

MIPSpro™ Power Fortran 77 Programmer's Guide

Document Number 007-2363-002

CONTRIBUTORS

Written by Chris Hogue

Production by Linda Rae Sande

Engineering contributions by Bron Nelson, Bill Johnson, and Marty Itzkowitz

© 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 Unpublished rights are reserved under the Copyright Laws of the United States.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 and IRIS are registered trademarks, and IRIX, CHALLENGE, Crimson, Indigo², and the POWER Series are trademarks of Silicon Graphics, Inc. MIPS, R4000, and R8000 are registered trademarks, and MIPSpro, R5000, and R10000 are trademarks of MIPS Technologies, Inc. Cray is a trademark of Cray Research. VAST is a trademark of Pacific Sierra Research, Inc. VMS is a trademark of Digital Equipment Corporation.

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

Contents

	List of Examples	ix
	List of Figures	xi
	List of Tables	xiii
	Introduction	xv
	Organization of Information	xv
	Additional Reading	xvi
	Typographical Conventions	xvii
1.	Overview of Power Fortran	1
	Overview	1
	Strategy for Using Power Fortran	2
	Command-Line Options	3
	Directives	3
	Assertions	6
	Summary	7
2.	How to Use Power Fortran	9
	Overview of Compilation	9
	Compiling Programs With Power Fortran	10
3.	Utilizing Power Fortran Output	15
	Overview of Output Files	15
	Formatting the Listing File	17
	Paginating the Listing	17
	Specifying Information to Include	17
	Disabling Message Classes	18

- Interpreting Default Listing Information 19
 - Viewing the Listing File 19
 - Field Descriptions 19
- Sample Listing Files 23
 - Indirect Indexing 24
 - Function Call 26
 - Reductions 28
- 4. Customizing Power Fortran Execution 33**
 - Overview of Customization 33
 - Controlling Code Execution 34
 - Running Code in Parallel 34
 - Specifying a Work Threshold 34
 - Enabling Parallel I/O 35
 - Controlling Power Fortran Code Transformations 36
 - Specifying a Complexity Limit 36
 - Setting the Optimization Level 36
 - Controlling Variations in Round Off 37
 - Performing Inlining and Interprocedural Analysis 38
- 5. Scalar Optimizations 39**
 - Overview of Scalar Optimization 39
 - Performing General Optimizations 41
 - Enabling Loop Fusion 41
 - Controlling Global Assumptions 41
 - Setting Invariant IF Floating Limits 42
 - Setting the Optimization Level 44
 - Controlling Variations in Round Off 45
 - Controlling Scalar Optimizations 48
 - Using Vector Intrinsics 48

	Performing Advanced Optimizations	51
	Using Aggressive Optimization	52
	Controlling Internal Table Size	53
	Performing Memory Management Transformations	53
	Enabling Loop Unrolling	55
	Recognizing Directives	57
	Specifying Recursion	58
6.	Inlining and Interprocedural Analysis	59
	Overview of Inlining and IPA	59
	Using Command-Line Options	60
	Specifying Routines for Inlining or IPA	61
	Specifying Occurrences for Inlining and IPA	61
	Specifying Where to Search for Routines	64
	Creating Libraries	66
	Conditions That Prevent Inlining and IPA	67
7.	Fine-Tuning Power Fortran	69
	Overview of Directives and Assertions	69
	Directives	70
	Assertions	71
	Circumventing Power Fortran	73
	C\$ DOACROSS	73
	The C\$& Directive	73
	The C*\$ NO SYNC Assertion	74
	Fine-Tuning Scalar Optimizations	74
	Controlling Internal Table Size	74
	Setting Invariant IF Floating Limits	74
	Optimization Level	75
	Variations in Round Off	76
	Controlling Scalar Optimizations	77
	Enabling Loop Unrolling	77
	Fine-Tuning Inlining and IPA	78

- Running Code Serially 79
 - C*\$* ASSERT DO (SERIAL) 79
 - CDIR\$ NEXT SCALAR 80
 - C*\$* ASSERT DO PREFER (SERIAL) 80
- Running Code in Parallel 80
 - C*\$* [NO] CONCURRENTIZE 80
 - CVD\$ CONCUR 81
 - C*\$* ASSERT DO PREFER (CONCURRENT) 81
- Using Equivalenced Variables 82
- Using Assertions 82
- Using Aliasing 82
- Fine-Tuning Global Assumptions 83
 - C*\$* ASSERT [NO] BOUNDS VIOLATIONS 83
 - C*\$* ASSERT NO EQUIVALENCE HAZARD 84
 - C*\$* ASSERT [NO] TEMPORARIES FOR CONSTANT ARGUMENTS 85
- Ignoring Data Dependencies 86
 - C*\$* ASSERT DO (CONCURRENT) 86
 - CDIR\$ IVDEP 87
 - C*\$* ASSERT CONCURRENT CALL 87
 - CVD\$ CNCALL 87
 - C*\$* ASSERT NO RECURRENCE 87
 - C*\$* ASSERT PERMUTATION 88
- A. Power Fortran Command-Line Options 89**
 - Overview of Options 89
 - Options Summary 90
 - Overview 90
- B. Power Fortran Directives 95**
 - Standard Directives 95
 - Cray Directives 98
 - VAST Directives 98

C.	Power Fortran Assertions	99
	Glossary	103
	Index	107

List of Examples

Example 3-1	Indirect Indexing	25
Example 3-2	Concurrent Function Call	27
Example 3-3	Roundoff Reduction	29
Example 5-1	Controlling Roundoff	46
Example 5-2	Vector Ininsics	49
Example 5-3	Loop Unrolling	57
Example 7-1	Inline Control	78
Example 7-2	Serial Execution	79
Example 7-3	Bounds Violations	84
Example 7-4	Equivalence Hazard	85

List of Figures

Figure 2-1 Compiling With Power Fortran 14

List of Tables

Table 1-1	Power Fortran Assertions and Their Duration	6
Table 2-1	Power Fortran Command-Line Options	11
Table 3-1	Listing File Include Options	17
Table 3-2	Listing File Message Disabling Options	18
Table 3-3	Listing File DO Loop Delimiters	20
Table 3-4	Power Fortran Action Abbreviations	22
Table 3-5	Power Fortran Reductions	31
Table 5-1	Optimization Options	40
Table 5-2	Vector Intrinsic Function Names	51
Table 5-3	Recommended Cache Option Settings	54
Table 6-1	Inlining and IPA Options	60
Table 6-2	Inlining and IPA Search Command-Line Options	64
Table 6-3	Filename Extensions	65
Table 7-1	Directives Summary	71
Table 7-2	Assertions and Their Duration	72
Table A-1	concurrentize Option	90
Table A-2	limit Option	90
Table A-3	lines Option	91
Table A-4	listoptions Option	91
Table A-5	minconcurrent Option	92
Table A-6	noconcurrentize Option	92
Table A-7	noparallelio Option	93
Table A-8	parallelio Option	93
Table A-9	sopt Option	93
Table A-10	suppress Option	94

Introduction

This guide describes the features of MIPSpro™ Power Fortran 77. For details about analyzing a program and converting the output for use on a multiprocessor system, refer to Chapter 7, “Fortran Enhancements for Multiprocessors,” of the *MIPSpro Fortran 77 Programmer’s Guide*.

Organization of Information

This guide contains the following chapters and appendixes:

Chapter 1, “Overview of Power Fortran,” explains the basic mechanism for invoking Power Fortran and includes a description of Power Fortran’s output files.

Chapter 2, “How to Use Power Fortran,” explains how to use Power Fortran.

Chapter 3, “Utilizing Power Fortran Output,” explains output produced by Power Fortran: the transformed source file, listing file, and WorkShop Pro MPF input file.

Chapter 4, “Customizing Power Fortran Execution,” describes how to use command-line options to optimize Power Fortran execution.

Chapter 5, “Scalar Optimizations,” describes the scalar optimizations that you can enable from the command line.

Chapter 6, “Inlining and Interprocedural Analysis,” explains how to perform inlining and interprocedural analysis by specifying options to the compiler.

Chapter 7, “Fine-Tuning Power Fortran,” describes how to optimize code by using Power Fortran directives and assertions.

Appendix A, “Power Fortran Command-Line Options,” lists and describes the command-line options unique to Power Fortran.

Appendix B, “Power Fortran Directives,” lists the Power Fortran directives you can use to modify the features of Power Fortran, that is, directives to increase the optimization level, increase the size of the loop that Power Fortran can analyze, or use more sophisticated (and time-consuming) ways of resolving superficial data dependencies that prevent Power Fortran from identifying a loop for parallel execution.

Appendix C, “Power Fortran Assertions,” lists the Power Fortran assertions you can include in a program to provide information that Power Fortran needs to identify loops that can run in parallel, despite apparent but sometimes non-existent data dependencies.

The Glossary lists and defines terminology related to Power Fortran.

Additional Reading

Refer to the *MIPSpro Fortran 77 Programmer’s Guide* for information on the following topics:

- how to compile and link a Fortran program
- alignments, sizes, and variable ranges for the various data types
- the coding interface between Fortran programs and programs written in C
- file formats, run-time error handling, and other information related to the IRIX operating system
- operating system functions and subroutines callable by Fortran programs
- scalar optimizations that can be controlled through command-line options and compiler directives
- Fortran directives for multiprocessing
- run-time error messages

Refer to the *MIPSpro Fortran 77 Language Reference Manual* for a description of the Fortran language as implemented on Silicon Graphics® workstations and servers.

Refer to the *MIPSpro Compiling and Performance Tuning Guide* for information on:

- an overview of the MIPSpro compiler system and general compiler system command-line options
- optimizing program performance

- using the performance tools, *prof* and *pixie*, of the compiler system
- using dynamic shared objects (DSOs)
- using the debugger, *dbx*
- the dump utilities, archiver, and other tools for maintaining Fortran programs
- writing and updating code that is portable to 64-bit systems

Refer to the *Developer Magic: WorkShop Pro MPF User's Guide* for information about using WorkShop Pro MPF, a graphical tool to help you better understand the structure and parallelization of multiprocessing applications.

Refer to the *MIPSpro 64-Bit Porting and Transition Guide* for information on:

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

Typographical Conventions

This guide uses the following conventions and symbols:

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 and manual titles.
<code>Courier</code>	Indicates command syntax, program listings, computer output, and error messages.
Courier bold	Indicates user input.
[]	Enclose optional command arguments.
()	Surround arguments or are empty if the function has no arguments following function/subroutine names. Surround reference 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 is an example illustrating the syntax conventions.

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

The previous syntax statement indicates that:

- the keyword **C*\$* NO IPA** or **C*\$* IPA** must be written as shown
- you can specify one or more *name*, each separated by a comma and all between parentheses
- you must specify one of the following: **HERE**, **ROUTINE**, or **GLOBAL**

The following statements are valid examples of the described syntax:

```
C*$* IPA(ALPHA,BETA) HERE
```

```
C*$* NOIPA GLOBAL
```

Overview of Power Fortran

This chapter contains the following sections:

- “Overview” describes how Power Fortran operates and suggests procedures for using it.
- “Strategy for Using Power Fortran” explains when and how to use Power Fortran.
- “Command-Line Options” lists and describes the command-line options.
- “Directives” explains what a directive is and lists the supported directives.
- “Assertions” explains what an assertion is and lists the supported assertions.
- “Summary” is a short summary of the capabilities of Power Fortran.

Overview

MIPSpro Power Fortran 77 is a Fortran 77 compiler that enables you to run existing Fortran 77 programs efficiently on the Silicon Graphics POWER Series™ multiprocessor systems. Power Fortran analyzes a program, identifies loops that are safe to execute in parallel (concurrently), and generates a parallel version of the program.

The Silicon Graphics MIPSpro Fortran 77 compiler can generate code to split loop processing across all the available multiple processors. You do not need a multiprocessor system to develop under Power Fortran (although there is a slight performance loss when running multiprocessed code on a single-processor system). You can develop and test a Fortran 77 program using Power Fortran on any Silicon Graphics system (including single-processor systems) and then execute the program on a multiprocessor system. The executable code automatically adjusts itself to use all the processors available on the workstation at run time. However, simply passing code through Power Fortran rarely produces all the increased performance available. There are often easily removed data dependencies that prevent Power Fortran from running a loop in parallel. Using the listing file, optionally generated by Power Fortran, you can find the real or potential data dependencies that prevented Power Fortran from running a loop in parallel. Refer to Chapter 3, “Utilizing Power Fortran Output,” for details about the listing file.

If the data dependency is real, you can often remove the dependency by making a small change to the code. If the data dependency was apparent but not real, you can explicitly instruct Power Fortran to run the code in parallel by inserting Power Fortran assertions. These assertions look like Fortran 77 comments.

With Power Fortran, you select the code to convert to run in parallel. Thus, you can convert the whole program or key parts of it by adding Power Fortran directives manually or by having Power Fortran convert only selected files. Also, you can run Power Fortran on some, all, or none of a program's source files. The object files produced using Power Fortran are fully compatible with other object files. You can freely combine them with object files that you prepared manually for parallel execution and with object files that run only serially.

You can also use Power Fortran with WorkShop Pro MPF™, an optional product available from Silicon Graphics. It provides a graphical interface to the analysis performed by Power Fortran and allows you to understand and control your program to be run in parallel. WorkShop Pro MPF also works with the WorkShop/Performance Analyzer to help you concentrate on those parts of the program that are taking the longest to execute.

Strategy for Using Power Fortran

Use Power Fortran to identify which loops of a Fortran 77 program can be run safely in parallel. In some instances, Power Fortran alone makes a significant amount of the code run in parallel. However, for many programs simple code changes let Power Fortran automatically run more of the code in parallel.

Knowing when and where to modify your code means understanding the information in the Power Fortran listing. Understanding the Power Fortran listing will make it easy to recognize where small changes to the code can make big differences in how much code can run in parallel. Refer to Chapter 3, "Utilizing Power Fortran Output," for information. Alternatively, you can use WorkShop Pro MPF to understand the code.

Power Fortran analyzes a program for data dependence. During this analysis, Power Fortran looks for Fortran 77 **DO** loops in which each iteration of the loop is independent of all other iterations. If each iteration of the loop is self-contained, the system can execute the iterations in any order (or even simultaneously on separate processors) and produce the same result after running all iterations.

Power Fortran can safely run data-independent loops in parallel. When Power Fortran finds a loop that contains iterations that are dependent on other iterations, it cannot safely run the loop in parallel but can tell you what is causing the problem. If Power Fortran cannot run a loop in parallel, the listing file explains where Power Fortran encountered problems.

Command-Line Options

To customize the way Power Fortran executes an entire program, you can specify various command-line options when you run Power Fortran (explained in Chapter 2, “How to Use Power Fortran.”) The six functional categories of command-line options are

- parallelization
- general optimization
- inlining and interprocedural analysis
- directive control
- listing
- advanced optimization

Many of these options are also recognized by the MIPSpro Fortran 77 compiler. This book describes only the options that are unique to Power Fortran. Chapter 4, “Customizing Power Fortran Execution,” explains when and how to use the various Power Fortran options, and Appendix A, “Power Fortran Command-Line Options,” provides a complete summary.

Directives

Power Fortran directives enable, disable, or modify a feature of Power Fortran. 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 the directive on or set a desired value for its level.

Power Fortran directives allow you to specify Power Fortran options in addition to, or instead of, command-line options. Directives placed on the first line of the input file are called *global directives*. Power Fortran interprets them as if they appear 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. Power Fortran 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 Power Fortran directives have corresponding command-line options. If you specify conflicting settings in the command line and a directive, Power Fortran 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, Power Fortran uses the minimum of the command-line setting and the directive setting.

Power Fortran supports the following standard directives:

<code>C** ARCLIMIT(<i>n</i>)*</code>	<code>C** NO IPA*</code>
<code>C** [NO] ASSERTIONS*</code>	<code>C** OPTIMIZE(<i>n</i>)*</code>
<code>C** CONCURRENTIZE</code>	<code>C** ROUNDOFF(<i>n</i>)*</code>
<code>C** EACH_INVARIANT_IF_GROWTH(<i>n</i>)*</code>	<code>C** SCALAR OPTIMIZE(<i>n</i>)*</code>
<code>C** INLINE*</code>	<code>C** UNROLL(<i>n</i>,<i>m</i>)*</code>
<code>C** IPA*</code>	<code>C\$*</code>
<code>C** LIMIT(<i>n</i>)</code>	<code>C\$DOACROSS</code>
<code>C** MAX_INVARIANT_IF_GROWTH(<i>n</i>)*</code>	<code>C\$&</code>
<code>C** MINCONCURRENT(<i>n</i>)*</code>	<code>C\$CHUNK*</code>
<code>C** NO CONCURRENTIZE</code>	<code>C\$COPYIN*</code>
<code>C** NO INLINE*</code>	<code>C\$MP_SCHEDTYPE*</code>

Note: The * denotes that the directive is also supported by the MIPSpro Fortran 77 compiler and therefore, described in the *MIPSpro Fortran 77 Programmer's Guide*.

In addition to the simple loop-level parallelism offered by the `C$DOACROSS` directive, Power Fortran 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.

Power Fortran supports the following PCF directives, which are described in the *MIPSpro Fortran 77 Programmer's Guide*:

- **C\$PAR BARRIER**
- **C\$PAR [END] CRITICAL SECTION**
- **C\$PAR [END] PARALLEL**
- **C\$PAR PARALLEL DO**
- **C\$PAR [END] PDO**
- **C\$PAR [END] PSECTION[S]**
- **C\$PAR SECTION**
- **C\$PAR [END] SINGLE PROCESS**
- **C\$PAR &**

Power Fortran supports the Cray™ directives listed below, which it maps to corresponding Power Fortran assertions. Refer to Chapter 7, “Fine-Tuning Power Fortran,” for details.

- **CDIR\$ NEXT SCALAR**
- **CDIR\$ NO RECURRENCE***
- **CDIR\$ IVDEP**

Power Fortran supports the following VAST™ directives, which it maps to corresponding Power Fortran assertions:

- **CVDS\$ CNCALL**
- **CVDS\$ CONCUR**
- **CVDS\$ [NO] DEPCHK***
- **CVDS\$ [NO] LSTVAL***

As with the command-line options, many directives are also recognized by the MIPSpro Fortran 77 compiler. This manual describes those directives that are supported only by Power Fortran. Refer to Appendix B, “Power Fortran Directives,” for a summary.

Assertions

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

As with a directive, Power Fortran treats an assertion as a global assertion if it comes before all comments and statements in the file. That is, Power Fortran treats the assertion as if it were repeated at the top of each program unit in the file. However, Power Fortran does not check the correctness of assertions.

Many assertions, like directives, are active until the end of the program unit (or file) or until you reset them. Other assertions are valid only for the **DO** loop before which they appear (such as **C*\$* ASSERT DO PREFER (CONCURRENT)**). This type of assertion applies to the next **DO** loop but not to any loop nested inside it.

Table 1-1 lists the accepted Power Fortran assertions and their longevity.

Table 1-1 Power Fortran Assertions and Their Duration

Assertion	Duration
C*\$* ASSERT [NO] ARGUMENT ALIASING ^a	Until reset
C*\$* ASSERT [NO] BOUNDS VIOLATIONS ^a	Until reset
C*\$* ASSERT CONCURRENT CALL	Next loop
C*\$* ASSERT DO (CONCURRENT)	Next loop
C*\$* ASSERT DO (SERIAL)	Next loop
C*\$* ASSERT DO PREFER (CONCURRENT)	Next loop
C*\$* ASSERT DO PREFER (SERIAL)	Next loop
C*\$* ASSERT [NO] EQUIVALENCE HAZARD ^a	Until reset
C*\$* ASSERT [NO] LAST VALUE NEEDED	Until reset
C*\$* ASSERT NO RECURRENCE	Next loop
C*\$* ASSERT NO SYNC	Next loop
C*\$* ASSERT RELATION (<i>name.xx. name</i>)	Next loop

Table 1-1 (continued) Power Fortran Assertions and Their Duration

Assertion	Duration
C*\$* ASSERT PERMUTATION (<i>name</i>) ^a	Next loop
C*\$* ASSERT [NO] TEMPORARIES FOR CONSTANT ARGUMENTS ^a	Next loop

a. The *MIPSpro Fortran 77 Programmer's Guide* describes this assertion.

As with the command-line options and directives, many assertions are also recognized by the MIPSpro Fortran 77 compiler. This manual describes those assertions that are supported only by Power Fortran.

Summary

Power Fortran provides information about the dependencies of loops in a Fortran 77 program. Often, Power Fortran can use this information to automatically run loops in parallel. But when Power Fortran is not able to convert the code for parallel execution automatically, it can tell you where it ran into problems. Often, you need only make a small change to remove the dependencies that prevent the loop from running in parallel. The better you understand the information Power Fortran gives you, the better equipped you will be to transform the program into an efficient parallel version.

For more information about parallel processing in general, see Chapter 7 in the *MIPSpro Fortran 77 Programmer's Guide*. Especially recommended are the sections "Analyzing Data Dependencies for Multiprocessing" and "Breaking Data Dependencies" for information about recognizing and repairing data dependency problems.

How to Use Power Fortran

This chapter contains the following sections:

- “Overview of Compilation” describes how to prepare for using Power Fortran.
- “Compiling Programs With Power Fortran” explains how to run Power Fortran.

Overview of Compilation

Simply running a program through Power Fortran might improve the performance of your program, but you can improve it far more if you understand the Power Fortran listing. From the listing, you can often identify small problems that prevent a loop from running safely in parallel. With a relatively small amount of work, you can remove these data dependencies and dramatically improve the program’s performance.

When trying to find loops to run in parallel, focus your efforts on the areas of the code that use the bulk of the run time. Spending time trying to run a routine in parallel that uses only one percent of the run time of the program cannot significantly improve the performance of your program.

To determine where your code spends its time, take an execution profile of the program. Use either pc sampling, through the `-p` option to `f77(1)`, or basic block profiling, through `pixie(1)`. Refer to the *MIPSpro Compiling and Performance Tuning Guide* for details about profiling. Alternatively, you can use the WorkShop Pro MPF Parallel Analyzer View to examine the performance of your program. Refer to the *Developer Magic: WorkShop Pro MPF User’s Guide* for details.

There are two schools of thought about profiling: conservative and optimistic. The conservative approach takes a profile of the original (nonparallel) job. You then run in parallel only the loops that account for most of the run time. The more optimistic approach runs the entire program through Power Fortran and then profiles the resulting multiprocessed job. The conservative approach reduces the chances that something might go wrong because it makes fewer changes to the code. It also focuses on the smallest number of lines of code that have the greatest effect.

Use the optimistic approach when you think that Power Fortran will do a good job with the existing program. You will save time by letting Power Fortran do what it can. You can then focus on those routines where Power Fortran had a problem. One situation in which Power Fortran frequently does a good job is when you convert programs that already run well on traditional vector architectures. Many such programs run in parallel without additional effort.

Whichever approach you choose, use the profile to focus your efforts on the most time-consuming routines. Once you find a time-consuming routine, submit that routine alone to Power Fortran. If the routine is in the middle of a large file, consider using *fsplit(1)* to isolate the individual routine. Compile the routine with the **-pfa keep** option and examine the listing file. The Power Fortran listing identifies the loops that Power Fortran can and cannot run in parallel. For loops that cannot run in parallel, the listing also tells you why Power Fortran could not convert the loop for parallel execution.

Compiling Programs With Power Fortran

This section describes the command-line syntax for compiling a Fortran 77 program with Power Fortran. You can pass these options to Power Fortran by adding the **-pfa** option to the *f77* command line. It invokes the various processing phases that compile, optimize, assemble, and link the program. For more information about the **-pfa** option, see the *f77(1)* reference page.

Syntax

```
f77 options -pfa[{list|keep}] [-pfa,option[=value] [,option[=value]]...] filename
```

where

<i>options</i>	Specifies any <i>f77</i> compiler options. Refer to the <i>f77(1)</i> reference page and the <i>MIPSpro Fortran 77 Programmer's Guide</i> for details.
-pfa	Requests automatic parallelization of loops, and enables any multiprocessing directives.
list	Specifies an annotated listing of the parts of the program that can (and cannot) run in parallel on multiple processors. The listing file has the suffix .l .
keep	Generates the listing file (.l), saves the transformed equivalent Fortran 77 program (.m), and creates an output file for use with WorkShop Pro MPF (.anl).

-pfa	Passes the specified command-line options to Power Fortran. Do not enter spaces between -pfa and any of the hyphens, <i>options</i> , equal signs, and <i>values</i> that follow it.
<i>option</i>	Specifies a Power Fortran command-line option listed in Table 2-1, for example, -concurrentize .
<i>value</i>	Specifies a value for a command-line option, for example, 1.
<i>filename</i>	Specifies the Fortran 77 source program. The filename must always use the .f , .F , .for , or .FOR suffix.

Table 2-1 lists the Power Fortran command-line options. Although the table lists the options in lowercase, you can specify them in uppercase as well.

Note: You can replace many of the Power Fortran command-line options listed in Table 2-1 with in-code directives. For information on these directives, see Chapter 7, “Fine-Tuning Power Fortran,” and Appendix B, “Power Fortran Directives.”

Table 2-1 Power Fortran Command-Line Options

Reference	Long Name	Short Name	Default Value
Parallelization	-[no]concurrentize	-[n]conc	-concurrentize
	-minconcurrent= <i>n</i>	-mc= <i>n</i>	-minconcurrent=500
	-[no]parallelio	-[no]pio	(option off)
General Optimization ^a	-assume= <i>list</i>	-as= <i>list</i>	-assume=el
	-fuse	-fuse	-fuse
	-optimize= <i>n</i>	-o= <i>n</i>	depends on -O <i>n</i>
	-roundoff= <i>n</i>	-r= <i>n</i>	depends on -O <i>n</i>
	-scaleropt= <i>n</i>	-so= <i>n</i>	depends on -O <i>n</i>
Directive Control ^a	-[no]directives= <i>list</i>	-[n]dr= <i>list</i>	-directives=ackpv

Table 2-1 (continued) Power Fortran Command-Line Options

Reference	Long Name	Short Name	Default Value
Inlining and Interprocedural Analysis ^a	-inline[= <i>list</i>]	-in[= <i>list</i>]	(option off)
	-ipa[= <i>list</i>]	-ipa[= <i>list</i>]	(option off)
	-inline_create= <i>name</i>	-incr= <i>name</i>	(option off)
	-ipa_create= <i>name</i>	-ipacr= <i>name</i>	(option off)
	-inline_from_files= <i>list</i>	-inff= <i>list</i>	(option off)
	-ipa_from_files= <i>list</i>	-ipaff= <i>list</i>	(option off)
	-inline_from_libraries= <i>list</i>	-infl= <i>list</i>	(option off)
	-ipa_from_libraries= <i>list</i>	-ipafl= <i>list</i>	(option off)
	-inline_loop_level= <i>n</i>	-inll= <i>n</i>	-inll=10
	-ipa_loop_level= <i>n</i>	-ipall= <i>n</i>	-ipall=10
	-inline_man	-inm	(option off)
	-ipa_man	-ipam	(option off)
	-inline_depth	-ind	-ind=10
Listing	-lines= <i>n</i>	-ln= <i>n</i>	-lines=55
	-listoptions= <i>list</i>	-lo= <i>list</i>	-listoptions=k
	-suppress= <i>list</i>	-su= <i>list</i>	(option off)
Advanced Optimization	-aggressive= <i>letter</i> ^a	-ag= <i>letter</i>	(option off)
	-arclimit= <i>n</i> ^a	-arclm= <i>n</i>	-arclimit=5000
	-cacheline= <i>n</i> ^a	-chl= <i>n</i>	-cacheline=4
	-cachesize= <i>n</i> ^a	-chs= <i>n</i>	-cachesize=256
	-chunk= <i>n</i> ^a	-chk= <i>n</i>	-chunk=1
	-dpreregisters= <i>n</i> ^a	-dpr= <i>n</i>	-dpreregisters=16
	-each_invariant_if_growth= <i>n</i> ^a	-eiifg= <i>n</i>	-each_invariant_if_growth=20
	-fpreregisters= <i>n</i> ^a	-fpr= <i>n</i>	-fpreregisters=16
	-limit= <i>n</i>	-lm= <i>n</i>	-limit=20000
	-max_invariant_if_growth= <i>n</i> ^a	-miifg= <i>n</i>	-max_invariant_if_growth=500
	-[no]recursion ^a	-[no]rc	-recursion
	-setassociativity= <i>n</i> ^a	sasc= <i>n</i>	-setassociativity=1
	-unroll= <i>n</i> ^a	-ur= <i>n</i>	-unroll=4
	-unroll2= <i>n</i> ^a	-ur2= <i>n</i>	-unroll2=100

a. Refer to the *MIPSpro Fortran 77 Programmer's Guide* for details about this option.

The **-pfa** option enables Power Fortran, which performs automatic parallelization plus **-pfa,-r=0,-so=3,-o=5**. This option also enables the multiprocessing directives that you can enable separately with the **-mp** option.

If you specify the **-On** option along with **-pfa**, the compiler performs the greater of the implied options. For example, specifying **-O1** (which is the same as **-pfa,-r=0,-so=2,-o=1**) and **-pfa** (which is the same as **-pfa,-r=0,-so=3,-o=5**) has the same effect as **-pfa,-r=0,-so=3,-o=5**.

Example

To compile the Fortran 77 program **prog.f** with Power Fortran and the **-minconcurrent=0** and **-parallelio** options, enter

```
% f77 -pfa -pfa,-minconcurrent=0,-parallelio prog.f
```

Figure 2-1 shows what happens when you compile a Fortran 77 program with **-pfa**. The first pass invokes the macro preprocessor *cpp* to handle *cpp* directives. (For more information, see the *cpp(1)* manual page.) The Power Fortran 77 analyzer, *pfa*, then takes the *cpp* output and inserts code that runs data-independent loops in parallel. This modified source code is then taken by the MIPSpro Fortran compiler, which generates intermediate code.

In addition to the intermediate code, the Power Fortran analyzer can generate the following files:

- listing file (**.l**)
- equivalent transformed file (**.m**)
- file for use with WorkShop Pro MPF (**.anl**)

For details and an example of the listing file, refer to Chapter 3, “Utilizing Power Fortran Output.”

Finally, the MIPSpro back end, *be*, processes the intermediate code to produce an object file (**.o**).

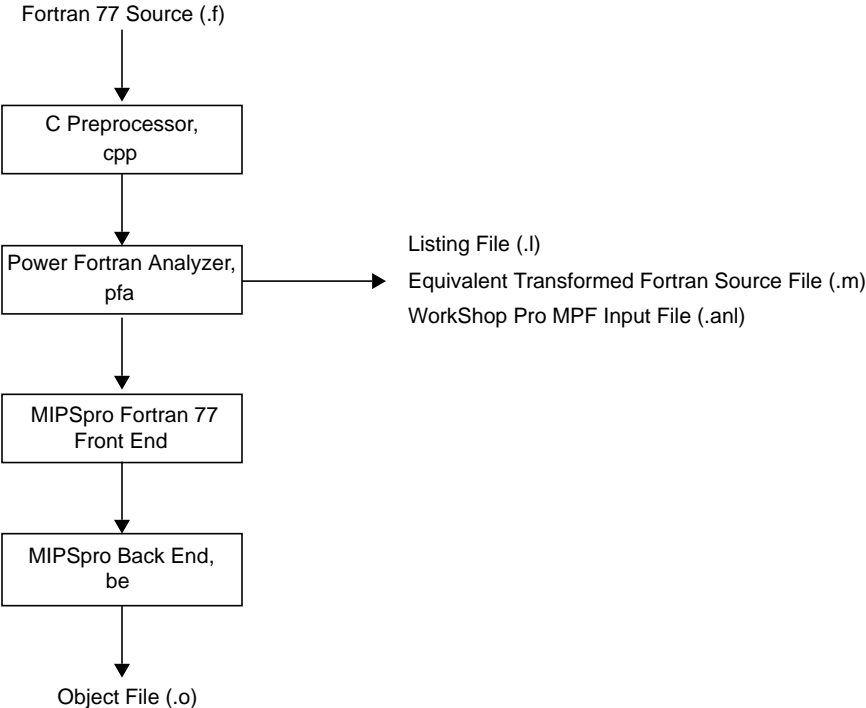


Figure 2-1 Compiling With Power Fortran

Utilizing Power Fortran Output

This chapter contains the following sections:

- “Overview of Output Files” discusses the Power Fortran output files and provides examples of them.
- “Formatting the Listing File” explains how to change the format of the standard listing file.
- “Interpreting Default Listing Information” explains the contents of the listing file.
- “Sample Listing Files” provides code examples along with an interpretation of each.

Overview of Output Files

Power Fortran can generate three types of output files:

- listing file (.l)
- transformed Fortran source file (.m) that contains the original source program with the multiprocessing directives inserted by Power Fortran
- an input file for use with WorkShop Pro MPF (.anl)

When you specify the **list** argument to **-pfa**, Power Fortran produces a line-numbered listing file. If you specify the **keep** argument instead, Power Fortran produces the numbered listing file, transformed Fortran source file, and the WorkShop Pro MPF file. (For details about invoking Power Fortran, refer to Chapter 2, “How to Use Power Fortran.”)

For example, consider the following code segment, **sample.f**:

```

subroutine sample (a,b,c)
dimension a(1000),b(1000),c(1000)
do 10 i = 1, 1000
10   a(i) = b(i) + c(i)
end
    
```

Compiling **sample.f** as follows

```
% f77 -pfa list -c sample.f
```

generates the following listing file, **sample.l**:

```

Footnotes Actions  Do Loops Line
          DIR                1 # 1 "sample.f"
          2      subroutine sample(a,b,c)
          3      dimension a(1000),b(1000),c(1000)
          SO C      +----- 4      do 10 i = 1,1000
          SO      *_____ 5 10   a (i) = b(i) + c(i)
          6      end

Abbreviations Used
SO      scalar optimization
DIR     directive
C       concurrentized

Loop Summary
      From To  Loop  Loop  at
Loop# line line label index nest Status
1      4   5   Do 10  I     1   concurrentized
    
```

Power Fortran placed a **C** before the first statement of the **DO** loop in the listing file, **sample.l**. The Abbreviations Used table shows that **C** stands for “concurrentized,” which means that Power Fortran determined that it can safely run the loop in parallel. The Loop Summary table at the bottom of **sample.l** shows that the status of the loop is concurrentized.

Note: The first line number directive appears in the listing because it was actually added by *cpp* before Power Fortran ran.

Formatting the Listing File

You can customize a Power Fortran listing file by

- paginating the listing
- selecting the information to be printed
- disabling specific message classes

Paginating the Listing

The `-lines=n` option (or `-ln=n`) paginates the listing for printing. Use this option to change the number of lines per page. Specifying `-lines=0` paginates at subroutine boundaries.

If you do not specify the `-lines` option, Power Fortran prints 55 lines per page.

Specifying Information to Include

The `-listoptions=list` option (or `-lo=list`) specifies the information to include in the listing file (`.l`), where *list* is any combination of the options in Table 3-1. The default is `-listoptions=ol`.

Table 3-1 Listing File Include Options

Value	Produces
c	Calling tree at the end of the program listing.
i	Transformed program file annotated with line numbers in the source program. Error messages and debugging information can refer to the original source rather than the transformed source. This argument is specified by default.
k	List of the Power Fortran options used at the end of each program unit.
l	Loop-by-loop optimization table.
n	Program unit names, as processed, to the standard error file. This option is added automatically as part of an <code>f77 -v</code> compilation.

Table 3-1 Listing File Include Options

Value	Produces
o	Annotated listing of the original program.
p	Processing performance statistics.
s	Summary of optimizations performed.
t	Annotated listing of the transformed program.

The following command compiles the program *source.f* with Power Fortran and includes an annotated listing of the original program and a summary of the optimizations performed in the listing file:

```
% f77 -pfa -pfa,-listoptions=ls source.f
```

Disabling Message Classes

Use the `-suppress=list` option (or `-su=list`) to disable individual classes of Power Fortran messages that are normally included in the listing (.l) file. These messages range from syntax warnings and error messages to messages about the optimizations performed. *list* is any combination of the options in Table 3-2.

Table 3-2 Listing File Message Disabling Options

Value	Message Class Disabled
d	Data dependence
e	Syntax error
i	Information
n	Unable to run loop in parallel
q	Questions
s	Standard messages
w	Warning of syntax error (Power Fortran adds the <code>-suppress=w</code> option automatically if you use the <code>-w</code> option to <i>f77</i>)

If you do not specify this option, Power Fortran prints messages of all classes.

Interpreting Default Listing Information

Knowing when and where to modify your code means understanding the information in the Power Fortran listing. This understanding allows you to recognize where small changes to the source code will make a big difference in how much code is run in parallel. The listing file generated by Power Fortran lists the optimizations Power Fortran made to the code. For example, a message could say that, although three loops could have run in parallel, Power Fortran converted only the one it determined most profitable.

This section explains how to view the listing file online and then lists and describes the various fields.

Viewing the Listing File

The listing file is in 132-column format. To view the file, open a window with 132 columns and 40 rows by entering

```
% wsh -s132,40
```

Field Descriptions

This section explains the contents of the listing file when you use the default values for the `-listoptions` command-line option (that is, `o` and `l`).

A default Power Fortran listing file includes

- line numbers
- **DO** loop markings
- footnotes
- syntax errors/warning messages
- action summary

Line Numbers

In the listing transformed by Power Fortran, a statement labeled with a line number, such as 21, is the same as line 21 from the original program or has been derived from that line. These line numbers are useful when inspecting the transformed program listing and

when debugging. Power Fortran sometimes generates several lines of code from a single line of the original program; in this case, each new line of code is labeled with the same number as the line of the original program from which it was generated. Consequently, many lines of the transformed program listing carry the same number because they are related to one line of the original program listing.

DO Loop Marking

The listing file displays **DO** loops graphically in a column headed **DO Loops**. Power Fortran surrounds each **DO** loop (up to nest level 10) with a loop delimiter character. The delimiters form brackets around each loop nest level. Each character listed in Table 3-3 has a specific meaning.

Table 3-3 Listing File DO Loop Delimiters

Character	Denotes
	Generic DO loop
*	Power Fortran can run loop in parallel
!	Syntax error

A statement contained within *n* **DO** loops has *n* of these loop delimiters on that line. For example, the following statements are contained within one **DO** loop and therefore have only one |:

```

DO Loops  Line
+-----  173      DO 100 M=2,MAX(MFLD,2)
|          174      IADR = ISECT(M)
|          175      IADR1= ISECT(M-1)
|          176      PNM(IADR)=(ANM(IADR) *PNM(IADR1))
|_____  177 100  PPNM(IADR)= -(ANM(IADR) *PNM(IADR1))
    
```

Footnotes

Power Fortran uses the footnotes listing to give important details concerning its actions. Power Fortran numbers and prints the footnotes at the bottom of each program unit under the Footnote List heading. References to the footnotes are displayed in the listing under the Footnotes column. For example, this footnote

```

13      DD          1790      IF (B(I) .LE. 6) IB(J*I) = I+J
    
```

appears under Footnote List at the end of the program unit

```
13: data dependence    Data dependence involving this line due
                       to variable IB.
```

In this example, **13** is the footnote number, **DD** (data dependence) is the explanation for the Power Fortran action, and the **IF** statement on line 1790 refers to the original source line number.

Syntax Errors/Warning Messages

When a program has syntax errors, the listing file describes the error next to the lines that start with the symbol **###** in the Footnotes column. These messages are also printed to **stderr**, which is usually your terminal.

For example:

```
Footnotes Actions   DO Loops   Line
                                     1      SUBROUTINE Z(A,B,N)
                                     2      REAL A(N), B(N)
+-----
!                                     3      DO 20 I=1,N
!                                     4      X=A(I)
!                                     5      Y=B(I)
! _____ 6      20 C(I)=X+Y

### line (6)
### error   Array not declared or statement function declared
            after executable statements.
### error   A do loop ends on a non-executable statement.
            7      PRINT *,X
            8      END
```

Action Summary

When Power Fortran translates or modifies a statement, it uses abbreviations in the Actions column of the listing file to identify the statements. Power Fortran lists an abbreviated explanation of its actions at the bottom of the listing. For the **DIR** and **V** classes, the class itself serves as the message without detailed messages. All other classes have associated messages.

Table 3-4 lists and explains the values that can appear in the Actions column.

Table 3-4 Power Fortran Action Abbreviations

Value	Meaning
DD	(Data Dependence) Indicates that data dependence prevented Power Fortran from running this statement in parallel.
DIR	(Directive) Used in conjunction with the footnotes and concerns compiler directives. If you code a compiler directive and that line does not have the DIR abbreviation in the listing, Power Fortran will not recognize the directive. Check the setting of the -directives command-line option and the syntax of the directive.
E	(Error) Indicates syntax errors. These messages can refer to missing or extra characters, illegal keywords, or text placed in the wrong column. Power Fortran cannot do anything with such code. The equivalent transformed source file (.m) contains a copy of this program unit that Power Fortran has not modified.
EX	(Extension) Shows where a construct in the original program is not allowed in the language Power Fortran produces. In some cases, an operation or type is allowed in the input language but not in the output language.
INF	(Information) Provides noncritical information.
I	(Insertion) Indicates that Power Fortran added a statement.
LR	(Loop Reordering) Indicates that Power Fortran has modified a Fortran 77 statement in the process of interchanging loops. If during optimization Power Fortran ascertains that an outer loop would be more efficient as an inner loop, and it can legally reorder the loops, Power Fortran places the outer loop inside. In the process of this reordering, Power Fortran might have to change loop bounds (for triangular loops), distribute loops, or float IF assignments. Only the statements modified for the exchange are marked.
MIS	(Miscellaneous) Indicates that some Power Fortran information has been lost. This message does not always mean that something is wrong with the program.
NX	(Nonconcurrent Statement) Indicates that Power Fortran did not try or was unable to run the statement in parallel. For example, when a subroutine call is involved in a loop, Power Fortran generates this message.

Table 3-4 (continued) Power Fortran Action Abbreviations

Value	Meaning
NO	(Program Too Large—Not Optimized) Indicates that the program unit being processed is too large for Power Fortran to optimize, because of Power Fortran data structure size limitations. When Power Fortran optimizes programs, it adds statements that might also overflow the fixed-size tables. In either case, Power Fortran stops optimization and passes the original program to the equivalent transformed source file (<i>.m</i>), informing you of this action. For Power Fortran to process the unit, you must split the program into smaller sections.
OE	(Option Error) Indicates a syntax error in a Power Fortran option. This error does not stop the processing of a program unit.
OTF	(Output Translation Failure) Marks statements that have constructs that exist in the input language but that cannot be represented in the output language.
Q	(Question) Indicates that Power Fortran tried to optimize a loop nest but discovered a data dependence it could not break at compile time without further information. You can usually answer this question with an appropriate assertion.
SO	(Scalar Optimization) Marks places in the transformed listing where Power Fortran has optimized a scalar loop.
STD	(Standardized) Marks where Power Fortran changed a program to improve the chance of finding code that it can optimize. This is often a conversion from an IF/GOTO into a block IF , loop rerolling, and conversion of an IF loop into a DO loop.
TE	(Translator Error) Indicates an internal Power Fortran error. Power Fortran writes the notification to the standard error file and writes a trace back to the output file. Notify Silicon Graphics if you see this sort of bug (so it can be corrected) and, if possible, send Silicon Graphics the code that caused the trace back as well as the trace back itself. If you can reproduce the error in a small program unit, send that small program unit as well.
W	(Warning) Contains syntax warnings.

Sample Listing Files

This section contains a few simple examples of Fortran code and the corresponding Power Fortran output. An actual source program would be much larger, and a single loop could contain several of the cases illustrated here. However, even in a large loop, you can deal with each problem individually.

Indirect Indexing

Power Fortran cannot determine if it can run a loop in parallel when the code uses indirect indexing. A loop is indirectly indexed when it uses the value from some auxiliary array as the index value rather than the **DO** loop variable. The code

```

subroutine foo2(a,b,index,n)
real a(n), b(n)
integer index(n)
c
do i = 1, n
  a(index(i)) = a(index(i)) + b(i)
enddo
end

```

when submitted to Power Fortran, produces this listing file:

```

Footnotes  Actions      DO Loops  Line
          DIR                1 # 1 "foo2.f"
          1  subroutine foo2(a,b,index,n)
          2  real a(n), b(n)
          3  integer index(n)
          4
1         Q SO          +----- 5  do i = 1, n
2         DD SO          !           6    a(index(i)) = a(index(i)) + b(i)
          !_____ 7  enddo
          8  end

```

Abbreviations Used
 DD data dependence
 Q question
 SO scalar optimization
 DIR directive

Footnote List
 1: question Is "INDEX" a permutation vector?
 2: data dependence Data dependence involving this line due to variable A.

Loop Summary

loop#	From line	To line	Loop label	Loop index	at nest	Status
1	6	8	Do	I	1	scalar mode preferable

DD in the Actions column on line 6 of the listing warns that the variable **a** might carry a dependency. A dependency exists when one iteration of the loop writes to a location that is used by a different iteration of the loop. In this example, if the values of **index(i)** are

ever the same for different values of **i**, then different iterations might use the same location in **a**. Therefore, this code contains a possible data dependence.

If you can guarantee that the values of **index(i)** are always different for each value of **i**, then there is no dependence (each iteration uses a different location in **a**). Question one on the Footnote List asks if **index(i)** is different for every value of **i**. A permutation vector is a list of numbers, each of which is different from the others. If you know that **index** is a permutation vector, then the loop is data-independent. An example of a permutation vector is a list of objects in which each object appears exactly once.

Explicitly state that **index** is a permutation vector by adding an assertion in the source.

Example 3-1 Indirect Indexing

```
subroutine foo2(a,b,index,n)
  real a(n), b(n)
  integer index(n)
c*$*assert permutation (index)
  do i = 1, n
    a(index(i)) = a(index(i)) + b(i)
  enddo
end
```

Now the listing file shows that Power Fortran finds the loop safe to run in parallel (indicated by the * **DO** loop delimiter):

Footnotes	Actions	DO Loops	Line
	DIR		1 # 1 "foo2.f"
			1 subroutine foo2(a,b,index,n)
			2 real a(n), b(n)
			3 integer index(n)
	DIR		4 c*\$*assert permutation (index)
			5
1	SO C	+-----	6 do i = 1, n
2	SO	*	7 a(index(i)) = a(index(i)) + b(i)
		*_____	8 enddo
			9 end

Abbreviations Used

SO scalar optimization
 DIR directive
 C concurrentized

Loop Summary						
loop#	From line	To line	Loop label	Loop index	at nest	Status
1	7	9	Do	I	1	concurrentized

Note: As with all assertions, Power Fortran does not verify the truth of this assertion. When you make an assertion, be certain that it is always true for all possible input data.

Function Call

This example shows what happens when a loop contains a call to an external routine. The Fortran 77 code

```

subroutine foo3 (a,b,c,n)
real a(n), b(n), c(n)
external force
c
do i = 1, n
  a(i) = force (b(i), c(i))
enddo
end
    
```

generates this listing:

Footnotes	Actions	DO Loops	Line
	DIR		1 # 1 "foo3.f"
			1 subroutine foo3(a,b,c,n)
			2 real a(n), b(n), c(n)
			3 external force
			4
1 2	NO SO NCS	+-----	5 do i = 1,n
3	NO SO NCS	!	6 a(i) = force (b(i), c(i))
		!_____	7 enddo
			8 end

Abbreviations Used

NO	not optimized
SO	scalar optimization
DIR	directive
NCS	non-concurrent-stmt

Footnote List

1: not optimized No optimizable statements found.
 2: not optimized This loop contains an unoptimizable call to "FORCE".
 3: not optimized This statement contains an unoptimizable call to
 "FORCE".

Loop Summary

Loop#	From line	To line	Loop label	Loop index	at nest	Status
1	6	8	Do	I	1	unoptimizable call (FORCE)

Calling the function **force** prevents Power Fortran from automatically running the loop in parallel. Power Fortran identifies the function call as a **non-concurrent-stmt**. By its nature, a nonconcurrent statement prevents Power Fortran from assuming the loop is safe to run in parallel because Power Fortran cannot see into the routine to look for data dependencies.

If you know that **force** generates no data dependencies, then explicitly state this fact for the nonconcurrent statement.

Example 3-2 Concurrent Function Call

```
subroutine foo3(a,b,c,n)
  real a(n), b(n), c(n)
  external force
c*$assert concurrent call
  do i = 1, n
    a(i) = force(b(i), c(i))
  enddo
end
```

Now that Power Fortran knows that the nonconcurrent statement involves no data dependency, Power Fortran will find the loop safe to run in parallel.

There is one subtlety in using the concurrent call assertion. When you use this assertion, Power Fortran makes no attempt to examine the called routine; it simply assumes that it is safe. However, Power Fortran is still left with the problem of correctly declaring the variables in the loop to be either **SHARE** or **LOCAL**. (Power Fortran does the best it can, but it can sometimes be fooled.) For example:

```
subroutine tricky (a,b,c,n,m)
  real a(*), b(*)
  external my_function
c**assert concurrent call
  do i = 1, n
    a(i) = my_function (b(i), m)
    b(i) = a(i) + m
  enddo
  m = 0
end
```

The question is whether the variable **m** should be **SHARE** or **LOCAL**. If the routine **my_function** reads only the old value of **m**, then it should be **SHARE**. If **my_function** writes a new value of **m**, then it should be **LOCAL**. In the absence of any more clues, Power Fortran must go by what it can see; and what it can see is that within the loop, there are no visible assignments to **m**, and so Power Fortran will declare it to be **SHARE**. If in fact **my_function** is writing the value of **m**, then this is incorrect.

In this case, to give Power Fortran the hint it needs, add a visible assignment to **m** at the top of the loop. For example, consider the following code:

```
do i = 1, n
  m = 0
  a(i) = my_function(b(i), m)
  b(i) = a(i) + m
enddo
```

Here, Power Fortran can see an assignment to **m** and so declares it to be **LOCAL**. Note that if **my_function** is both reading the old value and writing a new value of **m**, then it was not legal to parallelize the loop.

Reductions

This example shows how Power Fortran produces a single value from a set of values. Because the entire set of values is reduced to a single value, these operations are called reductions. Consider the following Fortran 77 code.

Example 3-3 Roundoff Reduction

```

subroutine foo4(a,b,n,sum)
  real a(n), b(n), sum
c
  sum = 0.0
  do i = 1, n
    sum = sum + a(i)*b(i)
  enddo
end

```

Using the previous code as input, Power Fortran produces this listing file:

Footnotes	Actions	DO Loops	Line
	DIR		1 # 1 "foo3.f"
			1 subroutine foo4(a,b,n,sum)
			2 real a(n), b(n), sum
			3
	SO		4 sum = 0.0
	SO	+-----	5 do i = i, n
1	DD SO	!	6 sum = sum + a(i)*b(i)
		!_____	7 enddo
			8 end

Abbreviations Used

DD data dependence
SO scalar optimization
DIR directive

Footnote List

1: data dependence Data dependence involving this line due to variable "SUM".

Loop Summary

Loop#	From line	To line	Loop label	Loop index	at nest	Status
1	6	8	Do	I	1	scalar mode preferable

Because different iterations of the loop read and write the same location (the variable **sum**), there is a dependence. However, this is a special case. Because **sum** just accumulates a total, you can accumulate subtotals in parallel and then combine the subtotals at the end.

Because the parallel version of the code adds the elements together in an order different from the single-process version, the round-off errors accumulate differently for the two versions of the code. Thus, the answer can differ slightly as you vary the number of processes used to run the code. In fact, if you use the dynamic scheduling option for the

code, the answer might vary slightly from one run of the program to the next, even if you use the same number of processes on the same machine.

Most applications can safely ignore this variation in round-off error. If you do not care about this round-off error, tell Power Fortran to use parallel subtotals. To tell Power Fortran not to worry about round-off error, use either the **C*\$*ROUND OFF(2)** directive or the *f77* command-line option **-pfa,-roundoff=2**. The resulting listing file is

```
Footnotes  Actions      DO Loops  Line
          DIR                1 # 1 "foo3.f"
          1      subroutine foo4(a,b,n,sum)
          2      real a(n), b(n), sum
          3
          SO                4      sum = 0.0
          SO C              +-----  5      do i = i, n
          SO                *                6          sum = sum + a(i)*b(i)
          *                *_____  7      enddo
          8      end
```

Abbreviations Used
 SO scalar optimization
 DIR directive
 C concurrentized

Footnote List
 1: data dependence Data dependence involving this line due to variable "SUM".

Loop Summary

Loop#	From line	To line	Loop label	Loop index	at nest	Status
1	6	8	Do	I	1	concurrentized

Be aware that the round-off error produced by the parallel reduction operation is not necessarily any worse than the round-off error already present in the original serial version. It is simply different. If your application did not worry about the round-off error in the original, there is no reason to suppose that it should worry about it in the parallel version. If, on the other hand, your application takes special steps to reduce round off (for example, adding the numbers together in order from smallest absolute value to largest), then you should not use parallel reductions.

The previous example is called a sum reduction because the reduction operator is +. Table 3-5 shows the types of reductions Power Fortran supports.

Table 3-5 Power Fortran Reductions

Type	Operator	Example
Sum	+	sum = sum + <i>expression</i>
Product	*	p = p* <i>expression</i>
Min	min()	a = min(a, <i>expression</i>)
Max	max()	x = max(x, <i>expression</i>)

All these reductions are under the control of the **-roundoff** command-line option, even though technically the min and max reductions do not involve round-off problems.

Customizing Power Fortran Execution

This chapter contains the following sections:

- “Overview of Customization” explains when to optimize Power Fortran execution.
- “Controlling Code Execution” describes how to control whether Power Fortran runs eligible loops in parallel.
- “Controlling Power Fortran Code Transformations” describes how to control the various transformations performed by Power Fortran.
- “Performing Inlining and Interprocedural Analysis” describes inlining and interprocedural analysis and explains how and when to perform these procedures.

Overview of Customization

You can insert comment statements into a Power Fortran program to control whether it runs loops in parallel, how it limits complexity or round-off, and when it performs inlining or interprocedural analysis. These comment statements apply to only certain portions of source code.

To customize how Power Fortran executes an entire program, you can specify various command-line options when you run Power Fortran as described in Chapter 2, “How to Use Power Fortran.” For a complete summary of the Power Fortran command-line options, refer to Appendix A, “Power Fortran Command-Line Options.”

This chapter describes options that are recognized only by Power Fortran. For details about options for controlling scalar optimizations in *pfu*, refer to Chapter 5, “Scalar Optimizations.”

Controlling Code Execution

When modifying most programs to allow loops to run in parallel, modify the code so that Power Fortran can automatically run the loop in parallel. To avoid forcing the loop to run in parallel, directly insert a **C\$ DOACROSS** directive. If you force code to run in parallel, you (and not Power Fortran) need to verify that no subsequent modification inserts data dependencies. Forcing these data dependencies in code to run in parallel can produce serious (and difficult-to-find) errors. Rewriting the loop so that Power Fortran recognizes the loop as safe to run in parallel allows Power Fortran to check future modifications for potential data dependencies.

This section describes how to control whether eligible loops are run in parallel and how to specify a work threshold for loops.

Running Code in Parallel

The **-concurrentize** option (or **-conc**) converts eligible loops to run in parallel. This is the default value for this option. The **-noconcurrentize** option (or **-nconc**) prevents Power Fortran from converting loops to run in parallel.

Loops requiring the addition of synchronization might run slower than the scalar original when concurrentized. In this case, you can specify the **-noconcurrentize** command-line option or the **C*\$ NO CONCURRENTIZE** directive for a particular loop.

Specifying a Work Threshold

The **-minconcurrent=*n*** option (or **-mc=*n***) specifies the minimum amount of work needed inside the loop to make executing a loop in parallel profitable. The positive integer *n* is a count of the number of operations (for example, add, multiply, load, store) in the loop, multiplied by the number of times the loop is executed. The higher the value for *n*, the larger (more iterations, more statements, or both) the loop body must be to be run in parallel.

If you do not specify this option, Power Fortran runs all loops containing 500 or more operations in parallel.

If the **DO** loop bounds are known at compilation time (that is, if they are constants), the compiler can compute the exact iteration count and decide whether to run the loop in parallel. If the **DO** loop bounds are unknown at compilation time, the compiler adds an **IF** clause to the **C\$DOACROSS** directive to test at run time if sufficient work exists. This is interpreted by the compiler as a request to generate two loops, one concurrentized and one left serial, and an **IF-THEN-ELSE** to make a run time check to decide whether to execute the loop in parallel. This case is called a two-version loop.

To disable the generation of two-version loops throughout the program, specify **-minconcurrent=0**; or to disable this action only in a few **DO** loops, specify the **C*\$*MINCONCURRENT(0)** directive.

For example, given the original loop

```

      DO 2 I =1,N
          X(I) = Y(I) * Z(I)
2      CONTINUE

```

Power Fortran generates the following transformed loop:

```

C$DOACROSS IF (N .GT. 100), SHARE (N,X,Y,Z), LOCAL(I)
      DO 3 I=1,N
          X(I) = Y(I)*Z(I)
3      CONTINUE

```

The **IF** clause ensures that n is large enough to make running the loop in parallel profitable (otherwise, Power Fortran runs the loop serially). If the loop bound is a small constant (such as 10) instead of n , Power Fortran would not generate a **DOACROSS** statement for the loop and the listing file states that the loop does not contain enough work. Conversely, if the bound is a large constant (such as 101), Power Fortran generates the **DOACROSS** statement without the **IF** clause.

Enabling Parallel I/O

The **-parallelio** option (or **-pio**) enables the parallelization of loops that contain I/O statements. The **no** version, which is the default, disables this optimization. Use this option only on systems with parallel I/O capabilities or where I/O statements in loops are not executed.

Controlling Power Fortran Code Transformations

This section discusses the various ways in which you can control the standard transformations that Power Fortran performs.

Specifying a Complexity Limit

The `-limit=n` option (or `-lm=n`) controls the amount of time Power Fortran can spend trying to determine whether a loop is safe to run in parallel. Power Fortran estimates how much time is required to analyze each loop nest construct. If an outer loop looks like it would take too much time to analyze, Power Fortran ignores the outer loop and recursively visits the inner loops.

Larger limits often allow Power Fortran to generate parallel code for deeply nested loop structures that it might not otherwise be able to run safely in parallel. However, with larger limits Power Fortran can also take more time to analyze a program. (The limit does not correspond to the **DO** loop nest level. It is an estimate of the number of loop orderings that Power Fortran can generate from a loop nest.) This option has the same effect as the global `C*$* LIMIT(n)` directive.

Note: You do not usually need to change these limits.

You can also change the thresholds for internal table size. Refer to the *MIPSpro Fortran 77 Programmer's Guide* for details.

Setting the Optimization Level

The `-optimize=n` option (or `-o=n`) sets the optimization level. The higher you set this level, the more code is optimized and the longer Power Fortran runs. Programs that are written for running in parallel often do not need advanced transformation. With these programs, a lower optimization level is enough. Valid values for *n* are as follows:

- | | |
|---|---|
| 0 | Avoids converting loops to run in parallel. |
| 1 | Converts loops to run in parallel without using advanced data dependence tests. Enables loop interchanging. |

- 2 Determines when scalars need last-value assignment using lifetime analysis. Also uses more powerful data dependence tests to find loops that can run safely in parallel. This level allows reductions in loops that execute concurrently but only if the **-roundoff** option is set to 2. (Refer to the following section for details about the **-roundoff** option.)
- 3 Breaks data dependence cycles using special techniques and additional loop interchanging methods, such as interchanging triangular loops. This level also implements special-case data dependence tests.
- 4 Generates two versions of a loop, if necessary, to break a data-dependent arc. This level also implements more-exact data dependence tests and allows special index sets (called wraparound variables) to convert more code to run in parallel.
- 5 Fuses two adjacent loops if it is legal to do so (that is, there are no data dependencies) and if the loops have the same control values. In certain limited cases, this level recognizes arrays as local variables. This level is the default.

Refer to the *MIPSpro Fortran 77 Programmer's Guide* for examples.

This option has the same effect as the global **C*\$* OPTIMIZE(n)** directive described in Chapter 7, "Fine-Tuning Power Fortran."

Controlling Variations in Round Off

The **-roundoff=n** option (or **-r=n**) controls the amount of variation in round off that Power Fortran allows. Valid values for *n* are these integers:

- 0-1 Suppresses any round-off transformations. This is the default.
- 2 Allows reductions to be performed in parallel. The valid reduction operators are **+**, *****, **min**, and **max**. This value is one of the most commonly-specified user options.
- 3 Recognizes **REAL** induction variables. Permits memory management transformations (refer to the *MIPSpro Fortran 77 Programmer's Guide* for details).

Refer to the *MIPSpro Fortran 77 Programmer's Guide* for examples.

When executing reductions in parallel, Power Fortran processes values in a different order from the original serial code. Round-off errors accumulate differently and produce a slightly different answer. Some algorithms are sensitive to this variation, and so, by default, Power Fortran does not run reductions in parallel. Usually, these tiny variations are irrelevant, and you can allow Power Fortran to process a reduction in parallel allowing more loops to be run in parallel.

Performing Inlining and Interprocedural Analysis

Function and subroutine calls create an obstacle to parallelization. Power Fortran provides three ways of dealing with this obstacle:

- Assert that the external routine is safe for concurrent execution (see “CVD\$ CNCALL” in Chapter 7).
- Inline the routine by replacing the call to the external routine with the actual code.
- Perform interprocedural analysis (IPA) by analyzing the external routine ahead of time and using the results of that analysis when a reference to the routine is encountered.

Inlining and IPA tend to be slow, memory-intensive operations. Attempting to inline all routines everywhere they occur can take a lot of time and use a lot of system resources. Inlining should usually be restricted to a few time-critical places. For details about inlining and IPA, and the related directives and command-line options, refer to Chapter 6, “Inlining and Interprocedural Analysis.”

Scalar Optimizations

This chapter contains the following sections:

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

Overview of Scalar Optimization

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

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

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

You can also initiate many of these optimizations with compiler directives (see Chapter 7, “Fine-Tuning Power Fortran.”)

Table 5-1 Optimization Options

Long Name	Short Name	Default Value
<code>-aggressive=letter</code>	<code>-ag=letter</code>	option off
<code>-arclimit=integer</code>	<code>-arclm=integer</code>	5000
<code>-[no]assume=list</code>	<code>-[n]as=list</code>	CEL
<code>-cacheline=integer</code>	<code>-chl=integer</code>	4
<code>-cachesize=integer</code>	<code>-chs=integer</code>	256
<code>-[no]directives=list</code>	<code>-[n]dr=list</code>	ackpv
<code>-dpreregisters=integer</code>	<code>-dpr=integer</code>	16
<code>-each_invariant_if_growth=integer</code>	<code>-eiifg=integer</code>	20
<code>-fpreregisters=integer</code>	<code>-fpr=integer</code>	16
<code>-fuse</code>	<code>-fuse</code>	option on with <code>-scalaropt=2</code> or <code>-optimize=5</code>
<code>-max_invariant_if_growth=integer</code>	<code>-miifg=integer</code>	500
<code>-optimize=integer</code>	<code>-o=integer</code>	depends on <code>-O</code> option
<code>-recursion</code>	<code>-rc</code>	option on
<code>-roundoff=integer</code>	<code>-r=integer</code>	depends on <code>-O</code> option
<code>-scalaropt=integer</code>	<code>-so=integer</code>	depends on <code>-O</code> option
<code>-setassociativity=integer</code>	<code>-sasc=integer</code>	1
<code>-unroll=integer</code>	<code>-ur=integer</code>	4
<code>-unroll2=weight</code>	<code>-ur2=weight</code>	100

The `-On` option directly initiates basic optimizations.

Performing General Optimizations

This section discusses the general optimizations that you can enable.

Enabling Loop Fusion

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

Controlling Global Assumptions

The `-assume=list` option (or `-as=list`) controls certain global assumptions of a program. You can also control most of these assumptions with various assertions (see “Assertions” on page 71). The default is `-assume=ccl`.

list can contain the following characters:

- a Allows procedure argument aliasing, which is when different subroutine or function parameters refer to the same object. This practice is forbidden by the Fortran 77 standard. This option provides a method of dealing with programs that use argument aliasing anyway.
- b Allows array subscripts to go outside the declared bounds.
- c Places constants used in subroutine or function calls in temporary variables.
- e Allows variables in **EQUIVALENCE** statements to refer to the same memory location inside one **DO** loop nest.
- l Uses temporary variables within an optimized loop and assigns the last value to the original scalar, if the compiler determines that the scalar can be reused before it is assigned.

By default, the compiler assumes that a program conforms to the Fortran 77 standard, that is, `-assume=el`, and includes `-assume=c` to simplify some analysis and inlining. You can disable the default values by specifying the `-noassume` option.

Example

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

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

Setting Invariant IF Floating Limits

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

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

The **-each_invariant_if_growth=*integer*** option (or **-eiifg=*integer***) controls the rewriting of **IF** statements nested within loops. This option specifies a limit on the number of executable statements in a nested **IF** statement. If the number of statements in the loop exceeds this limit, the compiler does not rewrite the code. If there are fewer statements, the compiler improves execution speed by interchanging the loop and **IF** statements.

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

This process becomes complicated when there is other code in the loop, since a copy of the other code must be included in both the **THEN** and **ELSE** loops.

For example, the following code:

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

becomes

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

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

This can be controlled on a loop-by-loop basis with the **C*\$*** **EACH_INVARIANT_IF_GROWTH** (*integer*) directive within the source (see “Setting Invariant IF Floating Limits” in Chapter 7).

You can limit the total amount of additional code generated in a program unit through invariant **IF** floating by specifying the **-max_invariant_if_growth** option.

The **-max_invariant_if_growth=integer** option (or **-miifg=integer**) specifies an upper bound on the total number of additional lines of code the compiler can generate in each program unit through invariant **IF** floating. This limit is applied on a per subroutine

basis. For example, if a subroutine is 400 lines long and `-miifg=500`, the compiler can add at most 100 lines in the process of invariant **IF** floating. The default for *integer* is 500.

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

This can be controlled on a loop-by-loop basis with the `C*$* MAX_INVARIANT_IF_GROWTH(integer)` directive within the source (see “Setting Invariant IF Floating Limits” in Chapter 7).

Setting the Optimization Level

The `-optimize=integer` option (or `-o=integer`) sets the optimization level. Each optimization level is cumulative (that is, level 5 performs everything up to and including level 5). You can also modify the optimization level on a loop-by-loop basis by using the `C*$* OPTIMIZE(integer)` directive within the source (see “Optimization Level” in Chapter 7).

Valid values for *integer* are as follows:

<code>fe0</code>	Disables optimization.
<code>1</code>	Performs only simple optimizations. Enables induction variable recognition.
<code>2</code>	Performs lifetime analysis to determine when last-value assignment of scalars is necessary.
<code>3</code>	Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wrap-around variables.
<code>4</code>	Generates two versions of a loop, if necessary, to break a data dependence arc.
<code>5</code>	Enables array expansion and loop fusion.

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

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

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

```

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

```

At **-optimize=1**, the compiler sees the summation in **ASUM** as an intractable data dependence between iterations and does not try to optimize the loop. Specifying **-optimize=2** (perform lifetime analysis and do not interchange around reduction) produces the following:

```

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

```

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

```

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

```

Controlling Variations in Round Off

The **-roundoff=integer** option (or **-r=integer**) controls the amount of variation in round-off error produced by optimization. If an arithmetic reduction is accumulated in a different order than in the scalar program, the round-off error is accumulated differently and the final result might differ from the output of the original program. Although the

difference is usually insignificant, certain restructuring transformations performed by the compiler must be disabled to obtain exactly the same answers as the scalar program.

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

- 0 Suppresses any transformations that change round-off error.
- 1 Performs expression simplification, which might generate various overflow or underflow errors, for expressions with operands between binary and unary operators, expressions that are inside trigonometric intrinsic functions returning integer values, and after forward substitution. Enables strength reduction. Performs intrinsic function simplification for *max* and *min*. Enables code floating if **-scaleropt** is at least 1. Allows loop interchanging around serial arithmetic reductions, if **-optimize** is at least 4. Allows loop rerolling, if **-scaleropt** is at least 2.
- 2 Allows loop interchanging around arithmetic reductions if **-optimize** is at least 4. For example, the floating point expression $A/B/C$ is computed as $A/(B*C)$.
- 3 Recognizes **REAL** (float) induction variables if **-scaleropt** is greater than 2 or **-optimize** is at least 1. Enables sum reductions. Enables memory management optimizations if **-scaleropt=3** (see “Performing Memory Management Transformations” on page 53 for details about memory management transformations).

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

Consider the following code segment.

Example 5-1 Controlling Roundoff

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


When `-roundoff=1`, the compiler does not transform the summation reduction. The compiler distributes the loop:

```

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

```

When `-roundoff=2` and `-optimize=5` (reduction variable identification and loop interchange around arithmetic reduction), the original code becomes:

```

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

```

When `-roundoff=3` and `-optimize=5`, the compiler recognizes **REAL** induction variables. In this example, the compiler performs forward substitution of the transformed induction variable **X**:

The following code

```

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

```

becomes

```

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

```

Controlling Scalar Optimizations

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

- 0 Disables all scalar optimizations.
- 1 Enables simple scalar optimizations—dead code elimination, global forward substitution of variables, and conversion of **IF-GOTO** to **IF-THEN-ELSE**.
- 2 Enables the full range of scalar optimizations—floating invariant **IF** statements out of loops, loop rerolling and unrolling (if `-roundoff` is greater than zero), array expansion, loop fusion, loop peeling, and induction variable recognition.
- 3 Enables memory management transformations if `-roundoff=3` (see “Performing Memory Management Transformations” on page 53 for details about memory management transformations). Performs dead-code elimination during output conversion.

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

Unlike the `-scalaropt` command-line option, the `C*$* SCALAR OPTIMIZE` directive sets the level of loop-based optimizations (for example, loop fusion) only, and not straight-code optimizations (for example, dead-code elimination).

Refer to “Controlling Scalar Optimizations” in Chapter 7 for details about the `C*$* SCALAR OPTIMIZE` directive.

Using Vector Ininsics

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

Finding Vector Intrinsic

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

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

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

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

Example 5-2 Vector Intrinsic

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

In the preceding function,

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

Limitations of the Vector Ininsics

The vector intrinsics are limited in the following ways:

- The SQRT function is not used automatically in the current release (but it can be called directly; see “Calling Vector Functions Directly” on page 50).
- The single-precision COS, SIN, and TAN functions are valid only for arguments whose absolute value is less than or equal to 2^{28} .
- The double-precision COS, SIN, and TAN functions are valid only for arguments whose absolute value is less than or equal to $\pi \cdot 2^{19}$.

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

Disabling Vector Ininsics

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

```
f77 -64 -mips4 -O3 -OPT:vector_intrinsics=OFF trig.f
```

Calling Vector Functions Directly

The vector intrinsic functions are C functions that can be called directly using the techniques discussed in the *MIPSpro Fortran 77 Programmer's Guide*. The prototype of one function is as follows:

```
__vsinf( void*from, void*dest, int count, int fromstride, int deststride )
```

Note the two leading underscore characters in the name. The arguments are

<i>from</i>	Address of the first element of the source array
<i>dest</i>	Address of first element of destination array
<i>count</i>	Number of elements to process
<i>fromstride</i>	Number of elements to advance in the source array
<i>deststride</i>	Number of elements to advance in the destination array

For example, the compiler converts a loop of this form:

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

into nonlooping code of this form:

```
real a(10000), b(10000)
call __VSINF$(%REF(A(1)),%REF(A(2)),%VAL(1000),%VAL(3),%VAL(2))
```

All the vector intrinsic functions have the same prototype as the one shown above for `__vsinf`. The names of the available vector functions are shown in Table 5-2.

Table 5-2 Vector Intrinsic Function Names

Operation	REAL*4 Function Name	REAL*8 Function Name
acos	__vacosf	__vacos
asin	__vasinf	__vasin
atan	__vatanf	__vatan
cos	__vcosf	__vcos
exp	__vexpf	__vexp
log	__vlogf	__vlog
sin	__vsinf	__vsin
sqrt	__vsqrtf	__vsqrt
tan	__vtanf	__vtan

Performing Advanced Optimizations

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

Using Aggressive Optimization

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

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

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

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

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

can degrade performance if your program contains the following statement:

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

All three of the arrays **a**, **b**, and **c** have the same starting virtual address modulo the cache size, and so every access to the array elements causes a cache miss. It would be much better to add some padding between each of the arrays to force the virtual addresses to be different. The `-aggressive=a` option does exactly this.

Unfortunately, this transformation is not always possible. Fortran allows different routines to have different definitions of **COMMON**. If some other routine contained the definition

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

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

Controlling Internal Table Size

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

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

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

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

Performing Memory Management Transformations

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

Memory Management Techniques

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

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

Loop blocking is a transformation that can be applied when the loop nesting depth is greater than the dimensions of the data arrays being manipulated. For example, the simple matrix multiply uses a nest of three loops operating on two-dimensional arrays. The simple approach repeatedly sweeps across the entire arrays. A better approach is to break the arrays up into blocks, each block being small enough to fit into the cache, and then make repeated sweeps over each (in-cache) block. (This technique is also sometimes

called “tiles” or “tiling.”) However, the code needed to implement a block style algorithm is often very complex and messy. This automatic transformation allows you to write the simpler method, and have the compiler transform it into the more complex and efficient block method.

Memory Management Options

The compiler recognizes the following memory management command-line options when specified with the `-pfa` option:

- `-cacheline` specifies the width of the memory channel between cache and main memory.
- `-cachesize` specifies the data cache size.
- `-fpreregisters` specifies an unrolling factor.
- `-dpreregisters` ensures that registers do not overflow during loop unrolling.
- `-setassociativity` specifies which memory management transformation to use.

The `-cacheline=integer` option (or `-chl=integer`) specifies the width of the memory channel, in bytes, between the cache and main memory. The default value for *integer* is 4. Refer to Table 5-3 for the recommended setting for your machine.

The `-cachesize=integer` option (or `-chs=integer`) specifies the size of the data cache, in kilobytes, for which to optimize. The default value for *integer* is 256 kilobytes. Refer to Table 5-3 for the recommended setting for your machine. You can obtain the cache size for a given machine with the `hinv(1)` command. This option is generally useful only in conjunction with the other memory management transformations.

Table 5-3 Recommended Cache Option Settings

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

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

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

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

Enabling Loop Unrolling

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

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

0	Uses default values to unroll.
1	Disables unrolling.
2- <i>n</i>	Unrolls, at most, this many iterations.

The `-unroll2=weight` (or `-ur2=weight`) option specifies an upper bound on the number of operations in a loop when unrolling it with the `-unroll` option. The default value for *weight* is 100. The compiler unrolls an inner loop until the number of operations (the

amount of work) in the unrolled loop is close to this upper bound, or until the number of iterations specified in the **-unroll** option is reached, whichever occurs first.

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

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

The following example uses the **C*\$* UNROLL** directive (see “Enabling Loop Unrolling” in Chapter 7) to specify 8 for the maximum number of iterations to unroll and 100 for the maximum “work per unrolled iteration.” (This is equivalent to specifying **-pfa,-unroll=8,-unroll2=100.**)

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

This example has:

```
1 assignment
0 IF statements
3 subscripts
2 arithmetic operators
```

—
6 is the weighted sum (the work for 1 iteration)

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

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

The result is the following example.

Example 5-3 Loop Unrolling

```

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

```

Recognizing Directives

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

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

The default value for *list* is **ackpv**. For example, `-pfa,-directives=k` enables Silicon Graphics directives only, whereas `-pfa,-directives=kas` enables Silicon Graphics directives and assertions and Sequent directives.

To disable all of the above options, enter `-nodirectives` or `-directives` (without any values for *list*) on the command line. Chapter 7, "Fine-Tuning Power Fortran," describes the Silicon Graphics, Cray, Sequent, and VAST directives the compiler accepts.

Assertions are similar in form to directives, but they assert program characteristics that the compiler can use in its optimizations. In addition to specifying **a** in *list*, you can control whether the compiler accepts assertions using the **C*\$* ASSERTIONS** and **C*\$* NO ASSERTIONS** directives (refer to “Using Assertions” in Chapter 7).

Specifying Recursion

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

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

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

Inlining and Interprocedural Analysis

This chapter contains the following sections:

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

Overview of Inlining and IPA

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

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

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

Using Command-Line Options

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

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

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

Table 6-1 Inlining and IPA Options

Long Option Name	Short Option Name	Default Value
<code>-inline[=<i>list</i>]</code>	<code>-inl[=<i>list</i>]</code>	option off
<code>-ipa[=<i>list</i>]</code>	<code>-ipa[=<i>list</i>]</code>	option off
<code>-inline_and_copy</code>	<code>-inc</code>	option off
<code>-inline_looplevel=<i>integer</i></code>	<code>-inll=<i>integer</i></code>	2
<code>-ipa_looplevel=<i>integer</i></code>	<code>-ipall=<i>integer</i></code>	2
<code>-inline_depth=<i>integer</i></code>	<code>-ind=<i>integer</i></code>	2
<code>-inline_man</code>	<code>-inm</code>	option off
<code>-ipa_man</code>	<code>-ipam</code>	option off
<code>-inline_from_files=<i>list</i></code>	<code>-inff=<i>list</i></code>	option off
<code>-ipa_from_files=<i>list</i></code>	<code>-ipaff=<i>list</i></code>	option off
<code>-inline_from_libraries=<i>list</i></code>	<code>-infl=<i>list</i></code>	option off
<code>-ipa_from_libraries=<i>list</i></code>	<code>-ipa=<i>list</i></code>	option off
<code>-inline_create[=<i>name</i>]</code>	<code>-incr[=<i>name</i>]</code>	option off
<code>-ipa_create[=<i>name</i>]</code>	<code>-ipacr[=<i>name</i>]</code>	option off

Specifying Routines for Inlining or IPA

The `-inline[=list]` option (or `-inl[=list]`) provides a list of routines to be expanded inline; the `-ipa[=list]` option provides a list of routines to be analyzed. The routine names in *list* must be separated by colons. If you do not specify a list of routines, the compiler expands all eligible routines. The compiler looks for the routines in the current source file, unless you specify an `-inline_from` or `-ipa_from` option. Refer to “Specifying Where to Search for Routines” on page 64 for details.

Example

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

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

Refer to “Conditions That Prevent Inlining and IPA” on page 67 for information about conditions that prevent inlining and IPA.

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

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

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

Specifying Occurrences for Inlining and IPA

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

Loop Level

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

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

Example

Consider the following code:

```
PROGRAM MAIN
  ..
  CALL A -----> SUBROUTINE A
  ..
  DO
    DO
      CALL B -----> SUBROUTINE B
    ENDDO
  ENDDO
      DO
        CALL C -----> SUBROUTINE C
      ENDDO
    ENDDO
```

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

For `-inline_looplevel=1`, only the routines referenced in the most deeply nested call sites are inlined (subroutine C in the above example). (If more than one routine is called at the same loop nest level, the compiler selects all of them when that level is inlined/analyzed.)

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

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

A routine that passes the **-inline_looplevel** test is inlined everywhere it is used, even places that are not in deeply-nested loops. If some, but not all, invocations of a routine are to be expanded, use the **C*\$* INLINE** or **C*\$* IPA** directives just before each **CALL**/reference to be expanded (refer to “Fine-Tuning Inlining and IPA” in Chapter 7).

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

Depth

The **-inline_depth=integer** option (or **-ind=integer**) restricts the number of times the compiler continues to attempt inlining already inlined routines. Valid values for integer are as follows:

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

When a routine is expanded inline, it can contain references to other routines. The compiler must decide whether to recursively expand these references (which might themselves contain yet other references, and so on). This option limits the number of times the compiler performs this recursive expansion. Note that the default setting is quite low; if you know inlining is useful for a particular program, increase this setting.

Note: There is no `-ipa_depth` option.

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

Manual Control

The `-inline_man` (or `-inm`) option enables recognition of the `C*$* INLINE` directive. This directive, described in “Fine-Tuning Inlining and IPA” in Chapter 7, allows you to select individual instances of routines to be inlined. The `-ipa_man` (or `-ipam`) option is the analogous option for the `C*$* IPA` directive.

Specifying Where to Search for Routines

The options listed in Table 6-2 tell the compiler where to search for the routines specified with the `-inline` or `-ipa` options. If you do not specify either option, the compiler searches the current source file by default.

Table 6-2 Inlining and IPA Search Command-Line Options

Long Option Name	Short Option Name
<code>-inline_from_files=list</code>	<code>-inff=list</code>
<code>-ipa_from_files=list</code>	<code>-ipaff=list</code>
<code>-inline_from_libraries=list</code>	<code>-infl=list</code>
<code>-ipa_from_libraries=list</code>	<code>-ipafl=list</code>

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

For example

```
-pfa, -inline_from_files=file1:file2:file3
```

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

Table 6-3 Filename Extensions

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

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

- “-” means current source file (as listed on the command line or specified in an `-input=file` command-line option)
- “.” means the current working directory

Example

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

```
% f77 -pfa,-inline,-inline_from_files=-:input.f calc.f
```

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

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

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

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

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

Creating Libraries

When performing inlining and IPA, the compiler analyzes the routines in the source program. Normally, inlining is done directly from a source file. However, when inlining the same set of routines in many different programs, it is more efficient to create a pre-analyzed library of the routines. Use the **-inline_create[=name]** option (or **-incr[=name]**) to create a library of prepared routines (for later use with the **-inline_from_libraries** option). The compiler assigns *name* to the library file it creates; for maximum compatibility, use the filename extension **.klib**. For example: **samp.klib**.

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

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

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

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

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

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

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

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

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

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

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

Conditions That Prevent Inlining and IPA

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

These conditions inhibit inlining:

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

Inlining is also inhibited when the routine to be inlined

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

Fine-Tuning Power Fortran

This chapter contains the following sections:

- “Overview of Directives and Assertions” explains the concept of directives and assertions.
- “Circumventing Power Fortran” explains how to use directives to bypass Power Fortran’s analysis and leave areas of code unchanged.
- “Fine-Tuning Scalar Optimizations” describes how you can use directives to fine-tune scalar optimizations.
- “Fine-Tuning Inlining and IPA” explains how to use directives for inlining and IPA.
- “Running Code Serially” explains how to use directives and assertions to stop Power Fortran from running specific code in parallel.
- “Running Code in Parallel” explains how to use directives and assertions to tell Power Fortran that it is safe to run specific parts of code in parallel.
- “Using Equivalenced Variables” explains how you can inform the compiler that your code uses or does not use equivalenced variables.
- “Using Assertions” tells how to enable or disable compiler recognition of assertions.
- “Using Aliasing” explains the assertions that enable or disable types of aliasing.
- “Fine-Tuning Global Assumptions” describes how you can use assertions to fine-tune global assumptions.
- “Ignoring Data Dependencies” explains how to tell Power Fortran that apparently data-dependent code is safe to run in parallel.

Overview of Directives and Assertions

After you run a Fortran source program through Power Fortran once, you can use directives and assertions to fine-tune program execution. The listing file shows where

and why Power Fortran did not parallelize the code. You can also use WorkShop Pro MPF to review the Power Fortran analysis of your program.

You can use directives and assertions to force Power Fortran to execute portions of code in various ways. Command-line directives apply to the program as a whole. You can use the **-pfa,-directives** command-line option to selectively enable or disable certain directives and assertions. Refer to “Recognizing Directives” on page 57 for information about the **-directives** option.

If you want finer control for parallelizing a critical loop or inlining a particular occurrence of a routine, specify directives and assertions directly in the code. You can also use directives and assertions to keep Power Fortran from converting code to run in parallel. In other cases you might want to explicitly force Power Fortran to run segments of code in parallel even though it normally would not.

Because Power Fortran does not check the correctness of assertions, they can be unsafe. If you specify an incorrect assertion, the code generated by Power Fortran might give different answers from the scalar program. If you suspect unsafe assertions are causing problems, use the **-nodirectives** command-line option or the **C*\$ NO ASSERTIONS** directive to tell Power Fortran to ignore all assertions.

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. Others override directives. Many directives have corresponding command-line options. If you specify conflicting settings in the command line and a directive, the compiler chooses the most restrictive setting.

For Boolean options, if either the directive or the command line has the option turned off, it is considered off. For options that require a numeric value, the compiler uses the minimum of the command-line setting and the directive setting.

Table 7-1 lists the directives supported by the compiler. In addition to the standard Silicon Graphics directives, the compiler supports the Cray and VAST directives listed in the table. The compiler maps these directives to corresponding Silicon Graphics assertions. Refer to “Assertions” on page 71 for details.

Table 7-1 Directives Summary

Directive	Compatibility
C*\$* ARCLIMIT(<i>n</i>)	Silicon Graphics
C*\$* [NO] ASSERTIONS	Silicon Graphics
C*\$* EACH_INVARIANT_IF_GROWTH(<i>n</i>)	Silicon Graphics
C*\$* [NO] INLINE	Silicon Graphics
C*\$* [NO] IPA	Silicon Graphics
C*\$* MAX_INVARIANT_IF_GROWTH(<i>n</i>)	Silicon Graphics
C*\$* OPTIMIZE(<i>n</i>)	Silicon Graphics
C*\$* ROUNDOFF(<i>n</i>)	Silicon Graphics
C*\$* SCALAR OPTIMIZE(<i>n</i>)	Silicon Graphics
C*\$* UNROLL(<i>integer</i> [, <i>weight</i>])	Silicon Graphics
CDIR\$ NO RECURRENCE	Cray
CVD\$ [NO] DEPCHK	VAST
CVD\$ [NO] LSTVAL	VAST

Assertions

Assertions provide the compiler with additional information about a source program. Sometimes assertions can improve optimization results. Use them only when speed is essential. Assertions can be unsafe because the compiler cannot verify the accuracy of the information provided. If you specify an incorrect assertion, the compiler-generated code might produce results different from those of the original serial program. If you suspect unsafe assertions are causing problems, use the `-pfa,-nodirectives` command-line option or the `C*$* NO ASSERTIONS` directive to tell the compiler to ignore all assertions.

Table 7-2 lists the supported assertions and their duration.

Table 7-2 Assertions and Their Duration

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

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

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

Many assertions, like directives, are active until the end of the program unit (or file) or until you reset them. Other assertions are active within a program unit, regardless of where they appear in that program unit.

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

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

To understand how the compiler interprets assertions, you must understand the concept of *assumed dependences*. The following loop contains two types of dependences:

```

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

```

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

Circumventing Power Fortran

Sometimes you might need to hand-tune a **DO** loop so that it runs in parallel. Use the directives in this section to prevent Power Fortran from analyzing your modified code.

C\$ DOACROSS

The **C\$ DOACROSS** directive tells the Fortran 77 compiler to generate parallel code for the following loop. When Power Fortran encounters this directive on input, it does not modify the accompanying loop and therefore does not interfere with any hand-tuning.

C\$ DOACROSS is the standard method for parallelism in Fortran. This directive is the same directive that Power Fortran generates as a result of its analysis. Refer to the *MIPSpro Fortran 77 Programmer's Guide* for more information about the **C\$ DOACROSS** directive and its optional clauses.

Power Fortran runs the following code as it appears:

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

The C\$& Directive

The **C\$&** directive continues the **C\$ DOACROSS** 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)
```

The C*\$* NO SYNC Assertion

Sometimes when Power Fortran concurrentizes a loop, it adds unnecessary synchronization directives or other synchronization code. Use the C*\$* **ASSERT NO SYNC** assertion to eliminate synchronization overhead.

Fine-Tuning Scalar Optimizations

The compiler supports several directives that allow you to fine-tune the scalar optimizations described in “Controlling Scalar Optimizations” on page 48.

Controlling Internal Table Size

The C*\$* **ARCLIMIT**(*integer*) directive sets the minimum size of the internal table that the compiler uses for data dependence analysis. The greater the value for *integer*, the more information the compiler can keep on complex loop nests. The maximum value and default value for *integer* is 5000. When you specify this directive globally, it has the same effect as the **-arclimit** command-line option (refer to “Controlling Internal Table Size” on page 53 for details).

Setting Invariant IF Floating Limits

The C*\$* **EACH_INVARIANT_IF_GROWTH** and the C*\$* **MAX_INVARIANT_IF_GROWTH** directives control limits on invariant **IF** floating. This process generally involves duplicating the body of the loop, which can increase the amount of code considerably. Refer to “Setting Invariant IF Floating Limits” on page 42 for details about invariant **IF** floating.

The C*\$* **EACH_INVARIANT_IF_GROWTH**(*integer*) directive limits the total number of additional lines of code generated through invariant **IF** floating in a loop. You can control this limit globally with the **-each_invariant_if_growth** command-line option (see “Setting Invariant IF Floating Limits” on page 42).

You can limit the maximum amount of additional code generated in a program unit through invariant **IF** floating with the C*\$* **MAX_INVARIANT_IF_GROWTH**(*integer*) directive. Use the **-max_invariant_if_growth** command-line option to control this limit globally (see “Setting Invariant IF Floating Limits” on page 42).

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

Example

Consider the following code:

```
C*$*EACH_INVARIANT_IF_GROWTH(integer)
C*$*MAX_INVARIANT_IF_GROWTH(integer)
    DO I = ...
C*$*EACH_INVARIANT_IF_GROWTH(integer)
C*$*MAX_INVARIANT_IF_GROWTH(integer)
        DO J = ...
C*$*EACH_INVARIANT_IF_GROWTH(integer)
C*$*MAX_INVARIANT_IF_GROWTH(integer)
            DO K = ...
                section-1
                IF ( ) THEN
                    section-2
                ELSE
                    section-3
                ENDIF
                section-4
            ENDDO
        ENDDO
    ENDDO
```

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

Optimization Level

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

- 0 Disables optimization.
- 1 Performs only simple optimizations. Enables induction variable recognition.

- 2 Performs lifetime analysis to determine when last-value assignment of scalars is necessary.
- 3 Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wrap-around variables.
- 4 Generates two versions of a loop, if necessary, to break a data dependence arc.
- 5 Enables array expansion and loop fusion.

Refer to “Controlling Scalar Optimizations” on page 48 for examples.

Variations in Round Off

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

- 0 Suppresses any transformations that change round-off error.
- 1 Performs expression simplification, which might generate various overflow or underflow errors, for expressions with operands between binary and unary operators, for expressions that are inside trigonometric intrinsic functions returning integer values, and after forward substitution. Enables strength reduction. Performs intrinsic function simplification for *max* and *min*. Enables code floating if **-scaleropt** is at least 1. Allows loop interchanging around serial arithmetic reductions, if **-optimize** is at least 4. Allows loop rerolling, if **-scaleropt** is at least 2.
- 2 Allows loop interchanging around arithmetic reductions if **-optimize** is at least 4. For example, the floating point expression **A/B/C** is computed as **A/(B*C)**.
- 3 Recognizes **REAL** (float) induction variables if **-scaleropt** is greater than 2 or **-optimize** is at least 1. Enables sum reductions. Enables memory management optimizations if **-scaleropt=3** (see “Performing Memory Management Transformations” on page 53 for details).

Controlling Scalar Optimizations

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

- | | |
|---|--|
| 0 | Disables all scalar optimizations. |
| 1 | Enables simple, loop-based, scalar optimization—changing IF loops to DO loops, simple code floating out of loops, and forward substitution of variables. |
| 2 | Enables the full range of loop-based scalar optimizations—induction variable recognition, loop rerolling, loop unrolling, loop fusion, and array expansion. |
| 3 | Enables memory management transformations if <code>-roundoff=3</code> . Refer to “Performing Memory Management Transformations” on page 53 for details. |

Enabling Loop Unrolling

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

The compiler unrolls the loop proceeding the `C*$* UNROLL` directive until either the number of operations in the loop equals the *weight* parameter or the number of iterations reaches the *integer* parameter, whichever occurs first. The `-unroll` and `-unroll2` command-line options act like a global `C*$* UNROLL` directive. See “Enabling Loop Unrolling” on page 55 for detailed examples.

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

Fine-Tuning Inlining and IPA

Chapter 6, “Inlining and Interprocedural Analysis,” explains how to use inlining and IPA on an entire program. You can fine-tune inlining and IPA using the `C*$* [NO] INLINE` and `C*$* [NO] IPA` directives.

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

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

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

where

<i>name</i>	Specifies the routines to be inlined. If you do not specify a name, this directive will affect all routines in the program.
HERE	Applies the INLINE directive only to the next line; occurrences of the named routines on that next line are inlined.
ROUTINE	Inlines the named routines everywhere they appear in the current routine.
GLOBAL	Inlines the named routines throughout the source file.

If you do not specify **HERE**, **ROUTINE**, or **GLOBAL**, the directive applies only to the next statement. The `C*$* NO INLINE` form overrides the `-inline` command-line option and so allows you to selectively disable inlining of the named routines at specific points.

Example 7-1 Inline Control

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

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


Using the specifier **ROUTINE** rather than **HERE** inlines both calls. This routine must be compiled with the **-inline_man** command-line option for the compiler to recognize the **C** INLINE** directive.

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

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

Running Code Serially

Use the following assertions and directives to keep Power Fortran from running specific code in parallel.

C ASSERT DO (SERIAL)**

The **C** ASSERT DO (SERIAL)** assertion tells Power Fortran to run the loop immediately following it serially. Power Fortran also does not try to run any enclosing loop in parallel. However, it can still convert any loops nested inside the serial loop to run in parallel. For example, consider the following code.

Example 7-2 Serial Execution

```

      DO 100 i = 1, n
        DO 100 j = 1, n
C**ASSERT DO (SERIAL)
          DO 200 k = 1, n
            X(i, j, k) = X(i, j, k) * Y(i, j)
200      CONTINUE
          DO 300 k = 1, n
            X(i, j, k) = X(i, j, k) + Z(i, k)
300      CONTINUE
100     CONTINUE
```

The assertion forces the **DO 100 I** loop, the **DO 100 J** loop, and the **DO 200 K** loop to be serial. The compiler can still optimize the **DO 300 K** loop. In this case, the compiler does not distribute the **I** or **J** loops to try to obtain an optimizable loop.

See also “**C** ASSERT DO PREFER (SERIAL)**” on page 80.

CDIR\$ NEXT SCALAR

MIPSpro Power Fortran 77 supports the corresponding Cray directive, **CDIR\$ NEXT SCALAR**. Power Fortran interprets this directive as if it were a **C*\$* ASSERT DO (SERIAL)** assertion and generates scalar code for the next **DO** loop.

C*\$* ASSERT DO PREFER (SERIAL)

The **C*\$* ASSERT DO PREFER (SERIAL)** assertion tells the compiler to prefer any ordering in which the loop following the assertion remains serial. Unlike **C*\$* ASSERT DO (SERIAL)**, this assertion does not inhibit optimization of outer loops. This assertion directs Power Fortran to leave the **DO** loop alone, regardless of the setting of the optimization level. You can use this assertion to control which loop (in a nest of loops) Power Fortran chooses to run in parallel.

The following code segment is an example of how to use the assertion:

```
DO 100 I = 1, N
C*$*ASSERT DO PREFER (SERIAL)
DO 100 J = 1, M
    A(I,J) = B(I,J)
100 CONTINUE
```

In the **DO** loop above, the assertion requests that the **J** loop be serial. In this construction, Power Fortran tries to run the **I** loop in parallel but not the **J** loop. This capability is useful when you know the value of **M** to be very small or less than **N**. This assertion applies only to the **DO** loop that appears directly after the assertion.

Running Code in Parallel

This section explains the directives and assertions that allow Power Fortran to determine that specific areas of code are safe to run in parallel.

C*\$* [NO] CONCURRENTIZE

The **C*\$* [NO] CONCURRENTIZE** directive converts eligible loops to run in parallel. The **NO** version prevents Power Fortran from converting loops to run in parallel. These directives override the **-[no] concurrentize** command-line option.

For example, if your program contains the `C*$* NO CONCURRENTIZE` directive, parallelization is disabled even if you compile with `-concurrentize`. When specified globally, these directives have the same effect as the `-concurrentize` and `-noconcurrentize` options (see “Running Code in Parallel” on page 34 for details).

CVD\$ CONCUR

Power Fortran supports the VAST directive `CVD$CONCUR`. This directive runs a loop in parallel to optimize performance. Power Fortran interprets this directive as if it were the `C*$*CONCURRENTIZE` directive.

C*\$* ASSERT DO PREFER (CONCURRENT)

The `C*$* ASSERT DO PREFER (CONCURRENT)` assertion directs Power Fortran to run a particular nested loop in parallel if possible. Power Fortran runs another of the nested loops in parallel only if a condition prevents running the selected loop in parallel.

This assertion applies only to the `DO` loop immediately after it.

Consider the following code:

```
C*$* ASSERT DO PREFER (CONCURRENT)
      DO 100 I = 1, N
        DO 100 J = 1, M
          A (I, J) = B (I, J)
100      CONTINUE
```

This code directs Power Fortran to prefer to run the `I` loop in parallel. However, if a data dependence conflict prevents running the `I` loop in parallel, Power Fortran might run the `J` loop in parallel.

The `-noconcurrentize` command-line option and the `C*$* NO CONCURRENTIZE` directive prevent Power Fortran from generating concurrent code, even if you specify the `C*$* ASSERT DO PREFER (CONCURRENT)` assertion.

Using Equivalenced Variables

The `C*$* ASSERT [NO] EQUIVALENCE HAZARD` assertion tells the compiler that your code does not use equivalenced variables to refer to the same memory location inside one loop nest. Normally, `EQUIVALENCE` statements allow your code to use different variable names to refer to the same storage location.

The `-pfa,-assume=e` command-line option acts like the global `C*$* ASSERT EQUIVALENCE HAZARD` assertion (see “Controlling Global Assumptions” on page 41). The `C*$* ASSERT EQUIVALENCE HAZARD` assertion is active until you reset it or until the end of the program.

Using Assertions

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

If you specify the `-directives` command-line option without the assertions parameter (that is, `a`), the compiler will ignore assertions regardless of whether the file contains the `C*$* ASSERTIONS` directive. Refer to “Recognizing Directives” on page 57 for details on the `-directives` command-line option.

Using Aliasing

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

The `C*$* ASSERT RELATION` assertion includes variable names (*name* and *xx*). When specified globally, this assertion is used only when the variable names appear in `COMMON` blocks or are dummy arguments to a subprogram. You cannot use global assertions to make relational assertions about variables that are local to a subprogram.

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

```

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

```

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

```

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

```

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

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

Fine-Tuning Global Assumptions

Use the assertions described in this section to fine-tune the global assumptions discussed in “Controlling Global Assumptions” on page 41.

`C*$* ASSERT [NO] BOUNDS VIOLATIONS`

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

The `-pfa,-assert=b` command-line option acts like a global `C*$* ASSERT BOUNDS VIOLATIONS` assertion.

In the example below, the compiler assumes the first loop nest is standard conformant, and therefore optimizes both loops. The loops can be interchanged to improve memory referencing because no **A(I,J)** overwrites an **A(I,J+1)**. In the second nest, the assertion warns the compiler that the loop limit of the first array index (**I**) might violate declared array bounds. The compiler plays it safe and optimizes only the right array index.

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

Example 7-3 Bounds Violations

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

The example above becomes:

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

C*\$* ASSERT NO EQUIVALENCE HAZARD

The **C*\$* ASSERT NO EQUIVALENCE HAZARD** assertion tells the compiler that equivalenced variables will not be used to refer to the same memory location inside one **DO** loop nest. Normally, **EQUIVALENCE** statements allow different variable names to refer to the same storage location. The **-pfa,-assume=e** command-line option acts like a

global **C*\$* ASSERT NO EQUIVALENCE HAZARD** assertion. This assertion is active until reset or until the end of the program.

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

Example 7-4 Equivalence Hazard

```

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

```

The example above becomes:

```

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

```

C*\$* ASSERT [NO] TEMPORARIES FOR CONSTANT ARGUMENTS

Sometimes the compiler does not perform certain transformations when their effects on the rest of the program are unclear. For example, usually the **IF-to-intrinsic** transformation changes the following code

```

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

```

into the following:

```

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

```

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

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

it would appear that the value of the constant 1 was being reassigned. Without additional information, the compiler does not perform transformations within the subroutine.

Most compilers automatically put constant actual arguments into temporary variables to protect against this case. The **C**ASSERT TEMPORARIES FOR CONSTANT ARGUMENTS** assertion or the **-pfa,-assume=c** command-line option (the default) informs the compiler that constant parameters are protected.

The **NO** version directs the compiler to avoid transformations that might change the values of constant parameters.

Ignoring Data Dependencies

Power Fortran avoids running code in parallel that it believes to be data-dependent. Use the assertions described in the following sections to override this behavior.

C ASSERT DO (CONCURRENT)**

Use the **C** ASSERT DO (CONCURRENT)** assertion to tell Power Fortran to ignore assumed data dependencies. Normally Power Fortran is conservative about converting loops to run in parallel.

When Power Fortran determines that it can run a loop in parallel, it categorizes the loop into one of three groups:

1. yes, it is safe to run the loop parallel
2. no, it is not safe to run the loop in parallel
3. not sure (cannot be determined)

Normally, Power Fortran does not run the loops it is unsure about in parallel. It assumes there are data dependencies. **C** ASSERT DO (CONCURRENT)** tells Power Fortran to go ahead and run these “not sure” loops in parallel.

Note: If Power Fortran identifies a loop as containing definite data dependencies (as opposed to dependencies it assumes, but is not sure of), it does not run the loop in parallel even if you specify a **C** ASSERT DO (CONCURRENT)** assertion.

CDIR\$ IVDEP

Power Fortran interprets the Cray directive **CDIR\$ IVDEP** as if it were a **C** ASSERT DO (CONCURRENT)** assertion. Some dependencies that are safe to run on Cray hardware are not safe to run on Silicon Graphics hardware. Therefore, to avoid incorrect parallelization of loops, recognition of this assertion is turned off by default.

C ASSERT CONCURRENT CALL**

The **C** ASSERT CONCURRENT CALL** tells Power Fortran to ignore assumed dependencies that are caused by a subroutine call or a function reference. However, you must ensure that the subroutines and referenced functions are safe for parallel execution. This assertion applies to all subroutine and function references in the accompanying loop, which must appear on the next line.

CVD\$ CNCALL

Power Fortran interprets the VAST directive **CDIR\$ CNCALL** as if it were a **C** ASSERT CONCURRENT CALL** assertion. Some dependencies that are safe to run on Cray hardware are not safe to run on Silicon Graphics hardware. Therefore, recognition of this assertion is turned off by default.

C ASSERT NO RECURRENCE**

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

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

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

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

C ASSERT PERMUTATION**

The **C** ASSERT PERMUTATION(array)** assertion tells Power Fortran that *array* contains no repeated values. This assertion permits Power Fortran to run in parallel certain kinds of loops that use indirect addressing; for example:

```
DO I = 1, N
  A(INDEX(I)) = A(INDEX(I)) + B(I)
ENDDO
```

You can run this loop in parallel only if the array **INDEX** has no repeated values (so that each **INDEX (I)** is unique). Power Fortran cannot determine this, so it does not run such a loop in parallel. However, if you know that every element of **INDEX()** is unique, you can insert the following line before the loop to permit the loop to be run in parallel:

```
C** ASSERT PERMUTATION (INDEX)
```

Power Fortran Command-Line Options

This appendix contains the following sections:

- “Overview of Options”
- “Options Summary”

Overview of Options

This appendix lists and describes the options supported by Power Fortran. The default settings are satisfactory for most programs. However, you can alter the defaults to customize output.

Table 2-1 on page 11 summarizes the Power Fortran command-line options. In that table, the Reference column lists functional categories of the following options:

- parallelization
- general optimization
- inlining and interprocedural analysis
- advanced optimization
- directive control
- listing

The next three columns list the long names, short names, and default values of the options. Following the table is an explanation of each option, including the option’s long and short names, its default, and, if applicable, the long and short names for the **NO** version of the option. Although the options are listed in uppercase letters, you can specify them in lowercase as well.

Note: You can replace many of the Power Fortran command-line options described in this chapter with in-code directives.

Options Summary

Overview

This section alphabetically lists and defines the command-line options that uniquely affect Power Fortran.

concurrentize

The **-concurrentize** option, described in Table A-1, converts eligible loops to run in parallel.

Table A-1 concurrentize Option

Long Option Name	Short Option Name	Default Value
-concurrentize	-c	-concurrentize

See also “noconcurrentize” on page 92.

limit

The **-limit** option, described in Table A-2, reduces Power Fortran processing time by limiting the amount of time Power Fortran can spend trying to determine whether a loop is safe to run in parallel.

Table A-2 limit Option

Long Option Name	Short Option Name	Default Value
-limit= <i>n</i>	-lm= <i>n</i>	-limit=5000

Power Fortran estimates how much time is required to analyze each loop nest construct. If an outer loop looks like it would take too much time to analyze, Power Fortran ignores the outer loop and recursively visits the inner loops.

Larger limits often allow Power Fortran to generate parallel code for deeply nested loop structures that it might not otherwise be able to run safely in parallel. However, with larger limits Power Fortran can also take more time to analyze a program. (The limit does

not correspond to the **DO** loop nest level. It is an estimate of the number of loop orderings that Power Fortran can generate from a loop nest.)

lines

The **-lines** option, described in Table A-3, paginates the listing for printing.

Table A-3 lines Option

Long Option Name	Short Option Name	Default Value
-lines=<i>n</i>	-ln=<i>n</i>	-lines=55

Use this option to change the number of lines per page. Specifying **-lines=0** paginates at subroutine boundaries.

listoptions

The **-listoptions** option, described in Table A-4, specifies the information to include in the listing file (.l).

Table A-4 listoptions Option

Long Option Name	Short Option Name	Default Value
-listoptions=<i>list</i>	-lo=<i>list</i>	-listoptions=ol

list consists of any combination of the following letters:

- c Calling tree at the end of the program listing.
- i Transformed program file annotated with line numbers in the source program. Error messages and debugging information can refer to the original source rather than the transformed source. This option is automatically specified.
- k Power Fortran option used at the end of each program unit.
- l Loop-by-loop optimization table.
- n Program unit names, as processed, to the standard error file. This option is added automatically as part of an *f77 -v* compilation.
- o Annotated listing of the original program.

- p Processing performance statistics.
- s Summary of optimization performed.
- t Annotated listing of the transformed program.

minconcurrent

The **-minconcurrent** option, described in Table A-5, establishes the minimum amount of work needed inside the loop to make executing a loop in parallel profitable.

Table A-5 minconcurrent Option

Long Option Name	Short Option Name	Default Value
-minconcurrent= <i>n</i>	-mc= <i>n</i>	500

If the loop does not contain at least this much work, the loop will not be run in parallel. If the loop bounds are not constants, an **IF** clause will be automatically added to the **DOACROSS** directive generated by Power Fortran to test at run time whether sufficient work exists.

The value *n* is a count of the number of operations (for example, add, multiply, load, store) in the loop, multiplied by the number of times the loop is executed.

noconcurrentize

The **-noconcurrentize** option, described in Table A-6, prevents Power Fortran from converting loops to run in parallel.

Table A-6 noconcurrentize Option

Long Option Name	Short Option Name	Default Value
-noconcurrentize	-nconc	none

See also “concurrentize” on page 90.

noparallel

The **-noparallel** option, described in Table A-7, disables the parallelization of loops that contain I/O statements.

Table A-7 noparallel Option

Long Option Name	Short Option Name	Default Value
-noparallel	-nopio	option off

Use this option only on systems with parallel I/O capabilities or where I/O statements in loops are not executed.

See also “parallel” on page 93.

parallel

The **-parallel** option, described in Table A-8, enables the parallelization of loops that contain I/O statements.

Table A-8 parallel Option

Long Option Name	Short Option Name	Default Value
-parallel	-pio	option off

Use this option only on systems with parallel I/O capabilities or where I/O statements in loops are not executed.

See also “noparallel” on page 93.

sopt

The **-sopt** option, described in Table A-9, requests execution of the scalar optimizer.

Table A-9 sopt Option

Long Option Name	Short Option Name	Default Value
-sopt[<i>option,...</i>]	-sopt	option off

The **-sopt** option passes these options to Power Fortran:

`-pfa, -roundoff=0, -scaleropt=3, -optimize=5`

suppress

The **-suppress** option, described in Table A-10, lets you disable individual classes of Power Fortran messages that are normally included in the listing (.l) file.

Table A-10 suppress Option

Long Option Name	Short Option Name	Default Value
<code>-suppress=list</code>	<code>-su=list</code>	option off

These messages range from syntax warnings and error messages to messages about the optimizations performed. *list* is of any combination of the following:

- d data dependence
- e syntax error
- l information
- n not able to run loop in parallel
- q questions
- s standard messages
- w warning of syntax error (Power Fortran adds the **-suppress=w** option automatically if you specify the **-w** option to *f77*)

Power Fortran Directives

This appendix contains the following sections:

- “Standard Directives”
- “Cray Directives”
- “VAST Directives”

Chapter 1, “Overview of Power Fortran,” describes the purpose of directives. For details about how to use directives, refer to Chapter 7, “Fine-Tuning Power Fortran.”

Standard Directives

This section lists and describes the following standard Power Fortran directives:

- **C*\$* CONCURRENTIZE**
- **C*\$* LIMIT**
- **C*\$* MINCONCURRENT**
- **C*\$* NO CONCURRENTIZE**
- **C*\$* OPTIMIZE**
- **C*\$* ROUNDOFF**
- **C\$* DOACROSS**
- **C\$&**

C*\$* CONCURRENTIZE

The **C*\$* CONCURRENTIZE** directive converts eligible loops to run in parallel. This directive, if specified globally, has the same effect as the **-concurrentize** command-line option. See also the section called “**C*\$* NO CONCURRENTIZE**” on page 96.

C*\$* LIMIT

The **C*\$* LIMIT(n)** directive reduces Power Fortran processing time by limiting the amount of time Power Fortran can spend on trying to determine whether a loop is safe to run in parallel. Power Fortran estimates how much time is required to analyze each loop nest construct. If an outer loop looks like it would take too much time to analyze, Power Fortran ignores the outer loop and recursively visits the inner loops.

Larger limits often allow Power Fortran to generate parallel code for deeply nested loop structures that it might not otherwise be able to run safely in parallel. However, with larger limits Power Fortran can also take more time to analyze a program. (The limit does not correspond to the **DO** loop nest level. It is an estimate of the number of loop orderings that Power Fortran can generate from a loop nest.)

This directive, when specified globally, has the same effect as the **-limit** command-line option.

C*\$* MINCONCURRENT

The **C*\$* MINCONCURRENT(n)** option establishes the minimum amount of work needed inside the loop to make executing a loop in parallel profitable. *n* is a count of the number of operations (for example, add, multiply, load, store) in the loop, multiplied by the number of times the loop is executed. If the loop does not contain at least this much work, the loop is not run in parallel. If the loop bounds are not constants, an **IF** clause is automatically added to the Power Fortran-generated **C\$ DOACROSS** directive to test at run time if sufficient work exists.

C*\$* NO CONCURRENTIZE

The **C*\$* NO CONCURRENTIZE** option prevents Power Fortran from converting loops to run in parallel. See also **C*\$* CONCURRENTIZE**.

C*\$* OPTIMIZE

The **C*\$* OPTIMIZE(n)** directive sets the optimization level. The higher this level, the more code is optimized and the longer Power Fortran runs. Valid values for *n* are:

- 0 Avoids converting loops to run in parallel.
- 1 Converts loops to run in parallel without using advanced data dependence tests. Enables loop interchanging.

- | | |
|---|---|
| 2 | Determines when scalars need last-value assignment using lifetime analysis. Also uses more powerful data dependences tests to find loops that can run safely in parallel. This level allows reductions in loops that execute concurrently but only if the round-off setting is at least 2. |
| 3 | Breaks data dependence cycles using special techniques and additional loop interchanging methods, such as interchanging triangular loops. This level also implements special-case data dependence tests. |
| 4 | Generates two versions of a loop, if necessary, to break a data dependent arc. This level also implements more exact data dependence tests and allows special index sets (called wraparound variables) to convert more code to run in parallel. |
| 5 | Fuses two adjacent loops if it is legal to do so (no data dependencies) and if the loops have the same control values. In certain limited cases, this also recognizes arrays as local variables. Also tells Power Fortran to try harder to run the outermost loop possible (of a set of loops) in parallel. |

Note: If you want to use unrolling, set the optimize level to at least 4 (the default optimization level is above this threshold).

C*\$* ROUNDOFF

The **C*\$* ROUNDOFF(n)** directive controls whether Power Fortran runs a reduction operation in parallel. Valid values for *n* are as follows:

- | | |
|-----|---|
| 0-1 | Suppresses any round-off changing transformations. |
| 2 | Allows reductions to be performed in parallel; a common option. The valid reduction operators are addition, multiplication, <i>min</i> , and <i>max</i> . |
| 3 | Recognizes REAL induction variables. Permits memory management transformations. |

C\$ DOACROSS and C\$&

The **C\$ DOACROSS** directive tells the compiler to generate parallel code for the loop that immediately follows the directive. Putting this directive in the original source marks the loop to run in parallel and signals Power Fortran not to modify the loop. Power Fortran generates and inserts this directive as part of its parallelism analysis.

The **C\$&** directive continues the **C\$ DOACROSS** directive onto multiple lines.

Cray Directives

Power Fortran supports the following Cray directives:

- **CDIR\$ IVDEP**
- **CDIR\$ NEXT SCALAR**

CDIR\$ IVDEP

Power Fortran interprets the **CDIR\$ IVDEP** directive as if it were a **C**\$* ASSERT DO (CONCURRENT)** assertion. (Refer to Appendix C, “Power Fortran Assertions.”)

CDIR\$ NEXT SCALAR

CDIR\$ NEXT SCALAR is a Cray directive that generates scalar code for the next **DO** loop. Power Fortran interprets this directive as if it were a **C**\$* ASSERT DO(SERIAL)** assertion. (Refer to Appendix C, “Power Fortran Assertions,” for details.)

VAST Directives

Power Fortran supports the following VAST directives:

- **CVD\$ CNCALL**
- **CVD\$ CONCUR**

CVD\$ CNCALL

Power Fortran interprets the **CVD\$ CNCALL** directive as if it were the **C**\$* ASSERT CONCURRENT CALL** assertion (described in “**CVD\$ CNCALL**” in Chapter 7). The **CVD\$ CNCALL** directive tells Power Fortran to ignore assumed dependencies caused by a subroutine call or function reference.

CVD\$ CONCUR

Power Fortran interprets this directive as if it were the **C**\$* CONCURRENTIZE** directive (described in “Standard Directives” on page 95). The **CVD\$CONCUR** directive runs a loop in parallel to optimize performance.

Power Fortran Assertions

This appendix lists and describes the following Power Fortran assertions alphabetically:

- **C*\$* ASSERT CONCURRENT CALL**
- **C*\$* ASSERT DO (CONCURRENT)**
- **C*\$* ASSERT DO (SERIAL)**
- **C*\$* ASSERT DO PREFER (CONCURRENT)**
- **C*\$* ASSERT DO PREFER (SERIAL)**
- **C*\$* ASSERT [NO] LAST VALUE NEEDED**
- **C*\$* ASSERT NO RECURRENCE**
- **C*\$* ASSERT NO SYNC**
- **C*\$* ASSERT PERMUTATION**
- **C*\$* ASSERT RELATION**

This chapter describes the assertions that are supported by Power Fortran. Chapter 1, “Overview of Power Fortran,” describes the purpose of assertions.

For details about using assertions, refer to Chapter 7, “Fine-Tuning Power Fortran.”

C*\$* ASSERT CONCURRENT CALL

C*\$* ASSERT CONCURRENT CALL tells Power Fortran to ignore assumed dependencies that are due to a subroutine call or a function reference. However, you must ensure that the subroutines and referenced functions are safe for parallel execution. This assertion applies to all subroutine and function references in the immediately following loop.

C*\$* ASSERT DO (CONCURRENT)

The **C*\$* ASSERT DO (CONCURRENT)** assertion tells Power Fortran to ignore assumed data dependencies. Normally, Power Fortran is conservative about what loops it converts to run in parallel. When Power Fortran analyzes a loop to see if it is safe to run in parallel, it categorizes the loop into one of three groups:

- yes (loop is safe to run in parallel)
- no
- not sure

Normally, Power Fortran does not run “not sure” loops in parallel. **C*\$* ASSERT DO (CONCURRENT)** tells Power Fortran to go ahead and run “not sure” loops in parallel.

Note: If Power Fortran identifies a loop as containing definite (as opposed to assumed) data dependencies, it does not run the loop in parallel even if a **C*\$* ASSERT DO (CONCURRENT)** assertion precedes the loop.

C*\$* ASSERT DO (SERIAL)

The **C*\$* ASSERT DO (SERIAL)** assertion tells Power Fortran to run the specified loop serially. Power Fortran does not try to convert the specified loop to run in parallel. Nor does it try to run any enclosing loop in parallel. However, Power Fortran can still convert any loops nested inside the serial loop to run in parallel.

C*\$* ASSERT DO PREFER (CONCURRENT)

The **C*\$* ASSERT DO PREFER (CONCURRENT)** assertion runs a particular nested loop in parallel whenever possible. Power Fortran runs other nested loops in parallel only if a condition prevents running the selected loop in parallel.

The **C*\$* ASSERT DO PREFER (CONCURRENT)** assertion applies only to the **DO** loop that it precedes. Power Fortran does not generate parallel code if you use the **-noconcurrentize** command-line option or the **C*\$* NO CONCURRENTIZE** directive.

C ASSERT DO PREFER (SERIAL)**

The **C** ASSERT DO PREFER (SERIAL)** assertion indicates that you want to execute a **DO** loop in serial mode. This assertion directs Power Fortran to leave the **DO** loop alone, regardless of the setting of the optimization level. You can use this assertion to control which loop (in a nest of loops) Power Fortran chooses to run in parallel.

C ASSERT [NO] LAST VALUE NEEDED**

The compiler usually uses a temporary variable within an optimized loop when it assigns a scalar in a loop that is concurrentized. It then assigns the last value of the variable to the original scalar if it is possible that the scalar might be reused before it is assigned again. The **C** ASSERT NO LAST VALUE NEEDED** assertion lets the compiler assume that such last-value assignments are unnecessary. This assertion is active until reset or until the end of the program.

C ASSERT NO RECURRENCE**

The **C** ASSERT NO RECURRENCE(variable)** assertion tells Power Fortran to ignore all data dependencies associated with *variable*. Power Fortran ignores not just assumed dependencies (as with the **C** ASSERT DO (CONCURRENT)** assertion) but also real dependencies. Use this assertion to force Power Fortran to parallelize a loop when other, gentler means have failed. Use this assertion with caution, as indiscriminate use can result in illegal parallel code.

C ASSERT NO SYNC**

Sometimes when Power Fortran concurrentizes a loop, it adds unnecessary synchronization directives or other synchronization code. You can use the **C** ASSERT NO SYNC** assertion to eliminate synchronization overhead.

C ASSERT PERMUTATION**

The **C** ASSERT PERMUTATION(array)** assertion tells Power Fortran that *array* contains no repeated values. This assertion permits Power Fortran to run in parallel certain kinds of loops that use indirect addressing.

C*\$* ASSERT RELATION

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

Glossary

action summary

The portion of the listing file that summarizes Power Fortran actions.

assertion

A Power Fortran directive that asserts something about the program. For example, an assertion can assert that a particular array is a permutation vector. Power Fortran does not verify the validity of assertions.

data independence

When no iteration of a loop writes to a memory location that is read or written by any other iteration of that loop.

directive

A command, specified within the source file, that requests a particular action from Power Fortran. For example, directives enable, disable, or modify a feature of Power Fortran.

global assertion

An assertion that is placed on the first line of the input file. Power Fortran interprets global assertions as if they appeared at the top of each program unit in the file. *See also* assertion.

global directive

Directives that are placed on the first line of the input file. Power Fortran interprets global directives as if they appeared at the top of each program unit in the file. *See also* directive.

inlining

The process of replacing a call to an external routine with the actual code.

equivalent transformed source file

A transformed version of a Fortran source program generated by Power Fortran. The name of this file has the suffix **.m**, such as *analysis.m*.

interprocedural analysis (IPA)

The process of analyzing an external routine ahead of time and using the results when the routine is referenced.

listing file

An annotated listing of the parts of a source program that can and cannot run in parallel on multiple processor generated by Power Fortran. This file has the suffix **.1**.

max reduction

A reduction that uses the *max()* intrinsic function. *See also* reduction.

min reduction

A reduction that uses the *min()* intrinsic function. *See also* reduction.

parallelize

Manipulating code so that it can be run in parallel.

permutation index

A permutation vector used to index into an array. Because all the numbers in the permutation vector are different, when used as indexes they all refer to different array elements.

permutation vector

Any list of numbers that are all different.

Power Fortran 77

A Fortran 77 compiler that analyzes a program, identifies loops that are safe to run in parallel (that is, they do not contain data dependencies), and generates a parallel version of the program.

product reduction

A reduction that uses the multiply operator ***. *See also* reduction.

profiling

A process that produces detailed information about program execution, such as details about areas of code where most of the execution time is spent. The *prof(1)* command produces profiling information.

reduction

An operation that reduces a set of values to one value.

round-off error

The inaccuracy resulting from rounding off values in a calculation.

sum reduction

A reduction that uses the add operator +. *See also* reduction.

WorkShop Pro MPF

An optional product that provides a graphical interface to the analysis performed by Power Fortran.

Index

A

- action summary, 21, 103
- aggressive option, 52
- aliasing, 82
- alignment
 - of COMMON blocks, 52
- .anl file, 10
- ANSI-X3H5 standard, 4
- arclimit option, 53
- argument aliasing, 41
- assertions
 - C*\$* ASSERT CONCURRENT CALL, 87, 99
 - C*\$* ASSERT DO (CONCURRENT), 86, 100
 - C*\$* ASSERT DO (SERIAL), 79, 100
 - C*\$* ASSERT DO PREFER (CONCURRENT), 81, 100
 - C*\$* ASSERT DO PREFER (SERIAL), 80, 101
 - C*\$* ASSERT LAST VALUE NEEDED, 101
 - C*\$* ASSERT NO RECURRENCE, 87, 101
 - C*\$* ASSERT NO SYNC, 101
 - C*\$* ASSERT PERMUTATION, 88, 101
 - C*\$* ASSERT RELATION, 82, 102
 - C*\$* ASSERT TEMPORARIES FOR CONSTANT ARGUMENTS, 85
 - definition, 103
 - duration of, 6
 - enabling recognition of, 57
 - overview, 71
 - purpose of, 6
- assumed dependences, 72

- assume option, 41, 82
- assumptions
 - controlling globally, 41

B

- be* back end process, 13

C

- C\$&, 73, 97
- C*\$* ARCLIMIT, 74
- C*\$* ASSERT CONCURRENT CALL, 87, 99
- C*\$* ASSERT DO (CONCURRENT), 86, 100
- C*\$* ASSERT DO (SERIAL), 79, 100
- C*\$* ASSERT DO PREFER (CONCURRENT), 81, 100
- C*\$* ASSERT DO PREFER (SERIAL), 80, 101
- C*\$* ASSERT LAST VALUE NEEDED, 101
- C*\$* ASSERT NO RECURRENCE, 87, 101
- C*\$* ASSERT NO SYNC, 101
- C*\$* ASSERT PERMUTATION, 88, 101
- C*\$* ASSERT RELATION, 82, 102
- C*\$* ASSERT TEMPORARIES FOR CONSTANT ARGUMENTS, 85
- C*\$* CONCURRENTIZE, 80, 95
- C*\$* EACH_INVARIANT_IF_GROWTH, 74
- C*\$* INLINE, 78
- C*\$* LIMIT, 96

- C*\$* MAX_INVARIANT_IF_GROWTH, 74
- C*\$* MINCONCURRENT, 96
- C*\$* NO CONCURRENTIZE, 80
- C*\$* NO INLINE, 78
- C*\$* NO IPA, 79
- C*\$* NO SYNC, 74
- C*\$* OPTIMIZE, 75, 96
- C*\$* ROUND OFF, 76, 97
- C*\$* SCALAROPTIMIZE, 77
- cache
 - setting up page mapping, 55
 - specifying size, 54
 - specifying width of memory channel, 54
- cacheline option, 54
- cachesize option, 54
- CDIR\$ IVDEP, 87, 98
- CDIR\$ NEXT SCALAR, 80, 98
- C\$ DOACROSS, 73, 97
- COMMON blocks
 - aligning, 52
- compiler options
 - pfa, 13, 39
- compiling programs with Power Fortran, 10
- concurrentize option, 34, 90
- controlling code execution, 34
 - running code in parallel, 34
 - specifying a work threshold, 34
- Cray directives
 - CDIR\$ IVDEP, 87, 98
 - CDIR\$ NEXT SCALAR, 98
- customizing execution, 33
 - controlling code execution, 34
 - overview, 33
- CVD\$ CNCALL, 87, 98
- CVD\$ CONCUR, 81, 98

D

- data dependencies
 - ignoring, 86
- data independence, 103
- default listing information interpretation
 - action summary, 21
 - DO loop marking, 20
 - field descriptions, 19
 - footnotes, 20
 - line numbers, 19
 - syntax error/warning messages, 21
 - viewing the listing file, 19
- dependences
 - assumed, 72
- directives
 - C&#, 73, 97
 - C*\$* ARCLIMIT, 74
 - C*\$* CONCURRENTIZE, 80, 95
 - C*\$* EACH_INVARIANT_IF_GROWTH, 74
 - C*\$* INLINE, 78
 - C*\$* LIMIT, 96
 - C*\$* MAX_INVARIANT_IF_GROWTH, 74
 - C*\$* MINCONCURRENT, 96
 - C*\$* NO CONCURRENTIZE, 80, 96
 - C*\$* NO INLINE, 78
 - C*\$* NO IPA, 79
 - C*\$* NO SYNC, 74
 - C*\$* OPTIMIZE, 75, 96
 - C*\$* ROUND OFF, 76, 97
 - C*\$* SCALAROPTIMIZE, 77
 - CDIR\$ IVDEP, 87, 98
 - CDIR\$ NEXT SCALAR, 80, 98
 - C\$ DOACROSS, 73, 97
 - CVD\$ CNCALL, 87, 98
 - CVD\$ CONCUR, 81, 98
 - definition, 103
 - enabling recognition of, 57
 - overview, 70
 - purpose of, 3

-directives option, 57

DO loop

marking in listing file, 20

double precision registers, 55

-dpreregisters option, 55

E

-each_invariant_if_growth option, 42

equivalent transformed source file, 103

error messages

in listing file, 21

example

Power Fortran command line, 13

F

fef77, 13

fef77p, 13

fine-tuning inlining and IPA, 78

floating point registers, 55

footnotes

in listing file, 20

formatting the listing file, 17

-fpreregisters option, 55

fsplit, 10

function call

generated by Power Fortran, 26

-fuse option, 41

G

global assertion, 103

global assumptions

controlling, 41

global directive, 103

I

indirect indexing, 24

-inline_and_copy option, 61

-inline_create option, 66

-inline_from_files option, 64

-inline_from_libraries option, 64

inlining, 59, 103

enabling with options, 60

fine-tuning, 78

performing, 38

specifying routines, 61

internal table size

controlling, 53

interprocedural analysis

performing with options, 60

interprocedural analysis (IPA), 59, 104

fine-tuning, 78

performing, 38

specifying routines, 61

invariant IF floating, 42, 74

-ipa_create option, 66

-ipa_from_files option, 64

-ipa_from_libraries option, 64

- L**
- limit option, 36, 90
 - lines option, 17, 91
 - listing file, 10, 104
 - action summary, 21
 - error/warning messages, 21
 - field descriptions, 19
 - footnotes, 20
 - include options, 17
 - interpreting default information, 19
 - samples, 23-30
 - viewing, 19
 - listing file formatting, 17
 - disabling message classes, 18
 - paginating the listing, 17
 - specifying information to include, 17
 - listoptions option, 17, 19, 91
 - loop blocking, 53
 - loop fusion, 41
 - loop unrolling, 53
 - enabling, 55
- M**
- max_invariant_if_growth option, 42
 - max reduction, 104
 - memory channel
 - specifying width, 54
 - memory management transformations, 53
 - options, 54
 - techniques, 53
 - messages
 - in listing file, 21
 - .m file, 10, 103
 - minconcurrent option, 92
 - min reduction, 104
- N**
- noassume option, 41
 - noconcurrentize option, 34, 92
 - noperallelio option, 93
- O**
- optimization
 - setting levels, 36
 - optimizations
 - aggressive, 52
 - changing levels, 75
 - controlling internal table size, 53
 - controlling levels, 44
 - invariant IF floating, 42
 - loop blocking, 53
 - loop fusion, 41
 - loop unrolling, 53, 55
 - memory management transformations, 53
 - recursion, 58
 - scalar, 77
 - optimize option, 36, 44
 - and -O compiler option, 44
 - optimizing
 - inlining and IPA, 59
 - overview of Power Fortran, 1
- P**
- paginating the listing file, 17
 - parallelio option, 35, 93
 - parallelize, 104
 - permutation index, 104
 - permutation vector, 104
 - pfa compiler option, 13
 - pfa option, 10, 11

- aggressive, 52
 - and scalar optimizations, 39
 - arclimit, 53
 - assume, 41, 82
 - cacheline, 54
 - cachesize, 54
 - directives, 57
 - dpregisters, 55
 - each_invariant_if_growth, 42
 - fpregisters, 55
 - fuse, 41
 - inline_create, 66
 - inline_from_files, 64
 - inline_from_libraries, 64
 - ipa_create, 66
 - ipa_from_files, 64
 - ipa_from_libraries, 64
 - max_invariant_if_growth, 42
 - optimize, 44
 - recursion, 58
 - roundoff, 45
 - scaleropt, 48
 - setassociativity, 55
 - unroll, 55
 - unroll2, 55
 - Power Fortran, 1, 104
 - action summary, 21
 - assertions, 99
 - purpose of, 6
 - circumventing, 73
 - command-line example, 13
 - command-line options, 3, 89
 - command-line syntax, 10
 - compiling with, 10
 - controlling code transformations, 36
 - customizing execution, 34
 - directives, 95-98
 - purpose of, 3
 - interpreting listing, 19
 - output files, 10
 - overview of operation, 1
 - overview of usage, 9
 - running from f77, 13
 - strategy for using, 2
 - summary, 7
 - table of action abbreviations, 22
 - utilizing output, 15
 - Power Fortran option
 - concurrentize, 34, 90
 - limit, 36, 90
 - lines, 17, 91
 - listoptions, 17, 91
 - minconcurrent, 92
 - noconcurrentize, 34, 92
 - noparallel, 93
 - optimize, 36
 - parallel, 35, 93
 - pfa, 10, 11
 - roundoff, 37
 - sopt, 93
 - suppress, 18, 94
 - product reduction, 104
 - profiling, 104
- ## R
- recursion
 - enabling, 58
 - recursion option, 58
 - reductions
 - definition, 105
 - example of, 28
 - sum, 31
 - types of, 31
 - registers
 - double precision, 55
 - floating point, 55

round off
 controlling from command line, 45
 controlling variations, 37
 error, 105
-roundoff option, 31, 37, 45
 and -O compiler option, 46
running code in parallel, 34, 80
running code serially, 79

S

sample listing files, 23
 function call, 26
 indirect indexing, 24
 reductions, 28
scalar optimizations
 controlling levels, 48
 controlling with directives, 77
 fine tuning, 74
-scaleropt option, 48
 and -O compiler option, 48
-setassociativity option, 55
setting optimization level, 36
-sopt option, 93
specifying a complexity limit, 36
specifying a work threshold, 34
standard directives, 95-97
 See also directives.
strategy for using Power Fortran, 2
sum reduction, 31, 105
-suppress option, 18, 94
syntax conventions, xvii

T

tiling, 54

U

-unroll2 option, 55
-unroll option, 55

V

VAST directives, 98
 CVD\$ CNCALL, 87, 98
 CVD\$ CONCUR, 98
 See also directives.
viewing the listing file, 19

W

warning messages
 in listing file, 21
WorkShop Pro MPF, 2, 105
 producing input file, 10
work threshold
 specifying, 34

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 007-2363-002.

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