = A(I) + PARTIAL_A(I,K)
         END DO
      END DO
```

You must trade off the various possible optimizations to find the combination that is right for the particular job.

Load Balancing

When the Fortran compiler divides a loop into pieces, by default it uses the simple method of separating the iterations into contiguous blocks of equal size for each process. It can happen that some iterations take significantly longer to complete than other iterations. At the end of a parallel region, the program waits for all processes to complete their tasks. If the work is not divided evenly, time is wasted waiting for the slowest process to finish.

Example

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

The previous code segment can be parallelized on the **I** loop. Because the inner loop goes from 1 to **I**, the first block of iterations of the outer loop will end long before the last block of iterations of the outer loop.

In this example, this is easy to see and predictable, so you can change the program:

```
     NUM_THREADS = MP_NUMTHREADS()
C$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, break it into interleaved blocks. Thus, each execution thread receives some small values of **I** and some large values of **I**, giving a better balance of work between the threads. Interleaving usually, but not always, cures a load balancing problem.

You can use the **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
```

The previous code has the same meaning as the rewritten form above.

Note that interleaving can cause poor cache performance because the array is no longer stepped through at stride 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 will have stride 1 to improve cache performance, while the chunks are interleaved to improve load balancing.

The way that iterations are assigned to processes is known as *scheduling*. Interleaving is one possible schedule. Both interleaving and the "simple" scheduling methods are examples of *fixed* schedules; the iterations are assigned to processes by a single decision made when the loop is entered. For more

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 pc sampling 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 **"C$DOACROSS"** 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, the time spent waiting for all processes to complete can be several hundred cycles, even with near perfect balance.

Advanced Features

A number of features are provided so that sophisticated users can override the multiprocessing defaults and customize the parallelism to their particular applications. This section provides a brief explanation of these features.

mp_block and mp_unblock

mp_block puts the slave threads into a blocked state using the system call **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.

mp_setup, mp_create, and mp_destroy

The **mp_setup**, **mp_create**, and **mp_destroy** subroutine 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; they must be re−created when needed. Use of these routines is discouraged because they degrade performance; the **mp_block** and **mp_unblock** routines should be used in almost all cases.

mp_setup takes no arguments. It creates the default number of processes as defined by previous calls to **mp_set_numthreads**, by the environment variable **MP_SET_NUMTHREADS** (described in **"Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP"**), 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.

mp_create 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 die, 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.

mp_blocktime

The Fortran slave threads spin wait until there is work to do. This makes them immediately available when a parallel region is reached. However, this consumes CPU resources. After enough wait time has passed, 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) man page for details). This makes the response time much longer when starting up a parallel region.

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, will still block the threads.

This automatic blocking is transparent to the user's program; blocked threads are automatically unblocked when a parallel region is reached.

mp_numthreads, mp_set_numthreads

Occasionally, you may want to know how many execution threads are available. **mp_numthreads** is a zero−argument integer function that returns the total number of execution threads for this job. The count includes the master thread.

mp_set_numthreads takes a single−integer argument. It 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. If the slave threads have already been created, this call will not change their number. It only has an effect when **mp_setup** is called.

mp_my_threadnum

mp_my_threadnum is a zero−argument function that allows a thread to differentiate itself while in a parallel region. 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: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP

The **MP_SET_NUMTHREADS**, **MP_BLOCKTIME**, and **MP_SETUP** environment variables act as an implicit call to the corresponding routine(s) of the same name 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 like the source 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: MP_SCHEDTYPE, CHUNK

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 **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.

mp_setlock, mp_unsetlock, mp_barrier

mp_setlock, **mp_unsetlock**, and **mp_barrier** are zero−argument subroutines that provide convenient (although limited) access to the locking and barrier functions provided by **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.

Local COMMON Blocks

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 Fortran programs into a parallel form.

The common block must be a named **COMMON** (blank **COMMON** may not be made local), and it must not be initialized 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

```
% f77 −mp a.o −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 local **COMMON** block. It can be a variable, an array, an individual element of an array, or the entire **COMMON** block.

For example,

```
C$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 standard system call **sproc**. It is recommended that programs not attempt to use both **C$DOACROSS** loops and **sproc** calls. It is possible, but there are several restrictions:

- Any 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−defined threads.
- Calls to routines such as **m_get_numprocs** do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the routine **kill** with the

routines. However, the Fortran threads are ordinary subprocesses; using the routine **kill** with the arguments **0** and **sig** (for example, **kill**(0,**sig**)) to signal all members of the process group might 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 **DOACROSS** routine. 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 spools 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 spooled routine. References to **SHARE** variables are resolved by referring back to the original routine.

Because the spooled 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 spooled routine has three arguments: the starting value for the index, the number of times to execute the loop, and a special flag word.

As an example, the following routine that appears on line 1000:

```
      SUBROUTINE EXAMPLE(A, B, C, N)
      REAL A(*), B(*), C(*)
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
```

produces this spooled routine to represent the loop:

```
      SUBROUTINE EXAMPLE.pregion
X ( _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 Spooled Routines

The set of processes that cooperate to execute the parallel Fortran job are members of a process share group created by the system call **sproc**. The process share group is created by special Fortran start–up 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 spooled 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 spooled routine with the appropriate arguments. When a slave completes its execution of the spooled routine, it 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.

PCF Directives

In addition to the simple loop–level parallelism offered by the **C$DOACROSS** directive (described in **"Parallel Loops"**), the compiler supports a more general model of parallelism. This model is based on the work done by the Parallel Computing Forum (PCF), which itself formed the basis for the proposed ANSI–X3H5 standard. The compiler supports this model through compiler directives, rather than extensions to the source language.

The main concept in this model is the *parallel region*, which can be any arbitrary section of code (not just a **DO** loop). Within the parallel region, there are special *work–sharing constructs* that can be used to divide the work among separate processes or threads. 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 then resumes executing the parallel region code. At the end of the parallel region, all the threads synchronize, and the master thread continues execution of the user program.

The PCF directives, summarized in **Table 7–1** , implement the general model of parallelism. They look like Fortran comments, with a **C** in column one. The compiler recognizes these directives when multiprocessing is enabled with either the **-mp** option. (Multiprocessing is also enabled with the **-pfa** option if you have purchased Power Fortran 77.) If multiprocessing is not enabled, the compiler treats these statements as comments. Therefore, you can compile identical source with a single–processing compiler or by Fortran without the multiprocessing option. The PCF directives start with the characters **C$PAR**.

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 [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 [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$PAR [END] PDO	Separate iterations of the enclosed loop are executed by different processes. This directive must be inside a parallel region.
C$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 [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.

Parallel Region

A parallel region encloses any number of PCF constructs (described in **"PCF Constructs"**). It signifies the boundary within which slave threads execute. 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 Parallel Fortran"**).

The preferred form of the directive has no commas between the clauses. The **SHARED** clause is preferred over **SHARE** and **LOCAL** is preferred over **PRIVATE**.

In the following code, all threads enter the parallel region and call the routine **foo**:

```
          subroutine ex1(index)
          integer i

C$PAR PARALLEL LOCAL(i)
          i = mp_my_threadnum()
          call foo(i)
C$PAR END PARALLEL

          end
```

PCF Constructs

The three types of PCF constructs are work–sharing constructs, critical sections, and barriers. All master and slave threads synchronize at the bottom of a work–sharing construct. None of the threads continue past the end of the construct until they all have completed execution within that construct.

The four work–sharing constructs are

- parallel **DO**
- PDO
- parallel sections
- single process

If specified, the PDO, parallel section, and single process constructs must appear inside of a parallel region; the parallel **DO** construct cannot. Specifying a parallel **DO** construct inside of a parallel region

produces a syntax error.

The critical section construct protects a block of code with a lock so that it is executed by only one thread at a time. Threads do not synchronize at the bottom of a critical section.

The barrier construct ensures that each process that is executing waits until all others reach the barrier before proceeding.

Parallel DO

The parallel **DO** construct is the same as the **C$DOACROSS** directive (described in **"C$DOACROSS"**) and conceptually the same as a parallel region containing exactly one PDO construct and no other code. Each thread inside the enclosing parallel region executes separate iterations of the loop within the parallel **DO** construct. The syntax of the parallel **DO** construct is

```
C$PAR PARALLEL DO [clause [[,] clause]...]
```

"C$DOACROSS" describes valid values for *clause* with the exception of the **MP_SCHEDTYPE=***mode* clause. For the **C$PAR PARALLEL DO** directive, **MP_SCHEDTYPE=** is optional; you can just specify *mode*.

PDO

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 Parallel Fortran"**). 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 parenthesis 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 the **DO** loop. The optional **NOWAIT** clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify **NOWAIT**, the processes will wait until all have reached the directive before proceeding.

As an example of the PDO construct, consider the following code:

```
          subroutine ex2(a,n)
          real a(n)

C$PAR PARALLEL local(i) shared(a)
C$PAR PDO
          do i = 1, n
            a(i) = a(i) + 1.0
          enddo
C$PAR END PARALLEL

          end
```

This sample code is the same as a **C$ DOACROSS** loop. 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 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$PAR PSECTION[S] [clause [[,]clause ]...
     code
[C$PAR SECTION
        code] ...
C$PAR END PSECTION[S] [NOWAIT]
```

where 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 **"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 contructs (described in **"Critical Section"**) but cannot contain any of the following types of constructs:

- PDO
- parallel **DO** or **C$ DOACROSS**
- single process

Each code block is executed in parallel (depending on the number of processes available). The code blocks are assigned to threads one at a time, in the order specified. Each code block is executed by only one thread.

For example, consider the following code:

```
          subroutine ex3(a,n1,b,n2,c,n3)
          real a(n1), b(n2), c(n3)

C$PAR PARALLEL local(i) shared(a,b,c)

C$PAR PSECTIONS
C$PAR SECTION
          do i = 1, n1
            a(i) = 0.0
          enddo
C$PAR SECTION
          do i = 1, n2
            b(i) = 0.5
          enddo
C$PAR SECTION
          call normalize(c,n3)
          do i = 1, n3
            c(i) = c(i) + 1.0
          enddo
C$PAR END PSECTION

C$PAR END PARALLEL

          end
```

The first thread to enter the parallel sections construct executes the first block, the second thread

executes the second block, and so on. This example has only three sections, so if more than three threads are in the parallel region, the fourth and higher threads wait at the **C$PAR END PSECTION** directive until all threads are finished. If the parallel region is being executed by only two threads, whichever thread finishes its block first continues and executes the remaining block.

This example uses **DO** loops, but a parallel section can be any arbitrary block of code. 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$PAR SINGLE PROCESS [clause [[,] clause]...]
      code
C$PAR END SINGLE PROCESS [NOWAIT]
```

where 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 **"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. For example, consider the following code:

```
          real function ex4(a,n, big_max, bmax_x, bmax_y)
          real a(n,n), big_max
          integer bmax_x, bmax_y

C$ volatile big_max, bmax_x, bmax_y
C$ volatile cur_max, index_x, index_y

          index_x = 0
          index_y = 0
          cur_max = 0.0

C$PAR PARALLEL local(i,j)
C$PAR& shared(a,n,index_x,index_y,cur_max,
C$PAR& big_max,bmax_x,bmax_y)


C$PAR PDO
          do j = 1, n
            do i = 1, n
              if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
                if (a(i,j) .gt. cur_max) then
                  index_x = i
                  index_y = j
```

```
                cur_max = a(i,j)
              endif
C$PAR END CRITICAL SECTION
            endif
          enddo
        enddo

C$PAR SINGLE PROCESS
        if (cur_max .gt. big_max) then
          big_max = (big_max + cur_max) / 2.0
          bmax_x = index_x
          bmax_y = index_y
        endif
C$PAR END SINGLE PROCESS

C$PAR PDO
        do j = 1, n
          do i = 1, n
            a(i,j) = a(i,j)/big_max
          enddo
        enddo

C$PAR END PARALLEL

        ex4 = cur_max

        end
```

The first thread to reach the single process section executes the code in that block. All other threads wait at the end of the block until the code has been executed.

This example contains a number of interesting points to be examined. First, note the use of the **VOLATILE** declaration. Any data item that might be written by one thread and then read by a different thread must be marked as **VOLATILE**. Making a variable **VOLATILE** can reduce opportunities for optimization, so the declarations are prefixed by **C$** to prevent the single–processor version of the code from being penalized. Refer to the *MIPSpro Fortran 77 Language Reference Manual* for more information about the **VOLATILE** statement.

Second, note the use of the odd looking repetition of the **IF** test in the first parallel loop:

```
            if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
              if (a(i,j) .gt. cur_max) then
```

This practice is usually called *test&test&set*. It is a multi–processing optimization. Note that the following straight forward code segment is incorrect:

```
        do i = 1, n
          if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
                index_x = i
                index_y = j
                cur_max = a(i,j)
C$PAR END CRITICAL SECTION
          endif
        enddo
```

Because many threads execute the loop in parallel, there is no guarantee that once inside the critical section, **cur_max** still has the same value it did in the **IF** test outside the critical section (some other thread may have updated it). In particular, **cur_max** may now have a value that is larger than **a(i,j)**. Therefore, the critical section must be locked before testing the value of **cur_max**. Changing the

previous code into the equally straightforward

```
            do i = 1, n
C$PAR CRITICAL SECTION
                if (a(i,j) .gt. cur_max) then
                  index_x = i
                  index_y = j
                  cur_max = a(i,j)
                endif
C$PAR END CRITICAL SECTION
         enddo
```

works correctly, but suffers from a serious performance penalty: the critical section lock must be acquired and released (an expensive operation) for each element of the array. Because the values are rarely updated, this process involves a lot of wasted effort. It is almost certainly slower than just executing the loop serially.

Combining the two methods, as in the original example, produces code that is both fast and correct. If the **IF** test outside of the critical section fails, you can be certain that the values will not be updated, and can proceed. You can expect that the outside **IF** test will account for the majority of cases. If the outer **IF** test passes, then the values *might* be updated, but you cannot always be certain. To ensure correctness, you must perform the test again after acquiring the critical section lock.

You can prefix one of the two identical **IF** tests with **C$** to reduce overhead in the non−multiprocessed case.

Lastly, note the difference between the single process and critical section constructs. If several processes arrive at a critical section construct, they execute the code one at a time. However, they will *all* execute the code. If several processes arrive at a single process construct, only one process executes the code. The other processes bypass the code and wait at the end of the construct for the chosen process to finish.

Critical Section

The critical section construct restricts execution of a block of code so that only one process can execute it at a time. Another process attempting to gain entry to the critical section must wait until the previous process has exited.

The critical section construct can appear anywhere in a program, including inside and outside a parallel region and 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. If you do not specify *lock_variable*, the compiler automatically supplies one.

Multiple critical section constructs inside the same parallel region are considered to be independent of each other unless they use the same explicit *lock_variable*.

Consider the following code:

```
          integer function num_exceptions(a,n,biggest_allowed)
          double precision a(n,n,n), biggest_allowed

          integer count
          integer lock_var

          volatile count

          count = 0
          lock_var = 0
```

```
C$PAR PARALLEL  local(i,j,k) shared(count,lock_var)

C$PAR PDO
          do 10 k = 1,n
            do 10 j = 1,n
              do 10 i = 1,n
                if (a(i,j,k) .gt. biggest_allowed) then

C$PAR CRITICAL SECTION (lock_var)
                  count = count + 1
C$PAR END CRITICAL SECTION (lock_var)

                else
                  call transform(a(i,j,k))
                  if (a(i,j,k) .gt. biggest_allowed) then

C$PAR CRITICAL SECTION (lock_var)
                    count = count + 1
C$PAR END CRITICAL SECTION (lock_var)

                  endif

                endif
   10     continue

C$PAR END PARALLEL

          num_exceptions = count

          return
          end
```

This example demonstrates the use of the lock variable (**lock_var**). A **C$PAR CRITICAL SECTION** directive ensures that no more than one process executes the enclosed block of code at a time. However, if there are multiple critical sections, different processes can be in different critical sections at the same time. This example does not allow different processes to be in different critical sections at the same time because both critical sections control access to the same variable (**count**). Specifying the same lock variable for both critical sections ensures that no more than one process is executing either of the critical sections that use that lock variable. Note that the **lock_var** must be **SHARED** (so that all processes use the same lock), and that **count** must be **volatile** (because other processes might change its value).

Barrier Constructs

A barrier construct ensures that each process waits until all processes reach the barrier before proceeding. The syntax of the barrier construct is

```
C$PAR BARRIER
```

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,

```
C$PAR PARALLEL local(i,j)
C$PAR& shared(a,n,index_x,index_y,cur_max,
C$PAR& big_max,bmax_x,bmax_y)
```

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:

```
          .
        .
        .
C$PAR PARALLEL
          if (mp_my_threadnum() .gt. 5) then
C$PAR SINGLE PROCESS
              many_processes = .true.
C$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. Because some of the threads never took the appropriate branch, they will never encounter the construct. However, the following kind of simple looping is supported:

```
       code
C$PAR PARALLEL local(i,j) shared(a)
          do i= 1,n
C$PAR PDO
            do j = 2,n
       code
```

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 and 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, "Fortran Enhancements 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 debug your program in a single–processor environment by calling the Fortran compiler with the *f77* command. You can also debug your program using the WorkShop Pro MPF debugger, which is sold as a separate product. After your program has executed successfully on a single processor, you can compile it for multiprocessing. Check the *f77*(1) manual page for multiprocessing options.

To turn on multiprocessing, add **-mp** to the *f77* command line. This option causes the Fortran compiler to generate multiprocessing code for the particular files being compiled. When linking, you can specify both object files produced with the **-mp** option and object files produced without it. If any or all of the files are compiled with **-mp**, the executable must be linked with **-mp** so that the correct libraries are used.

Using the -static Option

A few words of caution about the **-static** compiler option: The multiprocessing implementation demands some use of the stack to allow multiple threads of execution to execute the same code simultaneously. Therefore, the parallel **DO** loops themselves are compiled with the **-automatic** option, even if the routine enclosing them is compiled with **-static**.

This means that **SHARE** variables in a parallel loop behave correctly according to the **-static** semantics but that **LOCAL** variables in a parallel loop do not (see **"Debugging Parallel Fortran"** for a description of **SHARE** and **LOCAL** variables).

Finally, if the parallel loop calls an external routine, that external routine cannot be compiled with **-static**. You can mix static and multiprocessed object files in the same executable; the restriction is that a static routine cannot be called from within a parallel loop.

Examples of Compiling

This section steps you through a few examples of compiling code using **-mp**. The following command line

```
% f77 -mp foo.f
```

compiles and links the Fortran program **foo.f** into a multiprocessor executable.

In this example

```
% f77 -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:

```
% f77 -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, running and synchronizing them, and task terminating 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 task 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 (the value returned by the system call **sysmp(MP_NAPROCS)**; see the *sysmp*(2) man page). You can override the default by setting the shell environment variable **MP_SET_NUMTHREADS**. If it is set, Fortran tasks use the specified number of execution threads regardless of the number of processors physically present on the machine. **MP_SET_NUMTHREADS**

can be from 1 to 64.

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*(1) and pc−sample profiling can be used. On jobs that use multiple threads, both these methods will create multiple profile data files (one for each thread). You can use the standard profile analyzer *prof*(1) to examine this output. (Refer to the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for details about using *prof*.)

The profile of a Fortran parallel job is different from a standard profile. As mentioned in **"Analyzing Data Dependencies for Multiprocessing"** , 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 standard 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.
- 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: Erroneous C$DOACROSS

In this example, the bug is that 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.

```
c$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, you could rewrite this as:

```
c$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 is giving 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**(1) to look for duplicates. If duplicates are found, then there is a data dependency.

Chapter 9

Fine–Tuning Program Execution

This chapter contains the following sections:

- **"Overview"** explains the concept of directives and assertions.
- **"Fine–Tuning Scalar Optimizations"** describes how you can use directives to fine–tune scalar optimizations.
- **"Fine–Tuning Inlining and IPA"** explains how you can use directives to fine tune inlining and IPA.
- **"Using Equivalenced Variables"** explains how you can inform the compiler that your code uses or does not use equivalenced variables.
- **"Using Assertions"** explains how you can enable or disable compiler recognition of assertions.
- **"Using Aliasing"** explains the assertions that enable or disable types of aliasing.
- **"Fine–Tuning Global Assumptions"** describes how to use assertions to fine–tune global assumptions.
- **"Ignoring Data Dependencies"** explains how to instruct the compiler to ignore data dependencies.

Overview

After running a Fortran source program through the compiler's scalar optimizations once, you can use

directives and assertions to fine−tune program execution by forcing the compiler to execute portions of code in various ways.

By default, the compiler recognizes all Silicon Graphics directives and assertions. You can use the **-WK,-directives** command line option to selectively enable/disable certain directives and assertions. Refer to **"Recognizing Directives" in Chapter 5** for information about the **-directives** option.

Directives

Directives enable, disable, or modify a feature of the compiler. Essentially, directives are command line options specified within the input file instead of on the command line. Unlike command line 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 allow you to enable, disable, or modify a feature of the compiler in addition to, or instead of, command line 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 command line 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 command line option or a global directive.)

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

Table 9−1 lists 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. Refer to **"Assertions"** for details.

Table 9−1 Directives Summary

Standard	Cray
C*$*ARCLIMIT(*n*)	CDIR$ NO RECURRENCE
C*$*[NO]ASSERTIONS	
C*$* EACH_INVARIANT_IF_GROWTH(*n*)	
C*$* [NO]INLINE	
C*$* [NO]IPA	
C*$* MAX_INVARIANT_IF_GROWTH(*n*)	
C*$* OPTIMIZE(*n*)	
C*$* ROUNDOFF(*n*)	
C*$* SCALAR OPTIMIZE(*n*)	
C*$* UNROLL(*integer*[,*weight*])	

Assertions

Assertions provide the compiler with additional information about the source program. Sometimes assertions can improve optimization results. Use them only when speed is essential.

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** command line option or the **C*$* NO ASSERTIONS** directive to tell the compiler to ignore all assertions.

Table 9−2 lists the supported assertions and their duration.

Table 9−2 Assertions and Their Duration

Assertion	Duration
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

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 the end of the program unit (or file) or until you reset them. 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 Silicon Graphics 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.

To understand the process the compiler uses in interpreting assertions, you must understand the concept of *assumed dependences*. The following loop contains two types of dependences:

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

X is an array, **n** and **m** are scalars, and nothing is known about the relationship between **n** and **m**. Between **X(i)** and **X(i–1)** there is a forward dependence, and the distance is one. Between **X(i)** and **X(m)**, the compiler tries to find a relation, but cannot, because it does not know the value of **m** in relation to **n**. The second dependence is called an assumed dependence, because it is assumed but cannot be proven to exist.

Fine–Tuning Scalar Optimizations

The compiler supports several directives that allow you to fine–tune the scalar optimizations described in **"Controlling Scalar Optimizations" in Chapter 5** .

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** command line option (refer to **"Controlling Internal Table Size" in Chapter 5** for details).

Setting Invariant IF Floating Limits

The **C*$* EACH_INVARIANT_IF_GROWTH** and the **C*$* MAX_INVARIANT_IF_GROWTH** directives control limits on invariant **IF** floating. 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" in Chapter 5** 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** command line option (see **"Setting Invariant IF Floating Limits" in Chapter 5**).

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** command line option to control this limit globally (see **"Setting Invariant IF Floating Limits" in Chapter 5**).

These directives are in effect until the end of the routine or until reset by a succeeding directive of the same type.

Example

Consider the following code:

```
C*$*EACH_INVARIANT_IF_GROWTH(integer)
C*$*MAX_INVARIANT_IF_GROWTH(integer)
        DO I = ...
C*$*EACH_INVARIANT_IF_GROWTH(integer)
C*$*MAX_INVARIANT_IF_GROWTH(integer)
            DO J = ...
C*$*EACH_INVARIANT_IF_GROWTH(integer)
C*$*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.

Optimization Level

The **C*$* OPTIMIZE**(*integer*) directive sets the optimization level in the same way as the **-optimize** command line 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 wrap−around variables.
4	Generates two versions of a loop, if necessary, to break a data dependence arc.
5	Enables array expansion and loop fusion.

Refer to **"Controlling Scalar Optimizations" in Chapter 5** for examples.

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** command line option. Valid values for *integer* are

0	Suppresses any transformations that change round−off error.
1	Performs expression simplification, which might generate various overflow or

underflow errors, for expressions with operands between binary and unary operators, for 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" in Chapter 5** for details about memory management transformations).

Controlling Scalar Optimizations

The **C*$* SCALAR OPTIMIZE**(*integer*) directive controls the amount of standard scalar optimizations that the compiler performs. Unlike the **-WK,-scalaropt** command line option, the **C*$* 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 Disables all scalar optimizations.

1 Enables simple, loop–based, scalar optimization —changing **IF** loops to **DO** loops, simple code floating out of loops, and forward substitution of variables.

2 Enables the full range of loop–based scalar optimizations— induction variable recognition, loop rerolling, loop unrolling, loop fusion, and array expansion.

3 Enables memory management transformations if **-roundoff=3**. Refer to **"Performing Memory Management Transformations" in Chapter 5** for details.

Enabling Loop Unrolling

The **C*$* UNROLL**(*integer*[,*weight*]) directive controls how the compiler unrolls scalar loops. Loops that cannot be optimized for concurrent execution usually execute more efficiently when they are unrolled. This directive is recognized only when you specify **-WK,-scalaropt=2**.

The compiler unrolls the loop proceeding the **C*$*UNROLL** directive until either the number of operations in the loop equals the *weight* parameter or the number of iterations reaches the *integer* parameter, whichever occurs first. The **-unroll** and **-unroll2** command line options act like a global **C*$* UNROLL** directive. See **"Enabling Loop Unrolling" in Chapter 5** for detailed examples.

The **C*$* UNROLL** directive is in effect only for the loop immediately following it, unlike other directives.

Fine–Tuning Inlining and IPA

Chapter 6, "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 command line option, respectively, on the command line. The **-inline_manual** and **-ipa_manual** command line options enable these directives without activating the automatic inlining/algorithms.

The **C*$* [NO] INLINE** directive behaves like the **-inline** command line 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]
```

where

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** command line option and so allows you to selectively disable inlining of the named routines at specific points.

Example

In the following code fragment, the **C*$*INLINE** directive inlines the first call to **beta** but not the second.

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

Using the specifier **ROUTINE** rather than **HERE** inlines both calls. This routine must be compiled with the **-inline_man** command line 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]
```

Using Equivalenced Variables

The **C*$* ASSERT [NO] EQUIVALENCE HAZARD** assertion tells the compiler that your code does not use equivalenced variables to refer to the same memory location inside one loop nest. Normally, **EQUIVALENCE** statements allow your code to use different variable names to refer to the same storage location. The **-WK,–assume=e** command line option acts like the global **C*$* ASSERT EQUIVALENCE HAZARD** assertion (see **"Controlling Global Assumptions"** in Chapter 4). The **C*$* ASSERT EQUIVALENCE HAZARD** assertion is active until you reset it or until the end of the program.

Using Assertions

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** command line option without the assertions parameter (that is, **a**), the compiler will ignore assertions regardless of whether the file contains the **C*$* ASSERTIONS** directive. Refer to **"Recognizing Directives" in Chapter 5** for details on the **-directives** command line option.

Using Aliasing

The compiler recognizes two assertions for use with aliasing.

C*$* ASSERT [NO] ARGUMENT ALIASING

The **C*$* ASSERT [NO] ARGUMENT ALIASING** assertion allows the compiler to make assumptions about subprogram arguments in a program. According to the Fortran 77 standard, you can alias a variable only if you do not modify (that is, write to) the aliased variable.

The following subroutine violates the standard, because variable **A** is aliased in the subroutine (through **C** and **D**) and variable **X** is aliased (through **X** and **E**):

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

```
COMMON X,Y
X =  ...
C =  ...
...
```

The command line option **-assume=a** acts like a global **C*$* ASSERT ARGUMENT ALIASING** assertion (see **"Controlling Global Assumptions" in Chapter 5**). A **C*$* ARGUMENT ALIASING** assertion is active until it is reset or until the next routine begins.

C*$* ASSERT RELATION

The **C*$* ASSERT RELATION**(*name.xx.name*) assertion indicates 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 variable names (*name* and *xx*). When specified globally, this assertion will only be used when the variable 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.

As an example of the use of the **C*$* ASSERT RELATION** assertion, consider the following code:

```
            DO 100 I = 1, N
               A (I) = A (I+M) + B (I)
100       CONTINUE
```

If you know that **M** is greater than **N**, use the following assertion to give this information to the compiler:

```
C*$* ASSERT RELATION (M .GT. N)
            DO 100 I = 1, N
               A (I) = A (I +M) + B (I)
100       CONTINUE
```

Knowing that **M** is greater than **N**, the compiler can generate parallel code for this loop. If **M** is less than **N** at run time, the answers produced by the code run in parallel could differ from the answers produced by the original code run serially.

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.

Fine–Tuning Global Assumptions

You can use the assertions described in this section to fine–tune the global assumptions discussed in **"Controlling Global Assumptions" in Chapter 5** .

C*$* ASSERT [NO]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** command line option acts like a global **C*$* ASSERT BOUNDS VIOLATIONS** assertion.

In the following example, the compiler assumes the first loop nest is standard–conforming, and therefore can optimize both loops. 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 right array index.

Note: The compiler always assumes that array references will be within the array itself, so the rightness index will be concurrentizable.

```
        DO 100 I = 1,M
          DO 100 J = 1,N
            A(I,J) = A(I,J) + B (I,J)
    100 CONTINUE
C
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
```

becomes

```
C$DOACROSS SHARE(N,M,A,B),LOCAL(J,I)
          DO 2 J=1,N
        DO 2 I=1,M
            A(I,J) = A(I,J) + B (I,J)
C
C*$*ASSERT BOUNDS VIOLATIONS
        DO 4 I=1,M
C$DOACROSS SHARE(N,I,A,B),LOCAL(J)
          DO 3 J=1,N
            A(I,J) = A(I,J) + B (I,J)
      3     CONTINUE
      4     CONTINUE
```

C*$* ASSERT NO EQUIVALENCE HAZARD

The **C*$* ASSERT NO EQUIVALENCE HAZARD** assertion tells the compiler that equivalenced variables will not be used to refer to the same memory location inside one **DO** loop nest. Normally, **EQUIVALENCE** statements allow different variable names to refer to the same storage location. The **-WK,-assume=e** command line option acts like a global **C*$* ASSERT NO EQUIVALENCE HAZARD** assertion. This assertion is active until reset or until the end of the program.

In the following example, if arrays **E** and **F** are equivalenced, but you know that the overlapping sections will not be referenced in this loop, then using **C*$* ASSERT NO EQUIVALENCE HAZARD** allows the compiler to concurrentize the loop:

```
          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
```

becomes

```
          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
```

C*$* ASSERT [NO] TEMPORARIES FOR 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 the following code:

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

into

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

But if the actual parameter for **I** were a constant such as the following,

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

it would appear that the value of the constant 1 was being reassigned. 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** command line 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.

Ignoring Data Dependencies

The **C*$* ASSERT NO RECURRENCE**(*variable*) assertion 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:

```
C*$* ASSERT NO RECURRENCE (X)
          DO 10 i=1,m,5
    10    X(k) = X(k) + X(i)
```

Not only does the compiler ignore the assumed dependence, it also ignores the real dependence caused by **X(k)** appearing on both sides of the assignment.

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.

Appendix A
Run–Time Error Messages

Table A–1 lists possible Fortran run–time I/O errors. Other errors given by the operating system may also occur (refer to the *intro*(2) manual page for details).

Each error is listed on the screen alone or with one of the following phrases appended to it:

apparent state: unit *num* named *user filename*

last format: *string*

lately (reading, writing) (sequential, direct, indexed)

formatted, unformatted (external, internal) IO

When the Fortran run–time system detects an error, the following actions take place:

- A message describing the error is written to the standard error unit (Unit 0).
- A core file, which can be used with *dbx* (the debugger) to inspect the state of the program at termination, is produced if the **f77_dump_flag** environment variable is defined and set to **y**.

When a run–time error occurs, the program terminates with one of the error messages shown in **Table A–1** . All of the errors in the table are output in the format *user filename* : *message*.

Table A–1 Run–Time Error Messages

Number	Message/Cause
100	`error in format` Illegal characters are encountered in FORMAT 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.
106	`can't backspace file` Cannot perform BACKSPACE/REWIND on file.
107	`null file name` Filename specification in OPEN statement is null.
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.
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.

132 `indexed file not open`
Cannot perform indexed I/O on an unopened file.

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.

134 `bad key description`
The key description is invalid.

135 `too many open indexed files`
Cannot have more than 32 open indexed files.

136 `corrupted isam file`
The indexed file format is not recognizable. This error is usually caused by a corrupted file.

137 `isam file not opened for exclusive access`
Cannot obtain lock on the indexed file.

138 `record locked`
The record has already been locked by another process.

138 `key already exists`
The key specification in the OPEN statement has already been specified.

140 `cannot delete primary key`
DELETE cannot be executed on a primary key.

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.

142 `cannot find request record`
The requested key for indexed READ does not exist.

143 `current record not defined`
Cannot execute REWRITE, UNLOCK, or DELETE before doing a READ to define the current record.

144 `isam file is exclusively locked`
The indexed file has been exclusively locked by another process.

145 `filename too long`
The indexed filename exceeds 128 characters.

148 `key structure does not match file structure`
Mismatch between the key specifications in the OPEN statement and the indexed file.

149 `direct access on an indexed file not allowed`
Cannot have direct−access I/O on an indexed file.

150 `keyed access on a sequential file not allowed`
Cannot specify keyed access together with sequential organization.

151 `keyed access on a relative file not allowed`
Cannot specify keyed access together with relative organization.

152 `append access on an indexed file not allowed`
Cannot specifiy append access together with indexed organization.

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.

155 `character key field value length too long`
The length of the character key value exceeds the length specification for that key.

156 `fixed record on sequential file not allowed`
RECORDTYPE='fixed' cannot be used with a sequential file.

157 `variable records allowed only on unformatted sequential file`
RECORDTYPE='variable' can only be used with an unformatted sequential file.

158 `stream records allowed only on formatted sequential file`

RECORDTYPE='stream_lf' can only be used with a formatted sequential file.

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 create or 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.

162 `carriage control not allowed for unformatted units`
CARRIAGECONTROL specifier can be used only on a formatted file.

163 `indexed files only`
Indexed I/O can be done only on logical units that have been opened for indexed (keyed) access.

164 `cannot use on indexed file`
Illegal I/O operation on an indexed (keyed) file.

165 `cannot use on indexed or append file`
Illegal I/O operation on an indexed (keyed) or append file.

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.

174 `cannot write to stdin`
Attempt to write to stdin.

176 `illegal specifier`
The I/O control list contains an invalid value for one of the I/O specifiers. For example, ACCESS='INDEXED'.

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 argument routines called in subroutines that have not been declared in a $VARARGS directive.

185 `internal error`
Internal run–time library error.