

MIPSpro™ Fortran 77 Programmer's Guide

Document Number 007-2361-004

CONTRIBUTORS

Written by Chris Hogue

Illustrated by Dany Galgani and Gloria Ackley

Production by Linda Rae Sande

Engineering contributions by Bill Johnson, Bron Nelson, Calvin Vu, Marty Itzkowitz,
Dick Lee, and Rohit Chandra

St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower
image courtesy of Xavier Berenguer, Animatica.

© 1994-1996 Silicon Graphics, Inc.— All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole
or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the technical data contained in this document by
the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the
Rights in Technical Data and Computer Software clause at DFARS 52.227-7013
and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR
Supplement. Unpublished rights are reserved under the Copyright Laws of the
United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline
Blvd., Mountain View, CA 94039-7311.

Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks, and
CASEVision, IRIS 4D, IRIS Power Series, IRIX, Origin2000, and POWER
CHALLENGE are trademarks of Silicon Graphics, Inc. MIPS, R4000, R4400, and
R8000 are registered trademarks and MIPSpro and R10000 are trademarks of MIPS
Technologies, Inc. UNIX is a registered trademark in the United States and other
countries, licensed exclusively through X/Open Company, Ltd. VMS and VAX are
trademarks of Digital Equipment Corporation.

Portions of this product and document are derived from material copyrighted by
Kuck and Associates, Inc.

MIPSpro™ Fortran 77 Programmer's Guide
Document Number 007-2361-004

Contents

List of Examples ix

List of Figures xi

List of Tables xiii

Introduction xv

Organization xv

Additional Reading xvi

Typographical Conventions xvii

1. **Compiling, Linking, and Running Programs** 1
 - Compiling and Linking 1
 - Drivers 1
 - Compilation 2
 - Compiling Multilanguage Programs 3
 - Linking Objects 5
 - Specifying Link Libraries 6
 - Driver Options 7
 - Compiling Simple Programs 7
 - Specifying Source File Format 8
 - Specifying Compiler Input and Output Files 8
 - Using a Defaults Specification File 9
 - Specifying Target Machine Features 9
 - Specifying Memory Allocation and Alignment 10
 - Specifying Debugging and Profiling 10
 - Specifying Optimization Levels 11
 - Controlling Compiler Execution 12
 - Object File Tools 13
 - Archiver 13

Run-Time Considerations	14
Invoking a Program	14
Maximum Memory Allocations	14
File Formats	17
Preconnected Files	18
File Positions	18
Unknown File Status	19
Quad-Precision Operations	19
Run-Time Error Handling	19
Floating Point Exceptions	20
2. Storage Mapping	21
Alignment, Size, and Value Ranges	21
Access of Misaligned Data	24
Accessing Small Amounts of Misaligned Data	25
Accessing Misaligned Data Without Modifying Source	25
3. Fortran Program Interfaces	27
How Fortran Treats Subprogram Names	27
Working with Mixed-Case Names	28
Preventing a Suffix Underscore with \$	28
Naming Fortran Subprograms from C	29
Naming C Functions from Fortran	29
Testing Name Spelling Using <i>nm</i>	29
Correspondence of Fortran and C Data Types	30
Corresponding Scalar Types	30
Corresponding Character Types	31
Corresponding Array Elements	31
How Fortran Passes Subprogram Parameters	33
Normal Treatment of Parameters	33
Calling Fortran from C	34
Calling Fortran Subroutines from C	34
Calling Fortran Functions from C	37

Calling C from Fortran	39
Normal Calls to C Functions	39
Using Fortran COMMON in C Code	41
Using Fortran Arrays in C Code	41
Calls to C Using LOC%, REF% and VAL%	42
Making C Wrappers with <i>mkf2c</i>	44
Using <i>mkf2c</i> and <i>extcentry</i>	48
Makefile Considerations	49
4. System Functions and Subroutines	51
Library Functions	51
Extended Intrinsic Subroutines	59
DATE	60
IDATE	60
ERRSNS	60
EXIT	61
TIME	61
MVBITS	62
Extended Intrinsic Functions	62
SECNDS	63
RAN	63
5. Fortran Enhancements for Multiprocessors	65
Overview	65
Directives	66
Parallel Loops	66
Writing Parallel Fortran	68
C\$DOACROSS	68
C\$&	74
C\$	75
C\$MP_SCHEDTYPE and C\$CHUNK	75
Nesting C\$DOACROSS	76
Analyzing Data Dependencies for Multiprocessing	76
Breaking Data Dependencies	82

- Work Quantum 87
- Cache Effects 88
 - Performing a Matrix Multiply 89
 - Understanding Trade-Offs 90
 - Load Balancing 91
 - Reorganizing Common Blocks To Improve Cache Behavior 93
- Advanced Features 93
 - mp_block and mp_unblock 93
 - mp_setup, mp_create, and mp_destroy 94
 - mp_blocktime 94
 - mp_numthreads, mp_set_numthreads 95
 - mp_suggested_numthreads 95
 - mp_my_threadnum 95
 - Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP 96
 - Environment Variables: MP_SUGNUMTHD, MP_SUGNUMTHD_MIN, MP_SUGNUMTHD_MAX, MP_SUGNUMTHD_VERBOSE 97
 - Environment Variables: MP_SCHEDTYPE, CHUNK 97
 - mp_setlock, mp_unsetlock, mp_barrier 98
 - Local COMMON Blocks 98
 - Compatibility With sproc 99
- DOACROSS Implementation 100
 - Loop Transformation 100
 - Executing Spooled Routines 101
- PCF Directives 102
 - Parallel Region 104
 - PCF Constructs 104
 - Restrictions 114
 - A Few Words About Efficiency 115

Synchronization Intrinsic	116
Atomic fetch-and-op Operations	117
Atomic op-and-fetch Operations	117
Atomic BOOL Operation	118
Atomic synchronize Operation	119
Atomic lock and unlock Operations	119
Example of Implementing a Pure Spin-Wait Lock	120
6. Parallel Programming on Origin2000	121
Performance Tuning of Parallel Programs on Origin2000	122
Improving Program Performance	122
Choosing Between Multiple Options	125
New Directives for Performance Tuning on Origin2000	126
Data Distribution Directives	127
Nested Doacross Directive	129
Affinity Scheduling	129
Data Affinity	130
Thread Affinity	132
Specifying Processor Topology With the ONTO Clause	132
Types of Data Distribution	133
Regular Data Distribution	133
Data Distribution With Reshaping	134
Query Intrinsic for Distributed Arrays	136
Implementation of Reshaped Arrays	137
Regular vs. Reshaped Data Distribution	139
Explicit Placement of Data	140
Optional Environment Variables and Compile-Time Options	141
Examples	142
Distributing Columns of a Matrix	143
Using Data Distribution and Data Affinity Scheduling	144
Parameter Passing	145
Redistributed Arrays	146
Irregular Distributions and Thread Affinity	147

- 7. **Compiling and Debugging Parallel Fortran** 149
 - Compiling and Running 149
 - Using the **-static** Option 150
 - Examples of Compiling 150
 - Profiling a Parallel Fortran Program 151
 - Debugging Parallel Fortran 152
 - General Debugging Hints 152
- A. **Run-Time Error Messages** 155
- Index** 163

List of Examples

Example 3-1	Example Subroutine Call	33
Example 3-2	Example Function Call	33
Example 3-3	Example Fortran Subroutine with COMPLEX Parameters	34
Example 3-4	C Declaration and Call with COMPLEX Parameters	35
Example 3-5	Example Fortran Subroutine with String Parameters	35
Example 3-6	C Program that Passes String Parameters	35
Example 3-7	C Program that Passes Different String Lengths	36
Example 3-8	Fortran Function Returning COMPLEX*16	37
Example 3-9	C Program that Receives COMPLEX Return Value	37
Example 3-10	Fortran Function Returning CHARACTER*16	38
Example 3-11	C Program that Receives CHARACTER*16 Return	38
Example 3-12	C Function Written to be Called from Fortran	39
Example 3-13	Common Block Usage in Fortran and C	41
Example 3-14	Fortran Program Sharing an Array in Common with C	42
Example 3-15	C Subroutine to Modify a Common Array	42
Example 3-16	Fortran Function Calls Using %VAL	43
Example 3-17	Fortran Call to <i>gmatch()</i> Using %REF	44
Example 3-18	C Function Using <i>varargs</i>	47
Example 3-19	C Code to Retrieve Hidden Parameters	47
Example 3-20	Source File for Use with <i>extcentry</i>	48
Example 5-1	Simple DOACROSS	72
Example 5-2	DOACROSS LOCAL	73
Example 5-3	DOACROSS LAST LOCAL	73
Example 5-4	Simple Independence	77
Example 5-5	Data Dependence	78
Example 5-6	Stride Not 1	78
Example 5-7	Local Variable	78

Example 5-8	Function Call	79
Example 5-9	Rewritable Data Dependence	80
Example 5-10	Exit Branch	80
Example 5-11	Complicated Independence	80
Example 5-12	Inconsequential Data Dependence	81
Example 5-13	Local Array	81
Example 5-14	Loop Carried Value	82
Example 5-15	Indirect Indexing	83
Example 5-16	Recurrence	84
Example 5-17	Sum Reduction	84
Example 5-18	Loop Interchange	87
Example 5-19	Conditional Parallelism	88
Example 5-20	Load Balancing	91

List of Figures

Figure 1-1	Compilation Process	2
Figure 1-2	Compiling Multilanguage Programs	4
Figure 1-3	Linking	5
Figure 3-1	Correspondence Between Fortran and C Array Subscripts	32
Figure 6-1	Origin2000 Memory Hierarchy	122
Figure 6-2	Cache Behavior and Solutions	124
Figure 6-3	Implementation of BLOCK Distribution	137
Figure 6-4	Implementation of CYCLIC(1) Distribution	138
Figure 6-5	Implementation of BLOCK-CYCLIC Distribution	139

List of Tables

Table 1-1	Link Libraries	6
Table 1-2	Compile Options for Source File Format	8
Table 1-3	Compile Options that Select Files	8
Table 1-4	Compile Options for Target Machine Features	9
Table 1-5	Compile Options for Memory Allocation and Alignment	10
Table 1-6	Compile Options for Debugging and Profiling	10
Table 1-7	Compile Options for Optimization Control	11
Table 1-8	Power Fortran Defaults for Optimization Levels	12
Table 1-9	Compile Options for Compiler Phase Control	12
Table 1-10	Preconnected Files	18
Table 2-1	Size, Alignment, and Value Ranges of Data Types	21
Table 2-2	Valid Ranges for REAL*4 and REAL*8 Data Types	22
Table 2-3	Valid Ranges for REAL*16 Data Type	23
Table 3-1	Corresponding Fortran and C Data Types	30
Table 3-2	How <i>mkf2c</i> treats Function Arguments	45
Table 4-1	Summary of System Interface Library Routines	52
Table 4-2	Overview of System Subroutines	59
Table 4-3	Information Returned by ERRSNS	61
Table 4-4	Arguments to MVBITS	62
Table 4-5	Function Extensions	62
Table 5-1	Summary of PCF Directives	103
Table 6-1	Summary of New Directives	127
Table A-1	Run-Time Error Messages	155

Introduction

This manual provides information on implementing Fortran 77 programs using the MIPSpro™ Fortran 77 compiler on IRIX™ 6.2 systems. 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,” provides 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, “Fortran Enhancements for Multiprocessors,” describes programming directives for running Fortran programs in a multiprocessor mode.
- Chapter 6, “Parallel Programming on Origin2000™,” describes the support provided for writing parallel programs on Origin2000 and how to improve program performance.

- Chapter 7, “Compiling and Debugging Parallel Fortran,” describes and illustrates compilation and debugging techniques for running Fortran programs in a multiprocessor mode.
- 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 77 language as implemented on Silicon Graphics® systems.

Refer to the *MIPS Compiling 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
- general discussion of performance tuning
- 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

For information on interfaces to programs written in assembly language, refer to the *MIPSpro 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.
<i>Italics</i>	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.
<code>Courier</code>	Indicates command syntax, program listings, computer output, and error messages.
<code>Courier bold</code>	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.

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 *elfdump*, *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 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]
```

For this format:

- 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**.

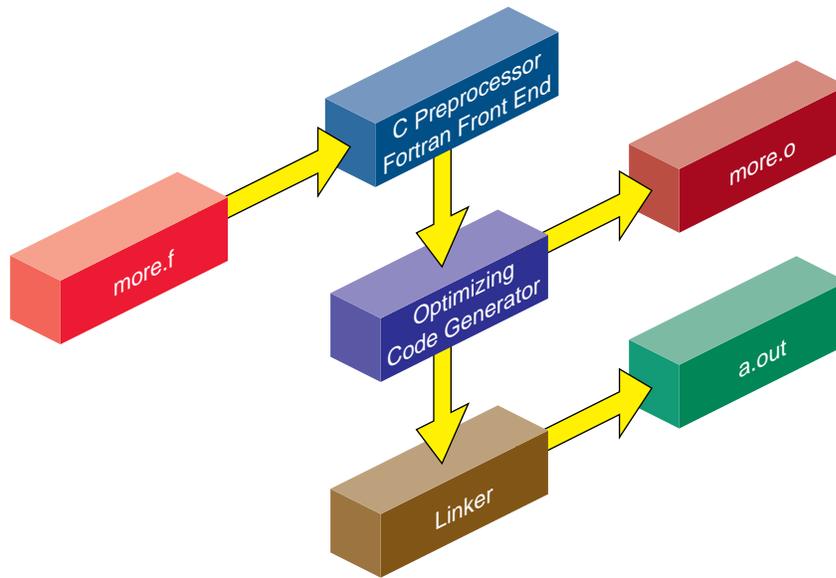


Figure 1-1 Compilation Process

Note the following:

- The source file ends with the required suffixes **.f**, **.F**, **.for**, **.FOR**, or **.i**.
- The Fortran front end has an integrated C preprocessor that provides full *cpp* capabilities.
- 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 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.

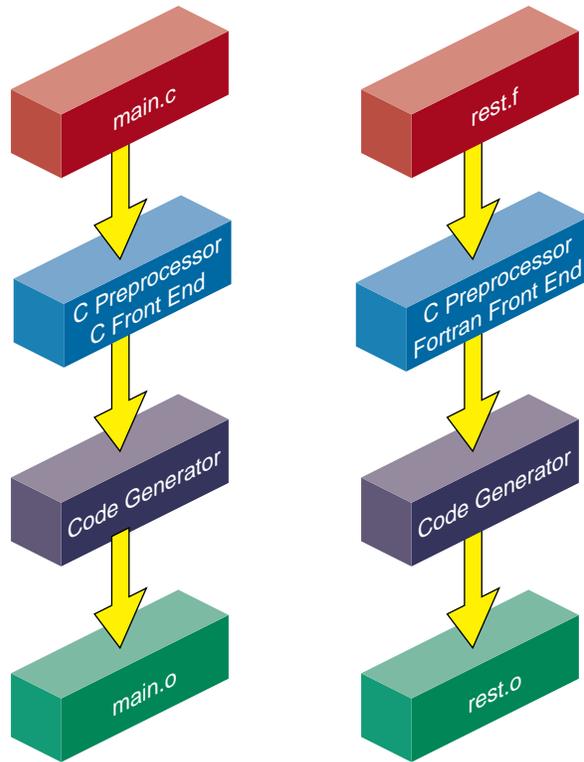


Figure 1-2 Compiling Multilanguage Programs

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.

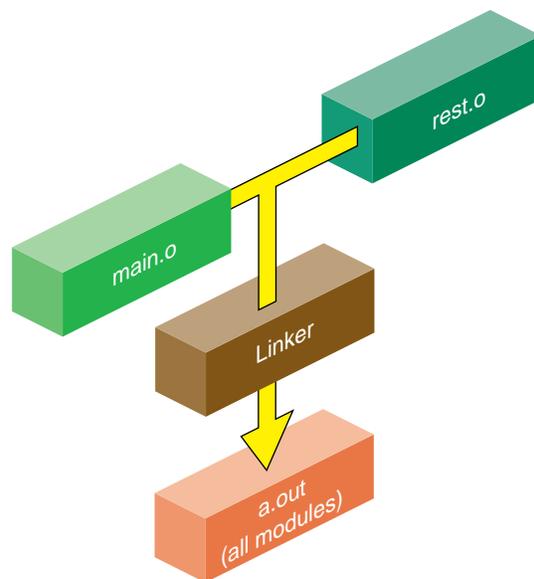


Figure 1-3 Linking

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

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

See the section called “FILES” in the *f77(1)* reference page for a complete list of the files used by the Fortran driver. Also refer to the *ld(1)* reference 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 reference 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 an overview of the Fortran-specific driver options. The `f77(1)` reference page has a complete description of the compiler options. This discussion only covers the relationships between some of the options, so as to help you make sense of the many options in the reference page. For for information you can review:

- The *MIPSpro Compiling and Performance Tuning Guide* for a discussion of the compiler options that are common to all MIPSpro compilers.
- The `pfa(1)` reference page for options related to the parallel optimizer.
- The `ld(1)` reference page for a description of the linker options.

Tip: The command `f77 -help` lists all compiler options for quick reference. Use the `-show` option to have the compiler document each phase of execution, showing the exact default and nondefault options passed to each.

Compiling Simple Programs

You need only a very few compiler options when you are compiling a simple program. Examples of simple programs include the following:

- Test cases used to explore algorithms or Fortran language features
- Programs that are principally interactive
- Programs whose performance is limited by disk I/O
- Programs you will execute under a debugger

In these cases you need only specify `-g` for debugging, the target machine architecture, and the word-length. For example, to compile a single source file to execute under `dbx` on a POWER CHALLENGE™ XL, you could use the following commands.

```
f77 -g -mips4 -n32 -o testcase testcase.f
dbx testcase
```

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

Specifying Source File Format

The options shown in Table 1-2 tell the compiler how to treat the program source file.

Table 1-2 Compile Options for Source File Format

Options	Purpose
<i>-ansi</i>	Report any nonstandard usages.
<i>-backslash</i>	Treat \ in character literals as a character, not as the start of an escape sequence.
<i>-col72, -col120, -extend_source, -noextend_source</i>	Specify margin columns of source lines.
<i>-d_lines</i>	Compile lines with D in column 1.
<i>-Dname, -Dname=def, -Uname</i>	Define, undefine names to the integrated C preprocessor.

Specifying Compiler Input and Output Files

The options summarized in Table 1-3 tell the compiler what output files to generate.

Table 1-3 Compile Options that Select Files

Options	Purpose
<i>-c</i>	Generate a single object file for each input file; do not link.
<i>-E</i>	Run only the macro preprocessor and write its output to standard output.
<i>-I, -Idir, -nostdinc</i>	Specify location of include files.
<i>-listing</i>	Request a listing file.
<i>-MDupdate</i>	Request Makefile dependency output data.
<i>-o</i>	Specify name of output file.
<i>-S</i>	Specify only assembly-language source output.

Using a Defaults Specification File

You can set the Application Binary Interface (ABI), instruction set architecture (ISA), and processor type without explicitly specifying them. Just set the environment variable `COMPILER_DEFAULTS_PATH` to a colon separated list of paths designating where the compiler is to look for a `compiler.defaults` file. If no `compiler.defaults` file is found or if the environment variable is not set, the compiler looks for `/etc/compiler.defaults`. If this file is not found, the compiler resorts to the built-in defaults.

The `compiler.defaults` file contains a `-DEFAULT: option` group specifier that specifies the default ABI, ISA, and processor. The compiler issues a warning if you specify anything other than `-DEFAULT: option` in the `compiler.defaults` file.

The format of the `-DEFAULT: option` group specifier is as follows:

```
-DEFAULT:[abi={32|n32|64}]:[isa=mipsn]:[proc={r4k|r5k|r8k|r10k}]
```

See the [f77\(1\)](#) reference page for an explanation of the `option` group.

Specifying Target Machine Features

The options summarized in Table 1-4 are used to specify the characteristics of the machine where the compiled program will be used.

Table 1-4 Compile Options for Target Machine Features

Options	Purpose
<code>-64</code> , <code>-n32</code> , <code>-32</code>	Whether target machine runs 64-bit mode, “new” 32-bit mode, or 32-bit mode. The <code>-64</code> option is allowed only with the <code>-mips3</code> and <code>-mips4</code> architecture options.
<code>-mips3</code> , <code>-mips4</code>	The instruction architecture available in the target machine: use <code>-mips3</code> for MIPS R4000 [®] and R4400 [®] machines; use <code>-mips4</code> for MIPS R8000 [®] and R10000 [™] machines.
<code>-TARG:option,...</code>	Specify certain details of the target CPU. Most of these options have correct default values based on the preceding options.
<code>-TENV:option,...</code>	Specify certain details of the software environment in which the source module will execute. Most of these options have correct default values based on other, more general values.

Specifying Memory Allocation and Alignment

The options summarized in Table 1-5 tell the compiler how to allocate memory and how to align variables in it. These options can have a strong effect on both program size and program speed.

Table 1-5 Compile Options for Memory Allocation and Alignment

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

Specifying Debugging and Profiling

The options summarized in Table 1-6 direct the compiler to include more or less extra information in the object file for debugging or profiling.

Table 1-6 Compile Options for Debugging and Profiling

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

For more information on debugging and profiling, see the manuals listed in the preface.

Specifying Optimization Levels

The MIPSpro Fortran 77 compiler contains three optimizer phases. One is part of the compiler “back end”; that is, it operates on the generated code, after all syntax analysis and source transformations are complete. The use of this standard optimizer, which is common to all MIPSpro compilers, is discussed in the *MIPSpro Compiling and Performance Tuning Guide*.

In addition, MIPSpro Fortran 77 contains two phases of accelerators, one for scalar optimization and one for parallel array optimization. These operate during the initial phases of the compilation, transforming the source statements before they are compiled to machine language. The options of the scalar optimizer are detailed in the `fopt(1)` reference page. The options of the parallel optimizer are detailed in the `pfa(1)` reference page.

Note: The reason these optimizer phases are documented in separate reference pages is that, when compiling for 32-bit machines, these phases use a separate product, the Power Fortran Accelerator, which has been integrated into the MIPSpro Fortran 77 compiler.

The options summarized in Table 1-7 are used to communicate to the different optimization phases.

Table 1-7 Compile Options for Optimization Control

Options	Purpose
<code>-O, -O0, -O1, -O2, -O3</code>	Select basic level of optimization, setting defaults for all optimization phases.
<code>-GCM:option,...</code>	Specify details of global code motion performed by the back-end optimizer.
<code>-OPT:option,...</code>	Specify miscellaneous details of optimization.
<code>-SWP:option,...</code>	Specify details of pipelining done by back-end optimizer.
<code>-pfa</code>	Request execution of the parallel source-to-source optimizer.
<code>-WK:option,...</code>	Pass options to parallel source-to-source optimizer of MIPSpro Power Fortran 77.

When you use `-O` to specify the optimization level, the compiler assumes default options for the accelerator phases. These defaults are listed in Table 1-8. To see all options that are passed to a compiler phase, use the `-show` option.

Table 1-8 Power Fortran Defaults for Optimization Levels

Optimization Level	Power Fortran Defaults Passed
<code>-O0</code>	<code>-WK,-roundoff=0,-scaleropt=0,-optimize=0</code>
<code>-O1</code>	<code>-WK,-roundoff=0,-scaleropt=0,-optimize=0</code>
<code>-O2</code>	<code>-WK,-roundoff=0,-scaleropt=0,-optimize=0</code>
<code>-O3</code>	<code>-WK,-roundoff=2,-scaleropt=3,-optimize=5</code>

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 *MIPSpro Compiling and Performance Tuning Guide* and the `ld(1)` reference page for more information.
- The `-jmpopt` option permits the linker to fill certain instruction delay slots not filled by the compiler front end. This option can improve the performance of smaller programs not requiring extremely large blocks of virtual memory. See the `ld(1)` reference page for more information.

Controlling Compiler Execution

The options summarized in Table 1-9 control the execution of the compiler phases.

Table 1-9 Compile Options for Compiler Phase Control

Options	Purpose
<code>-E, -P</code>	Execute only the integrated C preprocessor.
<code>-fe</code>	Stop compilation immediately after the front-end (syntax analysis) runs.
<code>-M</code>	Run only the macro preprocessor.

Table 1-9 (continued) Compile Options for Compiler Phase Control

Options	Purpose
<i>-Yc,path</i>	Load the compiler phase specified by <i>c</i> from the specified <i>path</i> .
<i>-Wc,option,...</i>	Pass the specified list of options to the compiler phase specified by <i>c</i> .

Object File Tools

The following tools provide information on object files as indicated:

<i>elfdump</i>	Lists headers, tables, and other selected parts of an ELF-format object or archive file.
<i>dis</i>	Disassembles object files into machine instructions.
<i>nm</i>	Prints symbol table information for object and archive files.
<i>file</i>	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.
<i>size</i>	Prints information about the text, rdata, data, sdata, bss, and sbss sections of the specified object or archive files. See the <i>a.out(4)</i> reference page for a description of the contents and format of section data.
<i>strip</i>	Removes symbol table and relocation bits.

For more information on these tools, see the *MIPSpro Compiling and Performance Tuning Guide* and the *dis(1)*, *elfdump(1)*, *file(1)*, *nm(1)*, *size(1)*, and *strip(1)* reference 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 *MIPSpro Compiling and Performance Tuning Guide* and the *ar(1)* reference 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.

Maximum Memory Allocations

The total memory allocation for a program, and individual arrays, can exceed 2 gigabytes (2 GB, or 2,048 MB).

Previous implementations of Fortran 77 limited the total program size, as well as the size of any single array, to 2 GB. The current release allows the total memory in use by the program to far exceed this. (For details on the memory use of individual scalar values, see “Alignment, Size, and Value Ranges” on page 21.)

Arrays Larger Than 2 Gigabytes

The MIPSPro 7.1 compilers support arrays that are larger than 2 gigabytes for programs compiled under the -64 ABI. The arrays can be local, global, and dynamically created as the following example demonstrates. (Note: Initializers are not provided for these arrays.) Large array support is limited to Fortran77, C, and C++.

For example:

```
$cat a2.c

#include <stdlib.h>

int i[0x100000008];

void foo()
{
int k[0x100000008];
k[0x100000007] = 9;
printf("%d \n", k[0x100000007]);
}

main()
{
char *j;
j = malloc(0x100000008);
i[0x100000007] = 7;
j[0x100000007] = 8;
printf("%d \n", i[0x100000007]);
printf("%d \n", j[0x100000007]);
foo();
}
```

You must run this program on a 64-bit operating system with IRIX version 6.2 (or higher). You can verify the system you have by typing `uname -a`. You must have enough swap space to support the working set size and you must have your shell limit `datasize`, `stacksize`, and `vmemoryuse` variables set to values large enough to support the sizes of the arrays (see `sh(1)` reference page).

The following example compiles and runs the above code after setting the `stacksize` to a correct value:

```
$uname -a
IRIX64 cydrome 6.2 03131016 IP19
$cc -64 -mips3 a2.c
$limit
cputime          unlimited
filesize         unlimited
datasize        unlimited
stacksize       65536 kbytesn
coredumpsize    unlimited
```

```
memoryuse
descriptors      200
vmemoryuse       unlimited
$limit stacksize unlimited
$limit
cputime          unlimited
filesize         unlimited
datasize         unlimited
stacksize        unlimited
coredumpsize     unlimited
memoryuse        754544 kbytes
descriptors      200
vmemoryuse       unlimited
$a.out
7
8
9
```

Local Variable (Stack Frame) Sizes

Arrays that are allocated on the process stack must not exceed 2 GB, but the total of all stack variables can exceed that limit. For example,

```
parameter (ndim = 16380)
integer*8 xmat(ndim,ndim), ymat(ndim,ndim), &
          zmat(ndim,ndim)
integer k(1073741824)
integer l(33554432, 256)
```

However, when an array is passed as an argument, it is not limited in size.

```
subroutine abc(k)
integer k(8589934592_8)
```

Static and Common Sizes

When compiling with the `-static` flag, global data is allocated as part of the compiled object (`.o`) file. The total size of any `.o` file may not exceed 2 GB. However, the total size of a program linked from multiple `.o` files may exceed 2 GB.

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

Pointer-based Memory

There is no limit on the size of a pointer-based array. For example,

```
integer *8 ndim
parameter (ndim = 20001)
pointer (xptr, xmat), (yptr, ymat), (zptr, zmat), &
      (aptr, amat)
xptra = malloc(ndim*ndim*8)
yptra = malloc(ndim*ndim*8)
zptr = malloc(ndim*ndim*8)
aptra = malloc(ndim*ndim*8)
```

It is important to make sure that malloc is called with an INTEGER*8 value. A count greater than 2 GB would be truncated if assigned to an INTEGER*4.

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 1-10 shows the standard preconnected files at program start.

Table 1-10 Preconnected Files

Unit #	Unit	Alternate Unit #
5	Standard input	* (in READ)
6	Standard output	* (in WRITE)
0	Standard error	** (in WRITE)

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` reference 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 User's Guide*.

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 the *dbx User's Guide*.

Floating Point Exceptions

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

The library provides the subroutine **handle_sigfpe** 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. The **-TENV:check_div** compile flag inserts checks for divide by zero or overflow. See the `handle_sigfpe(3F)` reference page for more information.

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 scalar data types. (For details on the maximum sizes of arrays, see “Maximum Memory Allocations” on page 14.)

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	$-2^{31} \dots 2^{31} - 1$
INTEGER*8		64 bits	Double word	$-2^{63} \dots 2^{63} - 1$
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

Table 2-1 (continued) Size, Alignment, and Value Ranges of Data Types

Type	Synonym	Size	Alignment	Value Range
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 option 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	1.797693134862315807937289714053023* 10 ³⁰⁸
Minimum normalized	2.0041683600089730005034939020703004* 10 ⁻²⁹²	2.0041683600089730005034939020703004* 10 ⁻²⁹²
Minimum denormalized	4.940656458412465441765687928682214* 10 ⁻³²⁴	2.225073858507201383090232717332404* 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 **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” on page 24 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 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)* reference 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 one of the alignment options (**-align8**, **-align16**) to *f77* when compiling any source file that references misaligned data (see the *f77(1)* reference page). If your program passes misaligned data to system libraries, you may also have to link it with the trap handler. See the *fixade(3f)* reference page for more information.

Fortran Program Interfaces

Sometimes it is necessary to create a program that combines modules written in Fortran and another language. For example,

- In a Fortran program, you need access to a facility that is only available as a C function, such as a member of a graphics library.
- In a program in another language, you need access to a computation that has been implemented as a Fortran subprogram, for example one of the many well-tested, efficient routines in the BLAS library.

Tip: Fortran subroutines and functions that give access to the IRIX system functions and other IRIX facilities already exist, and are documented in Chapter 4 of this manual.

This chapter focuses on the interface between Fortran and the most common other language, C. However other language can be called, for example C++.

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

How Fortran Treats Subprogram Names

The Fortran compiler normally changes the names of subprograms and named common blocks while it translates the source file. When these names appear in the object file for reference by other modules, they are normally changed in two ways:

- converted to all lowercase letters
- extended with a final underscore (`_`) character

Normally the following declarations

```
SUBROUTINE MATRIX
function MixedCase()
COMMON /CBLK/a,b,c
```

produce the identifiers **matrix_**, **mixedcase_**, and **blk_** (all lowercase with appended underscore) in the generated object file.

Note: Fortran intrinsic functions are not named according to these rules. The external names of intrinsic functions as defined in the Fortran library are not directly related to the intrinsic function names as they are written in a program. The use of intrinsic function names is discussed in the *MIPSpro Fortran 77 Language Reference Manual*.

Working with Mixed-Case Names

There is no way by which you can make the Fortran compiler generate an external name containing uppercase letters. If you are porting a program that depends on the ability to call such a name, you will have to write a C function that takes the same arguments but which has a name composed of lowercase letters only. This C function can then call the function whose name contains mixed-case letters.

Note: Previous versions of the Fortran 77 compiler for 32-bit systems supported the `-U` compiler option, telling the compiler to not force all uppercase input to lowercase. As a result, uppercase letters could be preserved in external names in the object file. As now implemented, this option does not affect the case of external names in the object file.

Preventing a Suffix Underscore with \$

You can prevent the compiler from appending an underscore to a name by writing the name with a terminal currency symbol (`$`). The `'$'` is not reproduced in the object file. It is dropped, but it prevents the compiler from appending an underscore. The declaration

```
EXTERNAL NOUNDER$
```

produces the name **nounder** (lowercase, but no trailing underscore) in the object file.

Note: This meaning of `'$'` in names applies only to subprogram names. If you end the name of a `COMMON` block with `'$'`, the name in the object file includes the `'$'` and ends with an underscore regardless.

Naming Fortran Subprograms from C

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

```
SUBROUTINE HYPOT()
```

would typically be declared in this C function (lowercase with trailing underscore):

```
extern int hypot_()
```

You must know if a subprogram is declared with a trailing ‘\$’ to suppress the underscore.

Naming C Functions from Fortran

The C compiler does not modify the names of C functions. C functions can have uppercase or mixed-case names, and they have terminal underscores only when the programmer writes them that way.

In order to call a C function from a Fortran program you must ensure that the Fortran compiler spells the name correctly. When you control the name of the C function, the simplest solution is to give it a name that consists of lowercase letters with a terminal underscore. For example, the following C function:

```
int fromfort_() {...}
```

could be declared in a Fortran program as follows:

```
EXTERNAL FROMFORT
```

When you do not control the name of a C function, you must cause the Fortran compiler to generate the correct name in the object file. Write the C function’s name using a terminal ‘\$’ character to suppress the terminal underscore. (You cannot cause the compiler to generate an external name with uppercase letters in it.)

Testing Name Spelling Using *nm*

You can verify the spelling of names in an object file using the *nm* command (or with the *elfdump* command with the *-t* or *-Dt* options). To see the subroutine and common names generated by the compiler, apply *nm* to the generated .o (object) or executable file.

Correspondence of Fortran and C Data Types

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

Corresponding Scalar Types

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

Table 3-1 Corresponding Fortran and C Data Types

Fortran Data Type	Corresponding C type
BYTE, INTEGER*1, LOGICAL*1	signed char
CHARACTER*1	unsigned char
INTEGER*2, LOGICAL*2	short
INTEGER ^a , INTEGER*4, LOGICAL ^a , LOGICAL*4	int or long
INTEGER*8, LOGICAL*8	long long
REAL ^a , REAL*4	float
DOUBLE PRECISION, REAL*8	double
REAL*16	long double
COMPLEX ^a , COMPLEX*8	typedef struct{float real, imag;} cpx8;
DOUBLE COMPLEX, COMPLEX*16	typedef struct{ double real, imag; } cpx16;
COMPLEX*32	typedef struct{long double real, imag;} cpx32;
CHARACTER*n (n>1)	typedef char fstr_n[n];

a. Assuming default precision

The rules governing alignment of variables within common blocks are covered under “Alignment, Size, and Value Ranges” on page 21.

Corresponding Character Types

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

A Fortran CHARACTER*n ($n > 1$) variable contains exactly n characters at all times. When a shorter character expression is assigned to it, it is padded on the right with spaces to reach n characters.

A C vector of characters is normally sized 1 greater than the longest string assigned to it. It may contain fewer meaningful characters than its size allows, and the end of meaningful data is marked by a null byte. There is no null byte at the end of a Fortran string. (The programmer can create a null byte using the Hollerith constant '\0' but this is not normally done.)

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

Corresponding Array Elements

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

To use a Fortran array in C,

- Reverse the order of dimension limits when declaring the array
- Reverse the sequence of subscript variables in a subscript expression
- Adjust the subscripts to origin-0 (usually, decrement by 1)

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

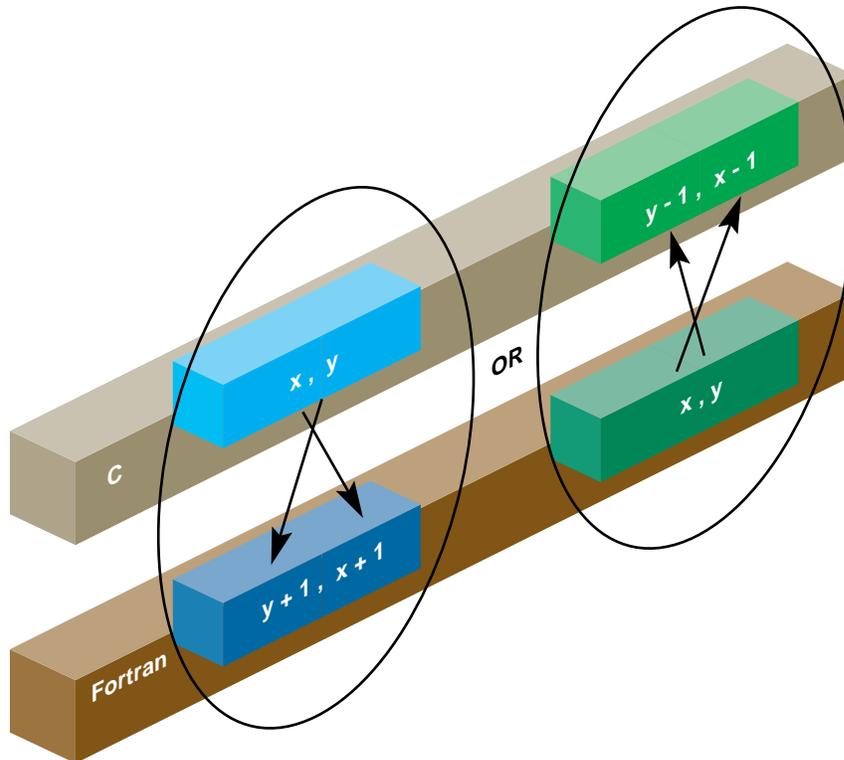


Figure 3-1 Correspondence Between Fortran and C Array Subscripts

For a coding example, see “Using Fortran Arrays in C Code” on page 41.

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

How Fortran Passes Subprogram Parameters

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

Normal Treatment of Parameters

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

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

Example 3-1 Example Subroutine Call

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

Code generated from the CALL in Example 3-1 prepares these 5 argument values:

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

Example 3-2 Example Function Call

```
CHARACTER*8 symb1,picksym  
CHARACTER*100 sentence  
INTEGER nsym  
symb1 = picksym(sentence,nsym)
```

Code generated from the function call in Example 3-2 prepares these 5 argument values:

1. The address of variable *syml*, the function result space
2. The length of *syml*, an integer value of 8
3. The address of *sentence*, the first explicit parameter
4. The address of *nsym*, the second explicit parameter
5. The length of *sentence*, an integer value of 100

You can force changes in these conventions using %VAL and %LOC; this is covered under “Calls to C Using LOC%, REF% and VAL%” on page 42.

Calling Fortran from C

There are two types of callable Fortran subprograms: subroutines and functions (these units are documented in the *MIPSpro Fortran 77 Language Reference Manual*). In C terminology, both types of subprogram are external functions. The difference is the use of the function return value from each.

Calling Fortran Subroutines from C

From the standpoint of a C module, a Fortran subroutine is an external function returning int. The integer return value is normally ignored by a C caller (its meaning is discussed in “Alternate Subroutine Returns” on page 36).

The following two examples show a simple Fortran subroutine and a sketch of a call to it.

Example 3-3 Example Fortran Subroutine with COMPLEX Parameters

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

Example 3-4 C Declaration and Call with COMPLEX Parameters

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

The Fortran subroutine in Example 3-3 is named in Example 3-4 using lowercase letters and a terminal underscore. It is declared as returning an integer. For clarity, the actual call is cast to (void) to show that the return value is intentionally ignored.

The trivial subroutine in the following example takes adjustable-length character parameters.

Example 3-5 Example Fortran Subroutine with String Parameters

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

Example 3-6 C Program that Passes String Parameters

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

The C program in Example 3-6 prepares CHARACTER*16 values and passes them to the subroutine in Example 3-5. Observe that the subroutine call requires 5 parameters, including the lengths of the two string parameters. In Example 3-6, the string length parameters are generated using `sizeof()`, derived from the typedef `fstr_16`.

Example 3-7 C Program that Passes Different String Lengths

```
extern int
prt_(char*pbef, float*pval, char*paft, int lbef, int laft);
main()
{
    float val = 2.1828e0;
    char *bef = "Start:";
    char *aft = ":End";
    (void)prt_(bef,&val,aft,strlen(bef),strlen(aft));
}
```

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

Alternate Subroutine Returns

In Fortran, a subroutine can be defined with one or more asterisks (*) in the position of dummy parameters. When such a subroutine is called, the places of these parameters in the CALL statement are supposed to be filled with statement numbers or statement labels. The subroutine returns an integer which selects among the statement numbers, so that the subroutine call acts as both a call and a computed go-to (for more details, see the discussions of the CALL and RETURN statements in the *MIPSpro Fortran 77 Language Reference Manual*).

Fortran does not generate code to pass statement numbers or labels to a subroutine. No actual parameters are passed to correspond to dummy parameters given as asterisks. When you code a C prototype for such a subroutine, simply ignore these parameter positions. A CALL statement such as

```
CALL NRET (*1,*2,*3)
```

is treated exactly as if it were the computed GOTO written as

```
GOTO (1,2,3), NRET()
```

The value returned by a Fortran subroutine is the value specified on the RETURN statement, and will vary between 0 and the number of asterisk dummy parameters in the subroutine definition.

Calling Fortran Functions from C

A Fortran function returns a scalar value as its explicit result. This corresponds exactly to the C concept of a function with an explicit return value. When the Fortran function returns any type shown in Table 3-1 other than CHARACTER*n (n>1), you can call the function from C and handle its return value exactly as if it were a C function returning that data type.

Example 3-8 Fortran Function Returning COMPLEX*16

```
COMPLEX*16 FUNCTION FSUB16(INP)
COMPLEX*16 INP
FSUB16 = INP
END
```

The trivial function shown in Example 3-8 accepts and returns COMPLEX*16 values. Although a COMPLEX value is declared as a structure in C, it can be used as the return type of a function.

Example 3-9 C Program that Receives COMPLEX Return Value

```
typedef struct{ double real, imag; } cpx16;
extern cpx16 fsub16_( cpx16 * inp );
main()
{
    cpx16 inp = { -3.333, -5.555 };
    cpx16 oup = { 0.0, 0.0 };
    printf("testing fsub16...");
    oup = fsub16_( &inp );
    if ( inp.real == oup.real && inp.imag == oup.imag )
        printf("Ok\n");
    else
        printf("Nope\n");
}
```

The C program in Example 3-9 shows how the function in Example 3-8 is declared and called. Observe that the parameters to a function, like the parameters to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Note: In IRIX 5.3 and earlier, you can *not* call a Fortran function that returns COMPLEX (although you can call one that returns any other arithmetic type). The register conventions used by compilers prior to IRIX 6.0 do not permit returning a structure value from a Fortran function to a C caller.

Example 3-10 Fortran Function Returning CHARACTER*16

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

The function in Example 3-10 has a CHARACTER*16 return value. When the Fortran function returns a CHARACTER*n (n>1) value, the returned value is not the explicit result of the function. Instead, you must pass the address and length of the result area as the first two parameters of the function.

Example 3-11 C Program that Receives CHARACTER*16 Return

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *pz,int lz,int *pj,int *pk,fstr_16*ps,int ls);
main()
{
    char work[64];
    fstr_16 inp,oup;
    int j=7;
    int k=11;
    strncpy(inp,"0123456789abcdef",sizeof(inp));
    fs16_ (oup, sizeof(oup), &j, &k, inp, sizeof(inp) );
    strncpy(work,oup,sizeof(oup));
    work[sizeof(oup)] = '\0';
    printf("FS16 returns <%s>\n",work);
}
```

The C program in Example 3-11 calls the function in Example 3-10. The address and length of the function result are the first two parameters of the function. (Since type *fstr_16* is an array, its name, *oup*, evaluates to the address of its first element.) The next three parameters are the addresses of the three named parameters; and the final parameter is the length of the string parameter.

Calling C from Fortran

In general, you can call units of C code from Fortran as if they were written in Fortran, provided that the C modules follow the Fortran conventions for passing parameters (see “How Fortran Passes Subprogram Parameters” on page 33). When the C program expects parameters passed using other conventions, you can either write special forms of CALL, or you can build a “wrapper” for the C functions using the *mkf2c* command.

Normal Calls to C Functions

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

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

```

/*
|| C functions to export the facilities of strtoll()
|| to Fortran 77 programs.  Effective Fortran declaration:
||
|| INTEGER*8 FUNCTION ISCAN(S,J)
|| CHARACTER*(*) S
|| INTEGER J
||
|| String S(J:) is scanned for the next signed long value
|| as specified by strtoll(3c) for a "base" argument of 0
|| (meaning that octal and hex literals are accepted).
||
|| The converted long long is the function value, and J is
|| updated to the non-space character following the last
|| converted character, or to 1+LEN(S).
||
|| Note: if this routine is called when S(J:J) is neither
|| whitespace nor the initial of a valid numeric literal,
|| it returns 0 and does not advance J.
*/
#include <ctype.h> /* for isspace() */
long long iscan_(char *ps, int *pj, int ls)
{
    int  scanPos, scanLen;
    long long ret = 0;
    char wrk[1024];
    char *endpt;
    /* when J>LEN(S), do nothing, return 0 */

```

```
if (ls >= *pj)
{
  /* convert J to origin-0, permit J=0 */
  scanPos = (0 < *pj)? *pj-1 : 0 ;

  /* calculate effective length of S(J:) */
  scanLen = ls - scanPos;

  /* copy S(J:) and append a null for strtoll() */
  strncpy(wrk,(ps+scanPos),scanLen);
  wrk[scanLen] = '\0';

  /* scan for the integer */
  ret = strtoll(wrk, &endpt, 0);

  /*
  || Advance over any whitespace following the number.
  || Trailing spaces are common at the end of Fortran
  || fixed-length char vars.
  */
  while(isspace(*endpt)) { ++endpt; }
  *pj = (endpt - wrk)+scanPos+1;
}
return ret;
}
```

The following program in demonstrates a call to the function in Example 3-12.

```
EXTERNAL ISCAN
INTEGER*8 ISCAN
INTEGER*8 RET
INTEGER J,K
CHARACTER*50 INP
INP = '1 -99 3141592 0xffff 033 '
J = 0
DO 10 WHILE (J .LT. LEN(INP))
  K = J
  RET = ISCAN(INP,J)
  PRINT *, K, ': ',RET,' -->',J
10 CONTINUE
END
```

Using Fortran COMMON in C Code

A C function can refer to the contents of a COMMON block defined in a Fortran program. The name of the block as given in the COMMON statement is altered as described in “How Fortran Treats Subprogram Names” on page 27 (that is, forced to lowercase and extended with an underscore). The name of the “blank common” is `_BLNK__` (one leading, two final, underscores).

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

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

An example is shown below.

Example 3-13 Common Block Usage in Fortran and C

```
INTEGER STKTOP,STKLEN,STACK(100)
COMMON /WITHC/STKTOP,STKLEN,STACK

struct fstack {
    int stktop, stklen;
    int stack[100];
}
extern fstack withc_;
int peektop_()
{
    if (withc_.stktop) /* stack not empty */
        return withc_.stack[withc_.stktop-1];
    else...
}
```

Using Fortran Arrays in C Code

As described under “Corresponding Array Elements” on page 31, a C program must take special steps to access arrays created in Fortran.

Example 3-14 Fortran Program Sharing an Array in Common with C

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

The Fortran fragment in Example 3-14 prepares a matrix in a common block, then calls a C subroutine to modify the array.

Example 3-15 C Subroutine to Modify a Common Array

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

The C function in Example 3-15 stores its third argument in the common array using the subscripts passed in the first two arguments. In the C function, the order of the dimensions of the array are reversed. The subscript values are reversed to match, and decremented by 1 to match the C assumption of 0-origin indexing.

Calls to C Using LOC%, REF% and VAL%

Using the special intrinsic functions %VAL, %REF, and %LOC you can pass parameters in ways other than the standard Fortran conventions described under “How Fortran Passes Subprogram Parameters” on page 33. These intrinsic functions are documented in the *MIPSpro Fortran 77 Language Reference Manual*.

Using %VAL

%VAL is used in parameter lists to cause parameters to be passed by value rather than by reference. Examine the following function prototype (from the random(3b) reference page).

```
char *initstate(unsigned int seed, char *state, int n);
```

This function takes an integer value as its first parameter. Fortran would normally pass the address of an integer value, but %VAL can be used to make it pass the integer itself. Example 3-16 demonstrates a call to function **initstate()** and the other functions of the **random()** group.

Example 3-16 Fortran Function Calls Using %VAL

```

C declare the external functions in random(3b)
C random() returns i*4, the others return char*
      EXTERNAL RANDOM$, INITSTATE$, SETSTATE$
      INTEGER*4 RANDOM$
      INTEGER*8 INITSTATE$,SETSTATE$
C We use "states" of 128 bytes, see random(3b)
C Note: An undocumented assumption of random() is that
C a "state" is dword-aligned! Hence, use a common.
      CHARACTER*128 STATE1, STATE2
      COMMON /RANSTATES/STATE1,STATE2
C working storage for state pointers
      INTEGER*8 PSTATE0, PSTATE1, PSTATE2
C initialize two states to the same value
      PSTATE0 = INITSTATE$(%VAL(8191),STATE1)
      PSTATE1 = INITSTATE$(%VAL(8191),STATE2)
      PSTATE2 = SETSTATE$(%VAL(PSTATE1))
C pull 8 numbers from state 1, print
      DO 10 I=1,8
          PRINT *,RANDOM$()
10    CONTINUE
C set the other state, pull 8 numbers & print
      PSTATE1 = SETSTATE$(%VAL(PSTATE2))
      DO 20 I=1,8
          PRINT *,RANDOM$()
20    CONTINUE
      END

```

The use of %VAL(8191) or %VAL(PSTATE1) causes that value to be passed, rather than an address of that value.

Using %REF

%REF is used in parameter lists to cause parameters to be passed by reference, that is, to pass the address of a value rather than the value itself.

Passing parameters by reference is the normal behavior of Silicon Graphics Fortran 77 compilers, so no effective difference exists between writing `%REF(parm)` and writing *parm* alone in a parameter list for non-character parameters. For character parameters, using `%REF(parm)` causes the character string length not to be added to the end of the parameter list as in the normal case. Thus, using the `%REF(parm)` guarantees that only the address of the parameter is parsed and nothing else.

When calling a C function that expects the address of a value rather than the value itself, you can write `%REF(parm)`. Examine this C prototype (see the `gmatch(3G)` reference page).

```
int gmatch (const char *str, const char *pattern);
```

This function `gmatch()` could be declared and called from Fortran.

Example 3-17 Fortran Call to `gmatch()` Using `%REF`

```
LOGICAL GMATCH$  
CHARACTER*8 FNAME,FPATTERN  
FNAME = 'foo.f\0'  
FPATTERN = '*.f\0'  
IF ( GMATCH$(%REF(FNAME),%REF(FPATTERN)) )...
```

The use of `%REF()` in Example 3-17 simply documents the fact that `gmatch()` expects addresses of character strings.

Using `%LOC`

`%LOC` returns the address of its argument. It can be used in any expression (not only within parameter lists), and is often used to set `POINTER` variables.

Making C Wrappers with `mkf2c`

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

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

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

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

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

Parameter Assumptions by *mkf2c*

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

Note: Through compiler release 6.0.2, *mkf2c* does not recognize the C data types “long long” and “long double” (INTEGER*8 and REAL*16). It treats arguments of this type as “long” and “double” respectively.

Table 3-2 How *mkf2c* treats Function Arguments

Data Type in C Prototype	Treatment by Generated Wrapper Code
unsigned char	Load CHARACTER*1 from memory to register, no sign extension
char	Load CHARACTER*1 from memory to register; sign extension only when <i>-signed</i> is specified
unsigned short, unsigned int	Load INTEGER*2 or INTEGER*4 from memory to register, no sign extension
short	Load INTEGER*2 from memory to register with sign extension
int, long	Load INTEGER*4 from memory to register with sign extension
long long	(Not supported through 6.0.2)
float	Load REAL*4 from memory to register, extending to double unless <i>-f</i> is specified
double	Load REAL*8 from memory to register
long double	(Not supported through 6.0.2)

Table 3-2 (continued) How *mkf2c* treats Function Arguments

Data Type in C Prototype	Treatment by Generated Wrapper Code
char <i>name</i> [], <i>name</i> [<i>n</i>]	Pass address of CHARACTER* <i>n</i> and pass length as integer parameter as Fortran does
char *	Copy CHARACTER* <i>n</i> value into allocated space, append null byte, pass address of copy

Character String Treatment by *mkf2c*

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

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

- The address passed to C points to the allocated memory
- The length of the value is not passed as an implicit argument
- There is a terminating null byte in the value
- Changes in the string are *not* reflected back to Fortran

A character array specified in the C function is processed by *mkf2c* just like a Fortran CHARACTER**n* value. In this case,

- The address prepared by Fortran is passed to the C function
- The length of the value is passed as an implicit argument (see “Normal Treatment of Parameters” on page 33)
- The character array contains no terminating null byte (unless the Fortran programmer supplies one)
- Changes in the array by the C function will be visible to Fortran

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

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

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

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

When *func1* is changed to use *varargs*, the C source file is as follows.

Example 3-18 C Function Using *varargs*

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

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

Example 3-19 C Code to Retrieve Hidden Parameters

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

Restrictions of *mkf2c*

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

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

Since *mkf2c* does not support structure-valued arguments, it does not support passing COMPLEX**n* values.

Using *mkf2c* and *extcentry*

mkf2c understands only a limited subset of the C grammar. This subset includes common C syntax for function entry point, C-style comments, and function bodies. However, it does not include constructs such as typedefs, external function declarations, or C preprocessor directives.

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

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

Example 3-20 Source File for Use with *extcentry*

```
typedef unsigned short grunt [4];
struct {
    long l, ll;
    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 */
```

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

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

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

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

Makefile Considerations

make(1) contains default rules to help automate the control of wrapper generation. The following example of a makefile illustrates the use of these rules. In the example, 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*.

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.
- “Extended Intrinsic Subroutines” describes the extensions to the Fortran intrinsic subroutines.
- “Extended Intrinsic Functions” describes the extensions to the Fortran functions.

Library Functions

The Fortran library functions provide an interface from Fortran programs to the IRIX system functions. System functions are facilities that are provided by the IRIX system kernel directly, as opposed to functions that are supplied by library code linked with your program. System functions are documented in volume 2 of the reference pages, with an overview in the intro(2) reference page.

Table 4-1 summarizes the functions in the Fortran run-time library. In general, the name of the interface routine is the same as the name of the system function that would be called from a C program. For details on a system interface routine use the command:

```
man 2 name_of_function
```

Note: You must declare the **time** function as **EXTERNAL**; if you do not, the compiler will assume you mean the VMS-compatible intrinsic **time** function rather than the IRIX system function. (In general it is a good idea to declare any library function in an **EXTERNAL** statement as documentation.)

Table 4-1 Summary of System Interface Library Routines

Function	Purpose
abort(3F)	abnormal termination
access	determine accessibility of a file
acct	enable/disable process accounting
alarm(3F)	execute a subroutine after a specified time
barrier(3P)	perform barrier operations
blockproc(2)	block processes
brk(2)	change data segment space allocation
chdir	change default directory
chmod(3F)	change mode of a file
chown(2)	change owner
chroot	change root directory for a command
close	close a file descriptor
creat	create or rewrite a file
ctime(3F)	return system time
dtime(3F)	return elapsed execution time
dup	duplicate an open file descriptor
etime(3F)	return elapsed execution time
exit(2)	terminate process with status
fcntl(2)	file control
fdate(3F)	return date and time in an ASCII string
fgetc(3F)	get a character from a logical unit
fork(2)	create a copy of this process
fputc(3F)	write a character to a Fortran logical unit

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
free_barrier(3P)	free barrier
fseek(3F)	reposition a file on a logical unit
fseek64(3F)	reposition a file on a logical unit for 64-bit architecture
fstat(2)	get file status
ftell(3F)	reposition a file on a logical unit
ftell64(3F)	reposition a file on a logical unit for 64-bit architecture
gerror(3F)	get system error messages
getarg(3F)	return command line arguments
getc(3F)	get a character from a logical unit
getcwd	get pathname of current working directory
getdents(2)	read directory entries
getegid(2)	get effective group ID
gethostid(2)	get unique identifier of current host
getenv(3F)	get value of environment variables
geteuid(2)	get effective user ID
getgid(2)	get user or group ID of the caller
gethostname(2)	get current host ID
getlog(3F)	get user's login name
getpgrp	get process group ID
getpid	get process ID
getppid	get parent process ID
getsockopt(2)	get options on sockets
getuid(2)	get user or group ID of caller
gmtime(3F)	return system time

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
iargc(3F)	return command line arguments
idate(3F)	return date or time in numerical form
ierrno(3F)	get system error messages
ioctl(2)	control device
isatty(3F)	determine if unit is associated with tty
itime(3F)	return date or time in numerical form
kill(2)	send a signal to a process
link(2)	make a link to an existing file
loc(3F)	return the address of an object
lseek(2)	move read/write file pointer
lseek64(2)	move read/write file pointer for 64-bit architecture
lstat(2)	get file status
ltime(3F)	return system time
m_fork(3P)	create parallel processes
m_get_myid(3P)	get task ID
m_get_numprocs(3P)	get number of subtasks
m_kill_procs(3P)	kill process
m_lock(3P)	set global lock
m_next(3P)	return value of counter
m_park_procs(3P)	suspend child processes
m_rele_procs(3P)	resume child processes
m_set_procs(3P)	set number of subtasks
m_sync(3P)	synchronize all threads
m_unlock(3P)	unset a global lock

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
mkdir(2)	make a directory
mknod(2)	make a directory/file
mount(2)	mount a filesystem
new_barrier(3P)	initialize a barrier structure
nice	lower priority of a process
open(2)	open a file
oserror(3F)	get/set system error
pause(2)	suspend process until signal
perror(3F)	get system error messages
pipe(2)	create an interprocess channel
plock(2)	lock process, test, or data in memory
prctl(2)	control processes
profil(2)	execution-time profile
ptrace	process trace
putc(3F)	write a character to a Fortran logical unit
putenv(3F)	set environment variable
qsort(3F)	quick sort
read	read from a file descriptor
readlink	read value of symbolic link
rename(3F)	change the name of a file
rmdir(2)	remove a directory
sbrk(2)	change data segment space allocation
schedctl(2)	call to scheduler control
send(2)	send a message to a socket

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
setblockproccnt(2)	set semaphore count
setgid	set group ID
sethostid(2)	set current host ID
setoserror(3F)	set system error
setpgrp(2)	set process group ID
setsockopt(2)	set options on sockets
setuid	set user ID
sginap(2)	put process to sleep
sginap64(2)	put process to sleep in 64-bit environment
shmat(2)	attach shared memory
shmdt(2)	detach shared memory
sighold(2)	raise priority and hold signal
sigignore(2)	ignore signal
signal(2)	change the action for a signal
sigpause(2)	suspend until receive signal
sigrelse(2)	release signal and lower priority
sigset(2)	specify system signal handling
sleep(3F)	suspend execution for an interval
socket(2)	create an endpoint for communication TCP
sproc(2)	create a new share group process
stat(2)	get file status
stime(2)	set time
symlink(2)	make symbolic link
sync	update superblock

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
sysmp(2)	control multiprocessing
sysmp64(2)	control multiprocessing in 64-bit environment
system(3F)	issue a shell command
taskblock(3P)	block tasks
taskcreate(3P)	create a new task
taskctl(3P)	control task
taskdestroy(3P)	kill task
tasksetblockcnt(3P)	set task semaphore count
taskunblock(3P)	unblock task
time(3F)	return system time (must be declared EXTERNAL)
ttynam(3F)	find name of terminal port
uadmin	administrative control
ulimit(2)	get and set user limits
ulimit64(2)	get and set user limits in 64-bit architecture
umask	get and set file creation mask
umount(2)	dismount a file system
unblockproc(2)	unblock processes
unlink(2)	remove a directory entry
uscalloc(3P)	shared memory allocator
uscalloc64(3P)	shared memory allocator in 64-bit environment
uscas(3P)	compare and swap operator
usclosetpollsema(3P)	detach file descriptor from a pollable semaphore
usconfig(3P)	semaphore and lock configuration operations
uscpssema(3P)	acquire a semaphore

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
uscsetlock(3P)	unconditionally set lock
usctlsema(3P)	semaphore control operations
usdumplock(3P)	dump lock information
usdumpsema(3P)	dump semaphore information
usfree(3P)	user shared memory allocation
usfreelock(3P)	free a lock
usfreepollsema(3P)	free a pollable semaphore
usfreesema(3P)	free a semaphore
usgetinfo(3P)	exchange information through an arena
usinit(3P)	semaphore and lock initialize routine
usinitlock(3P)	initialize a lock
usinitsema(3P)	initialize a semaphore
usmalloc(3P)	allocate shared memory
usmalloc64(3P)	allocate shared memory in 64-bit environment
usmallopt(3P)	control allocation algorithm
usnewlock(3P)	allocate and initialize a lock
usnewpollsema(3P)	allocate and initialize a pollable semaphore
usnewsema(3P)	allocate and initialize a semaphore
usopenpollsema(3P)	attach a file descriptor to a pollable semaphore
uspsema(3P)	acquire a semaphore
usputinfo(3P)	exchange information through an arena
usrealloc(3P)	user share memory allocation
usrealloc64(3P)	user share memory allocation in 64-bit environment
ussetlock(3P)	set lock

Table 4-1 (continued) Summary of System Interface Library Routines

Function	Purpose
ustestlock(3P)	test lock
ustestsema(3P)	return value of semaphore
usunsetlock(3P)	unset lock
usvsema(3P)	free a resource to a semaphore
uswsetlock(3P)	set lock
wait(2)	wait for a process to terminate
write	write to a file

Extended Intrinsic Subroutines

This section describes the intrinsic subroutines that are extensions to Fortran 77 (the intrinsic *functions* that are standard to Fortran 77 are documented in Appendix A of the *MIPSpro Fortran 77 Language Reference Manual*). The rules for using the names of intrinsic subroutines are also discussed in that appendix.

Table 4-2 gives an overview of the intrinsic subroutines and their function; they are described in detail in the sections following the topics.

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
<i>arg1</i>	IRIX global variable <i>errno</i> , which is then reset to zero after the call
<i>arg2</i>	Zero
<i>arg3</i>	Zero
<i>arg4</i>	Logical unit number of the file that was being processed when the error occurred
<i>arg5</i>	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
<i>source</i>	Integer variable or array element	Source location of bit field to be transferred.
<i>sbit</i>	Integer expression	First bit position in the field to be transferred from <i>source</i> .
<i>length</i>	Integer expression	Length of the field to be transferred from <i>source</i> .
<i>destination</i>	Integer variable or array element	Destination location of the bit field
<i>dbit</i>	Integer expression	First bit in <i>destination</i> to which the field is transferred.

Extended Intrinsic Functions

Table 4-5 gives an overview of the intrinsic 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

RAN generates a pseudo-random number. The format is as follows:

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

The argument *s* is an **INTEGER*4** variable or array element. This variable serves as a seed in determining the next random number. It should initially be set to a large, odd integer value. You can compute multiple random number series by supplying different variables or array elements as the seed argument to different calls of **RAN**.

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

The algorithm used in **RAN** is the linear congruential method. The code is similar to the following fragment:

```
S = S * 1103515245L + 12345  
RAN = FLOAT(IAND(RSHIFT(S,16),32767))/32768.0
```

RAN is supplied for compatibility with VMS. For demanding applications, consider using the functions described in the random(3b) reference page. These can all be called using techniques described under “Using %VAL” on page 42.

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 PCF implements a general model of parallelism.
- “Synchronization Intrinsic” describes synchronization operations.

Overview

The MIPSpro Fortran 77 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 6,

“Parallel Programming on Origin2000,” describes the support provided for writing parallel programs on Origin2000. Chapter 7, “Compiling and Debugging Parallel Fortran,” gives compilation and debugging instructions for parallel processing.

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.

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” on page 70). 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. By default, the multiprocessing code does not use more than 8 processors. If you want to use more processors, set the environment variable `MP_SET_NUMTHREADS` (see “Environment Variables: `MP_SET_NUMTHREADS`, `MP_BLOCKTIME`, `MP_SETUP`” on page 96 for more information). 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 be run on single-processor machines. So the above loop is 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 to the number of processors on the machine or 8, whichever is smaller. If you want, you can override the default and explicitly control the number of threads of execution used by a parallel job.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of when any other iteration of the loop is executed. Not all **DO** loops have this property, and loops without it cannot be correctly executed in parallel. However, many of the loops encountered in practice fit this model. Further, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially in parallel.

To provide compatibility for existing parallel programs, Silicon Graphics has adopted 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” on page 102). Additionally, a number of special routines exist that permit more direct control over the parallel execution (refer to “Advanced Features” on page 93 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” on page 98. 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 ... ] ) ]  
[ {SHARE | SHARED} (item[ , item ... ] ) ]  
[ {LASTLOCAL | LAST LOCAL} (item[ , item ... ] ) ]  
[ REDUCTION (item[ , item ... ] ) ]  
[ MP_SCHEDTYPE=mode ]  
[ CHUNK=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 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” on page 93). 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 5-17 in “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” on page 91 for more details.

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

```
[SIMPLE | STATIC]
[DYNAMIC]
[INTERLEAVE INTERLEAVED]
[GUIDED GSS]
[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. 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” on page 93 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 5-1 Simple DOACROSS

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 5-2 DOACROSS LOCAL

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 5-8 in “Analyzing Data Dependencies for Multiprocessing” on page 76 for more information on this example.

Example 5-3 DOACROSS LAST LOCAL

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” on page 70. 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 loop, 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.

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 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**” on page 70 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 determine 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” on page 82 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” on page 68). 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 5-4 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 5-5 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 5-6 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 5-7 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 5-8 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 user's responsibility 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 5-9 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 5-14 in “Breaking Data Dependencies” on page 82).

Example 5-10 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 5-11 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 5-12 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 5-13 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” on page 68). 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 5-14 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 “Rewritable Data Dependence” on page 80. **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 5-15 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 5-16 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 5-17 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
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*PIECE_SIZE -PIECE_SIZE +1, MIN(K*PIECE_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 this 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 5-18 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 5-19 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 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.

Topics covered in this section include:

- “Performing a Matrix Multiply” on page 89
- “Understanding Trade-Offs” on page 90
- “Load Balancing” on page 91
- “Reorganizing Common Blocks To Improve Cache Behavior” on page 93

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 5-18 in “Work Quantum” on page 87 (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 5-17 of “Breaking Data Dependencies” on page 82 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 5-20 Load Balancing

```

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 complex loops, it may be desirable to use **DYNAMIC** or **GSS** schedules.

Comparing the output from `pixie(1)` 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” on page 68 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.

Reorganizing Common Blocks To Improve Cache Behavior

You can use the **-OPT:reorg_common** option, which reorganizes common blocks to improve the cache behavior of accesses to members of the common block. This option produces consistent results as long as the code follows the standard and array references are made within the bounds of the array. It produces unexpected results if you violate the standard, for example, if you access an array out of its declared bounds.

The option is enabled by default at **-O3** only if all files referencing the common block are compiled at that optimization level. It is disabled if any file with the common block is compiled at either **-O2** and below, **-OPT:reorg_common=OFF**, or **-WI,-noivpad**.

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 if 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 **MP_SET_NUMTHREADS** environment variable (described in “Environment Variables: **MP_SET_NUMTHREADS**, **MP_BLOCKTIME**, **MP_SETUP**” on page 96), 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) reference 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_suggested_numthreads

The **mp_suggested_numthreads**(uint32) interface is available to any user program. You can call this routine at any time, and if possible, the suggested number of threads will be used at the start of the *next* parallel region. The implementation reserves the right to ignore the suggestion if it feels there is an over-riding reason to do so. You can call **mp_suggested_numthreads** directly. You can not vary the number of threads during execution of a parallel region. You can call **mp_suggested_numthreads** while in a parallel region, but the value will not take effect until the next one.

The automatic adjustment feature is enabled by setting the environment variable **MP_SUGNUMTHD** or **MPC_SUGNUMTHD**. The **mp_suggested_numthreads** interface is available whether or not the automatic feature is turned on.

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 *cs*h 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 multiple multiprocessors and multiple 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_SUGNUMTHD, MP_SUGNUMTHD_MIN, MP_SUGNUMTHD_MAX, MP_SUGNUMTHD_VERBOSE

Prior to the current (6.02) compiler release, the number of threads utilized during execution of a multiprocessor job was generally constant, set for example using `MP_SET_NUMTHREADS`.

In an environment with long running jobs and varying workloads, it may be preferable to vary the number of threads during execution of some jobs.

Setting `MP_SUGNUMTHD` causes the run-time library to create an additional, asynchronous process that periodically wakes up and monitors the system load. When idle processors exist, this process increases the number of threads, up to a maximum of `MP_SET_NUMTHREADS`. When the system load increases, it decreases the number of threads, possibly to as few as 1. When `MP_SUGNUMTHD` has no value, this feature is disabled and multithreading works as before.

The environment variables `MP_SUGNUMTHD_MIN` and `MP_SUGNUMTHD_MAX` are used to limit this feature as desired. When `MP_SUGNUMTHD_MIN` is set to an integer value between 1 and `MP_SET_NUMTHREADS`, the process will not decrease the number of threads below that value.

When `MP_SUGNUMTHD_MAX` is set to an integer value between the minimum number of threads and `MP_SET_NUMTHREADS`, the process will not increase the number of threads above that value.

If you set any value in the environment variable `MP_SUGNUMTHD_VERBOSE`, informational messages are written to `stderr` whenever the process changes the number of threads in use.

Calls to `mp_numthreads` and `mp_set_numthreads` are taken as a sign that the application depends on the number of threads in use. The number in use is frozen upon either of these calls; and if `MP_SUGNUMTHD_VERBOSE` is set, a message to that effect is written to `stderr`.

Environment Variables: 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 **-Wl,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 -Wl,Xlocal,foo_
```

makes the **COMMON** block **/foo/** a local **COMMON** block in the resulting **a.out** file. You can specify multiple **-Wl,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.

Note: The **C\$COPYIN** directive cannot be executed from inside a parallel region.

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 arguments **0** and **sig** (for example, **kill(0,sig)**) to signal all members of the process group might kill 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” on page 66), 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 5-1, 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 MIPSpro 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 5-1 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” on page 104). 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 ...]) ]  
[ {SHARE | SHARED} (item [, item ...]) ]
```

The **IF**, **LOCAL**, and **SHARED** clauses have the same meaning as in the **C\$DOACROSS** directive (refer to “Writing Parallel Fortran” on page 68).

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**” on page 68) 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**” on page 68 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**, *sched*, and *chunk* have the same meaning as in the **C\$DOACROSS** directive (refer to “Writing Parallel Fortran” on page 68). 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**” on page 68). Note in particular that it is legal to declare a data item as **LOCAL** in a parallel sections construct even if it was declared as **SHARED** in the enclosing parallel region.

The optional **NOWAIT** clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify **NOWAIT**, the processes will wait until all have reached the **END PSECTION** directive before proceeding.

Parallel sections must appear within a parallel region. They can contain critical section constructs (described in “Critical Section” on page 112) 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**” on page 68). 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 **CS** 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 **CS** 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.

Synchronization Intrinsic

The intrinsics described in this section provide a variety of primitive synchronization operations. Besides performing the particular synchronization operation, each of these intrinsics has two key properties:

- The function performed is guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load-linked and/or store-conditional instructions in a loop).
- Associated with each intrinsic are certain *memory barrier* properties that restrict the movement of memory references to *visible data* across the intrinsic operation (by either the compiler or the processor).

A *visible memory reference* is a reference to a data object potentially accessible by another thread executing in the same shared address space. A visible data object can be one of the following types:

- Fortran COMMON data
- data declared extern
- volatile data
- static data (either file-scope or function-scope)
- data accessible via function parameters
- automatic data (local-scope) that has had its address taken and assigned to some object that is visible (recursively)

The memory barrier semantics of an intrinsic can be one of the following types:

- *acquire barrier*, which disallows the movement of memory references to visible data from after the intrinsic (in program order) to before the intrinsic (this behavior is desirable at lock-acquire operations)
- *release barrier*, which disallows the movement of memory references to visible data from before the intrinsic (in program order) to after the intrinsic (this behavior is desirable at lock-release operations)
- *full barrier*, which disallows the movement of memory references to visible data past the intrinsic (in either direction), and is thus both an acquire and a release barrier. A barrier only restricts the movement of memory references to visible data across the intrinsic operation: between synchronization operations (or in their absence), memory references to visible data may be freely reordered subject to the usual data-dependence constraints.

Caution: Conditional execution of a synchronization intrinsic (such as within an **if** or a **while** statement) does not prevent the movement of memory references to visible data past the overall **if** or **while** construct.

Atomic fetch-and-op Operations

```
<type> __fetch_and_add (<type>* ptr, <type> value)
<type> __fetch_and_sub (<type>* ptr, <type> value)
<type> __fetch_and_or (<type>* ptr, <type> value)
<type> __fetch_and_and (<type>* ptr, <type> value)
<type> __fetch_and_xor (<type>* ptr, <type> value)
<type> __fetch_and_nand(<type>* ptr, <type> value)
```

Where <type> can be one of:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically performs the specified operation with the given value on *ptr, and returns the old value of *ptr.

```
{ tmp = *ptr; *ptr <op>= value; return tmp; }
```

2. Full barrier.

Atomic op-and-fetch Operations

```
<type> __add_and_fetch (<type>* ptr, <type> value)
<type> __sub_and_fetch (<type>* ptr, <type> value)
<type> __or_and_fetch (<type>* ptr, <type> value)
<type> __and_and_fetch (<type>* ptr, <type> value)
<type> __xor_and_fetch (<type>* ptr, <type> value)
<type> __nand_and_fetch(<type>* ptr, <type> value)
```

Where `<type>` can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically performs the specified operation with the given value on `*ptr`, and returns the new value of `*ptr`.

```
{ *ptr <op>= value; return *ptr; }
```

2. Full barrier.

Atomic BOOL Operation

```
BOOL __compare_and_swap (<type>* ptr, <type> oldvalue, <type> newvalue)
```

Where `<type>` can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically do the following: compare `*ptr` to old value. If equal, store the new value and return 1, otherwise return 0.

```
if (*ptr != oldvalue) return 0;
```

```
else {
```

```
    *ptr = newvalue
```

```
    return 1;
```

```
}
```

2. Full barrier.

Atomic synchronize Operation

```
__synchronize ()
```

Behavior:

1. Full barrier.

Atomic lock and unlock Operations

```
<type> __lock_test_and_set (<type>* ptr, <type> value)
```

Where <type> can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically store the supplied value in *ptr and return the old value of *ptr.

```
{ tmp = *ptr; *ptr = value; return tmp; }
```

2. Acquire barrier.

```
void __lock_release (<type>* ptr)
```

Where <type> can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Set *ptr to 0.

```
{ *ptr = 0 }
```

2. Release barrier.

Example of Implementing a Pure Spin-Wait Lock

The following example shows implementation of a spin-wait lock.

```
int lockvar = 0;
while (__lock_test_and_set (&lockvar, 1) != 0); /* acquire the lock */
    ... read and update shared variables ...
__lock_release (&lockvar); /* release the lock */
```

The memory barrier semantics of the intrinsics guarantee that no memory reference to visible data is moved out of the above critical section, either before of the lock-acquire or after the lock-release.

Note: Pure spin-wait locks can perform poorly under heavy contention.

If the data structures protected by the lock are known precisely (for example, *x*, *y*, and *z* in the example below), then those data structures can be precisely identified as follows:

```
while (__lock_test_and_set (&lockvar, 1, x, y, z) != 0);
    ... read/modify the variables x, y, and z ...
__lock_release (&lockvar, x, y, z);
```

Parallel Programming on Origin2000

This chapter describes the support provided for writing parallel programs on Origin2000. It assumes that you are familiar with basic parallel constructs. For more information on parallel constructs, refer to Chapter 5, “Fortran Enhancements for Multiprocessors.”

Topics covered in this chapter include:

- “Performance Tuning of Parallel Programs on Origin2000” on page 122
- “Data Distribution Directives” on page 127
- “Nested Doacross Directive” on page 129
- “Affinity Scheduling” on page 129
- “Specifying Processor Topology With the ONTO Clause” on page 132
- “Types of Data Distribution” on page 133
- “Optional Environment Variables and Compile-Time Options” on page 141
- “Examples” on page 142

A subset of the mechanisms described in this chapter are supported for C and C++, and are described in the *C Language Reference Manual*, Chapter 11. You can find additional information on parallel programming in “Models of Parallel Computation” in *Topics in IRIX Programming*.

Note: The multiprocessing features described in this chapter require support from the MP run-time library (*libmp*). IRIX operating system versions 6.3 (and above) are automatically shipped with this new library. If you wish to access these features on a machine running IRIX 6.2, then contact your local Silicon Graphics service provider or SGI Customer Support (1-800-800-4744) for *libmp*.

Performance Tuning of Parallel Programs on Origin2000

Origin2000 provides cache-coherent, shared memory in the hardware. Memory is physically distributed across processors. Consequently, references to locations in the remote memory of another processor take substantially longer (by a factor of two or more) to complete than references to locations in local memory. This can severely affect the performance of programs that suffer from a large number of cache misses. Figure 6-1 shows a simplified version of the Origin2000 memory hierarchy.

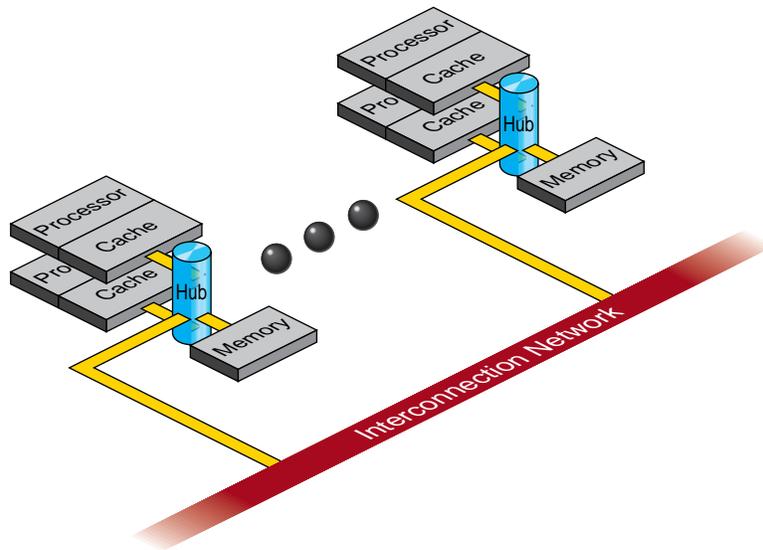


Figure 6-1 Origin2000 Memory Hierarchy

Improving Program Performance

To obtain good performance in such programs, it is important to schedule computation and distribute data across the underlying processors and memory modules, so that most cache misses are satisfied from local rather than remote memory. The primary goal of the programming support, therefore, is to enable user control over data placement and computation scheduling.

Cache behavior continues to be the largest single factor affecting performance, and programs with good cache behavior usually have little need for explicit data placement. In programs with high cache misses, if the misses correspond to true data communication between processors, then data placement is unlikely to help. In these cases, it may be necessary to redesign the algorithm to reduce inter-processor communication. Figure 6-2 shows this scenario.

If the misses are to data referenced primarily by a single processor, then data placement may be able to convert remote references to local references, thereby reducing the latency of the miss. The possible options for data placement are automatic page migration or explicit data distribution, either regular or reshaped (both of these are described in “Regular Data Distribution” on page 133 and “Data Distribution With Reshaping” on page 134). The differences between these choices are shown in Figure 6-2.

Automatic page migration requires no user intervention and is based on the run-time cache miss behavior of the program. It can therefore adjust to dynamic changes in the reference patterns. However, the page migration heuristics are deliberately conservative, and may be slow to react to changes in the references patterns. They are also limited to performing page-level allocation of data.

Regular data distribution (performing just page-level placement of the array) is also limited to page-level allocation, but is useful when the page migration heuristics are slow and the desired distribution is known to the programmer.

Finally, reshaped data distribution changes the layout of the array thereby overcoming the page-level allocation constraints; however, it is useful only if a data structure has the same (static) distribution for the duration of the program. Given these differences, it may be necessary to use each of these options for different data structures in the same program.

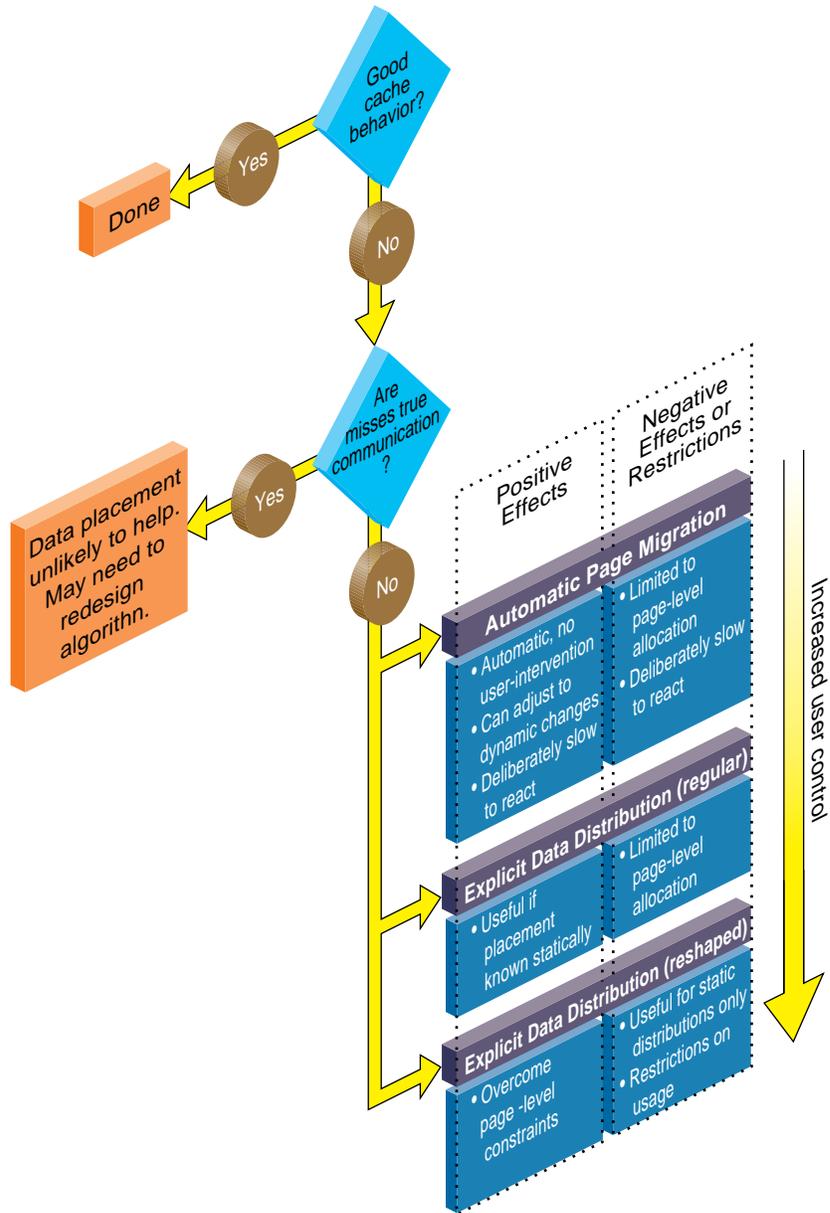


Figure 6-2 Cache Behavior and Solutions

Choosing Between Multiple Options

For a given data structure in the program, you can choose from the options described above based on the following criteria:

- If the program repeatedly references the data structure and benefits from reuse in the cache, then data placement is not needed.
- If the program incurs a large number of cache misses on the data structure, then you should identify the desired distribution in the array dimensions (such as BLOCK or CYCLIC, described in “Data Distribution Directives” on page 127) based on the desired parallelism in the program.

For example,

```
c$doacross
do i=2,n
  do j=2,n
    A(i,j) = 3*i + 4*j + A(i, j-1)
  enddo
enddo
```

suggests a distribution A(BLOCK, *), whereas:

```
do i=2,n
c$doacross
  do j=2,n
    A(i,j) = 3*i + 4*j + A(i-1, j)
  enddo
enddo
```

suggests a distribution of A(*, BLOCK).

- Having identified the desired distribution, you can select either regular and reshaped distribution based on the size of an individual processor’s portion of the distributed array. Regular distribution is useful only if each processor’s portion is substantially larger than the page-size in the underlying system (16KBytes on the Origin2000). Otherwise regular distribution is unlikely to be useful, and you should use **distribute_reshape**, where the compiler changes the layout of the array to overcome page-level constraints.

For example, consider the following code:

```
real*8 A(m, n)
c$ distribute A(BLOCK, *)
```

In this example, the size of each processor's portion is approximately m/P elements ($8*(m/P)$ bytes), where P is the number of processors. If m is 1000,000 then each processor's portion is likely to exceed a page and regular distribution is sufficient. If instead m is 10,000 then `distribute_reshape` is required to obtain the desired distribution.

In contrast, consider the following distribution:

```
c$ distribute A(*, BLOCK)
```

In this example, the size of each processor's portion is approximately $(m*n)/P$ elements ($8*(m*n)/P$ bytes). So if n is 100 (for instance), regular distribution may be sufficient even if m is only 10,000.

As this example illustrates, distributing the outer dimensions of an array increases the size of an individual processor's portion (favoring regular distribution), whereas distributing the inner dimensions is more likely to require reshaped distribution.

Finally, the IRIX operating system on Origin2000 follows a default "first-touch" page-allocation policy; that is, each page is allocated from the local memory of the processor that incurs a page-fault on that page. Therefore, in programs where the array is initialized (and consequently first referenced) in parallel, even a regular distribution directive may not be necessary, since the underlying pages are allocated from the desired memory location automatically due to the first-touch policy.

New Directives for Performance Tuning on Origin2000

The programming support consists of extensions to the existing Power Fortran/C directives/pragmas. Table 6-1 summarizes the new directives. Like the other Power Fortran/C directives, these new directives are ignored except under multiprocessor compilation.

Table 6-1 Summary of New Directives

Directive	Description
c\$distribute A (<dist>, <dist>, • • •) ^a	Data distribution
c\$redistribute A(<dist>, <dist>)	Dynamic data redistribution
c\$dynamic A	Redistributable annotation
c\$distribute_reshape B(<dist>)	Data distribution with reshaping
c\$page_place (<addr>, <sz>, <thread>)	Explicit placement of data
c\$doacross affinity (i) = data (A(i))	Data-affinity scheduling
c\$doacross affinity (i) = thread (<expr>)	Thread-affinity scheduling
c\$doacross nest (i,j)	Nested doacross

a. <dist> can be one of BLOCK, CYCLIC, CYCLIC(<expr>), or “*”. CYCLIC by itself implies a chunk size of 1. For performance reasons, CYCLIC(3) and CYCLIC(*k*) (where *k* has a run-time value of 3), may be incompatible when passing a reshaped array as a parameter to another routine.

The data distribution directives and doacross nest directive have an optional ONTO clause (described in “Specifying Processor Topology With the ONTO Clause” on page 132) to control the partitioning of processors across multiple dimensions.

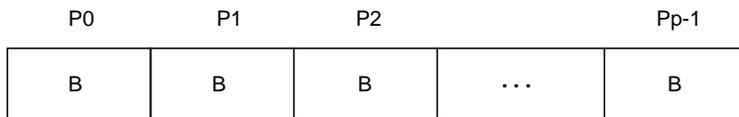
Data Distribution Directives

The data distribution directives allow you to specify High Performance Fortran-like distributions for array data structures. For irregular data structures, directives are provided to explicitly place data directly on a specific processor.

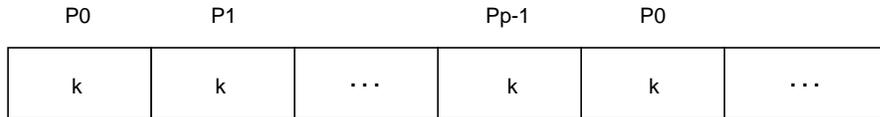
The **distribute**, **dynamic**, and **distribute_reshape** directives are declarations that must be specified in the declaration part of the program, along with the array declaration. The **redistribute** directive is an executable statement and can appear in any executable portion of the program.

You can specify a data distribution directive for any local, global, or common-block array. Each dimension of a multi-dimensional array can be independently distributed. The possible distribution types for an array dimension are BLOCK, CYCLIC (<expr>), and * (asterisk not distributed). (A CYCLIC distribution with a chunk size that is either greater than 1 or is determined at runtime is sometimes also called BLOCK-CYCLIC.)

A BLOCK distribution partitions the elements of the dimension of size N into P blocks (one per processor), with each block of size $B = \text{ceiling}(N/P)$.



A CYCLIC(k) distribution partitions the elements of the dimension into pieces of size k each, and distributes them sequentially across the processors.



A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the environment variable **MP_SET_NUMTHREADS**. If a distributed array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For instance, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension ($4 \times 4=16$), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension. You can override this default and explicitly control the number of processors in each dimension using the **ONTO** clause along with a data distribution directive.

Nested Doacross Directive

The nested **doacross** directive allows you to exploit nested concurrency in a limited manner. Although true nested parallelism is not supported, you can exploit parallelism across iterations of a perfectly nested loop-nest. For example:

```
c$doacross nest(i, j)
do i = 1, n
  do j = 1, m
    A(i,j) = 0
  enddo
enddo
```

This directive specifies that the entire set of iterations across the (i, j) loops can be executed concurrently. The restriction is that the `do-i` and `do-j` loops must be *perfectly nested*, that is, no code is allowed between either the `do-i` and `do-j` statements or the `enddo-i` and `enddo-j` statements. You can also supply the nest clause with the PCF directive **pdo**.

The existing clauses such as **local** and **shared** behave as before. You can combine a nested **doacross** with an affinity clause (as shown below), or with a schedtype of simple or interleaved (**dynamic** and **gss** are not currently supported). The default is simple scheduling, except when accessing reshaped arrays (see “Affinity Scheduling” on page 129).

```
c$doacross nest(i, j) affinity(i,j) = data(A(i,j))
do i = 2, n-1
  do j = 2, m-1
    A(i,j) = A(i,j) + i*j
  enddo
enddo
```

Affinity Scheduling

The goal of affinity scheduling is to control the mapping of iterations of a parallel loop for execution onto the underlying threads. Specify affinity scheduling with an additional clause to a **doacross** directive. An affinity clause, if supplied, overrides the SCHEDTYPE clause. This section describes the following topics:

- “Data Affinity” on page 130
- “Thread Affinity” on page 132

Data Affinity

The following code shows an example of data affinity:

```
c$distribute A(block)
c$doacross affinity(i) = data(A(a*i+b))
do i = 1, n
  A(a*i+b) = 0
enddo
```

The multiplier for the loop index variable (*a*) and the constant term (*b*) must both be literal constants, with *a* greater than zero.

The effect of this clause is to distribute the iterations of the parallel loop to match the data distribution specified for the array *A*, such that iteration *i* is executed on the processor that owns element *A(a*i+b)* based on the distribution for *A*. The iterations are scheduled based on the specified distribution, and are not affected by the actual underlying data-distribution (which may differ at page boundaries, for example).

In case of a multi-dimensional array, affinity is provided for the dimension that contains the loop-index variable. The loop-index variable cannot appear in more than one dimension in an affinity directive. For example:

```
c$distribute A (block, cyclic(1))
c$doacross affinity (i) = data (A(i+3, j))
do i = 1,n
  do j = 1,n
    A(i+3, j) = A(i+3,j-1)
  enddo
enddo
```

In this example, the loop is scheduled based on the block-distribution of the first dimension. Information on **doacross** is in “Nested Doacross Directive” on page 129. The affinity clause is also available with the PCF **pdo** directive.

The default schedtype for parallel loops is SIMPLE. However, under **-O3** compilation, loops that reference reshaped arrays default to data-affinity scheduling for the most frequently accessed reshaped array in the loop (chosen heuristically by the compiler). To obtain SIMPLE scheduling even at **-O3**, you can explicitly specify the schedtype on the parallel loop.

Data affinity for loops with non-unit stride can sometimes result in non-linear affinity expressions. In such situations the compiler issues a warning, ignores the affinity clause, and defaults to simple scheduling.

Data Affinity for Redistributed Arrays

By default, the compiler assumes that a distributed array is not dynamically redistributed, and directly schedules a parallel loop for the specified data affinity. In contrast, a redistributed array can have multiple possible distributions, and data affinity for a redistributed array must be implemented in the run-time system based on the particular distribution.

However, the compiler does not know whether or not an array is redistributed, since the array may be redistributed in another function (possibly even in another file). Therefore, you must explicitly specify the **c\$dynamic** declaration for redistributed arrays. This directive is required only in those functions that contain a **doacross** loop with data affinity for that array (see “Nested Doacross Directive” on page 129 for additional information). This informs the compiler that the array can be dynamically redistributed. Data affinity for such arrays is implemented through a run-time lookup.

Implementing data affinity through a run-time lookup incurs some extra overhead compared to a direct compile-time implementation. You can avoid this overhead when a subroutine contains data affinity for a redistributed array and the distribution of the array for the entire duration of that subroutine is known. In this situation, you can supply the **c\$distribute** directive with the particular distribution and omit the **c\$dynamic** directive.

By default, the compiler assumes that a distributed array is **not** redistributed at runtime. As a result, the distribution is known at compile time, and data affinity for the array can be implemented directly by the compiler. In contrast, since a redistributed array can have multiple possible distributions at runtime, data affinity for a redistributed array is implemented in the run-time system based on the distribution at runtime, incurring extra run-time overhead.

If an array is redistributed in the program, then you can explicitly specify a **c\$dynamic** directive for that array. The only effect of the **c\$dynamic** directive is to implement data affinity for that array at runtime rather than at compile time. If you know an array has a specified distribution throughout the duration of a subroutine, then you do not have to supply the **c\$dynamic** directive. The result is more efficient compile time affinity scheduling.

Since reshaped arrays cannot be dynamically redistributed, this is an issue only for regular data distribution.

Data Affinity for a Formal Parameter

You can supply a **c\$distribute** directive on a formal parameter, thereby specifying the distribution on the incoming actual parameter. If different calls to the subroutine have parameters with different distributions, then you can omit the **c\$distribute** directive on the formal parameter; data affinity loops in that subroutine are automatically implemented through a run-time lookup of the distribution. (This is permissible only for regular data distribution. For reshaped array parameters, the distribution must be fully specified on the formal parameter.)

Thread Affinity

Similar to data affinity, you can specify thread affinity as an additional clause on a **doacross** directive (refer to “Nested Doacross Directive” on page 129 for details). The syntax for thread affinity is as follows:

```
c$doacross affinity (i) = thread(<expr>)
```

The effect of this directive is to execute iteration *i* on the thread number given by the user-supplied expression (modulo the number of threads). Since the threads may need to evaluate this expression in each iteration of the loop, the variables used in the expression (other than the loop induction variable) must be declared shared and must not be modified during the execution of the loop. Violating these rules can lead to incorrect results.

If the expression does not depend on the loop induction variable, then all iterations will execute on the same thread and will not benefit from parallel execution.

Specifying Processor Topology With the ONTO Clause

This clause allows you to specify the processor topology when two (or more) dimensions of processors are required. For instance, if an array is distributed in two dimensions, then you can use the **ONTO** clause to specify how to partition the processors across the distributed dimensions. Or, in a nested **doacross** with two or more nested loops, you can use the **ONTO** clause to specify the partitioning of processors across the multiple parallel loops.

For example:

```
C Assign processor in the ratio 1:2 to the two dimension
real*8 A (100, 200)
c$distribute A (block, block) onto (1, 2)
```

```
C Use 2 processors in the do-i loop, and the remaining in the do-j loop
c$doacross nest (i, j) onto (2, *)
do i = 1, n
  do j = 1, m
    . . .
  enddo
enddo
```

Types of Data Distribution

There are two types of data distribution: *regular* and *reshaped*. The following sections describe each of these distributions.

Regular Data Distribution

The regular data distribution directives try to achieve the desired distribution solely by influencing the mapping of virtual addresses to physical pages without affecting the layout of the data structure. Since the granularity of data allocation is a physical page (at least 16 Kbytes), the achieved distribution is limited by the underlying page-granularity. However, the advantages are that regular data distribution directives can be added to an existing program without any restrictions, and can be used for affinity scheduling (see “Data Affinity” on page 130).

Distributed arrays can be dynamically redistributed with the following redistribute statement:

```
c$redistribute A (block, cyclic(k))
```

The **redistribute** is an executable statement that changes the distribution “permanently” (or until another **redistribute** statement). It also affects subsequent affinity scheduling.

The **c\$dynamic** directive specifies that the named array is redistributed in the program, and is useful in controlling affinity scheduling for dynamically redistributed arrays. It is discussed in “Data Affinity for Redistributed Arrays” on page 131.

Data Distribution With Reshaping

Similar to regular data distribution, the **reshape** directive specifies the desired distribution of an array. In addition, however, the **reshape** directive declares that the program makes no assumptions about the storage layout of that array. The compiler performs aggressive optimizations for reshaped arrays that violate standard Fortran-77 layout assumptions but guarantee the desired data distribution for that array.

The **reshape** directive accepts the same distributions as the regular data distribution directive, but uses a different keyword, as shown below:

```
c$distribute_reshape (block, cyclic(1))
```

Restrictions on Reshaped Arrays

Since the **distribute_reshape** directive specifies that the program does not depend on the storage layout of the reshaped array, restrictions on the arrays that can be reshaped include the following:

- The distribution of a reshaped array cannot be changed dynamically (that is, there is no **redistribute_reshape** directive).
- Initialized data cannot be reshaped.
- Arrays that are explicitly allocated through **alloca/malloc** and accessed through pointers cannot be reshaped.
- An array that is equivalenced to another array cannot be reshaped.
- I/O for a reshaped array cannot be mixed with namelist I/O or a function call in the same I/O statement.
- A COMMON block containing a reshaped array cannot be linked **-Xlocal**.

Caution: This user error is **not** caught by the compiler/linker.

If a reshaped array is passed as an actual parameter to a subroutine, two possible scenarios exist:

- The array is passed in its entirety (`call func(A)` passes the entire array `A`, whereas `call func(A(i,j))` passes a portion of `A`). The compiler automatically clones a copy of the called subroutine and compiles it for the incoming distribution. The actual and formal parameters must match in the number of dimensions, and the size of each dimension.

You can restrict a subroutine to accept a particular reshaped distribution on a parameter by specifying a **distribute_reshape** directive on the formal parameter within the subroutine. All calls to this subroutine with a mismatched distribution will lead to compile- or link-time errors.

- A portion of the array can be passed as a parameter, but the callee must access only a single processor's portion. If the callee exceeds a single processor's portion, then the results are undefined. You can use intrinsics to access details about the array distribution (described in "Query Intrinsics for Distributed Arrays" on page 136).

Error-Detection Support

Most errors in accessing reshaped arrays are caught either at compile time or at link time. These include:

- Inconsistencies in reshaped arrays across COMMON blocks (including across files)
- Declaring a reshaped array EQUIVALENCED to another array
- Inconsistencies in reshaped distributions on actual and formal parameters
- Other errors such as disallowed I/O statements involving reshaped arrays, reshaping initialized data, or reshaping dynamically allocated data

Errors such as matching the declared size of an array dimension typically are caught only at runtime. The compiler option, **-MP:check_reshape=on**, generates code to perform these tests at runtime. These run-time checks are not generated by default, since they incur overhead, but are useful during program development.

The run-time checks include:

- Inconsistencies in array-bound declarations on each actual and formal parameter
- Inconsistencies in declared bounds of a formal parameter that corresponds to a portion of a reshaped actual parameter.

Query Ininsics for Distributed Arrays

You can use the following set of intrinsics to obtain information about an individual dimension of a distributed array. Fortran array dimensions are numbered starting at 1. All routines work with 64-bit integers as shown below, and return -1 in case of an error (except `dsm_this_startingindex` where -1 may be a legal return value).

```
extern INT64 dsm_numthreads (void* array, INT64 dim)
    Called with a distributed array and a dimension number. Returns the
    number of threads in that dimension.

extern INT64 dsm_chunksize (void* array, INT64 dim)
    Returns the chunk size (ignoring partial chunks) in the given dimension
    for each of block, cyclic(..), and star distributions.

extern INT64 dsm_this_chunksize (void* array, INT64 dim, INT64 index)
    Returns the chunk size for the chunk containing the given index value
    for each of block, cyclic(..), and star. This value may be different from
    dsm_chunksize due to edge effects that may lead to a partial chunk.

extern INT64 dsm_rem_chunksize (void* array, INT64 dim, INT64 index)
    Returns the remaining chunk size from index to the end of the current
    chunk, inclusive of each end point. Essentially it is the distance from
    index to the end of that contiguous block, inclusive.

extern INT64 dsm_this_startingindex (void* array, INT64 dim, INT64 index)
    Returns the starting index value of the chunk containing the supplied
    index.

extern INT64 dsm_numchunks (void* array, INT64 dim)
    Returns the number of chunks (including partial chunks) in given dim
    for each of block, cyclic(..), and star distributions.

extern INT64 dsm_this_threadnum (void* array, INT64 dim, INT64 index)
    Returns the thread number for the chunk containing the given index
    value for each of block, cyclic(..), and star distributions.

extern INT64 dsm_distribution_block (void* array, INT64 dim)
extern INT64 dsm_distribution_cyclic (void* array, INT64 dim)
extern INT64 dsm_distribution_star (void* array, INT64 dim)
    Boolean routines to query the distribution of a given dimension.
```

```
extern INT64 dsm_isreshaped (void* array)
    Boolean routine to query whether reshaped or not.

extern INT64 dsm_isdistributed (void* array)
    Boolean routine to query whether distributed (regular or reshaped) or
    not.
```

Implementation of Reshaped Arrays

The compiler transforms a reshaped array into a pointer to a “processor array.” The processor array has one element per processor, with the element pointing to the portion of the array local to the corresponding processor.

Figure 6-3 shows the effect of the **distribute_reshape** directive with a BLOCK distribution on a 1-dimensional array. N is the size of the array dimension, P is the number of processors, and B is the block-size on each processor, $\text{ceiling} = (N/P)$.

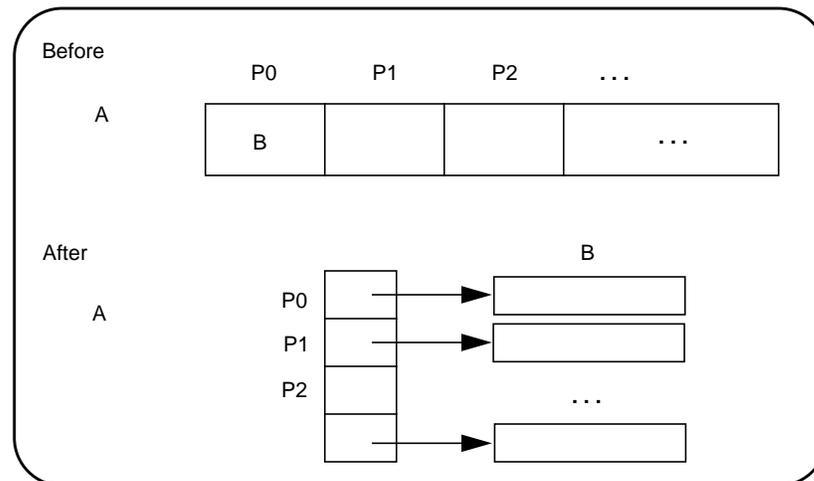


Figure 6-3 Implementation of BLOCK Distribution

With this implementation, an array reference $A(i)$ is transformed into a two-dimensional reference $A[i/B][i\%B]$ (in C syntax with C dimension order), where B is the size of each block, and given by $\text{ceiling}(N/P)$. Thus $A[i/B]$ points to a processor’s local portion of the array, and $A[i/B][i\%B]$ refers to a specific element within the local processor’s portion.

A CYCLIC distribution with a chunk size of 1 is implemented as shown in Figure 6-4.

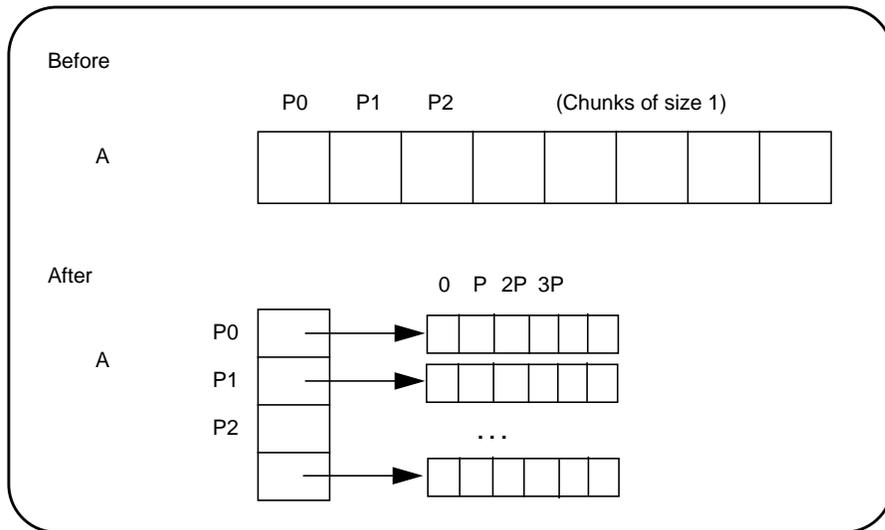


Figure 6-4 Implementation of CYCLIC(1) Distribution

An array reference, $A(i)$, is transformed to $A[i \% P] [i / P]$ where P is the number of threads in that distributed dimension.

Finally, a CYCLIC distribution with a chunk size that is either a constant greater than 1 or a run-time value (also called BLOCK-CYCLIC) is implemented as Figure 6-5 shows.

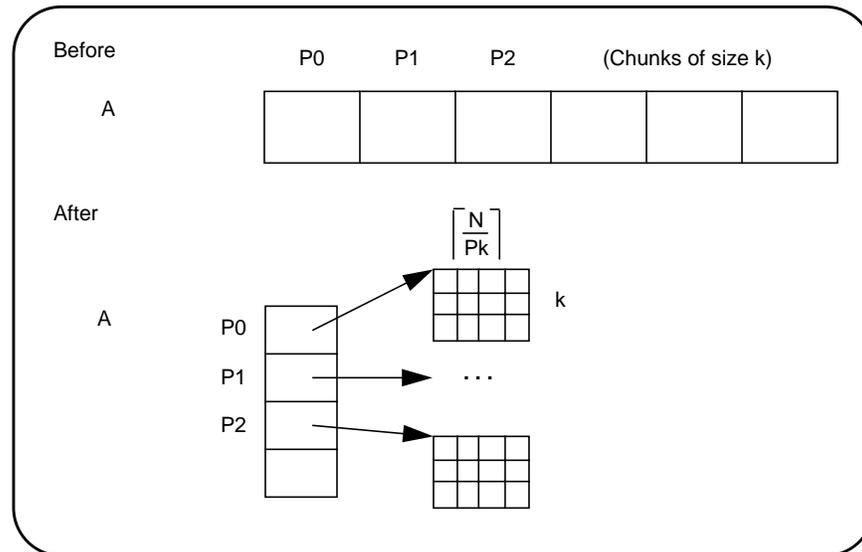


Figure 6-5 Implementation of BLOCK-CYCLIC Distribution

An array reference, $A(i)$, is transformed to the three-dimensional reference $A[(i/k)\%P][i/(Pk)][i\%k]$, where P is the total number of threads in that dimension, and k is the chunk size.

The compiler tries to optimize these divide/modulo operations out of inner loops through aggressive loop transformations such as blocking and peeling.

Regular vs. Reshaped Data Distribution

In summary, consider the differences between regular and reshaped data distribution. The advantage of regular distributions is that they do not impose any restrictions on the distributed arrays and can be freely applied in existing codes. Furthermore, they work well for distributions where page granularity is not a problem. For example, consider a BLOCK distribution of the columns of a two-dimensional Fortran array of size $A(r, c)$ (column-major layout) and distribution $(*, \text{BLOCK})$. If the size of each processor's portion, $\text{ceiling} = (c/P) * r * \text{element_size}$ is significantly greater than the page size (16KB on Origin2000), then regular data distribution should be effective in placing the data in the desired fashion.

However, regular data distribution is limited by page-granularity considerations. For instance, consider a (BLOCK, BLOCK) distribution of a two-dimensional array where the size of a column is much smaller than a page. Each physical page is likely to contain data belonging to multiple processors, making the data-distribution quite ineffective. (Data distribution may still be desirable from affinity-scheduling considerations, described in “Affinity Scheduling” on page 129.)

Reshaped data distribution addresses the problems of regular distributions by changing the layout of the array in memory so as to guarantee the desired distribution. However, since the array no longer conforms to standard Fortran-77 storage layout, there are restrictions on the usage of reshaped arrays.

Given both types of data distribution, you can choose between the two based on the characteristics of the particular array in an application.

Explicit Placement of Data

For irregular data structures, you can explicitly place data in the physical memory of a particular processor using the following directive:

```
c$page_place (<addr>, <size>, <threadnum>)
```

where *<addr>* is the starting address, *<size>* is the size in bytes, and *<threadnum>* is the number of the destination processor. This directive causes all the pages spanned by the virtual address range [*<addr>* • • • *<addr>*+*<size>*] to be allocated from the local memory of processor number *<threadnum>*. It is an executable statement; therefore, you can use it to place either statically or dynamically allocated data.

An example of this directive is as follows:

```
real*8 a(100)
c$page_place (a, 800, 3)
```

Optional Environment Variables and Compile-Time Options

You can control various run-time features through the following optional environment variables:

_DSM_OFF Disables non-uniform memory access (NUMA) specific calls (for example, to allocate pages from a particular memory).

_DSM_VERBOSE Prints messages about parameters being used during execution.

_DSM_PPM Specifies the number of processors to use per memory module. Must be set to an integer value; to use only one processor per memory module, set this variable to 1.

_DSM_FOP Enables the use of hardware **fetchop** operations to implement the synchronization for the completion of a parallel region, more efficient than the default software implementation using synchronization through shared-memory locations.

_DSM_MIGRATION Automatic page migration is *off* by default. This variable, if set, must be set to one of the following:

OFF disables migration (the default).

ON enables migration except for explicitly placed data (using `page_place` or a data distribution directive).

ALL_ON enables migration for ALL data.

_DSM_ROUND_ROBIN

Requests round-robin data allocation across memories (rather than first-touch) for all of stack, data, and text segments. The default is first-touch.

MP_SUGNUMTHD

If set, this variable enables the use of *dynamic threads* in the multiprocessor (MP) runtime. Essentially, with dynamic threads the MP runtime automatically adjusts the number of threads used for a parallel loop at runtime based on the overall system load. This feature improves the overall throughput of the system. Furthermore, by avoiding excessive concurrency, this feature can reduce delays at synchronization points within a single application.

PAGESIZE_STACK, PAGESIZE_DATA, PAGESIZE_TEXT

Specifies the desired page size in kilobytes. Must be set to an integer value.

Useful compile-time options include:

-MP:dsm={on, off} (default on)

All the data-distribution and scheduling features described in this chapter are enabled by default under **-mp** compilation. To disable all the DSM-specific directives (for example, distribution and affinity scheduling), compile with **-MP:dsm=off**.

-MP:clone={on, off} (default on)

The compiler automatically clones procedures that are called with reshaped arrays as parameters for the incoming distribution. However, if you have explicitly specified the distribution on all relevant formal parameters, then you can disable auto-cloning with **-MP:clone=off**. The consistency checking of the distribution between actual and formal parameters is *not* affected by this flag, and is always enabled.

-MP:check_reshape={on, off} (default off)

Enables generation of the run-time consistency checks across procedure boundaries when passing reshaped arrays (or portions thereof) as parameters.

Examples

The examples in this section include the following:

- “Distributing Columns of a Matrix” on page 143
- “Using Data Distribution and Data Affinity Scheduling” on page 144
- “Parameter Passing” on page 145
- “Redistributed Arrays” on page 146
- “Irregular Distributions and Thread Affinity” on page 147

Distributing Columns of a Matrix

The example below distributes sequentially the columns of a matrix. Such a distribution places data effectively only if the size of an individual column exceeds that of a page.

```
real*8 A(n, n)
C Distribute columns in cyclic fashion
c$distribute A (*, CYCLIC(1))

C Perform Gaussian elimination across columns
C The affinity clause distributes the loop iterations based
C on the column distribution of A
do i = 1, n
c$doacross affinity(j) = data(A(i,j))
  do j = i+1, n
    ... reduce column j by column i ...
  enddo
enddo
```

If the columns are smaller than a page, then it may be beneficial to reshape the array. This is easily specified by changing the keyword from **distribute** to **distribute_reshape**.

In addition to overcoming size constraints as shown above, the **distribute_reshape** directive is useful when the desired distribution is *contrary* to the layout of the array. For instance, suppose you want to distribute the *rows* of a two-dimensional matrix. In the following example, the **distribute_reshape** directive overcomes the storage layout constraints to provide the desired distribution.

```
real*8 A(n, n)
C Distribute rows in block fashion
c$distribute_reshape A (BLOCK, *)
real*8 sum(n)
c$distribute sum(BLOCK)

C Perform sum-reduction on the elements of each row
c$doacross local(j) affinity(i) = data(A(i,j))
do i = 1,n
  do j = 1,n
    sum(i) = sum(i) + A(i,j)
  enddo
enddo
```

Using Data Distribution and Data Affinity Scheduling

This example demonstrates regular data distribution and data affinity. This example, run on a 4-processor Origin2000 server, uses simple block scheduling. Processor 0 will calculate the results of the first 25,000 elements of A, processor 1 will calculate the second 25,000 elements of A, and so on. Arrays B and C are initialized using one processor; hence all of the memory pages are touched by the master processor (processor 0) and are placed in processor 0's local memory.

Using data distribution changes the placement of memory pages for arrays A, B, and C to match the data reference pattern. Thus, the code runs 33% faster on a 4-processor Origin2000 (than it would run using SMP directives without data distribution).

Without Data Distribution

```
real*8 a(1000000), b(1000000)
real*8 c(1000000)
integer i

c$par parallel shared(a, b, c) local(i)
c$par pdo
  do i = 1, 100000
    a(i) = b(i) + c(i)
  enddo
c$par end parallel
```

With Data Distribution

```
real*8 a(1000000), b(1000000)
real*8 c(1000000)
integer i
c$distribute a(block),b(block),c(block)

c$par parallel shared(a, b, c) local(i)
c$par pdo affinity( i ) = data( a(i) )
  do i = 1, 100000
    a(i) = b(i) + c(i)
  enddo
c$par end parallel
```

Parameter Passing

A distributed array can be passed as a parameter to a subroutine that has a matching declaration on the formal parameter:

```

real*8 A (m, n)
c$distributed_reshape A (block, *)
call foo (A, m, n)
end

subroutine foo (A, p, q)
real*8 A (p, q)
c$distributed_reshape A (block, *)
c$doacross affinity (i) = data (A(i, j))
  do i = 1, P
    enddo
end

```

Since the array is reshaped, it is *required* that the **distributed_reshape** directive in the caller and the callee match exactly. Furthermore, all calls to subroutine `foo()` must pass in an array with the exact same distribution.

If the array was only distributed (that is, not reshaped) in the above example, then the subroutine `foo()` can be called from different places with different incoming distributions. In this case, you can omit the distribution directive on the formal parameter, thereby ensuring that any data affinity within the loop is based on the distribution (at runtime) of the incoming actual parameter.

```

real*8 A(m, n), B (p, q)
real*8 A (block, *)
real*8 B (cyclic(1), *)
call foo (A, m, n)
call foo (B, p, q)
-----
subroutine foo (X, s, t)
real*8 X (s, t)

c$doacross affinity (i) = data (X(i+2, j))
do i = . . .
enddo

```

Redistributed Arrays

This example shows how an array is redistributed at runtime:

```
subroutine bar (X, n)
real*8 X(n, n)
...
c$redistribute X (*, cyclic(<expr>))
...
end
-----
subroutine foo
real*8 LocalArray (1000, 1000)
c$distribute LocalArray (*, BLOCK)
C the call to bar() may redistribute LocalArray
c$dynamic LocalArray
...
call bar (LocalArray, 100)
C The distribution for the following doacross
C is not known statically
c$doacross affinity (i) = data (A(i, j))
end
```

The next example illustrates a situation where the **c\$dynamic** directive can be optimized away. The main routine contains a local array **A** that is both distributed and dynamically redistributed. This array is passed as a parameter to `foo()` before being redistributed, and to `bar()` after being (possibly) redistributed. The incoming distribution for `foo()` is statically known; you can specify a **c\$distribute** directive on the formal parameter, thereby obtaining more efficient static scheduling for the affinity **doacross**. The subroutine `bar()`, however, can be called with multiple distributions, requiring run-time scheduling of the affinity **doacross**.

```
program main
c$distribute A (block, *)
c$dynamic A
call foo (A)
if (x.ne.17) then
  c$redistribute A (cyclic(x), *)
endif
call bar (A)
end
```

```

subroutine foo (A)
  C Incoming distribution is known to the user
  c$distributed A(block, *)
  c$doacross affinity (i) = data (A(i, j))
  ...
end

subroutine bar (A)
  C Incoming distribution is not known statically
  c$dynamic A
  c$doacross affinity (i) = data (A(i, j))
  ...
end

```

Irregular Distributions and Thread Affinity

The example below consists of a large array that is conceptually partitioned into unequal portions, one for each processor. This array is indexed through an index array `idx`, which stores the starting index value and the size of each processor's portion.

```

real*8 A(N)
  C idx ---> index array containing start index into A (idx(p, 0))
  C and size (idx(p, 1)) for each processor
real*4 idx (P, 2)
c$page_place (A(idx(0, 0)), idx(0, 1)*8, 0)
c$page_place (A(idx(1, 0)), idx(1, 1)*8, 1)
c$page_place (A(idx(2, 0)), idx(2, 1)*8, 2)
...
c$doacross affinity (i) = thread(i)
do i = 0, P-1
  ... process elements on processor i . . .
  ... A(idx(i, 0)) to A(idx(i,0)+idx(i,1)) . . .
enddo

```

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 5, “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)* reference 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” on page 152 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** did not have to be compiled with the `-mp` option (although it could be).

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)` reference 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 (`pc-sampling` can help show the system overhead). 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. Also, `timex(1)` indicates whether or not the parallelized versions performed better overall than the serial version.

The profile of a Fortran parallel job is different from a standard profile. As mentioned in “Analyzing Data Dependencies for Multiprocessing” on page 76, 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 a 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.

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)` and `perror(3F)` reference pages 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. The errors 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.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
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.
108	can't stat file The directory information for the file is not accessible.
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.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
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.
128	unknown system error An unexpected error was returned by IRIX.
129	requires seek ability The file is on a device that cannot do direct access.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
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.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
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 specify 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.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
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.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
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.
175	stat call failed in f77inode The directory information for the file is unreadable.
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.
186	Illegal input character in formatted read. The numeric input field in a formatted file contains non-blank characters beyond the maximum usable field width of 83 characters.

Table A-1 (continued) Run-Time Error Messages

Number	Message/Cause
187	Position specifier is allowed on sequential files only Cannot have a position specifier in a format statement for a non-sequential file.
188	Position specifier has an illegal value The position specifier has a value that does not make sense.
189	Out of memory The system is out of memory for allocation. Try to allocate more swap space.
195	Cannot keep a file opened as a scratch file. If a file is opened as a scratch file, it cannot be kept when closed, and is automatically deleted.

Index

Symbols

__mp_parallel_do, 151
__mp_slave_wait_for_work, 151
_DSM_FOP environment variable, 141
_DSM_MIGRATION environment variable, 141
_DSM_OFF environment variable, 141
_DSM_PPM environment variable, 141
_DSM_ROUND_ROBIN environment variable, 141
_DSM_VERBOSE environment variable, 141

A

ABI specification, 9
affinity scheduling, 129-132
-align16 compiler option, 25
-align8 compiler option, 25
alignment, 23, 24
ANSI Fortran
 data alignment, 24
ANSI-X3H5 standard, 67, 102
archiver, ar, 13
arrays
 2 gigabyte, 14
 and data affinity, 131
 data distribution directive, 128
 declaring, 23
 processor arrays, 137
 query dimensions, 136

arrays (*continued*)
 redistributed, 131
 reshaping, 134
 reshaping and error detection, 135
 restrictions on reshaping, 134
assembly language routines, 19
atomic BOOL operation, 118
atomic fetch-and-op operations, 117
atomic lock and unlock operations, 119
atomic op-and-fetch operations, 117
atomic synchronize operation, 119
-automatic compiler option, 150

B

barrier construct, 105, 114
barrier function, 98
-bestG compiler option, 12
blocking slave threads, 93
BOOL operation, 118

C

C\$, 75
C\$&, 74
cache, 88
 improve, 93
 misses, 122

- C\$CHUNK, 75
 - C compiler option, 153
 - c compiler option, 4
 - C\$COPYIN, 99
 - C\$DOACROSS, 68
 - and REDUCTION, 70
 - continuing with C\$&, 74
 - IF clause, 69
 - LASTLOCAL clause, 69
 - loop naming convention, 100
 - nesting, 76
 - c\$dynamic directive, 133
 - CHUNK, 72, 92, 98
 - C\$MP_SCHEDTYPE, 75
 - common block reorganization, 93
 - COMMON blocks, 70, 153
 - making local to a process, 98
 - shared, 24
 - compilation, 2
 - COMPILER_DEFAULTS_PATH environment variable, 9
 - compiler.defaults* file, 9
 - compiler options, 7
 - align16, 23, 25
 - align8, 23, 25
 - automatic, 150
 - bestG, 12
 - C, 153
 - c, 4
 - G, 12
 - jmopt, 12
 - l, 6
 - MP
 - check_reshape, 142
 - clone, 142
 - dsm, 142
 - mp, 101, 102, 149, 153
 - pfa, 102
 - static, 79, 150, 153
 - compile-time options
 - multiprocessing, 142
 - parallel programming, 142
 - COMPLEX, 23
 - COMPLEX*16, 23
 - COMPLEX*32, 23
 - computation scheduling
 - user control, 122
 - constructs
 - work-sharing, 105
 - core files, 19
 - producing, 155
 - C\$PAR & directive, 114
 - C\$PAR BARRIER, 114
 - C\$PAR CRITICAL SECTION, 112
 - C\$PAR PARALLEL, 104
 - C\$PAR PARALLEL DO, 105
 - C\$PAR PDO, 105
 - C\$PAR PSECTIONS, 107
 - C\$PAR SINGLE PROCESS, 109
 - critical section, 105
 - and SHARED, 113
 - PCF construct, 112
 - critical section construct, 102
 - differences between single process, 112
- ## D
- data
 - explicit placement, 140
 - placement in memory, 140
 - placement of, 140
 - data affinity, 130
 - formal parameter, 132
 - redistributed arrays, 131

- data dependencies, 78
 - analyzing for multiprocessing, 76
 - breaking, 82
 - complicated, 80
 - inconsequential, 81
 - rewritable, 80
- data distribution, 133-140
 - differences, 139
 - regular, 133
 - reshape, 134
- data distribution directives, 127-128
- data independence, 76
- data placement
 - user control, 122
- data structures
 - irregular, 140
- data types
 - alignment, 23, 24
- DATE, 60
- dbx, 155
- debugging
 - parallel Fortran programs, 152
- defaults
 - specification file, 9
- dimensions
 - arrays, 136
 - of arrays, 136
- direct files, 17
- directives
 - affinity scheduling, 129-132
 - and data affinity, 130
 - C\$, 75
 - C\$&, 74
 - C\$CHUNK, 75
- directives (*continued*)
 - C\$DOACROSS, 68
 - c\$dynamic, 133
 - C\$MP_SCHEDTYPE, 75
 - data distribution, 127-128
 - distribute, 127
 - distribute_reshape, 127, 137
 - doacross, 129
 - dynamic, 127
 - list of, 68
 - nested doacross, 129
 - ONTO clause, 132
 - overview, 66
 - redistribute, 133
 - regular data distribution, 133
 - reshape, 134
 - reshape data distribution, 134
 - see also* PCF directives
 - thread affinity, 132
- dis object file tool, 13
- distribute_reshape directive, 127, 137
- distribute directive, 127
- distributed shared memory, 141
- DOACROSS, 75
 - and multiprocessing, 100
- doacross directive, 129
- DO loops, 66, 77, 87, 154
- driver options, 7
- drivers, 1
 - defaults, 9
- dump object file tool, 13
- dynamic directive, 127
- dynamic scheduling, 71

E

- environment variable
 - COMPILER_DEFAULTS_PATH, 9
- environment variables, 151
 - CHUNK, 97
 - f77_dump_flag, 19, 155
 - MP_BLOCKTIME, 96
 - MP_SCHEDTYPE, 97
 - MP_SET_NUMTHREADS, 96
 - MP_SETUP, 96
 - parallel programming, 141
- equivalence statements, 153
- error detection
 - reshaped arrays, 135
- error handling, 19
- error messages
 - run-time, 155
- ERRSNS, 60
- examples
 - multiprocessing, 143
- executable object, 3
- EXIT, 61
- external files, 17

F

- f77
 - as driver, 1
 - supported file formats, 17
 - syntax, 1
- f77_dump_flag, 19, 155
- fetch-and-op operations, 117
- file, object file tool, 13

files

- compilation specification, 9
- direct, 17
- external, 17
- position when opened, 18
- preconnected, 18
- sequential unformatted, 17
- supported formats, 17
- UNKNOWN status, 19
- formal parameter
 - and data affinity, 132
- formats
 - files, 17
- Fortran
 - ANSI, 24
- functions
 - in parallel loops, 79
 - intrinsic, 62, 79
 - SECNDS, 63
 - library, 51, 79
 - RAN, 63
 - side effects, 79

G

- G compiler option, 12
- global data area
 - reducing, 12
- guided self-scheduling, 71

H

- handle_sigfpes, 20

I

IDATE, 60
 IF clause
 and C\$DOACROSS, 69
 IGCLD signal
 intercepting, 99
 interleave scheduling, 71
 interleaving, 92
 intrinsics, 116-120
 example, 120
 intrinsic subroutines
 DATE, 60
 ERRSNS, 60
 EXIT, 61
 IDATE, 60
 MVBITS, 62
 TIME, 61
 irregular data structures, 140
 ISA specification, 9

J

-jmpopt compiler option, 12

L

LASTLOCAL, 69, 77
 LASTLOCAL clause, 69
 -l compiler option, 6
 libfpe.a, 20
 libraries
 link, 6
 specifying, 6
 library
 multiprocessing library, *libmp*, 121
 library functions, 51

linking, 5
 link libraries, 6
 load balancing, 91
 LOCAL, 69, 77
 lock and unlock operations, 119
 lock example, 120
 LOGICAL, 23
 loop interchange, 87
 loops, 66
 data dependencies, 77
 transformation, 100

M

m_fork
 and multiprocessing, 100
 makefiles, 49
 master processes, 67, 101
 memory
 2 gigabyte arrays, 14
 array sizes, 14
 distributed shared, 141
 placement of data, 140
 misaligned data, 24
 -MP
 check_reshape compiler option, 142
 clone compiler option, 142
 dsm compiler option, 142
 MP
 compiler options, 142
 libmp library, 121
 mp_barrier, 98
 mp_block, 93
 mp_blocktime, 94
 MP_BLOCKTIME environment variable, 96
 mp_create, 94
 mp_destroy, 94

mp_my_threadnum, 95
mp_numthreads, 95
MP_SCHEDTYPE, 70, 75, 98
MP_SET_NUMTHREADS, 96
mp_set_numthreads, 95
 and MP_SET_NUMTHREADS, 96
mp_setlock, 98
MP_SETUP, 96
mp_setup, 94
mp_simple_sched
 and loop transformations, 100
 tasks executed, 101
mp_slave_control, 101
mp_suggested_numthreads, 95
MP_SUGNUMTHD environment variable, 141
mp_unblock, 93
mp_unsetlock, 98
-mp compiler option, 101, 102, 149, 153
multi-language programs, 3
multiprocessing
 and DOACROSS, 100
 and load balancing, 91
 associated overhead, 87
 control of, 121-147
 data distribution, 133-140
 enabling, 149
 enabling directives, 101
 environment variables, 141
 libmp library, 121
 options, 142
MVBITS, 62

N

nested doacross directive, 129
nm, object file tool, 13

NOWAIT clause, 106, 107, 109
NUM_THREADS, 96

O

object files, 3
 tools for interpreting, 13
object module, 3
objects
 linking, 5
ONTO clause, 132
op-and-fetch operations, 117
optimizing programs
 -*OPT* option
 reorg_common, 93
options
 control multiprocessing, 142
 parallel programming, 142
-*OPT* option
 reorg_common option, 93
Origin2000
 memory model, 122
 parallel programming, 121-147
 performance tuning, 121-147
 programming examples, 143

P

PAGESIZE environment variables, 141
parallel DO construct, 105
parallel Fortran
 directives, 68
parallel programming
 environment variables, 141
 examples, 143
 options, 142

- parallel programming on Origin2000, 121-147
 - parallel programs
 - improving performance, 122
 - tuning, 122
 - parallel region, 91, 102, 104
 - and SHARED, 104
 - efficiency of, 115
 - restrictions, 115
 - parallel sections construct, 107
 - assignment of processes, 108
 - parameter, formal
 - data affinity, 132
 - PCF constructs
 - and efficiency, 115
 - barrier, 105, 114
 - critical section, 105, 112
 - differences between single process and critical section, 112
 - NOWAIT, 106, 107, 109
 - parallel DO, 105
 - parallel regions, 104, 115
 - parallel sections, 107
 - PDO, 105
 - restrictions, 114
 - single process, 109
 - types of, 104
 - PCF directives
 - C\$PAR &, 114
 - C\$PAR BARRIER, 114
 - C\$PAR CRITICAL SECTION, 112
 - C\$PAR PARALLEL, 104
 - C\$PAR PARALLEL DO, 105
 - C\$PAR PDO, 105
 - C\$PAR PSECTIONS, 107
 - C\$PAR SINGLE PROCESS, 109
 - enabling, 102
 - overview, 102
 - PCF standard, 67
 - PDO construct, 105
 - performance
 - and cache behavior, 123
 - directives, 126-142
 - improving, 12, 122
 - performance tuning
 - examples, 143
 - pfa compiler option, 102
 - Power Fortran, 77
 - preconnected files, 18
 - processes
 - master, 67, 101
 - slave, 67, 101
 - processor
 - arrays, 137
 - topology
 - processor
 - ONTO clause, 132
 - processor specification, 9
 - prof
 - and parallel Fortran, 151
 - profiling
 - parallel Fortran program, 151
 - programs
 - multi-language, 3
- Q**
- quad-precision operations, 19
 - query intrinsics
 - distributed arrays, 136
- R**
- RAN, 63
 - rand
 - and multiprocessing, 79

- REAL*16
 - range, 22
 - REAL*4
 - range, 22
 - REAL*8
 - alignment, 23
 - range, 22
 - records, 17
 - recurrence
 - and data dependency, 84
 - redistribute directive, 133
 - reduction
 - and data dependency, 84
 - listing associated variables, 70
 - sum, 86
 - REDUCTION clause
 - and C\$DOACROSS, 70
 - regular data distribution
 - vs. reshaped distribution, 139
 - reorganize common blocks, 93
 - reshaped arrays, 134
 - error detection, 135
 - restrictions, 134
 - reshaped data distribution
 - vs. regular distribution, 139
 - reshape directive, 134
 - round-to-nearest mode, 19
 - run-time error handling, 19
 - run-time scheduling, 71
- S**
- scheduling methods, 70, 93, 100
 - dynamic, 71
 - guided self-scheduling, 71
 - interleave, 71
 - run-time, 71
 - simple, 71
 - SECNDS, 63
 - self-scheduling, 71
 - sequential unformatted files, 17
 - SHARE, 69, 77
 - SHARED
 - and critical section, 113
 - and parallel region, 104
 - shared memory
 - distributed, 141
 - SIGCLD, 94
 - simple scheduling, 71
 - single process
 - PCF construct, 109
 - single process construct, 109
 - differences between critical section, 112
 - size
 - arrays, 136
 - size, object file tool, 13
 - slave threads, 67, 101
 - blocking, 93, 94
 - source files, 3
 - specifying compilation mode, 9
 - spin-wait lock example, 120
 - spooled routines, 100
 - sproc
 - and multiprocessing, 99
 - associated processes, 101
 - static compiler option, 79, 150, 153
 - strip, object file tool, 13
 - subroutines
 - intrinsic, 79
 - system
 - DATE, 60
 - ERRSNS, 60
 - EXIT, 61
 - IDATE, 60
 - MVBITS, 62

sum reduction, example, 86
symbol table information
 producing, 13
synchronization intrinsics, 116-120
synchronize operation, 119
synchronizer, 151
syntax conventions, xvii
system interface, 51

T

test&test&set, 111
thread
 master, 67
 slave, 67
thread affinity
 doacross directive, 132
threads
 and processors, 67
 number of, 67
 override the default, 67
TIME, 61
trap handling, 20

U

ussetlock, 98
usunsetlock, 98

V

variables
 in parallel loops, 77
 local, 79
 to control multiprocessing, 141
VOLATILE
 and critical section, 113
 and multiple threads, 110

W

-Wl,Xlocal,data loader directive, 98
work quantum, 87
work-sharing constructs, 102
 restrictions, 114
 types of, 105

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2361-004.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
 - On the Internet: techpubs@sgi.com
 - For UUCP mail (through any backbone site): *[your_site]!sgi!techpubs*
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389