POWER Fortran Accelerator™ User's Guide

Introduction

This guide describes the features of the Silicon Graphics POWER Fortran Accelerator ™ (PFA). For details about analyzing a program and converting it for use on a multiprocessor system, refer to Chapter 5, "Fortran Enhancements for Multiprocessors," of the *Fortran 77 Programmer's Guide.*

Organization

This guide contains the following chapters and appendixes:

Chapter 1, "Overview of PFA," explains the basic mechanism for invoking PFA and includes a description of PFA's listing and intermediate files.

Chapter 2, "How to Use PFA," explains how to use PFA directly and as part of a Fortran compile.

Chapter 3, "Utilizing PFA Output," explains output produced by PFA: the intermediate file and the listing file.

Chapter 4, "Customizing PFA Execution,"describes how to use command line options to optimize PFA execution.

Chapter 5, "Fine−Tuning PFA," describes how to optimize code by using PFA directives and assertions.

Appendix A, "PFA Command Line Options," lists the five types of PFA command line options: parallelization, optimization, Fortran 77 language control, directives, and listing.

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

Appendix C, "PFA Assertions," lists the PFA assertions you can include in a program to provide information that PFA 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 PFA.

Related Documentation

The following documents contain information relevant to PFA:

- *Fortran 77 Programmer's Guide,* Silicon Graphics, Inc., document number 007−0711−030.
- *Fortran 77 Language Reference Manual*, Silicon Graphics, Inc., document number 007−0710−040.
- *IRIS−4D Series Compiler Guide*, Silicon Graphics, Inc., document number 007−0905−030.

Typographical Conventions

This guide uses the following conventions and symbols:

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

Bold	Indicates literal command line options, filenames, keywords, function/subroutine names, pathnames, and directory names.
Italics	Represents user−defined values. Replace the item in italics with a legal value. Italics are also used for command names, manual page names, and manual titles.
`Courier`	Indicates command syntax, program listings, computer output, and error messages.
`Courier bold`	

[ ]	Enclose optional command arguments.
()	Surround arguments or are empty if the function has no arguments following function/subroutine names. Surround manual page section in which the command is described following IRIX commands.
|	Sseparates 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 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*$* NOIPA** or **C*$*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
```

Chapter 1

Overview of PFA

This chapter contains the following sections:

- **"Overview"** describes how PFA operates and suggests procedures for using it.
- **"Strategy for Using PFA"** explains when and how to use PFA.
- **"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 PFA.

Overview

PFA is a Fortran 77 source−to−source preprocessor that enables you to run existing Fortran 77 programs efficiently on the Silicon Graphics POWER Series™ multiprocessor systems. PFA analyzes a program and identifies loops that do not contain data dependencies. Such loops are safe to execute in parallel (concurrently). PFA automatically inserts special compiler directives in a modified copy of the original source code. (PFA produces a number of files containing code and other information you need to run a program concurrently on multiple processors.)

Interpreting the PFA−generated compiler directives, the Silicon Graphics Fortran 77 compiler can generate code to split loop processing across all the available multiple processors. Because the directives inserted by PFA look like standard Fortran 77 comment statements, PFA does not affect the portability of the code to non-Silicon Graphics, Inc. (SGI), systems.

In addition, you do not need a multiprocessor system to develop under PFA (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 PFA on any IRIS−4D™ Series workstation (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. (You can also manually specify the number of processors to use; see the *Fortran 77 Programmer's Guide*.) However, simply passing code through PFA rarely produces all the increased performance available. There are often easily removed data dependencies that prevent PFA from running a loop in parallel. Using the listing file, optionally generated by PFA, you can find the real or potential data dependencies that prevented PFA from running a loop in parallel. Refer to **Chapter 3, "Utilizing PFA 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 PFA to run the code in parallel by inserting PFA assertions. These assertions look like Fortran 77 comments.

With PFA, you select the code to convert to run in parallel. Thus, you can convert the whole program or key parts of it by adding PFA directives manually or by having PFA convert only selected files. In addition, you can run PFA on some, all, or none of a program's source files. The object files produced using PFA 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.

Strategy for Using PFA

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

Knowing when and where to modify your code means understanding the information in the PFA listing. Understanding the PFA 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 PFA Output"** ," for information.

PFA analyzes a program for data dependence. During this analysis, PFA 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.

When PFA finds a loop with data independence, PFA knows it can safely run the loop in parallel. When PFA 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 PFA cannot run the loop in parallel, the listing file will explain where PFA encountered problems.

Command Line Options

To customize the way PFA executes an entire program, you can specify various command line options when you run PFA directly or when you specify PFA as part of a compile (**Chapter 2, "How to Use PFA,"** explains both procedures). The five functional categories of command line options are

- parallel execution
- general optimization
- Fortran 77 language control
- directive control
- listing

Chapter 4, "Customizing PFA Execution," explains when and how to use the various options, and **Appendix A, "PFA Command Line Options,"** provides a complete summary.

Directives

PFA directives enable, disable, or modify a feature of PFA. 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.

PFA directives allow you to specify PFA options in addition to, or instead of, command line options. Directives placed on the first line of the input file are called *global directives*. PFA 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. PFA 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 PFA directives have corresponding command line options. If you specify conflicting settings in the command line and a directive, PFA 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, PFA uses the minimum of the command line setting and the directive setting.

Table 1–1 lists the directives supported by PFA. In addition to the standard directive, PFA supports the Cray™ and VAST™ directives listed in the table. PFA maps these directives to corresponding PFA assertions. Refer to **Chapter 5, "Fine–Tuning PFA"** ," for details.

Table 1–1 PFA Directives

Standard	Cray	VAST
C*$*ARCLIMIT(*n*)	CDIR$ NEXT SCALAR	CVD$ CONCUR
C*$*CONCURRENTIZE	CDIR$ IVDEP	CVD$LSTVAL
C*$*INLINE		CFVD$NOLSTVAL
C*S*IPA		
C*$*LIMIT(*n*)		
C*$*MINCOMCURRENT(*n*)		
C*$*NONCONCURRENTIZE		
C*$*NOINLINE		
C*$*NOIPA		
C*$*OPTIMIZE(*n*)		
C*$*ROUNDOFF(*n*)		
C*$*SCALAR OPTIMIZE(*n*)		
C*$*UNROLL(*n*)		
C*$*UNROLL(*n*,*m*)		
C$DOACROSS		
C$&		

Refer to **Appendix B, "PFA Directives,"** for a list and description of PFA directives.

Assertions

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

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

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

C*$* ASSERT RELATION (*name .xx. name*) assertions 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 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 1–2 lists PFA assertions and their duration.

Table 1–2 PFA Assertions and Their Duration

Assertion	Duration
C*$* ASSERT DO (SERIAL)	Next Loop
C*$* ASSERT DO (CONCURRENT)	Next Loop
C*$* ASSERT DO PREFER (SERIAL)	Next Loop

C*$* ASSERT DO PREFER (CONCURRENT)	Next Loop
C*$* ASSERT [NO] EQUIVALENCE HAZARD	Until Reset
C*$* ASSERT [NO] ARGUMENT ALIASING	Until Reset
C*$* ASSERT RELATION (*name .xx. name*)	Next Loop
C*$* ASSERT CONCURRENT CALL	Next Loop
C*$* ASSERT NO RECURRENCE	Next Loop
C*$* ASSERT PERMUTATION (*name*)	Next Loop

Summary

PFA provides information about the dependencies of loops in a Fortran 77 program. Often, PFA can use the information to run loops in parallel automatically. But when PFA 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 PFA 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 5 in the *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.

Chapter 2

How to Use PFA

This chapter contains the following sections:

- **"Overview"** describes how to prepare for using PFA.
- **"Compiling Programs With PFA"** explains how to run PFA as part of a Fortran compile.
- **"Using PFA Directly"**explains how to run PFA independent of the Fortran driver.

Overview

Simply running a program through PFA might buy you some improved performance, but you can get far more if you understand the PFA 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 in parallel a routine that uses only 1 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−sample profiling (through the **−p** option to *f77*(1)) or basic block profiling (through *pixie*(1)). Refer to Chapter 2, "Improving Program Performance," of the *IRIS−4D Compiler Guide* for details about profiling.

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 PFA 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 PFA will do a good job with the existing program. You will save time by letting PFA do what it can. You can then focus on those routines where PFA had a problem. One situation in which PFA 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 PFA. 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 PFA listing identifies the loops that PFA can and cannot run in parallel. For loops that cannot run in parallel, the PFA listing also tells you why it could not convert the loop for parallel execution.

Compiling Programs With PFA

The following is the command line syntax for compiling a Fortran 77 program with PFA and command line options. You can pass these options to PFA by adding the **-WK** option to the *f77* command line. It invokes the various processing phases that compile, optimize, assemble, and link edit the program. For more information about the **-WK** option, see the *f77*(1) manual page.

Syntax

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

where

-pfa	Invokes the POWER Fortran Accelerator, *pfa*. Enables any multiprocessing directives.
list	Runs *pfa* and generates 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	Runs *pfa*, generates the listing file (**.l**), and saves the intermediate transformed Fortran 77 program. The intermediate file has the suffix **.m**.
-WK	Passes the specified command line options to PFA. Do not enter spaces between **–WK** and any of the hyphens, *options*, equal signs, and *values* that follow it.
-*option*	Specifies a PFA command line option listed in **Table 2–1** , for example, **–IGNOREOPTIONS**.
value	Specifies a value for a command line option, for example, 10.
-pfaprepass	Passes the code through PFA an extra time. The first time through (the prepass), PFA uses the options specified in the **-pfaprepass** option but does not insert **C$ DOACROSS** directives. The output of this operation is then passed back through PFA, using the options specified in the **-WK** option. Only rarely should you need to use this option, and there is good reason to avoid it. Normally, PFA does all it can in a single run–through. In rare circumstances an extra pass can be beneficial. However, the PFA algorithms do not necessarily converge, and multiple passes over the code can change it for the worse.The syntax of this option is the same as the **–WK** option.
*filename***.f**	Specifies the Fortran 77 source program. The filename must always use the **.f** suffix.

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

Note: You can replace many of the PFA command line options listed in **Table 2–1** with in–code directives. For information on these directives, see **Chapter 5, "Fine–Tuning PFA,"** and **Appendix B, "PFA Directives."**

Table 2–1 PFA Command Line Options

Reference	Long Name	Short Name
Parallelization	[NO]CONCURRENTIZE	[N]CONC
	MINCONCURRENT=*n*	MC=*n*
Optimization	ARCLIMIT	ARCLM=*n*
	LIMIT=*n*	LM=*n*
	OPTIMIZE=*n*	O=*n*
	ROUNDOFF=*n*	R=*n*
	SCALAROPT=n	SO=*n*
	UNROLL=*n*	UR=*n*
	UNROLL2=*n*	UR22=*n*
Fortran 77 Language Control	ASSUME=*list*	AS=*list*
	[NO]DLINES	[N]DL

	[NO]ONETRIP	[N]I	
	SAVE=*c*	SV=*c*	
	SCAN=*n*	SCAN=*n*	
	SYNTAX=*c*	SY=*c*	
			(
Inlining and	INLINE[=*list*]	*INIPA*INCR=*name*	*(
Interprocedural	IPA[=*names*]	IPACR=*name*	
Analysis	INLINE_CREATE=*name*	INFF=*list*	(
	IPA_CREATE=*name*	IPAFF=*list*	
	INLINE_FROM_FILES=*list*	INFL=*list*	
	IPA_FROM_FILES=*list*	IPAFL=*list*	(
	INLINE_FROM_LIBRARIES=*l ist*	INLL=*n*	
	IPA_FROM_LIBRARIES=*list*	IPALL=*n*	
	INLINE_LOOP_LEVEL=*n*	INM	(
	IPA_LOOP_LEVEL=*n*	IPAM	
	INLINE_MAN	IND	
	IPA_MAN		(
	INLINE_DEPTH		
			(
			(
			(
			*(
			I
			I
			(
			I
			I
			I
Directives	[NO]DIRECTIVES=*list*	[N]DR=*list*	
I/O	INPUT=file.f	*file*.f	*f
	[NO]FORTRAN=file	[N]F=*file*	*il
	[NO]LIST=file	[N]L=*file*	
			.
			f
			*f
			il
			.

Listing	LINES=*n*	LN=*n*
	LISTOPTIONS=*list*	LO=*list*
	SUPPRESS=*list*	SU=*list*
Obsolete	CREATE	CR
	LIBRARY=*file*	LIB=*file*
	[NO]EXPAND=*list*	EX=*list*
	LIMIT2=*n*	LM2=*n*

Example

To compile the Fortran 77 program **prog.f** with PFA and the **–UNROLL=8** option, enter

```
% f77 -pfa -WK,-UNROLL=8 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.) PFA then takes the *cpp* output and inserts code that runs data–independent loops in parallel. PFA can also generate a listing file (with the **.l** suffix) and an intermediate file (with the **.m** suffix). For details, refer to **Chapter 3, "Utilizing PFA Output."**

Finally, the Fortran 77 compiler, *f77*, compiles the transformed PFA–generated file to produce an object file.

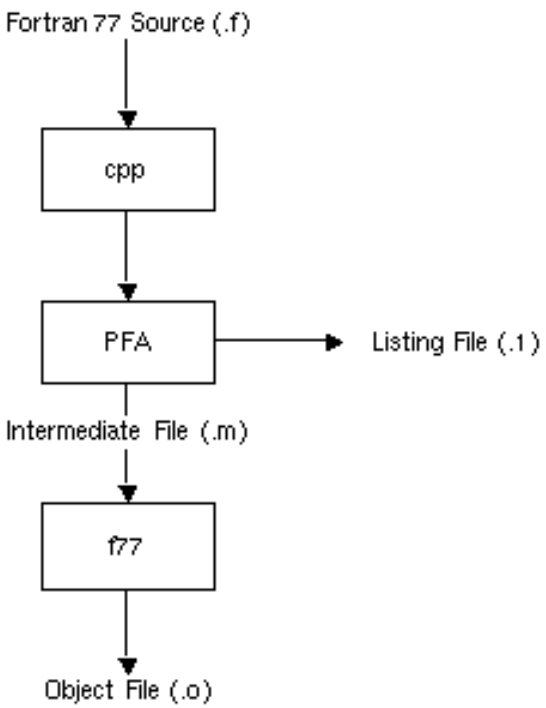

Figure 2–1 Compiling With PFA

Using PFA Directly

Although you normally run PFA as part of an *f77* compile, the two instances when you should run PFA directly are

- When creating an inlining or IPA library (refer to **Chapter 4, "Customizing PFA Execution."**)
- If you want to "capture" the output of PFA and review it to determine further optimizations

Running the *pfa*(1) command directly, using the following syntax, produces both the **.m** and the **.l** files.

Syntax

`/usr/lib/pfa [-option [-option]...] filename.f`

where

–option	Specifies a PFA command line option listed in **Table 2–1** , for example, **–INLINE**.
filename.**f**	Specifies the Fortran 77 source program. The filename must have the **.f** suffix.

Example

The following command runs PFA directly using the **–unroll** and **–roundoff** options:

```
% /usr/lib/pfa -ur=4 -r=2 sample.f
```

Chapter 3

Utilizing PFA Output

This chapter contains the following sections:

- **"Overview"** discusses the PFA 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 sample listing files along with an interpretation of each.

Overview

PFA generates two files, a listing file (**.l**) and an intermediate file (**.m**). Invoking PFA as part of a Fortran compilation produces a line–numbered listing file when you use the **–pfa list** option. If you specify the **–keep** option, PFA produces both the numbered listing file and the intermediate file. PFA automatically produces both files when you invoke it directly. (For details about invoking PFA, refer to **Chapter 2, "How to Use PFA."**)

For example, consider the following program, **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 keep sample.f
```

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

```
Actions   Do Loops Line
DIR                  1 # 1  "sample.f"
                     2       subroutine sample(a,b,c)
```

```
                        3       dimension a(1000),b(1000),c(1000)
c         +--------  4       do 10 i = 1,1000
            *________   5  10   a (i) = b(i) + c(i)
                        6       end
Abbreviations Used
     DIR    directive
     C      concurrentized
Loop Summary

         From  To    Loop     Loop
Loop#  line  line  label    index   Status
1      4     5     DO 10    I       concurrentized
```

and the intermediate file, **sample.m**:

```
#  1  "sample.f"
#  1  "sample.f"
              subroutine sample(a,b,c)
              DIMENSION A(1000), B(1000), C(1000)
#  3  "sample.f"
C$DOACROSS SHARE(A,B,C),LOCAL(I)
#  3  "sample.f"
              DO 2 I=1,000
#  4  "sample.f"
              A(I) = B(I) + C(I)
#  4  "sample.f"
              2 CONTINUE
              end
```

PFA 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 PFA 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.

PFA inserted the statement starting with **C$DOACROSS** before the **DO** statement in the intermediate file, **sample.m**. The Fortran 77 compiler directive **C$DOACROSS** tells *f77* that the next **DO** loop can run in parallel. The phrase **SHARE (A,B,C)** informs the Fortran 77 compiler that all processes that execute the **DO** loop share the arrays **A**, **B**, and **C**. The phrase **LOCAL(I)** indicates that every process executing the **DO** loop keeps a local variable **I**. The lines of the form **# 4 "sample.f"** are called line number directives. They relate the transformed source back to the original source.

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

Formatting the Listing File

You customize a PFA 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 to change the number of lines per page. Specifying **–LINES=0** paginates at subroutine boundaries.

If you do not specify the **–LINES** option, PFA 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** .

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. Running PFA as part of an *f77* compile automatically adds this option.
K	Print out of the PFA 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 *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.

Disabling Message Classes

Use the **–SUPPRESS=***list* option (or **–su=***list*) to disable individual classes of PFA 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 (PFA adds the **–SUPPRESS=W** option automatically if you use the **–w** option to *f77*)

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

Interpreting Default Listing Information

Knowing when and where to modify your code means understanding the information in the PFA 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 PFA–generated listing file lists the optimizations PFA made to the code. For example, a message could say that, although three loops could have run in parallel, PFA 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 **.l** file when you use the default values for the **–LISTOPTIONS** command line option (that is, **O** and **L**).

A default PFA file listing includes

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

Line Numbers

A statement in the PFA listing 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 PFA–transformed program listing and when debugging. PFA 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 PFA–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. The PFA surrounds each **DO** loop (up to nest level 10) with a loop delimiter character. Each character listed in **Table 3–3** has a specific meaning.

Table 3–3 Listing File DO Loop Delimiters

Character	**Denotes**
|	Generic DO loop
*	PFA 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,

```
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

PFA uses the footnotes listing to give important details concerning its actions. PFA 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 PFA's 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 will usually be 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 PFA translates or modifies a statement, it uses abbreviations in the Actions column of the listing file to identify the statements. PFA 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 and no detailed messages follow. All other classes have associated messages.

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

Table 3–4 PFA Action Abbreviations

Value	Meaning
DD	(Data Dependence) Indicates that data dependence prevented PFA 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, PFA 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. PFA cannot do anything with such code. The intermediate (**.m**) file contains a copy of this program unit that PFA has not modified.
EX	(Extension) Shows where a construct in the original program is not allowed in the language PFA 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 PFA added a statement.
LR	(Loop Reordering) Indicates that PFA has modified a Fortran 77 statement in the process of interchanging loops. If during optimization PFA ascertains that an outer loop would be more efficient as an inner loop, and it can legally reorder the loops, PFA places the outer loop inside. In the process of this reordering, PFA 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 PFA information has been lost. This message does not always mean that something is wrong with the program.
NX	(Nonconcurrent Statement) Indicates that PFA did not try or was unable to run the statement in parallel. For example, when a subroutine call is involved in a loop, PFA generates this message.
NO	(Program Too Large—Not Optimized) Indicates that the program unit being processed is too large for PFA to optimize, because of PFA's data structure size limitations. When PFA optimizes programs, it adds statements that might also overflow the fixed–size tables. In either case, PFA stops optimization and passes the original program to the intermediate (**.m**) file, informing you of this action. For PFA to process the unit, you must split the program into smaller sections.
OE	(Option Error) Indicates a syntax error in a PFA option. This error does not stop 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 PFA 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 PFA has optimized a scalar loop.

STD (Standardized) Marks where PFA 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 to a **DO** loop.

TE (Translator Error) Indicates an internal PFA error. PFA writes the notification to the standard error file and writes a trace back to the output file. Notify SGI if you see this sort of bug (so it can be corrected) and, if possible, send SGI 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 PFA 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

PFA 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 Fortran 77 code

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

when submitted to PFA, results in the listing file

```
                          10
                          11
                          12    subroutine foo2(w,b,index,n)
                          13    real w(n), b(n)
                          14    integer index(n)
                          15
1 Q       +-------     16    do i = 1, n
2 DD      !            17      w(index(i)) = w(index(i)) + b(i)
            !_______     18    enddo
                          19    end
Abbreviations Used
DD   data dependence
Q   question
Footnote List
1: question           Is INDEX a permutation vector?
2: data dependence    Data dependence involving this line due
                        to variable W.
DO Loop Summary
loop# from   to   DO label index   workload status
```

```
1     16     18   DO       I       dependencies prevent
                                      parallelism
```

DD in the Actions column on line 17 of the listing warns that the variable **w** 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 **w**. 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 **w**). 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

```
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 PFA finds the loop safe to run in parallel (indicated by the * **DO** loop delimiter)

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

Abbreviations Used

DIR   directive
C     concurrentized

Loop Summary

          From   To     Loop    Loop
Loop#   line   line   label   index   Status
1       7      9      Do      I       concurrentized
```

Note: As with all assertions, PFA does not verify the truth of this assertion. When you make an assertion, be certain that the assertion 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
do i = 1, n
a(i) = force (b(i), c(i))
enddo
end
```

generates the listing

```
Actions DO Loops     Line
DIR                  1 #  1 "foo3.f"
                       2    subroutine foo3(a,b,c,n)
                       3    real a(n), b(n), c(n)
                       4    external force
                       5
NCS     +------        6    do i = 1, n
NO NCS  !              7       a(i) = force(b(i), c(i))
          !______      8    enddo
                       9    end
Abbreviations Used
NO   not optimized
DIR   directive
NCS   non-concurrent-stmt
Footnote List
1: not optimized       No optimizable statements found.
2: not optimized       Unoptimizable call to "FORCE" found.
Loop Summary
          From   To     Loop    Loop
Loop#   line   line   label   index   Status
1       6      8      Do      I       unoptimizable
                                         call (FORCE)
```

Calling the function **force** prevents PFA from automatically running the loop in parallel. PFA identifies the function call as a **non−concurrent−stmt**. By its nature, a nonconcurrent statement prevents PFA from assuming the loop is safe to run in parallel because PFA 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

```
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 PFA knows that the nonconcurrent statement involves no data dependency, PFA will find the loop safe to run in parallel.

There is one subtlety in using the concurrent call assertion. When you use this assertion, PFA makes no attempt to examine the called routine; it simply assumes that it is safe. However, PFA is still left with the problem of correctly declaring the variables in the loop to be either **SHARE** or **LOCAL**. (PFA 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** only reads 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, PFA 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 PFA 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 PFA 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, PFA can see an assignment to **m** and so will declare 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 PFA 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 Fortran 77 code

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

Using the previous code as input, PFA produces the listing file

```
DIR             1 # 1 "foo4.f"
                  2       subroutine foo4(a,b,n,sum)
                  3       real a (n),b(n), sum
                  4
                  5       sum = 0.0
         +-----   6       do i = i, n
1 DD     !        7         sum = sum + a(i)*b(i)
         !_____   8       enddo
                  9       end
Abbreviations Used
DD        data dependence
DIR       directive
Footnote List
1: data dependence       Data dependence involving this
                           line due to variable "SUM".
Loop Summary
          From  To    Loop   Loop
Loop#   line  line  label  index   Status
1       6     8     Do     I       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 a different order than 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, you can tell PFA to use parallel subtotals. To tell PFA not to worry about round−off error, you can use either the **C*$*ROUNDOFF=2** directive or the *f77/pfa* command line option **−WK**, **−roundoff=2**.

The resulting listing file is

```
DIR               1 # 1 "foo4.f"
                    2       subroutine foo4(a,b,n,sum)
                    3       real a(n), b(n), sum
                    4
                    5       sum = 0.0
C       +------     6       do i = 1, n
          *           7          sum = sum + a(i)*b(i)
          *______     8       enddo
                      9       end
Abbreviations Used
DIR       directive
C         concurrentized
Loop Summary
          From  To    Loop   Loop
Loop#  line  line  label  index  Status
1      6     8     Do     I      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 will simply be 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 PFA supports.

Table 3−5 Reduction Types

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

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.

Chapter 4
Customizing PFA Execution

This chapter contains the following sections:

- **"Overview"** explains when to optimize PFA execution.
- **"Controlling Code Execution"** describes how to control whether PFA runs eligible loops in parallel.
- **"Controlling PFA Code Transformations"** describes how to control the various transformations

performed by PFA.

- **"Performing Inlining and Interprocedural Analysis"** describes inlining and interprocedural analysis and explains how and when to perform these procedures.
- **"Controlling Fortran Language Elements"** explains how to control standard Fortran elements with command line options to PFA.
- **"Controlling Directives and Assertions"** explains how to override PFA directives and assertions with command line options.
- **"Controlling PFA I/O"** explains how to customize the names of PFA input and output files.
- **"Obsolete Syntax"** lists obsolete PFA command line options.

Overview

To customize how PFA executes an entire program, you can specify various command line options when you run PFA directly or when you specify PFA as part of a compile. **Chapter 2, "How to Use PFA,"** explains both procedures. For a complete summary of the PFA command line options, refer to **Appendix A, "PFA Command Line Options."**

Controlling Code Execution

When modifying most programs to allow loops to run in parallel, modify the code so that PFA can automatically run the loop in parallel. Avoid forcing the loop to run in parallel by directly inserting a **C$DOACROSS** directive. If you force code to run in parallel, you (and not PFA) 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 PFA recognizes the loop as safe to run in parallel allows PFA 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 **–C**) converts eligible loops to run in parallel. This is the default value for this option. The **–NOCONCURRENTIZE** option (or **–NCONC**) prevents PFA from converting loops to run in parallel.

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 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 will be executed.

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 PFA–generated **C$ DOACROSS** directive to test at run time if sufficient work exists.

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

For example, given the original loop

```
        do 2 i =1,n
           x(i) = y(i) * z(i)
2     continue
```

PFA 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, PFA will run the loop serially). If the loop bound is a small constant (such as 10) instead of *n*, PFA would not generate a **DOACROSS** statement for the loop and the listing file will state that the loop does not contain enough work. Conversely, if the bound is a large constant (such as 100), then PFA generates the **DOACROSS** statement without the **IF** clause.

Controlling PFA Code Transformations

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

Controlling Size/Complexity Thresholds

You can control the thresholds for internal table size and routine complexity in order to analyze larger and more complex routines.

Controlling Internal Table Size

The **–ARCLIMIT=***n* option (or **–ARCLM=***n*) controls the size of the internal table used to store data dependence information (arcs). If this table overflows, PFA stops analyzing the loop and the PFA listing file shows the message

```
too many stmts/dd arcs
```

Increasing **ARCLIMIT** might allow PFA to analyze the loop but at the cost of additional processing time.

Specifying a Complexity Limit

The **–LIMIT=***n* option (or **–LM=***n*) controls the amount of time PFA can spend trying to determine whether a loop is safe to run in parallel. PFA 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, PFA ignores the outer loop and recursively visits the inner loops.

Larger limits often allow PFA 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 PFA 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 PFA 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.

Setting the Optimization Level

The **–OPTIMIZE=***n* option (or **–O=***n*) sets the optimization level. The higher you set the optimization level, the more code is optimized and the longer PFA 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

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.

This option has the same effect as the global **C*$* OPTIMIZE(*n*)** directive described in **Chapter 5, "Fine−Tuning PFA."**

Note: If you want to use the **−UNROLL** command line option, set the **−OPTIMIZE** option to 4 or higher (the default optimization level is above this threshold).

Controlling Variations in Round Off

The **−ROUNDOFF=***n* option (or **−R=***n*) controls the amount of variation in round off that PFA will allow. Valid values for *n* are the 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 addition, multiplication, 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 **"Memory Management Transformations"**).

When executing reductions in parallel, PFA 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, PFA does not run reductions in parallel. Usually, these tiny variations are irrelevant, and you can allow PFA to process a reduction in parallel allowing more loops to be run in parallel.

Controlling the Number of Scalar Optimizations

The **−SCALAROPT=***n* option (or **−SO=***n*) controls the amount of standard scalar optimizations attempted by PFA. Valid values for *n* are the integers

0 Performs no scalar transformations.

1 Enables dead code elimination, pulling loop invariants, forward substitution, and conversion of **IF−GOTO** into **IF−THEN−ELSE**.

2 Enables induction variable recognition, loop unrolling, loop fusion, array expansion, scalar promotion, and floating invariant **IF** tests. (Loop fusion also requires **−OPTIMIZE=5**.)

3 Enables the memory management transformations (refer to **"Memory Management Transformations"**). (Memory management also requires **−ROUNDOFF=3**.) This is the default value.

Enabling Loop Unrolling

The **−UNROLL=***n* option (or **−UR=***n*) unrolls scalar inner loops when PFA cannot run the loops in parallel. *n* specifies the number of times to replicate the loop body. The default is 4. Specify a small power of two for the unroll value, such as two, four, or eight. Disable unrolling by setting **−UNROLL=1**.

The **−UNROLL2=***m* option (or **−UR2=***m*) allows you to adjust the number of operations used by the **−UNROLL** option. Selecting a larger value for **−UNROLL2** allows PFA to unroll loops containing more calculations. This form of unrolling applies only to the innermost loops in a nest of loops. You can unroll loops whether they execute serially or concurrently.

PFA counts the number of array references and arithmetic operations in the loop. It unrolls the loop until it reaches either the number of operations specified by the **−UNROLL2** option or the number of iterations specified by **−UNROLL**.

When PFA unrolls a loop, it replicates the body of the loop a certain number of times, making the loop run faster. However, unrolling loops also increases the program size.

For example, if the original program is

```
do i = 1,100
     a(i) = b(i) + c(i)*d(i)
enddo
```

the unrolled program (unrolling of order 4) is

```
do i = 1,100,4
     a(i) = b(i) + c(i)*d(i)
     a(i+1) = b(i+1) + c(i+1)*d(i+1)
     a(i+2) = b(i+2) + c(i+2)*d(i+2)
     a(i+3) = b(i+3) + c(i+3)*d(i+3)
enddo
```

The second (unrolled) version runs faster than the original version. The reason for the improvement is that SGI processors have separate add and multiply hardware, allowing addition and multiplication operations to run simultaneously. In the original program, the processor has to do the multiplication, wait for it to complete, then do the addition. In the second case, the processor can do the first multiplication, wait for it to complete, then overlap the second multiplication and the first addition, then the third multiplication and the second addition, and so on.

The additions require nearly no additional time because all but the last one are completed within the time it takes the (previous) multiplication to complete. If the loop already contains many computations (for example, many lines of code, many additions and multiplications), then unrolling it might help a little but not much.

Memory Management Transformations

When **–ROUNDOFF** and **–SCALAROPT** are both set to 3, PFA attempts to do outer loop unrolling (to improve register utilization) and automatic loop blocking (also called tiling) to improve cache utilization.

Outer loop unrolling is a standard hand–optimization technique. Note that the **–UNROLL** and **–UNROLL2** options apply to inner–loop unrolling. Outer–loop unrolling can occur even if inner–loop unrolling is disabled.

Loop blocking is a complex transformation that is applicable when the loop nesting depth is greater than the dimensions of the data arrays being manipulated. The canonical example is the simple matrix multiply, where a three–deep nest of loops operates on two–dimensional arrays.

The simple method repeatedly sweeps over the entire array. If the array is too large to fit into the cache, this can result in a large amount of memory traffic. A better method is to break the arrays up into blocks, where each block is small enough to fit into the cache, and then sweep over each block in turn (rather than over the whole array). The code to do this is often ugly and complicated. PFA attempts to ease the burden of writing block–style algorithms by automatically generating the block version from the simple version. Note, however, that blocking does not help the more common case where the algorithm touches each array element exactly once (for example, a two–dimensional array inside of a two–deep loop nest). Because in this case the data is not being reused, blocking does not apply.

For example, given the loop nest

```
do k  =1,n
     do j= 1,n
        do i  =1,n
           a(i,j) = a(i,j) + b(i,k)*c(k,j)
        enddo
     enddo
enddo
```

using the option **–r=3**, PFA produces the listing below:

```
     II3 = 1
     II1 = MOD (N - 1, 682) + 1
     II2 = II1
     II10 = N - 7
```

```
      II11= (II10 + 7) / 8
      DO 4  II4=1, N, 682
      II8 = II3 + II2 - 1
      DO 2 K=1, II10, 8
C$DOACROSS SHARE(N,K,C,II3,II8,A,B),LOCAL(DD1,DD2,C$& DD3,
 DD4,DD5,DD6,DD7,DD8,DD9,J,I)
      DO 2 J=1,N
      DD2 = C(K,J)
      DD3 = C(K+1,J)
      DD4 = C(K+2,J)
      DD5 = C(K+3,J)
      DD6 = C(K+4,J)
      DD7 = C(K+5,J)
      DD8 = C(K+6,J)
      DD9 = C(K+7,J)
         DO 2 I=II3, II8, 1
         DD1 = A(I,J)
            DD1 = DD1 +  B(I,K) * DD2
            DD1 = DD1 +  B(I, K+1) *  DD3
            DD1 = DD1 +  B(I, K+2) *  DD4
            DD1 = DD1 +  B(I, K+3) *  DD5
            DD1 = DD1 +  B(I, K+4) *  DD6
            DD1 = DD1 +  B(I, K+5) *  DD7
            DD1 = DD1 +  B(I, K+6) *  DD8
            DD1 = DD1 +  B(I, K+7) *  DD9
        A(I,J) = DD1
2     CONTINUE
        II7 = II11 * 8 + 1
        II9 = II3 + II2 - 1
        DO 3 K=II7, N, 1
C$DOACROSS SHARE(N,K,C,II3,II9,A,B),LOCAL(DD10,J,I)
      DO 3 J=1,N
      DD10 = C(K,J)
         DO 3 I=II3,II9,1
            A(I,J) = A(I,J) + B(I,K) * DD10
3      CONTINUE
        II3 = II3 + II2
        II2 = 682
4      CONTINUE
```

Obviously, PFA's version is more complicated than the original, but it runs significantly faster.

Performing Inlining and Interprocedural Analysis

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

- Assert that the external routine is safe for concurrent execution (see **"C*$* ASSERT CONCURRENT CALL"**).
- 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.

This section discusses the three steps for inlining or IPA:

1. Specify which routines will be inlined (or interprocedurally analyzed).

2. Specify which source files and libraries will be searched to find the routines.
3. Specify which occurrences of those routines are to be inlined (or analyzed).

Specifying Routines for Inlining or IPA

PFA supports the **–INLINE=***list* option (or **–IN=***list*) that specifies the routines to be inlined and the **–IPA=***list* option for IPA. *list* is a colon–separated list of routines to be inlined. For example,

```
–INLINE=jump:more
```

If you do not specify *list*, PFA will attempt to inline all eligible routines.

Specifying Where to Search for Routines

The options listed in **Table 4–1** tell PFA where to search for the routines specified with the **–INLINE** or **–IPA** option. If you do not specify either option, PFA searches the current source file by default.

Table 4–1 Inlining and IPA Search Command Line Options

Long Option Name	Short Option Name	Default Value
–*INLINE_FROM_FILES=list*	–*INFF=list*	Current Source File
–IPA_FROM_FILES=*list*	–*IPAFF=list*	Current Source File
–INLINE_FROM_LIBRARIES=*list*	–INFL=*list*	None
–IPA_FROM_LIBRARIES=*list*	–IPAFL=*list*	None

If one of the names in *list* is a directory, then all appropriate files in that directory will be used. PFA assumes files with the extension **.f** are Fortran source and files with the extension **.klib** are PFA–produced libraries.

Specify multiple files and directories with the same option by using a colon–separated list. For example,

```
–INLINE_FROM_FILES=file1:file2:file3
```

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 a Library

When performing inlining and IPA, PFA 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 preanalyzed 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). PFA assigns a *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.

The library used to do IPA does not have to be generated from the same source that will be linked into the running program. Using this capability can cause errors, but it can also be useful. For example, you could write a library of hand–optimized assembly language routines, then construct a PFA–compatible IPA library using Fortran routines that mimic the behavior of the assembly code. Thus, you can do parallelism analysis with IPA correctly but still call the hand–optimized assembly routines. Use the following procedure to create and use a PFA library:

1. Create a library by passing the source program directly through *pfa*. Library creation is done by PFA and should not be done at the same time as an ordinary compilation. For example, the following command line creates a library called **samp.klib** for the source program **samp.f**:

   ```
   % /usr/lib/pfa –INLINE_CREATE=samp.klib samp.f
   ```

2. Compile the program with *pfa*:

   ```
   % f77 –pfa keep –WK,–INFL=samp.klib samp.f
   ```

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

Specifying Occurrences

The loop level, depth, and manual options allow you to control which occurrences of the routines specified with the **–INLINE** or **–IPA** option are actually dealt with when the **–INLINE** or **–IPA** options are used.

Loop Level

The **–INLINE_LOOPLEVEL=***n* (or **–INLL=***n*) and **–IPA_LOOPLEVEL=***n* (or **–IPALL=***n*) options allow you to limit PFA to work only on occurrences within deeply nested loops. Thus, a value of 1 restricts PFA to deal with routines only at the single–most deeply nested level; a value of 2 restricts PFA to the deepest and second–deepest levels; and so on.

To determine most deeply nested, PFA constructs a call graph to account for nesting due to loops that occur farther up the call chain. If you do not specify either option, the loop level is 10.

Depth

The **–INLINE_DEPTH=***n* (or **–IND**) option restricts the number of times PFA will continue to attempt inlining on already inlined routines. For example, suppose you use PFA to inline the routine **foo**. However, **foo** itself contains a call to **bar**. Should PFA now attempt a second inlining depth and inline **bar** ? And if **bar** calls **baz**, should PFA inline three deep? This option provides control over this process, as routines are only inlined to the specified depth.

As a special case, if you specify the value -1, only routines that do not reference other routines are inlined (that is, only leaf routines are inlined). Note that the extension to ð -2, -3, and so on is not supported, only -1. Note also that there is *no* **–IPA_DEPTH** option.

Manual

The **–INLINE_MAN** option turns on recognition of the **C*$*INLINE** directive. This directive (described in **Chapter 5, "Fine–Tuning PFA"**) allows you select individual occurrences of routines to be inlined. **–IPA_MAN** is the analogous option for the **C*$*IPA** directive (also described in **Chapter 5, "Fine–Tuning PFA."**).

Conditions That Prevent Inlining or IPA

Several conditions make a routine ineligible for inline expansion or IPA:

- Dummy arguments do not match the actual arguments in number, type, shape, or size.
- The calling program and called routine have conflicting declarations for the same **COMMON** block.
- The calling program and the called routine have conflicting **EQUIVALENCE** statements.
- The routine to be inlined has a **SAVE**, **ENTRY**, or **NAMELIST** statement.
- The routine to be inlined has a **DATA** loaded variable.
- The routine to be inlined is too long (the limit is about 600 lines).

Controlling Fortran Language Elements

This section explains how to control various Fortran 77 language elements.

Global Assumptions

The **–ASSUME=***list* option (or **–AS=***list*) controls certain global assumptions of a program. *list* consists of any combination of the following values:

E — Allows equivalence variables to refer to the same memory location inside one loop. For more information, see **Chapter 5, "Fine–Tuning PFA."**

L — Instructs PFA to use a temporary variable within the optimized loop and assign the last

value to the original scalar if PFA determines that scalar can be reused before it is assigned. This value is important when a scalar is assigned in a loop run in parallel. For more information, see **Chapter 5, "Fine−Tuning PFA."**

P — Allows for parameter aliasing in a subprogram. For more information, see **Chapter 5, "Fine−Tuning PFA."**

By default, PFA assumes that a program conforms to the ANSI (and VMS™) standard; therefore, the default is **−ASSUME=EL**.

Debugging Lines

The **−DLINES** option tells PFA to treat the letter **D** in column one as if the letter were a character space. PFA then parses the rest of that line as a normal Fortran 77 statement. The **−NODLINES** option tells PFA to treat these lines as though they were comments. These options are useful for excluding or including debugging lines. *f77* passes this option to PFA automatically when you specify the *f77* **−d_lines** option.

DO Loop Execution

The **−ONETRIP** option (or **−I**) provides compatibility with older versions of Fortran where a **DO** loop is always executed at least once. The **−NOONETRIP** (or **−N1**) option conforms to the Fortran 77 standard.

This option, which is the default, does not execute a **DO** loop whose termination condition is initially satisfied. *f77* passes the **−ONETRIP** option to PFA automatically when you specify the *f77* **−one_trip** option.

Variable Saving Across Invocations

The **−SAVE=***c* option (or **−SV=***c*) specifies whether a procedure's variables are saved across invocations. *c* is one of the following values:

A — Performs a lifetime analysis on a procedure's variables to determine those that need to have their value saved across invocations of the procedure. When it finds such a variable, PFA generates a **SAVE** statement for the variable.

M — Does not generate **SAVE** statements. This is the default value.

Significant Columns

The **−SCAN=***n* option controls the number of columns that PFA assumes to be significant. PFA ignores anything beyond the specified column number. The default value for *n* is 72. Specifying any of the following *f77* options automatically sets this option: **−col72**, **−col120**, or **−extend_source**.

Fortran Standard

Setting the **−SYNTAX=***c* option (or **−SY=***c*) alters the interpretation of the Fortran input to be in compliance with other standards. *c* is one of the following values:

A — Interprets the source in strict compliance with the ANSI Fortran 77 standard.

V — Interprets the source in compliance with the VMS Fortran standard but without the additional SGI extensions.

If you do not specify this option, PFA uses the same rules as the standard SGI Fortran compiler (refer to the *Fortran 77 Programmer's Guide* for details).

Controlling Directives and Assertions

This section discusses the options you can use to select whether PFA accepts a specific directive or assertion. You can use these options to override directives and assertions that are specified in the source program.

Selecting Directives and Assertions

The **–DIRECTIVES=***list* option specifies the directives and assertions to accept. The **–NODIRECTIVES** option tells PFA to ignore all directives and assertions. This option is useful when you suspect unsafe directives are causing problems with program execution.

Note: Some directives are called assertions because they assert program characteristics that PFA cannot verify. (For example, an assertion could assert that subroutine *x* contains no data dependencies.) However, you might want PFA to use it when optimizing. Refer to **Chapter 1, "Overview of PFA,"** for more information about directives and assertions.

Valid values for *list* are any combination of the values

A Accepts assertions.

C Accepts Cray **CDIR$** directives; **CDIR$IVDEP** ignores certain data dependencies in a loop. But because of differences between SGI hardware and a Cray machine, these data dependencies are not always safe to ignore on SGI hardware. To be safe, PFA does not recognize the **CDIR$IVDEP** directive by default. You can, at your own risk, turn on Cray–directive recognition, which will cause PFA to treat this Cray directive as if it were a **C*$*ASSERT DO (CONCURRENT)** assertion.

K Accepts **C*$*** directives.

S Accepts **C$** directives. PFA recognizes the directives **C$DOACROSS**, **C$**, and **C$&**. (For more information, see the *Fortran 77 Programmer's Guide*.) If a **C$DOACROSS** directive appears, PFA does not examine or alter the loop to which the directive applies. This allows you to mix code that you converted to parallel execution with code that PFA converted to parallel execution.

V Accepts VAST **CVD$** directives.

For example, specifying **–DIRECTIVES=K** enables PFA directives only, whereas **–DIRECTIVES=CK** enables both Cray and PFA directives. Adding **A** to the **DIRECTIVES** sequence also enables PFA assertions. Any combination of options is acceptable.

If you do not specify either option, PFA will accept all assertions, PFA **C*$*** directives, all **C$** directives, and VAST **CVD$** directives.

Controlling PFA I/O

This section describes command line options you can use to name PFA input and output. You do not need to use these options unless you want to change the default names. In particular, some versions of the *make*(1) utility assume that files ending in **.1** are *lex*(1) input files. To perform automatic makes without overwriting the PFA listing file, use a different suffix for the listing filename.

Use the **–INPUT=***file***.f** option to specify the name of the Fortran source program PFA input file. If you do not specify this option, PFA assumes that a command line argument not preceded by a dash is the input filename.

The **–FORTRAN=***file* option specifies the name of the PFA intermediate file (that is, the transformed source). If you do not specify this filename, PFA names the intermediate *file***.m**, where file is the name of the input file. For details about the intermediate file, refer to **Chapter 3, "Utilizing PFA Output."**

The **–LIST=***file* option specifies the name of the PFA listing file. If you do not specify this filename, PFA names the listing file *file***.l**, where *file* is the name of the input file. For details about the listing file, refer to **Chapter 3, "Utilizing PFA Output."**

Obsolete Syntax

Table 4–2 lists obsolete PFA command line options.

Table 4–2 Obsolete Options

Long Option Name	Short Option Name	Default Value
–EXPAND	–X, –EX	off
–CREATE	–CR	off
–LIBRARY	–LIB	off

–LIMIT2	–LM2	5000

PFA now accepts new syntax for some of the command line options (particularly the syntax for inlining). For compatibilIty with the older versions, these options are translated into their newer equivalents in **Table 4–3** . Whenever possible do not use the older syntax; support for it might be withdrawn in the future.

Table 4–3 Obsolete Options and Their Equivalents

Old Version	New Version
–EXPAND=*A*	–INLINE
–EXPAND=*M*	–INLINE_MAN
–LIBRARY=*name*	–INLINE_FROM_LIBRARIES=*name*
–CREATE –LIBRARY=*name*	–INLINE_CREATE=*name*
–LIMIT2=*n*	–ARCLIMIT=*n*

Chapter 5

Fine–Tuning PFA

This chapter contains the following sections:

- **"Overview"** explains how to fine–tune program execution using directives and assertions.
- **"Fine–Tuning Inlining and IPA"** describes how to use directives to use inlining and IPA more specifically than with command line options.
- **"Circumventing PFA"** explains how to use directives to bypass PFA's analysis and leave areas of code unchanged.
- **"Running Code Serially"** explains how to use directives and assertions to stop PFA from running specific code in parallel.
- **"Running Code in Parallel"** explains how to use directives and assertions to tell PFA that it is safe to run specific parts of code in parallel.
- **"Ignoring Data Dependencies"** explains how to tell PFA that apparently data–dependent code is safe to run in parallel.
- **"Using Equivalenced Variables"** explains how to assert that your code uses or does not use equivalenced variables.
- **"Using Aliasing"** describes the assertions used with aliasing.

Overview

After you run a Fortran source program through PFA once, you can use directives and assertions to fine–tune program execution. The listing file will show where and why PFA did not parallelize the code.

You can use directives and assertions to force PFA to execute portions of code in various ways. Command line directives apply to the program as a whole.

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 PFA from converting code to run in parallel. In other cases you might want to explicitly force PFA to run segments of code in parallel even though it normally would not.

Fine–Tuning Inlining and IPA

Chapter 4, "Customizing PFA Execution," explains how to use inlining and IPA on an entire program (refer to **"Performing Inlining and Interprocedural Analysis"**). You can fine–tune inlining and IPA using the **C*$*[NO] INLINE** and **C*$*[NO] IPA** directives.

The **C*$* [NO] INLINE** directive behaves much the same as the **–INLINE** command line option, but with the directive you can specify which occurrences of a routine are actually inlined. The format for this directive is

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

where

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

The **C*$*NOINLINE** form overrides the **–INLINE** command line option and so allows you to selectively disable inlining of the named routines at specific points.

Example

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

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

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

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}
```

Circumventing PFA

Sometimes you might need to hand–tune a **DO** loop so that it will run in parallel. Use the directives in this section to prevent PFA 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 PFA 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 PFA generates as a result of its analysis. Refer to the *Fortran 77 Programmer's Guide* for more information about the
C$ DOACROSS directive and its optional clauses.

PFA runs the following code as it appears:

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

C$&

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)
```

Running Code Serially

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

C*$* ASSERT DO (SERIAL)

The **C*$* ASSERT DO (SERIAL)** assertion tells PFA to run the specified loop serially. PFA does not try to convert the specified loop to run in parallel. It also does not try to run any enclosing loop in parallel. However, PFA can still convert any loops nested inside the serial loop to run in parallel.

CDIR$ NEXT SCALAR

Silicon Graphics PFA supports the corresponding Cray directive, **CDIR$ NEXT SCALAR**. PFA 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 indicates that you want to execute a **DO** loop in serial mode. This assertion directs PFA 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) PFA chooses to run in parallel. The following example program segment shows 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, PFA 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 PFA 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 PFA from converting loops to run in parallel. These directives, when specified globally, have the same effect as the **–CONCURRENTIZE** and **–NOCONCURRENTIZE** options (see **Chapter 2, "How to Use PFA."**).

CVD$ CONCUR

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

C*$* ASSERT DO PREFER (CONCURRENT)

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

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 PFA to prefer to run the **I** loop in parallel. However, if a data dependence conflict prevents running the **I** loop in parallel, PFA might run the **J** loop in parallel. The **C*$* ASSERT DO PREFER (CONCURRENT)** assertion applies only to the **DO** loop immediately before it.

Ignoring Data Dependencies

PFA 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)

The **C*$* ASSERT DO (CONCURRENT)** assertion tells PFA to ignore assumed data dependencies. Normally, PFA is conservative about converting loops to run in parallel.

When PFA 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, PFA does not run "not sure" loops in parallel. It assumes there are data dependencies. **C*$* ASSERT DO (CONCURRENT)** tells PFA to go ahead and run "not sure" loops in parallel.

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

CDIR$ IVDEP

PFA 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 SGI hardware. Therefore, recognition of this assertion is turned off by default.

C*$* ASSERT CONCURRENT CALL

The **C*$* ASSERT CONCURRENT CALL** tells PFA 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 accompanying loop, which must appear on the next line.

C*$* ASSERT NO RECURRENCE

The **C*$* ASSERT NO RECURRENCE(***variable***)** assertion tells PFA to ignore all data dependencies associated with *variable.* PFA ignores not just assumed dependencies (as with the **C*$* ASSERT DO (CONCURRENT)** assertion) but also real dependencies. Use this assertion to force PFA 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 PERMUTATION

The **C*$* ASSERT PERMUTATION(***array***)** assertion tells PFA that *array* contains no repeated values. This assertion permits PFA 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). PFA 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 PFA to run the loop in parallel:

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

Using Equivalenced Variables

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

Using Aliasing

PFA has several assertions for use with aliasing.

C*$* ASSERT [NO] ARGUMENT ALIASING

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

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

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

The command line option **–ASSUME=P** acts like a global **C*$* ASSERT ARGUMENT ALIASING** assertion (see **Chapter 4, "Customizing PFA Execution."**). A **C*$* ARGUMENT ALIASING** assertion is active until it is reset or until the next routine begins.

C*$* ASSERT RELATION

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

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 PFA:

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

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

Appendix A
PFA Command Line Options

This appendix contains the following sections:
- **"Overview"**
- **"Options Summary"**
- **"Obsolete Syntax"**

This appendix lists and describes the options to PFA. The default settings are satisfactory for most programs. However, you can alter the defaults to customize output. PFA accepts several command line options. **Table A–1** lists the default settings for each option.

Overview

Table A–1 summarizes the PFA command line options. The Reference column lists the functional categories of the following options:
- parallel execution
- general optimization
- Fortran 77 language control
- 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.

Note: You can replace many of the PFA command line options described in this chapter with in–code directives.

Table A–1 PFA Command Line Options

Reference	Long Name	Short Name
Parallelization	[NO]CONCURRENTIZE	[N]CONC
	MINCONCURRENT=*n*	MC=*n*
Optimization	ARCLIMIT	ARCLM=*n*
	LIMIT=*n*	LM=*n*
	OPTIMIZE=*n*	O=*n*
	ROUNDOFF=*n*	R=*n*
	SCALAROPT=n	SO=*n*
	UNROLL=*n*	UR=*n*
	UNROLL2=*n*	UR22=*n*
Fortran 77 Language	ASSUME=*list*	AS=*list*
Control	[NO]DLINES	[N]DL
	[NO]ONETRIP	[N]I

	SAVE=*c*	SV=*c*
	SCAN=*n*	SCAN=*n*
	SYNTAX=*c*	SY=*c*
Inlining and	INLINE[=*list*]	*INIPA*INCR=*name*
Interprocedural Analysis	IPA[=*names*]	IPACR=*name*
	INLINE_CREATE=*name*	INFF=*list*
	IPA_CREATE=*name*	IPAFF=*list*
	INLINE_FROM_FILES=*list*	INFL=*list*
	IPA_FROM_FILES=*list*	IPAFL=*list*
	INLINE_FROM_LIBRARIES=*list*	INLL=*n*
	IPA_FROM_LIBRARIES=*list*	IPALL=*n*
	INLINE_LOOP_LEVEL=*n*	INM
	IPA_LOOP_LEVEL=*n*	IPAM
	INLINE_MAN	IND
	IPA_MAN	
	INLINE_DEPTH	
Directives	[NO]DIRECTIVES=*list*	[N]DR=*list*
I/O	INPUT=file.f	*file*.f
	[NO]FORTRAN=file	[N]F=*file*
	[NO]LIST=file	[N]L=*file*
Listing	LINES=*n*	LN=*n*
	LISTOPTIONS=*list*	LO=*list*
	SUPPRESS=*list*	SU=*list*
Obsolete	CREATE	CR
	LIBRARY=*file*	LIB=*file*
	[NO]EXPAND=*list*	EX=*list*
	LIMIT2=*n*	LM2=*n*

Options Summary

This section lists and defines all PFA command line options alphabetically.

ARCLIMIT

The **–ARCLIMIT** option, described in **Table A–2** , controls the size of the internal table used to store data dependence information (arcs).

Table A–2 ARCLIMIT Option

Long Option Name	Short Option Name	Default Value
–ARCLIMIT=*n*	–ARCLM=*n*	5000

ASSUME

The **–ASSUME** option, described in **Table A–3** , controls certain global assumptions of a program.

Table A–3 ASSUME Option

Long Option Name	Short Option Name	Default Value
–ASSUME=*list*	–AS=*list*	EL

You can also use various assertions to control these assumptions. *list* is any combination of the following values:

E	Means that equivalence variables can refer to the same memory location inside one loop.
L	Is important when a scalar is assigned in a loop run in parallel. If **ASSUME** is L, PFA uses a temporary variable within the optimized loop and assigns the last value to the original scalar if PFA determines that scalar can be reused before it is assigned.
P	Allows for parameter aliasing in a subprogram.

CONCURRENTIZE

The **–CONCURRENTIZE** option, described in **Table A–4** , converts eligible loops to run in parallel.

Table A–4 CONCURRENTIZE Option

Long Option Name	Short Option Name	Default Value
–CONCURRENTIZE	–C	–CONCURRENTIZE

See also **NOCONCURRENTIZE**.

DIRECTIVES

The **–DIRECTIVES** option, described in **Table A–5** , specifies the directives and assertions to accept.

Table A–5 DIRECTIVES Option

Long Option Name	Short Option Name	Default Value
–DIRECTIVES=*list*	–DR=*list*	AKSV

list consists of any combination of

A	Accepts assertions.
C	Accepts Cray **CDIR$** directives. Because of differences between SGI and Cray hardware, certain data dependencies that **CDIR$IVDEP** ignores in a loop are not always safe to ignore on SGI hardware. PFA does not recognize the **CDIR$IVDEP** directive by default. You can, however, turn on Cray–directive recognition, which will cause PFA to treat the Cray directive as a **C*$*ASSERT DO (CONCURRENT)** assertion.
K	Accepts PFA **C*$*** directives.
S	Accepts **C$** directives. PFA recognizes the directives **C$&C**, **C$**, and **$DOACROSS**. (For more information, see the *Fortran 77 Programmer's Guide*.) If a **C$DOACROSS** directive appears, PFA does not examine or alter the loop to which the directive applies. This allows you to mix code you converted to parallel execution with code that PFA converted to parallel execution.
V	Accepts VAST **CVD$** directives.

See also **NODIRECTIVES**.

DLINES

The **–DLINES** option, described in **Table A–6** , tells PFA to treat letter **D** in column one as if the letter were a character space.

Table A–6 DLINES Option

Long Option Name	Short Option Name	Default Value
–DLINES	–DL	–NODLINES

PFA then parses the rest of that line as a normal Fortran 77 statement. See also **NODLINES**.

FORTRAN

The **–FORTRAN** option, described in **Table A–7** , specifies the name of the PFA–transformed source.

Table A–7 FORTRAN Option

Long Option Name	Short Option Name	Default Value
–FORTRAN=*filename*	–F=*filename*	*filename*. m

filename is the name of the Fortran source.

INLINE

The **–INLINE** option, described in **Table A–8** , specifies the routines to be inlined.

Table A–8 INLINE Option

Long Option Name	Short Option Name	Default Value
–INLINE[=*list*]	–IN[=*list*]	none

If this option is given with a (colon–separated) list of routine names, then only those routines will be inlined. If it is given without a list of names, then PFA will attempt to inline all eligible routines.

INLINE_CREATE

The **–INLINE_CREATE** option, described in **Table A–9** , creates a library of prepared routines for later use with the **–INLINE_FROM_LIBRARIES** option.

Table A–9 INLINE_CREATE Option

Long Option Name	Short Option Name	Default Value
–INLINE_CREATE=*name*	–INCR=*name*	option off

You are not required to create a library to do inlining; you can inline directly from a source file.

Constructing a library will save time if the inlining operation is to be done repeatedly. PFA analyzes the current source file and places the appropriate information into the file named with the **–INLINE_CREATE** option. For maximum compatibility, the filename extension **.klib** is recommended: for example, **samp.klib**.

INLINE_DEPTH

The **–INLINE_DEPTH** option, described in **Table A–10** , restricts the number of times PFA will continue to attempt inlining on already inlined routines.

Table A–10 INLINE_DEPTH Option

Long Option Name	Short Option Name	Default Value
–INLINE_DEPTH=*n*	–IND=*n*	10

This option controls this process, as routines are only inlined to the specified depth.

As a special case, if you specify the value -1, only routines that do not reference other routines are inlined (that is, only leaf routines are inlined). The only valid negative number is -1; do not specify -2, -3, and so on. Note that there is no **–IPA_DEPTH** option.

INLINE_FROM_FILES

The **–INLINE_FROM_FILES** option, described in **Table A–11** , specifies where to look for routines named in the **–INLINE** option.

Table A–11 INLINE_FROM_FILES Option

Long Option Name	Short Option Name	Default Value
–INLINE_FROM_FILES=*list*	–INFF=*list*	current source file

Files with the extension **.f** are assumed to be Fortran source, while files with the extension **.klib** are assumed to be PFA–produced libraries. Specify multiple files and directories by using a colon–separated list.

Note: This option alone does not initiate inlining. It only specifies where to look for the routines. Use this option with the **–INLINE** option.

INLINE_FROM_LIBRARIES

The **–INLINE_FROM_LIBRARIES** option, described in **Table A–12** , specifies where to look for the routines named in the **–INLINE** option.

Table A–12 INLINE_FROM_LIBRARIES Option

Long Option Name	Short Option Name	Default Value
–INLINE_FROM_LIBRARIES=*list*	–INFF=*list*	current source file

Files with the extension **.f** are assumed to be Fortran source, while files with the extension **.klib** are assumed to be PFA–produced libraries. Specify multiple libraries by using a colon–separated list.

Note: This option alone does not initiate inlining. It only specifies where to look for the routines. Use

this option with the **–INLINE** option.

INLINE_LOOPLEVEL

The **–INLINE_LOOPLEVEL** option, described in **Table A–13** , restricts PFA occurrences within deeply nested loops.

Table A–13 INLINE_LOOPLEVEL Option

Long Option Name	Short Option Name	Default Value
–INLINE_LOOPLEVEL=*n*	–INLL=*n*	10

Thus, a value of 1 restricts PFA to deal with routines only at the singlemost deeply nested level; a value of 2 restricts PFA to the deepest and second–deepest levels; and so on.

To determine what is most deeply nested, PFA constructs a call graph to account for nesting due to loops that occur farther up the call chain.

INLINE_MAN

The **–INLINE_MAN** option, described in **Table A–14** , turns on recognition of the **C*$*INLINE** directive.

Table A–14 INLINE_MAN Option

Long Option Name	Short Option Name	Default Value
–INLINE_MAN	–INM	option off

The **C*$*INLINE** directive allows you select individual occurrences of routines to be inlined.

INPUT

The **–INPUT** option, described in **Table A–15** , specifies the name of the PFA input file.

Table A–15 INPUT Option

Long Option Name	Short Option Name	Default Value
–INPUT=*filename*.f	–I=*filename*.f	*filename*.f

It is not necessary to precede the input filename with this option; PFA assumes that a command line argument not preceded by a dash is the input filename.

IPA

The **–IPA** option, described in **Table A–16** , specifies the routine's IPA.

Table A–16 IPA Option

Long Option Name	Short Option Name	Default Value
–IPA[=*list*]	–IPA[=*list*]	none

If this option is given with a colon–separated list of routine names, then only those routines will be IPAed. If it is given without a list of names, then PFA will attempt to IPA all eligible routines.

IPA_CREATE

The **–IPA_CREATE** option, described in **Table A–17** , creates a library of prepared routines for later use with the **–IPA_FROM_LIBRARIES** option.

Table A–17 IPA_CREATE Option

Long Option Name	Short Option Name	Default Value
–IPA_CREATE=*name*	–IPACR=*name*	option off

You are not required to create a library to do inlining; you can inline directly from a source file. Constructing a library will save time if the inlining operation is to be done repeatedly. PFA analyzes the current source file and places the appropriate information into the file named with the **–INLINE_CREATE** option. For maximum compatibility, the filename extension **.klib** is recommended: for example, **samp.klib**.

Libraries created for IPA only contain summary information and so can be used only for IPA.

IPA_FROM_FILES

The **–IPA_FROM_FILES** option, described in **Table A–18** , specifies where to look for the routines named in the **–IPA** option.

Table A–18 IPA_FROM_FILES Option

Long Option Name	Short Option Name	Default Value
–IPA_FROM_FILES=*list*	–IPAFF=*list*	current source file

Files with the extension **.f** are assumed to be Fortran source, while files with the extension **.klib** are assumed to be PFA–produced libraries. Specify multiple files using a colon–separated list.

Note: This option alone does not initiate IPA. It only specifies where to look for the routines. Use this option in conjunction with the **–IPA** option.

IPA_FROM_LIBRARIES

The **–IPA_FROM_LIBRARIES** option, described in **Table A–19** , specifies where to look for the routines named in the **–IPA** option.

Table A–19 IPA_FROM_LIBRARIES Option

Long Option Name	Short Option Name	Default Value
–IPA_FROM_LIBRARIES=*list*	–IPAFL=*list*	none

Files with the extension **.f** are assumed to be Fortran source, while files with the extension **.klib** are assumed to be PFA–produced libraries. Specify multiple libraries by using a colon–separated list.

Note: This option alone does not initiate IPA. It only specifies where to look for the routines. Use this option in conjunction with the **–IPA** option.

IPA_LOOPLEVEL

The **–IPA_LOOPLEVEL** option, described in **Table A–20** , restricts PFA to occurrences within deeply nested loops.

Table A–20 IPA_LOOPLEVEL Option

Long Option Name	Short Option Name	Default Value
–IPA_LOOPLEVEL=*n*	–IPALL=*n*	10

A value of 1 restricts PFA to routines only at the singlemost deeply nested level; a value of 2 restricts PFA to the deepest and second–deepest levels; and so on. To determine what is most deeply nested, PFA constructs a call graph to account for nesting due to loops that occur further up the call chain.

IPA_MAN

The **–IPA_MAN** option, described in **Table A–21** , turns on recognition of the **C*$*IPA** directive.

Table A–21 IPA_MAN Option

Long Option Name	Short Option Name	Default Value
–IPA_MAN	–IPAM	option off

The **C*$*IPA** directive allows you select individual occurrences of routines to be IPAed.

LIMIT

The **–LIMIT** option, described in **Table A–22** , reduces PFA processing time by limiting the amount of time PFA can spend on trying to determine whether a loop is safe to run in parallel.

Table A–22 LIMIT Option

Long Option Name	Short Option Name	Default Value
–LIMIT=*n*	–LM=*n*	LIMIT=5000

PFA 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, PFA ignores the outer loop and recursively visits the inner loops.

Larger limits often allow PFA 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 PFA 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 PFA can generate from a loop nest.)

LINES

The **–LINES** option, described in **Table A–23** , paginates the listing for printing.

Table A–23 LINES Option

Long Option Name	Short Option Name	Default Value
–LINES=*n*	–LN=*n*	LINES=55

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

LIST

The **–LIST** option, described in **Table A–24** , specifies the name of the PFA listing file.

Table A–24 LIST Option

Long Option Name	Short Option Name	Default Value
–LIST=*filename*	–L=*filename*	LIST=*filename*.l

filename is the name of the Fortran source.

LISTOPTIONS

The **–LISTOPTIONS** option, described in **Table A–25** , specifies the information to include in the listing file *(***.l***)*.

Table A–25 LISTOPTIONS Option

Long Option Name	Short Option Name	Default Value
–LISTOPTIONS=*list*	–LO=*list*	OL

list consists of any combination of

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. When PFA is run as part of an *f77* compilation, this option is added automatically.
K	PFA 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–26** , establishes the minimum amount of work needed inside the loop to make executing a loop in parallel profitable.

Table A–26 MINCONCURRENT Option

Long Option Name	Short Option Name	Default Value
–MINCONCURRENT=*n*	–MC=*n*	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 PFA–generated **DOACROSS** directive to test at run time whether sufficient work exists.

The **MINCONCURRENT** value 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 will be executed.

NOCONCURRENTIZE

The **–NOCONCURRENTIZE** option, described in **Table A–27** , prevents PFA from converting loops to run in parallel.

Table A–27 NOCONCURRENTIZE Option

Long Option Name	Short Option Name	Default Value
–NOCONCURRENTIZE	–NCONC	none

See also **CONCURRENTIZE**.

NODIRECTIVES

The **–NODIRECTIVES** option, described in **Table A–28** , tells PFA to ignore all directives and assertions.

Table A–28 NODIRECTIVES Option

Long Option Name	Short Option Name	Default Value
–NODIRECTIVES	–NDR	none

See also **DIRECTIVES**.

NODLINES

The **–NODLINES** option, described in **Table A–29** , tells PFA to treat lines starting with **D** as though they were comments.

Table A–29 NODLINES Option

Long Option Name	Short Option Name	Default Value
–NODLINES	–NDL	–NODLINES

See also **DLINES**.

NOONETRIP

The **–NOONETRIP** option, described in **Table A–30** , conforms to the Fortran 77 standard, which specifies that a **DO** loop whose termination condition is initially satisfied is not executed.

Table A–30 NOONETRIP Option

Long Option Name	Short Option Name	Default Value
–NOONETRIP	–N1	–NOONETRIP

See also **ONETRIP**.

ONETRIP

The **–ONETRIP** option, described in **Table A–31** , allows compatibility with older versions of Fortran where a **DO** loop is always executed at least once.

Table A–31 ONETRIP Option

Long Option Name	Short Option Name	Default Value
–ONETRIP	–1	–NOONETRIP

See also **NOONETRIP**.

OPTIMIZE

The **–OPTIMIZE** option, described in **Table A–32** , sets the optimization level.

Table A–32 OPTIMIZE Option

Long Option Name	Short Option Name	Default Value
–OPTIMIZE=*n*	–O=*n*	5

The higher you set the optimization level, the more code is optimized and the longer PFA runs.

Valid values for *n* are the integers

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 be run safely in parallel. This level allows reductions in loops that execute concurrently, but only if **–ROUNDOFF** is set to 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 dependence 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 level recognizes arrays as local variables. Level 5 also tells PFA to try harder to take the outermost loop possible (of a set of nested loops) and run it in parallel.

Note: If you want to use the **–UNROLL** command line option, you must set the **–OPTIMIZE** option to 4 or higher (the default optimization level is above this threshold).

ROUNDOFF

The **–ROUNDOFF** option, described in **Table A–33** , controls whether PFA runs a reduction operation in parallel.

Table A–33 ROUNDOFF Option

Long Option Name	Short Option Name	Default Value
–ROUNDOFF=*n*	–R=*n*	0

Valid values for *n* are

0-1	Suppresses any round–off changing transformations.
2	Allows reductions to be performed in parallel. The valid reduction operators are addition, multiplication, min, and max. **–ROUNDOFF=2** is one of the most common user options.
3	Recognizes **REAL** induction variables. Permits the memory management transformations.

SAVE

Table A–34 describes the **–SAVE** option.

Table A–34 SAVE Option

Long Option Name	Short Option Name	Default Value
–SAVE=*c*	–SV=*c*	A

Either of the following values are valid for *c*:

A	Performs lifetime analysis on a procedure‘s variables to try and determine those that need to have their value saved across invocations of the procedure. When it finds such a variable, PFA generates a **SAVE** statement for the variable. This is the default value.
M	Does not generate **SAVE** statements.

SCALAROPT

The **–SCALAROPT=***n* option, described in **Table A–35** , controls the amount of standard scalar

optimizations attempted by PFA.

Table A–35 SCALAROPT Option

Long Option Name	Short Option Name	Default Value
–SCALAROPT=*n*	–SO=*n*	3

Valid values for *n* are

0	Performs no scalar transformations.
1	Enables dead code elimination, pulling loop variables, forward substitution, and conversion of **IF–GOTO** into **IF–THEN–ELSE**.
2	Enables induction variable recognition, loop unrolling, loop fusion, array expansion, scalar promotion, and floating invariant **IF** tests. (Loop fusion also requires **–OPTIMIZE=5**.)
3	Enables the memory management transformations. (Memory management also requires **–ROUNDOFF=3**.)

SCAN

The **–SCAN** option, described in **Table A–36** , controls the number of columns that are assumed to be significant (PFA ignores anything beyond the specified column).

Table A–36 SCAN Option

Long Option Name	Short Option Name	Default Value
–SCAN=*n*	–SCAN=*n*	72

Specifying any of the following *f77* options automatically sets this option: **–col72**, **–col120**, or **–extend_source**.

SUPPRESS

The **–SUPPRESS** option, described in **Table A–37** , lets you individually disable classes of PFA messages that are normally included in the listing (**.l**) file.

Table A–37 SUPPRESS Option

Long Option Name	Short Option Name	Default Value
–SUPPRESS=*list*	–SU=*list*	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
I	Information
N	Not able to run loop in parallel
Q	Questions
S	Standard messages
W	Warning of syntax error (PFA adds the **–SUPPRESS=W** option automatically if you use the **–w** option to *f77*)

SYNTAX

Setting the **–SYNTAX** option, described in **Table A–38** , alters the interpretation of the Fortran input to be in compliance with other standards.

Table A–38 SYNTAX Option

Long Option Name	Short Option Name	Default Value
–SYNTAX=*c*	–SY=*c*	SGI Fortran syntax

c is any of the following values:

A	Interprets the source in strict compliance with the ANSI Fortran 77 standard.
V	Interprets the source in compliance with the VMS Fortran standard but without the additional SGI extensions.

UNROLL

The **–UNROLL** option, described in **Table A–39** , unrolls scalar inner loops when PFA cannot run the loops in parallel.

Table A–39 UNROLL Option

Long Option Name	Short Option Name	Default Value
–UNROLL=*n*	–UR=*n*	4

When PFA unrolls a loop, it replicates the body of the loop a certain number of times, making the loop run faster. (In previous releases of PFA the default value was 1.)

UNROLL2

The **–UNROLL2** option, described in **Table A–40** , allows you to adjust the number of operations used by the **–UNROLL** option.

Table A–40 UNROLL2 Option

Long Option Name	Short Option Name	Default Value
–UNROLL2=*m*	–UR2=*m*	50

Selecting a larger value for **–UNROLL2** allows PFA to unroll loops containing more calculations.

This form of unrolling applies only to the innermost loops in a nest of loops. You can unroll loops whether they execute serially or concurrently.

f 77 passes this option to PFA automatically when you specify the *f77* **–d lines** option.

Obsolete Syntax

Table A–41 describes obsolete PFA syntax.

Table A–41 Obsolete Options

Long Option Name	Short Option Name	Default Value
–EXPAND=*list*	–X=*list*, –EX=*list*	option off
–CREATE	–CR	option off
–LIBRARY=*file*	–LIB=*file*	option off
–LIMIT2=*n*	–LM2	5000

This version of PFA has altered some of the command line syntax (particularly the syntax for inlining).

For compatibility with the older versions, **Table A–42** lists the options that are translated into their newer equivalents.

Table A–42 Obsolete Options and Their Equivalents

Old Version	New Version
–EXPAND=A	–INLINE
–EXPAND=M	–INLINE_MAN
–LIBRARY=*name*	–INLINE_CREATE=*name*
–LIMIT2=*n*	–ARCLIMIT=*n*

Whenever possible do not use this older syntax. Support for it might be withdrawn in the future.

Appendix B
PFA Directives

This appendix contains the following sections:

- **"Standard Directives"**
- **"Cray Directives"**
- **"VAST Directives"**

This appendix lists and describes the three types of PFA directives:

- Standard
- Cray
- VAST

Chapter 1, "Overview of PFA," describes the purpose of directives. For details about how to use directives, refer to **Chapter 5, "Fine–Tuning PFA."**

Standard Directives

This section lists and describes the following standard PFA directives alphabetically:

- **C*$*ARCLIMIT**
- **C*$*CONCURRENTIZE**
- **C*$*INLINE**
- **C*$*IPA**
- **C*$*LIMIT**
- **C*$*MINCONCURRENT**
- **C*$*NOCONCURRENTIZE**
- **C*$*NOINLINE**
- **C*$*NOIPA**
- **C*$*OPTIMIZE**
- **C*$*ROUNDOFF**
- **C*$*SCALAR OPTIMIZE**
- **C*$*UNROLL**
- **C$*DOACROSS**
- **C$&**

C*$* ARCLIMIT

The **C*$*ARCLIMIT(*n*)** directive controls the size of the internal table used to store data dependence information (arcs). *n* is an integer. This directive, when specified globally, has the same effect as the **–ARCLIMIT** command line option.

C*$* CONCURRENTIZE

The **C*$*CONCURRENTIZE** directive converts eligible loops to run in parallel. This directive, when specified globally, has the same effect as the **–C*$*CONCURRENTIZE** command line option. See also **C*$*NOCONCURRENTIZE**.

C*$* INLINE

The **C*$*INLINE** directive behaves much like the **–INLINE** command line option but specifies which occurrences of a routine are actually inlined. The format for this directive is

```
C*$* INLINE [(name[,name  ... ] ) ] {HERE | ROUTINE | GLOBAL}
```

where

name	Specifies the routines to be inlined. If you do not specify a name, all routines will be affected.
HERE	Inlines 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.

See also **C*$*NOINLINE**.

For details about inlining, refer to **Chapter 4, "Customizing PFA Execution."** For details about using the **C*$*INLINE** directive, refer to **Chapter 5, "Fine–Tuning PFA."**

C*$* IPA

The **C*$* IPA** directive behaves much like the **–IPA** command line option but specifies on which occurrences of a routine to use IPA. The format for this directive is

```
C* $ * IPA [ (name [, name ... ])]  {HERE|ROUTINE|GLOBAL}
```

where

name	Specifies the routines to be IPAed. If you do not specify a name, all routines will be affected.
HERE	Uses IPA only on occurrences of the named routines that appear on the next line.
ROUTINE	Uses IPA on the named routines everywhere they appear in the current routine
GLOBAL	Uses IPA on the named routines throughout the source file.

See also **C*$*NOIPA**.

For details about interprocedural analysis, refer to **Chapter 4, "Customizing PFA Execution."** For details about using the **C*$*IPA** directive, refer to **Chapter 5, "Fine–Tuning PFA."**

C*$* LIMIT

The **C*$*LIMIT(***n***)** directive reduces PFA processing time by limiting the amount of time PFA can spend on trying to determine whether a loop is safe to run in parallel. PFA 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, PFA ignores the outer loop and recursively visits the inner loops.

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

C*$* NOCONCURRENTIZE

The **C*$*NONCONCURRENTIZE** option prevents PFA from converting loops to run in parallel. See also **C*$*CONCURRENTIZE**.

C*$* NOINLINE

The **C*$*NOINLINE** directive behaves much like the **–NOINLINE** command line option, but with the directive you can specify which occurrences of a routine are not inlined. The format for this directive is

```
C*$* NOINLINE [(name [,name ... ])] {HERE|ROUTINE|GLOBAL}
```

where

name	Specifies the routines to be inlined. If you do not specify a name all routines will be affected.
HERE	Disables inlining of occurrences of the named routines only on the next line.
ROUTINE	Disables inlining of the named routines everywhere they appear in the current routine.
GLOBAL	Disables inlining of the named routines throughout the source file.

C*$*NOINLINE overrides the **–INLINE** command line option and so allows you to disable inlining of the named routines at specific points.

C*$* NOIPA

The **C*$*NOIPA** directive behaves much like the **–NOIPA** command line option, but with the directive

you can specify on which occurrences of a routine to not use IPA. The format for this directive is

```
C*$* NOIPA [(name [, name ... ])] { HERE|ROUTINE|GLOBAL}
```

where

name	Specifies the routines to disable IPA. If you do not specify a name all routines will be affected.
HERE	Disables IPA of occurrences of the named routines only on the current routine
ROUTINE	Disables IPA of the named routines everywhere they appear in the current routine.
GLOBAL	Disables IPA of the named routines throughout the source file.

C*$*NOIPA overrides the **–IPA** command line option and so allows you to disable IPA of the named routines at specific points.

C*$*OPTIMIZE

The **C*$*OPTIMIZE(*n*)** directive sets the optimization level. The higher the optimization level, the more code is optimized and longer PFA runs. Valid values for *n* are the integers

0	Avoids converting loops to run in parallel.
1	Converts loops to run in parallel without using advanced data dependence tests. Enable 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 level recognizes arrays as local variables. Level 5 also tells PFA 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 PFA runs a reduction operation in parallel. Valid values for *n* are

0-1	Suppresses any round–off changing transformations.
2	Allows reductions to be performed in parallel. The valid reduction operators are addition, multiplication, min, and max. **–ROUNDOFF=2** is one of the most common user options.
3	Recognizes **REAL** induction variables. Permits the memory management transformations.

C*$*SCALAR OPTIMIZE

The **C*$*SCALAR OPTIMIZE (*n*)** directive controls the amount of standard scalar optimizations attempted by PFA. Valid values for *n* are

0	Performs no scalar transformations.
1	Enables dead code elimination, pulling loop invariants, forward substitution, and conversion of **IF–GOTO** into **IF–THEN–ELSE**.
2	Enables induction variables recognition, loop unrolling, loop fusion, array expansion, scalar promotion, and floating invariant **IF** tests. (Loop fusion also requires **–OPTIMIZE=5**.)

3 Enables the memory management transformations. (Memory management also requires **–ROUNDOFF=3**.)

C*$*UNROLL

The **C*$*UNROLL (*n*)** directive unrolls scalar inner loops when PFA cannot run the loops in parallel. When PFA unrolls a loop, it replicates the body of the loop a certain number of times, making the loop ran faster. In this form, *n* has the same meaning as in the **–UNROLL=***n* command line option.

The **C*$*UNROLL(*n, m*)** option allows you to adjust the number of operations used when unrolling. In this form, *n* is as above and *m* is as in the **–UNROLL2=***m* command line option.

This form of unrolling applies only to the innermost loops in a nest of loops. You can unroll loops whether they execute serially or concurrently.

C$ DOACROSS

The **C$ DOACROSS** directive tells the Fortran 77 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 PFA not to modify the loop.

Note: PFA generates the **C$ DOACROSS** directive and inserts it into the code as the result of PFA's parallelism analysis.

C$&

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

Cray Directives

PFA supports the following Cray directives:

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

CDIR$ IVDEP

PFA interprets the **CDIR$ IVDEP** directive as if it were a **C*$* ASSERT DO (CONCURRENT***)* assertion. (Refer to **Appendix C, "PFA Assertions,"** for details.)

CDIR$ NEXT SCALAR

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

VAST Directives

PFA supports the **CVD$CONCUR** VAST directive. The **CVD$CONCUR** directive runs a loop in parallel to optimize performance. PFA interprets this directive as if it were the **C*$*CONCURRENTIZE** directive (described in **"Standard Directives"**).

Appendix C

PFA Assertions

This appendix lists and describes the following PFA assertions alphabetically:

- **C*$* ASSERT ARGUMENT ALIASING**
- **C*$* ASSERT DO (SERIAL)**
- **C*$* ASSERT DO (CONCURRENT)**

- **C*$* ASSERT DO PREFER (SERIAL)**
- **C*$* ASSERT DO PREFER (CONCURRENT)**
- **C*$* ASSERT EQUIVALENCE HAZARD**
- **C*$* ASSERT NO ARGUMENT ALIASING**
- **C*$* ASSERT NO EQUIVALENCE HAZARD**
- **C*$* ASSERT RELATION** (*name .xx. name*)
- **C*$* ASSERT CONCURRENT CALL**
- **C*$* ASSERT NO RECURRENCE**
- **C*$* ASSERT PERMUTATION** (*name*)

Chapter 1, "Overview of PFA," describes the purpose of assertions. For details about using assertions, refer to **Chapter 5, "Fine–Tuning PFA."**

C*$* ASSERT ARGUMENT ALIASING

The **C*$* ASSERT ARGUMENT ALIASING** assertion allows PFA to make assumptions about subprogram arguments in a program. According to the Fortran 77 standard, you can alias a variable only if you do not modify (that is, write to) the aliased variable. This assertion tells PFA that the subprogram on the following line violates the Fortran 77 standard in this regard.

C*$* ASSERT DO (SERIAL)

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

C*$* ASSERT DO (CONCURRENT)

The **C*$* ASSERT DO (CONCURRENT)** assertion tells PFA to ignore assumed data dependencies. Normally, PFA is conservative about what loops it converts run in parallel. When PFA 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, PFA does not run "not sure" loops in parallel. **C*$* ASSERT DO (CONCURRENT)** tells PFA to go ahead and run "not sure" loops in parallel.

Note: If PFA 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 PREFER (SERIAL)

The **C*$* ASSERT DO PREFER (SERIAL)** assertion indicates that you want to execute a **DO** loop in serial mode. This assertion directs PFA 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) PFA chooses 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. PFA runs other nested loops in parallel only if a condition prevents running the selected loop in parallel.

The **C*$* ASSERT DO PREFER (CONCURRENT)** assertion applies only to the **DO** loop that it precedes. PFA does not generate parallel code if you use the **–NOCONCURRENTIZE** command line option or the **C*$* NOCONCURRENTIZE** directive.

C*$* ASSERT EQUIVALENCE HAZARD

The **C*$* ASSERT EQUIVALENCE HAZARD** assertion allows equivalence variables to refer to the same memory location inside one loop. This assertion, when specified globally, has the same effect as the **–ASSUME=E** command line option. The **C*$* ASSERT EQUIVALENCE HAZARD** assertion is active until you reset it or until the end of the program unit. See also **C*$* ASSERT NO EQUIVALENCE HAZARD**.

C*$* ASSERT CONCURRENT CALL

C*$* ASSERT CONCURRENT CALL tells PFA 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 NO ARGUMENT ALIASING

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

C*$* ASSERT NO EQUIVALENCE HAZARD

The **C*$* ASSERT NO EQUIVALENCE HAZARD** assertion tells PFA 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 memory location.

C*$* ASSERT NO RECURRENCE

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

C*$* ASSERT PERMUTATION

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

C*$* ASSERT RELATION

The **C*$* ASSERT RELATION (***name1 .xx. name2***)** assertion explicitly states the relationship between *name1* and *name2*. *name1* and *name2* are two variables or a variable and a constant, and *xx* is any of the following: **GT**, **GE**, **EQ**, **NE**, **LT**, or **LE**. This assertion applies only to the **DO** statement it precedes.

If you specify this assertion globally, the program uses the assertion only when *name1* and *name2* appear in **COMMON** blocks or are dummy argument names to the subprogram.

Glossary

action summary
: The portion of the listing file that summarizes PFA's actions.

assertion
: A PFA directive that asserts something about the program. For example, an assertion can assert that a particular array is a permutation vector. PFA 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 PFA. For example, directives enable, disable, or modify a feature of PFA.

global assertion
An assertion that is placed on the first line of the input file. PFA interprets global assertions as if they appear 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. PFA interprets global directives as if they appear 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.

intermediate file
A transformed version of a Fortran source program generated by PFA. This file name has the suffix *.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 PFA. 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 Accelerator (PFA)
A source−to−source preprocessor that analyzes a program and identifies loops that do not contain data dependencies.

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*.