MIPSpro™ Power Fortran 77 Programmer's Guide

Introduction

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

Organization of Information

This guide contains the following chapters and appendixes:

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

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

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

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

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

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

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

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

The Glossary lists and defines terminology related to Power Fortran.

Additional Reading

Refer to the *MIPspro Fortran 77 Programmer's Guide* for information on the following topics:

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

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

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

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

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

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

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

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

Typographical Conventions

This guide uses the following conventions and symbols:

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

Here 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 Power Fortran

This chapter contains the following sections:

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

Overview

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

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

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

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

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

Strategy for Using Power Fortran

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

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

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

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

Command Line Options

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

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

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

Directives

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

Power Fortran directives allow you to specify Power Fortran options in addition to, or instead of, command line options. Directives placed on the first line of the input file are called *global directives*. Power Fortran interprets them as if they appear at the top of each program unit in the file. Use global directives to ensure that the program is compiled with the correct command line options. Directives appearing anywhere else in the file apply only until the end of the current program unit. Power Fortran resets the value of the directive to the global value at the start of the next program unit. (Set the global value using a command line option or a global directive.)

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

Power Fortran supports the following standard directives:

C*$*ARCLIMIT(n)*
C*$*[NO]ASSERTIONS*
C*$*CONCURRENTIZE
C*$* EACH_INVARIANT_IF_GROWTH(n)*
C*$*INLINE*
C*$*IPA*
C*$*LIMIT(n)
C*$*MAX_INVARIANT_IF_GROWTH(n)*
C*$*MINCONCURRENT(n)*
C*$*NOCONCURRENTIZE
C*$*NOINLINE*
C*$*NOIPA*
C*$*OPTIMIZE(n)*
C*$*ROUNDOFF(n)*
C*$*SCALAR OPTIMIZE(n)*
C*$*UNROLL(n[,m])*
C$*
C$DOACROSS
C$&
C$CHUNK*
C$COPYIN*
C$MP_SCHEDTYPE*

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

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

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

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

Power Fortran supports the Cray[TM] directives listed below, which it maps to corresponding Power Fortran assertions. Refer to **Chapter 5, "Fine–Tuning Power Fortran,"** for details.

- **CDIR$ NEXT SCALAR**
- **CDIR$ NO RECURRENCE**[*]
- **CDIR$ IVDEP**

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

- **CVD$ CNCALL**
- **CVD$ CONCUR**
- **CVD$ [NO]DEPCHK**[*]
- **CVD$ [NO]LSTVAL**[*]

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

Assertions

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

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

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

Table 1–1 lists the assertions Power Fortran accepts and their duration.

Table 1–1 Power Fortran Assertions and Their Duration

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

C*$* ASSERT [NO] TEMPORARIES FOR CONSTANT ARGUMENTS[a]	Next loop

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

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

Summary

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

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

Chapter 2
How to Use Power Fortran

This chapter contains the following sections:

- **"Overview"** describes how to prepare for using Power Fortran.
- **"Compiling Programs With Power Fortran"** explains how to run Power Fortran.

Overview

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

When trying to find loops to run in parallel, focus your efforts on the areas of the code that use the bulk of the run time. Spending time trying to run a routine in parallel that uses only 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 sampling (through the -**p** option to *f77*(1)) or basic block profiling (through *pixie*(1)). Refer to the *MIPSpro Compiling, Debugging, and Performance Tuning Guide* for details about profiling. Alternatively, you can use the WorkShop Pro MPF Parallel Analyzer View to examine the performance of your program. Refer to the *CASEVision/WorkShop Pro MPF User's Guide* for details.

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

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

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

Compiling Programs With Power Fortran

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

Syntax

```
f77 f77_options -pfa[{list|keep}] [-WK,option[=value] [,option[=value]]...]
filename
```

where

f77_options Specifies any *f77* compiler options. Refer to the *f77*(1) manual page and the *MIPSpro Fortran 77 Programmer's Guide* for details.

-pfa Requests automatic parallelization of loops. Enables any multiprocessing directives.

list Specifies an annotated listing of the parts of the program that can (and cannot) run in parallel on multiple processors. The listing file has the suffix **.l**.

keep Generates the listing file (**.l**), saves the transformed equivalent Fortran 77 program (**.m**), and creates an output file for use with WorkShop Pro MPF (**.anl**).

-WK Passes the specified command line options to Power Fortran. Do not enter spaces between **-WK** and any of the hyphens, *options*, equal signs, and *values* that follow it.

option Specifies a Power Fortran command line option listed in **Table 2−1** , for example, **-concurrentize**.

value Specifies a value for a command line option, for example, 1.

filename Specifies the Fortran 77 source program. The filename must always use the **.f**, **.F**, **.for**, or **.FOR** suffix.

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

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

Table 2−1 Power Fortran Command Line Options

Reference	Long Name	Short Name	Default Value
Parallelization	-[no]concurrentize	-[n]conc	-concurrentize
	-minconcurrent=*n*	-mc=*n*	-minconcurrent=500
	-[no]parallelio	-[no]pio	(option off)
General Optimization[a]	-assume=*list*	-as=*list*	-assume
	-fuse	-fuse	

	-optimize=*n*	-o=*n*	=el
	-roundoff=*n*	-r=*n*	-fuse
	-scalaropt=*n*	-so=*n*	depends on -O*n*
			depends on -O*n*
			depends on -O*n*
Inlining and Interprocedural Analysis[a]	-inline[=*list*]	-in[=*list*]	(option off)
	-ipa[=*list*]	-ipa[=*list*]	(option off)
	-inline_create=*name*	-incr=*name*	(option off)
	-ipa_create=*name*	-ipacr=*name*	(option off)
	-inline_from_files=*list*	-inff=*list*	(option off)
	-ipa_from_files=*list*	-ipaff=*list*	(option off)
	-inline_from_libraries=*list*	-infl=*list*	(option off)
	-ipa_from_libraries=*list*	-ipafl=*list*	(option off)
	-inline_loop_level=*n*	-inll=*n*	-inll=10
	-ipa_loop_level=*n*	-ipall=*n*	-ipall=10
	-inline_man	-inm	(option off)
	-ipa_man	-ipam	(option off)
	-inline_depth	-ind	-ind=10
Directive Control[a]	-[no]directives=*list*	-[n]dr=*list*	-directives=ackpv
Listing	-lines=*n*	-ln=*n*	-line

	-listoptions=*list* -suppress=*list*	-lo=*list* -su=*list*	s=55 -listoptions=k (option off)
Advanced Optimization	-aggressive=*letter*[a] -arclimit=*n*[a] -cacheline=*n*[a] -cachesize=*n*[a] -chunk=*n*[a] -dpregisters=*n*[a] -each_invariant_if_growth=*n*[a] -fpregisters=*n*[a] -limit=*n* -max_invariant_if_growth=*n*[a] -[no]recursion[a]-setassociativity=*n*[a] -unroll=*n*[a] -unroll2=*n*[a]	-ag=*letter* -arclm=*n* -chl=*n* -chs=*n* -chk=*n* -dpr=*n* -eiifg=*n* -fpr=*n* -lm=*n* -miifg=*n* -[no]rc sasc=*n* -ur=*n* -ur2=*n*	(option off) -arclimit=5000 -cacheline=4 -cachesize=256 -chunk=1 -dpregisters=16 -each_invariant_if_growth=20 -fpregisters=16 -limit=20000 -max_invariant_if_growth=500 -recursion -setass

ass
ocia
tivity
=1
-unr
oll=
4
-unr
oll2
=10
0

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

The **-pfa** and **-O***n* options enable certain scalar optimizations that are equivalent to various combinations of the **-WK** option with the **-scalaropt** (or **-so**), **-roundoff** (or **-r**), and **-optimize** (or **-o**) options (described in detail in the *MIPSpro Fortran 77 Programmer's Guide*).

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

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

Example

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

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

Figure 2-1 shows what happens when you compile a Fortran 77 program with **-pfa**. The first pass invokes the macro preprocessor *cpp* to handle *cpp* directives. (For more information, see the *cpp*(1) manual page.) The Power Fortran 77 compiler, *fef77p*, then takes the *cpp* output and inserts code that runs data-independent loops in parallel. It also generates intermediate code.

Note: The MIPSpro Power Fortran 77 compiler, *fef77p*, is a separate compiler from the regular MIPSpro Fortran 77 compiler, *fef77*.

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

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

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

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

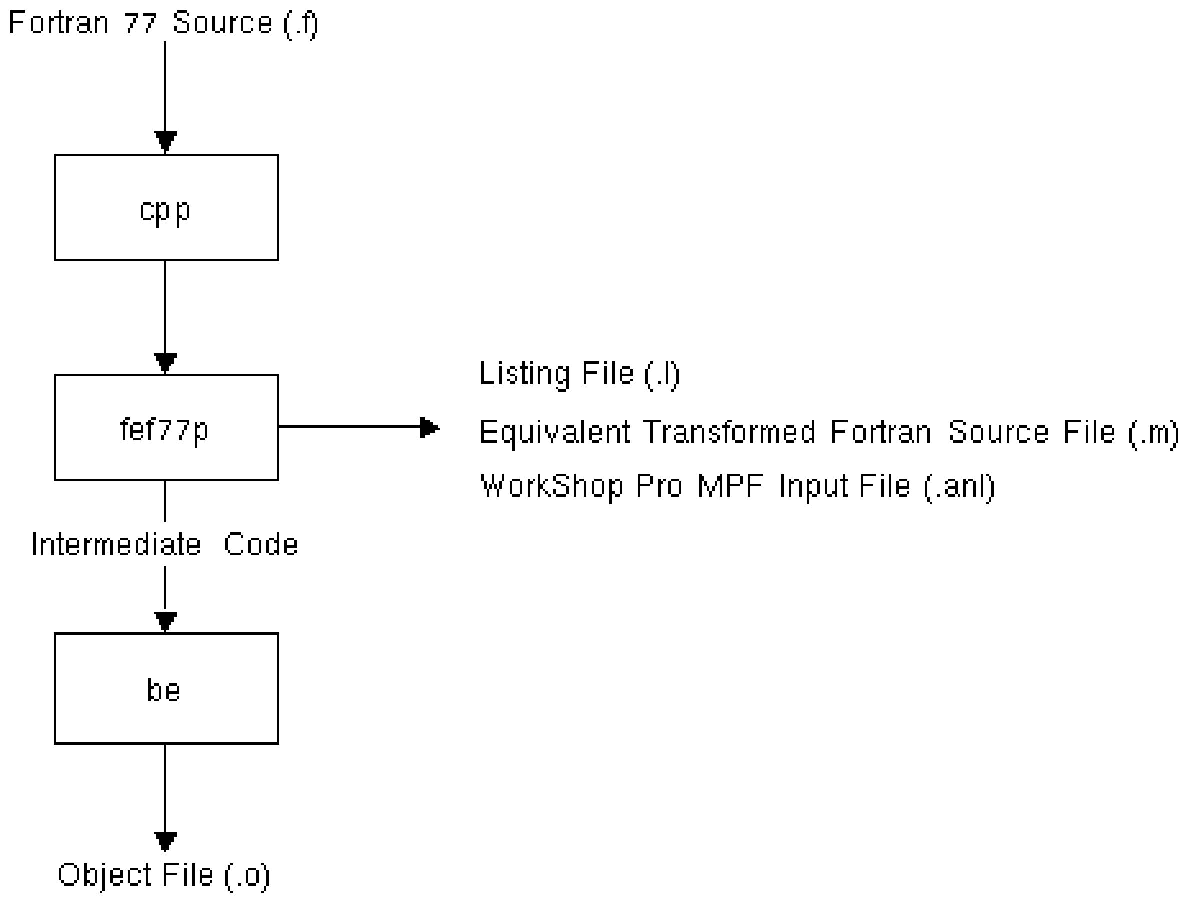

Figure 2−1 Compiling With Power Fortran

Chapter 3

Utilizing Power Fortran Output

This chapter contains the following sections:

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

Overview

Power Fortran can generate three types of output files:

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

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

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

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

Compiling **sample.f** as follows

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

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

```
Footnotes Actions   Do Loops Line
            DIR                 1  # 1  "sample.f"
                                2       subroutine sample(a,b,c)
                                3       dimension  a(1000),b(1000),c(1000)
            SO C      +-------- 4       do 10 i = 1,1000
            SO        *________ 5  10   a (i) = b(i) + c(i)
                                6       end
Abbreviations Used
     SO     scalar optimization
     DIR    directive
     C      concurrentized
Loop Summary
         From  To    Loop     Loop    at
Loop#  line  line  label    index   nest   Status
1      4     5     Do 10    I       1      concurrentized
```

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

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

Formatting the Listing File

You can customize a Power Fortran listing file by

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

Paginating the Listing

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

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

Specifying Information to Include

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

Table 3–1 Listing File Include Options

Value	Produces
c	Calling tree at the end of the program listing.
i	Transformed program file annotated with line numbers in the source program. Error messages

and debugging information can refer to the original source rather than the transformed source. This argument is specified by default.

k	List of the Power Fortran options used at the end of each program unit.
l	Loop−by−loop optimization table.
n	Program unit names, as processed, to the standard error file. This option is added automatically as part of an *f77* -v compilation.
o	Annotated listing of the original program.
p	Processing performance statistics.
s	Summary of optimizations performed.
t	Annotated listing of the transformed program.

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

```
% f77 −pfa −WK,−listoptions=ls source.f
```

Disabling Message Classes

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

Table 3−2 Listing File Message Disabling Options

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

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

Interpreting Default Listing Information

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

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

Viewing the Listing File

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

```
% wsh −s132,40
```

Field Descriptions

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

A default Power Fortran listing file includes

- line numbers

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

Line Numbers

A statement in the listing transformed by Power Fortran labeled with a line number, such as 21, is the same as line 21 from the original program or has been derived from that line. These line numbers are useful when inspecting the transformed program listing and when debugging. Power Fortran sometimes generates several lines of code from a single line of the original program; in this case, each new line of code is labeled with the same number as the line of the original program from which it was generated. Consequently, many lines of the transformed program listing carry the same number because they are related to one line of the original program listing.

DO Loop Marking

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

Table 3–3 Listing File DO Loop Delimiters

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

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

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

Footnotes

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

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

appears under Footnote List at the end of the program unit

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

In this example, **13** is the footnote number, **DD** (data dependence) is the explanation for Power Fortran'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 Power Fortran translates or modifies a statement, it uses abbreviations in the Actions column of the listing file to identify the statements. Power Fortran lists an abbreviated explanation of its actions at the bottom of the listing. For the **DIR** and **V** classes, the class itself serves as the message without detailed messages. All other classes have associated messages.

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

Table 3–4 Power Fortran Action Abbreviations

Value	Meaning
DD	(Data Dependence) Indicates that data dependence prevented Power Fortran from running this statement in parallel.
DIR	(Directive) Used in conjunction with the footnotes and concerns compiler directives. If you code a compiler directive and that line does not have the DIR abbreviation in the listing, Power Fortran will not recognize the directive. Check the setting of the -directives command line option and the syntax of the directive.
E	(Error) Indicates syntax errors. These messages can refer to missing or extra characters, illegal keywords, or text placed in the wrong column. Power Fortran cannot do anything with such code. The equivalent transformed source file (.m) contains a copy of this program unit that Power Fortran has not modified.
EX	(Extension) Shows where a construct in the original program is not allowed in the language Power Fortran produces. In some cases, an operation or type is allowed in the input language but not in the output language.
INF	(Information) Provides noncritical information.
I	(Insertion) Indicates that Power Fortran added a statement.
LR	(Loop Reordering) Indicates that Power Fortran has modified a Fortran 77 statement in the process of interchanging loops. If during optimization Power Fortran ascertains that an outer loop would be more efficient as an inner loop, and it can legally reorder the loops, Power Fortran places the outer loop inside. In the process of this reordering, Power Fortran might have to change loop bounds (for triangular loops), distribute loops, or float IF assignments. Only the statements modified for the exchange are marked.
MIS	(Miscellaneous) Indicates that some Power Fortran information has been lost. This message does not always mean that something is wrong with the program.
NX	(Nonconcurrent Statement) Indicates that Power Fortran did not try or was unable to run the statement in parallel. For example, when a subroutine call is involved in a loop, Power Fortran generates this message.
NO	(Program Too Large—Not Optimized) Indicates that the program unit being processed is too large for Power Fortran to optimize, because of Power Fortran's data structure size limitations. When Power Fortran optimizes programs, it adds statements that might also overflow the fixed–size tables. In either case, Power Fortran stops optimization and passes the original program to the equivalent transformed source file (.m), informing you of this action. For Power Fortran to process the unit, you must split the program into smaller sections.
OE	(Option Error) Indicates a syntax error in a Power Fortran option. This error does not stop processing of a program unit.
OTF	(Output Translation Failure) Marks statements that have constructs that exist in the input

	language but that cannot be represented in the output language.
Q	(Question) Indicates that Power Fortran tried to optimize a loop nest but discovered a data dependence it could not break at compile time without further information. You can usually answer this question with an appropriate assertion.
SO	(Scalar Optimization) Marks places in the transformed listing where Power Fortran has optimized a scalar loop.
STD	(Standardized) Marks where Power Fortran changed a program to improve the chance of finding code that it can optimize. This is often a conversion from an IF/GOTO into a block IF, loop rerolling, and conversion of an IF loop to a DO loop.
TE	(Translator Error) Indicates an internal Power Fortran error. Power Fortran writes the notification to the standard error file and writes a trace back to the output file. Notify Silicon Graphics if you see this sort of bug (so it can be corrected) and, if possible, send Silicon Graphics the code that caused the trace back as well as the trace back itself. If you can reproduce the error in a small program unit, send that small program unit as well.
W	(Warning) Contains syntax warnings.

Sample Listing Files

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

Indirect Indexing

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

The Fortran 77 code

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

when submitted to Power Fortran, produces the listing file

```
Footnotes  Actions     DO Loops   Line
             DIR                    1  # 1 "foo2.f"
                                    1    subroutine foo2(a,b,index,n)
                                    2    real a(n), b(n)
                                    3    integer index(n)
                                    4
1          Q SO        +-------   5    do i = 1, n
2          DD SO       !          6        a(index(i)) = a(index(i)) + b(i)
                         !________  7    enddo
                                    8    end
Abbreviations Used
   DD       data dependence
   Q        question
   SO       scalar optimization
   DIR      directive
Footnote List
   1: question            Is "INDEX" a permutation vector?
   2: data dependence     Data dependence involving this line due
                          to variable A.
Loop Summary
```

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

DD in the Actions column on line 6 of the listing warns that the variable **a** might carry a dependency. A dependency exists when one iteration of the loop writes to a location that is used by a different iteration of the loop. In this example, if the values of **index(i)** are ever the same for different values of **i**, then different iterations might use the same location in **a**. Therefore, this code contains a possible data dependence.

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

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

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

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

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

Abbreviations Used
   SO    scalar optimization
   DIR   directive
   C     concurrentized

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

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

Function Call

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

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

generates the listing

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

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

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

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

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

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

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

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

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

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

The question is whether the variable **m** should be **SHARE** or **LOCAL**. If the routine **my_function** 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, Power Fortran must go by what it can see; and what it can see is that within the loop, there are no visible assignments to **m**, and so Power Fortran will declare it to be **SHARE**. If in fact **my_function** is writing the value of **m**, then this is incorrect. In this case, to give Power Fortran the hint it needs, add a visible assignment to **m** at the top of the loop.

For example, consider the following code:

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

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

Reductions

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

Consider the 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, Power Fortran produces the listing file

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

```
Abbreviations Used
   DD        data dependence
   SO        scalar optimization
   DIR       directive

Footnote List
1: data dependence       Data dependence involving this
                           line due to variable "SUM".
Loop Summary
         From  To    Loop   Loop    at
Loop#  line  line  label  index   nest    Status
1      6     8     Do     I       1       scalar mode preferable
```

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

Because the parallel version of the code adds the elements together in 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 Power Fortran to use parallel subtotals. To tell Power Fortran not to worry about round−off error, you can use either the **C*$*ROUNDOFF(2)** directive or the *f77* command line option -**WK**,-**roundoff=2**.

The resulting listing file is

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

Abbreviations Used
   SO        scalar optimization
   DIR       directive
   C         concurrentized

Footnote List
1: data dependence       Data dependence involving this
                           line due to variable "SUM".
Loop Summary
         From  To    Loop   Loop    at
Loop#  line  line  label  index   nest    Status
1      6     8     Do     I       1       concurrentized
```

Be aware that the round−off error produced by the parallel reduction operation is not necessarily any worse than the round−off error already present in the original serial version. It 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 Power Fortran supports.

Table 3–5 Power Fortran Reductions

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 Power Fortran Execution

This chapter contains the following sections:

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

Overview

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

This chapter describes the options that are recognized only by Power Fortran. For details about the options that control scalar optimizations, refer to the *MIPSpro Fortran 77 Programmer's Guide*.

Controlling Code Execution

When modifying most programs to allow loops to run in parallel, modify the code so that Power Fortran can automatically run the loop in parallel. 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 Power Fortran) need to verify that no subsequent modification inserts data dependencies. Forcing these data dependencies in code to run in parallel can produce serious (and difficult–to–find) errors. Rewriting the loop so that Power Fortran recognizes the loop as safe to run in parallel allows Power Fortran to check future modifications for potential data dependencies.

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

Running Code in Parallel

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

Loops requiring the addition of synchronization might run slower than the scalar original when concurrentized. In this case, you can specify the **-noconcurrentize** command line option or the **C*$*NOCONCURRENTIZE** directive for a particular loop.

Specifying a Work Threshold

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

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

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

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

For example, given the original loop

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

Power Fortran generates the following transformed loop:

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

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

Enabling Parallel I/O

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

Controlling Power Fortran Code Transformations

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

Specifying a Complexity Limit

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

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

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

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

Setting the Optimization Level

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

0	Avoids converting loops to run in parallel.
1	Converts loops to run in parallel without using advanced data dependence tests. Enables loop interchanging.
2	Determines when scalars need last−value assignment using lifetime analysis. Also uses more powerful data dependence tests to find loops that can run safely in parallel. This level allows reductions in loops that execute concurrently but only if the **-roundoff** option is set to 2. (Refer to the following section for details about the **-roundoff** option.)
3	Breaks data dependence cycles using special techniques and additional loop interchanging methods, such as interchanging triangular loops. This level also implements special−case data dependence tests.
4	Generates two versions of a loop, if necessary, to break a data−dependent arc. This level also implements more−exact data dependence tests and allows special index sets (called wraparound variables) to convert more code to run in parallel.
5	Fuses two adjacent loops if it is legal to do so (that is, there are no data dependencies) and if the loops have the same control values. In certain limited cases, this level recognizes arrays as local variables. This level is the default.

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

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

Controlling Variations in Round Off

The **-roundoff=***n* option (or **-r=***n*) controls the amount of variation in round off that Power Fortran 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 +, *, *min*, and *max*. This value is one of the most commonly−specified user options.
3	Recognizes **REAL** induction variables. Permits memory management transformations (refer to the *MIPSpro Fortran 77 Programmer's Guide* for details.)

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

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

Performing Inlining and Interprocedural Analysis

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

- Assert that the external routine is safe for concurrent execution (see **"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. For details about inlining and IPA, and the related directives and command line options, refer to the *MIPSpro Fortran 77 Programmer's Guide*.

Chapter 5

Fine−Tuning Power Fortran

This chapter contains the following sections:

- **"Overview"** explains how to fine−tune program execution using directives and assertions.
- **"Circumventing Power Fortran"** explains how to use directives to bypass Power Fortran's analysis and leave areas of code unchanged.
- **"Running Code Serially"** explains how to use directives and assertions to stop Power Fortran from running specific code in parallel.
- **"Running Code in Parallel"** explains how to use directives and assertions to tell Power Fortran that it is safe to run specific parts of code in parallel.
- **"Ignoring Data Dependencies"** explains how to tell Power Fortran that apparently data−dependent code is safe to run in parallel.

Overview

After you run a Fortran source program through Power Fortran once, you can use directives and assertions to fine−tune program execution. The listing file will show where and why Power Fortran did not parallelize the code. You can also use WorkShop Pro MPF to review Power Fortran's analysis of your program.

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

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

Because Power Fortran does not check the correctness of assertions, they can be unsafe. If you specify an incorrect assertion, the code generated by Power Fortran might give different answers from the scalar program. If you suspect unsafe assertions are causing problems, use the **-nodirectives** command line option or the **C*$* NO ASSERTIONS** directive to tell Power Fortran to ignore all assertions (both described in the *MIPSpro Fortran 77 Programmer's Guide*).

Circumventing Power Fortran

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 Power Fortran from analyzing your modified code.

C$ DOACROSS

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

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

Power Fortran runs the following code as it appears:

```
C$ DOACROSS
```

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

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)
```

C*$* NO SYNC

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

Running Code Serially

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

C*$* ASSERT DO (SERIAL)

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

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

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

See also **"C*$* ASSERT DO PREFER (SERIAL)"** .

CDIR$ NEXT SCALAR

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

C*$* ASSERT DO PREFER (SERIAL)

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

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

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

Running Code in Parallel

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

C*$*[NO]CONCURRENTIZE

The **C*$*[NO]CONCURRENTIZE** directive converts eligible loops to run in parallel. The **NO** version prevents Power Fortran from converting loops to run in parallel. These directives override the -**[no]concurrentize** command line option. For example, if your program contains the **C*$*NOCONCURRENTIZE** directive, parallelization is disabled even if you compile with **-concurrentize**. When specified globally, these directives have the same effect as the **-concurrentize** and **-noconcurrentize** options (see **"Running Code in Parallel" in Chapter 4** for details).

CVD$ CONCUR

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

C*$* ASSERT DO PREFER (CONCURRENT)

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

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

Consider the following code:

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

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

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

Using Aliasing

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

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

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

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

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

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

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

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

Ignoring Data Dependencies

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

C*$* ASSERT DO (CONCURRENT)

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

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

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

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

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

CDIR$ IVDEP

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

C*$* ASSERT CONCURRENT CALL

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

CVD$ CNCALL

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

C*$* ASSERT NO RECURRENCE

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

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

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

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

C*$* ASSERT PERMUTATION

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

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

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

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

Appendix A

Power Fortran Command Line Options

This appendix contains the following sections:

- **"Overview"**
- **"Options Summary"**

Overview

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

Table A–1 summarizes the Power Fortran command line options.The Reference column lists the functional categories of the following options:

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

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

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

Table A−1 Power Fortran Command Line Options

Reference	Long Name	Short Name	Default Value
Parallelization	-[no]concurrentize -minconcurrent=*n* -[no]parallelio	-[n]conc -mc=*n* -[no]pio	-concurrentize -minconcurrent=500 (option off)
General Optimization[a]	-assume=*list* -fuse -optimize=*n* -roundoff=*n* -scalaropt=*n*	-as=*list* -fuse -o=*n* -r=*n* -so=*n*	-assume=el -fuse depends on -O*n* depends on -O*n* depends on -O*n*
Inlining and Interprocedural Analysis[a]	-inline[=*list*] -ipa[=*list*] -inline_create=*name* -ipa_create=*name* -inline_from_files=*list* -ipa_from_files=*list* -inline_from_libraries=*list* -ipa_from_libraries=*list* -inline_loop_level=*n* -ipa_loop_level=*n*	-in[=*list*] -ipa[=*list*] -incr=*name* -ipacr=*name* -inff=*list* -ipaff=*list* -infl=*list* -ipafl=*list* -inll=*n* -ipall=*n*	(option off) (option off) (option off) (option off) (opti

	-inline_man -ipa_man -inline_depth	-inm -ipam -ind	on off) (option off) (option off) (option off) (option off) -inll=10 -ipall=10 (option off) (option off) -ind=10
Directive Control[a]	-[no]directives=*list*	-[n]dr=*list*	-directives=ackpv
Listing	-lines=*n* -listoptions=*list* -suppress=*list*	-ln=*n* -lo=*list* -su=*list*	-lines=55 -listoptions=k (option off)
Advanced Optimization	-aggressive=*letter*[a] -arclimit=*n*[a] -cacheline=*n*[a] -cachesize=*n*[a] -chunk=*n*[a] -dpregisters=*n*[a] -each_invariant_if_growth=*n*[a] -fpregisters=*n*[a] -limit=*n* -max_invariant_if_growth=*n*[a] -[no]recursion[a]-setassociativity=*n*[a] -unroll=*n*[a] -unroll2=*n*[a]	-ag=*letter* -arclm=*n* -chl=*n* -chs=*n* -chk=*n* -dpr=*n* -eiifg=*n* -fpr=*n* -lm=*n* -miifg=*n* -[no]rc sasc=*n* -ur=*n* -ur2=*n*	(option off) -arclimit=5000 -cacheline=4 -cachesize=256 -chu

nk=1
-dpregisters=16
-each_invariant_if_growth=20
-fpregisters=16
-limit=20000
-max_invariant_if_growth=500
-recursion
-setassociativity=1
-unroll=4
-unroll2=100

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

Options Summary

Overview

This section alphabetically lists and defines the command line options that are supported only by Power Fortran. Refer to the *MIPSpro Fortran 77 Programmer's Guide* for information about the remaining command line options.

concurrentize

The **-concurrentize** option, described in **Table A−2** , converts eligible loops to run in parallel.

Table A−2 concurrentize Option

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

See also **"noconcurrentize"** .

limit

The **-limit** option, described in **Table A−3** , reduces Power Fortran processing time by limiting the amount of time Power Fortran can spend trying to determine whether a loop is safe to run in parallel.

Table A−3 limit Option

Long Option Name	Short Option Name	Default Value
-limit=*n*	-lm=*n*	-limit=5000

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

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

lines

The **-lines** option, described in **Table A−4** , paginates the listing for printing.

Table A−4 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.

listoptions

The **-listoptions** option, described in **Table A−5** , specifies the information to include in the listing file *(***.l***)*.

Table A−5 listoptions Option

Long Option Name	Short Option Name	Default Value
-listoptions=*list*	-lo=*list*	-listoptions=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. This option is automatically specified.
k	Power Fortran option used at the end of each program unit.
l	Loop−by−loop optimization table.
n	Program unit names, as processed, to the standard error file. This option is added automatically as part of an *f77* **-v** compilation.
o	Annotated listing of the original program.
p	Processing performance statistics.
s	Summary of optimization performed.
t	Annotated listing of the transformed program.

minconcurrent

The **-minconcurrent** option, described in **Table A−6** , establishes the minimum amount of work needed inside the loop to make executing a loop in parallel profitable.

Table A−6 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 Power−Fortran−generated **DOACROSS** directive to test at run time whether sufficient work exists.

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

noconcurrentize

The **-noconcurrentize** option, described in **Table A−7** , prevents Power Fortran from converting loops to run in parallel.

Table A−7 noconcurrentize Option

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

See also **"concurrentize"** .

noparallelio

The **-noparallelio** option, described in **Table A−9** , disables the parallelization of loops that contain I/O statements.

Table A−8 noparallelio Option

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

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

See also **"parallelio"** .

parallelio

The **-parallelio** option, described in **Table A−9** , enables the parallelization of loops that contain I/O statements.

Table A−9 parallelio Option

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

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

See also **"noparallelio"** .

suppress

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

Table A−10 suppress Option

Long Option Name	Short Option Name	Default Value
-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 (Power Fortran adds the **-suppress=w** option automatically if you specify the -**w** option to *f77*)

Appendix B

Power Fortran Directives

This appendix contains the following sections:

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

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

Standard Directives

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

- **C*$*CONCURRENTIZE**
- **C*$*LIMIT**
- **C*$*MINCONCURRENT**
- **C*$*NOCONCURRENTIZE**
- **C*$*OPTIMIZE**
- **C*$*ROUNDOFF**
- **C$*DOACROSS**
- **C$&**

For details about directives that do not relate specifically to multiprocessing, refer to the *MIPSpro Fortran 77 Programmer's Guide*.

C*$* CONCURRENTIZE

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

C*$* LIMIT

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

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

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

C*$* MINCONCURRENT

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

C*$* NOCONCURRENTIZE

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

C*$*OPTIMIZE

The **C*$*OPTIMIZE(*n*)** directive sets the optimization level. The higher the optimization level, the more code is optimized and longer Power Fortran 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 Power Fortran to try harder to run the outermost loop possible (of a set of loops) in parallel.

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

C*$*ROUNDOFF

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

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$ 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 Power Fortran not to modify the loop.

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

C$&

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

Cray Directives

Power Fortran supports the following Cray directives:
- **CDIR$ IVDEP**
- **CDIR$ NEXT SCALAR**

CDIR$ IVDEP

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

CDIR$ NEXT SCALAR

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

VAST Directives

Power Fortran supports the following VAST directives:
- **CVD$ CNCALL**
- **CVD$ CONCUR**

CVD$ CNCALL

Power Fortran interprets the **CVD$ CNCALL** directive as if it were the **C*$* ASSERT CONCURRENT CALL** assertion (described in **"C*$* ASSERT CONCURRENT CALL" in Chapter 5**). The **CVD$ CNCALL** directive tells Power Fortran to ignore assumed dependencies caused by a subroutine call or function reference.

CVD$ CONCUR

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

Appendix C

Power Fortran Assertions

This appendix lists and describes the following Power Fortran assertions alphabetically:
- **C*$* ASSERT CONCURRENT CALL**
- **C*$* ASSERT DO (CONCURRENT)**
- **C*$* ASSERT DO (SERIAL)**
- **C*$* ASSERT DO PREFER (CONCURRENT)**
- **C*$* ASSERT DO PREFER (SERIAL)**
- **C*$* ASSERT [NO] LAST VALUE NEEDED**
- **C*$* ASSERT NO RECURRENCE**
- **C*$* ASSERT NO SYNC**
- **C*$* ASSERT PERMUTATION**
- **C*$* ASSERT RELATION**

This chapter describes the assertions that are supported only by Power Fortran. **Chapter 1, "Overview of Power Fortran,"** describes the purpose of assertions and provides a comprehensive list of the assertions supported by Power Fortran and the MIPSpro Fortran 77 compiler. For details about using assertions, refer to **Chapter 5, "Fine−Tuning Power Fortran."**

C*$* ASSERT CONCURRENT CALL

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

C*$* ASSERT DO (CONCURRENT)

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

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

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

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

C*$* ASSERT DO (SERIAL)

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

C*$* ASSERT DO PREFER (CONCURRENT)

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

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

C*$* ASSERT DO PREFER (SERIAL)

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

C*$* ASSERT [NO] LAST VALUE NEEDED

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

C*$* ASSERT NO RECURRENCE

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

C*$* ASSERT NO SYNC

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

C*$* ASSERT PERMUTATION

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

C*$* ASSERT RELATION

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

Glossary

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

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

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

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

global assertion
: An assertion that is placed on the first line of the input file. Power Fortran interprets global assertions as if they 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. Power Fortran 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.

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

interprocedural analysis (IPA)
: The process of analyzing an external routine ahead of time and using the results when the routine is referenced.

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

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

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

parallelize
Manipulating code so that it can be run in parallel.

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

permutation vector
Any list of numbers that are all different.

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

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

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

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

round−off error
The inaccuracy resulting from rounding off values in a calculation.

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

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