MIPSpro™ POWER Fortran 90 Programmer's Guide

Document Number 007-2760-001

CONTRIBUTORS

Written by David Cortesi (based on Power Fortran 77 text by Chris Hogue)
Production by Derrald Vogt
Engineering contributions by Ron Price, Ron Shapiro, Rohit Chandra..
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson,
Erik Lindholm, and Kay Maitz

MIPSpro™ POWER Fortran 90 Programmer's Guide
Document Number 007-2760-001

Contents

List of Examples

List of Tables

About This Guide

This Guide presents the features of MIPSpro™ Power Fortran 90, an extension to MIPSpro Fortran 90 that can automatically prepare existing Fortran 90 programs to execute in parallel on multiprocessor systems such as the Silicon Graphics, Inc. POWER Challenge™ and POWER Onyx™.

Using normal MIPSpro Fortran 90, you can modify a program using explicit source directives and assertions so that, when you compile the program with appropriate driver options, the program executes in parallel on a multiprocessor. These explicit source directives, assertions, and driver options are only summarized in this document; they are covered in detail in the *MIPSpro Fortran 90 Programmer's Guide*.

This Guide describes the unique features of MIPSpro Power Fortran 90, especially the ability to automatically analyze and modify a program for parallel execution.

Organization

The contents of this Guide are organized as follows.

- Chapter 1, "Using Power Fortran 90," gives an overview of the product, defines important terms, and shows how to invoke the product.
- Chapter 2, "Using Listing and Temporary Files," shows how to use the files written by Power Fortran 90, especially the listing file.
- Chapter 3, "Working With Power Fortran 90," tells how to use source directives and driver options to control the operations of Power Fortran 90.
- Appendix A, "Power Fortran Assertions," summarizes for reference the source assertions supported by Power Fortran 90.

- Appendix B, “Power Fortran Directives,” summarizes for reference the source directives supported by Power Fortran 90.
- Appendix C, “Power Fortran Driver Options,” summarizes for reference the driver options supported by Power Fortran 90.

Additional Reading

Refer to the *Fortran 90 Handbook* by Adams, Brainerd, et. al. (McGraw-Hill 1992; ISBN 0-07-000406-4) for a description of the Fortran language as implemented by MIPSpro Fortran 90.

Refer to the *MIPSpro Fortran 90 Programmer's Guide* for information on:

- compiling, linking, and running Fortran programs
- alignment, sizes, and value ranges of data types
- the interface between Fortran programs and programs written in other languages
- file management, run-time error handling, and other details related to the IRIX operating system
- IRIX system functions and subroutines callable by Fortran programs
- scalar optimizations that can be controlled through command line options and compiler directives
- source directives for multiprocessing, including PCF directives
- run-time error messages

Refer to the *CASEVision™/WorkShop Pro MPF User's Guide* for information about using WorkShop Pro MPF, a visual analyzer for parallel Fortran programs. Refer to the *dbx User's Guide* for information on the debugging tool *dbx*.

Refer to the *MIPS Compiling and Performance Tuning Guide* for an overview of the MIPSpro compiler system and general compiler system command line options, as well as information on:

- using the performance tools *prof* and *pixie*
- using dynamic shared objects (DSOs)

- the dump utilities, archiver, and other tools for maintaining Fortran programs

Refer to the *MIPSpro Porting and Transition Guide* for information on 64-bit systems versus 32-bit systems, including:

- writing and updating code that is portable to 64-bit systems
- language implementation differences
- porting source code to the 64-bit system
- compilation and run-time issues

Conventions Used in This Guide

These are the typographical conventions used in this guide.

Table Intro-1 Conventions

Purpose	Example
Names of Fortran keywords and procedures, and procedure names defined in example code	A function such as AINT must be named in an INTRINSIC statement. The module NEW_TYPE defines type TAX_PAYER.
Names of Fortran variables from example code.	The assignment of *A*(*j*-1) into *A*(*j*) creates a dependency.
Names of commands and options entered on the IRIX command line	The compiler driver is *f90*. Use *elfdump -t* to list external names in an object file.
Titles of manuals	Refer to the *dbx User's Guide*.
Filenames and pathnames	The compiler automatically includes *libftn90.so*, *libftn.so*, and *libm.so* from */usr/lib64*.
Full lines of example code or commands, including variable elements you supply	`f90 -g -mips4` *sourcename*`.f`
Exact quotes of computer output	`off end of record`

Chapter 1

Using Power Fortran 90

This chapter contains the following main topics:

- "Power Fortran 90 Concepts" on page 1 surveys the main concepts behind Power Fortran 90.
- "The Power Fortran Design Cycle" on page 2 gives an overview of how you normally use Power Fortran 90.
- "Tuning Power Fortran 90" on page 3 summarizes the control inputs you can use.
- "Relation to Parallel Directives" on page 4 reviews the relationship between Power Fortran and other parallel-execution directives.

Power Fortran 90 Concepts

MIPSpro Power Fortran 90 (called Power Fortran hereafter) is an extended version of the MIPSpro Fortran 90 compiler that enables you to compile existing Fortran 90 programs so that they use multiple CPUs concurrently when running on a Silicon Graphics multiprocessor system. Power Fortran analyzes a program, identifies loops that can safely execute in parallel, and generates a parallel version of the program.

When you execute the parallelized program on a multiprocessor, the program dynamically adjusts itself to use all the CPUs available on the system at the time, and so runs faster than when it runs serially on one CPU.

Power Fortran does not require a multiprocessor system to compile. The parallel version of your program can run on a uniprocessor with some slight loss of speed. Thus you can compile and test using Power Fortran on any Silicon Graphics workstation or server.

You select the code to be converted. You can convert the whole program or you can select key parts of it, either by adding directives or by applying Power Fortran only to selected source files. Object files produced by Power Fortran are fully compatible with object files that run only serially, and with object files that you prepared manually for parallel execution.

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.

The Power Fortran Design Cycle

Simply passing code through Power Fortran rarely produces all the increased performance available. Power Fortran analyzes the way your program uses variables, and it must make conservative assumptions about how one variable depends on others. As a result, Power Fortran often detects data dependencies that prevent it from converting a loop to parallel operation.

You use the listing file generated by Power Fortran to find out which loops it can run in parallel and which it cannot. When a loop cannot be parallelized, you use the listing to find out why. Often the reason is a real or assumed dependency that you can easily remove. By making small changes in the program, for example by adding an assertion statement, you enable Power Fortran to parallelize much more of the program.

You should not apply Power Fortran until you have a complete, working version of your Fortran 90 program. There are three reasons for this:

1. Power Fortran compilation takes longer than normal compilation, so it is good to be done with the frequent recompilations of the testing phase.
2. Debugging a parallel program can be difficult, so it is good to know that all functions of the program work correctly in serial execution before you attempt parallel execution.

3. The point of parallelization is improved performance, so it is good to have a complete test setup and to know the serial execution times for different inputs—otherwise you will not know whether parallel execution is doing any good or not.

When the program is ready, use Power Fortran in the following cycle:

1. Compile the program using MIPSpro Power Fortran 90, specifying the *-pfalist* driver option to generate a listing file.
2. Examine the listing file, which shows the structure of the program and how Power Fortran dealt with each loop.
3. If there are any obvious data dependencies or other changes to make, apply them and repeat from step 1.
4. Run the parallelized program against your test cases and measure its performance. If you think there is more to be gained, modify the source program and repeat from step 1.

Chapter 2, "Using Listing and Temporary Files," describes the contents of the Power Fortran listing and how you interpret it.

Tuning Power Fortran 90

You tune and control the work of Power Fortran in three ways:

- Using driver options, you set global limits and policies on the way Power Fortran works. For example, you use the *-minconcurrent* option to set a limit on the simplest loop that is worth parallelizing.
- Using assertions, which are Fortran comment lines that tell Power Fortran about your program, you inform Power Fortran of how to resolve data dependencies, and you mark specific loops to be ignored or to be parallelized.
- Using directives, which are Fortran comment lines that instruct Power Fortran, you set special handling of sections of the source code.

Chapter 3, "Working With Power Fortran 90," tells how to use driver options, directives, and assertions to fine-tune the output of Power Fortran.

Relation to Parallel Directives

When Power Fortran detects a loop that can be run in parallel, it generates directive statements and inserts them into the program. These directives are interpreted by MIPSpro Fortran 90. It is the Fortran 90 compiler that implements the parallel code, based on the generated directives.

You can write and insert parallel-execution directives yourself, and Fortran 90 will act on them. This process is documented in the *MIPSpro Fortran 90 Programmer's Guide*.

In most cases it is better to let Power Fortran automatically choose the loops to run in parallel, rather than forcing a loop to run in parallel by directly inserting parallel directives. When you specify the code to run in parallel, you need to verify that no subsequent modification inserts a data dependency, causing a serious (and hard-to-find) error. When you rewrite a loop so that Power Fortran recognizes the loop as safe to run in parallel, Power Fortran can check future modifications for data dependencies.

Chapter 2

Using Listing and Temporary Files

This chapter has the following major topics:

- "Selecting Files to Generate" on page 6 tells how to select output files.
- "Using the Transformed Source File" on page 7 describes the source file generated by Power Fortran.
- "Using the Workshop Pro MPF File" on page 8 covers that file.
- "Controlling the Power Fortran Listing File" on page 9 discusses listing options.
- "Interpreting the Listing" on page 12 discusses the listing contents.
- "Sample Listings" on page 18 illustrates the use of the listing file in solving some Power Fortran problems.

Selecting Files to Generate

Power Fortran generates the types of output files shown in Table 2-1.

Table 2-1 Output Files

File Contents	Suffix	Driver Option Required
Listing of program structure and message	*.L*	*-pfalist* or *-pfakeep*
Transformed Fortran source file	*.m*	*-pfakeep*
Input file for WorkShop Pro MPF	*.anl*	*-pfakeep*

Suppose that the file *sample.f90* contains the tiny subroutine in Example 2-1.

Example 2-1 Contents of File *sample.f90*

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

When *sample.f* is compiled using the following command:

```
f90 -64 -pfakeep -c sample.f
```

the output includes the following files:

- *sample.L* contains the listing file
- *sample.m* contains the transformed source
- *sample.anl* contains the WorkShop Pro input file

Using the Transformed Source File

The transformed source file contains the original source lines plus parallel-execution directives generated by Power Fortran. The *sample.m* file produced from Example 2-1 contains the lines shown in Example 2-2.

Example 2-2 Transformed Source File

```
C     KAP/SGI_F90 MP        00.00 k240000 190195o5r0so3
20-Jun-1995 14:54:32
# 1 "pfa1.f90"
      subroutine SAMPLE ( A, B, C )
       integer I
       real A(1000), B(1000), C(1000)
C$PAR parallel shared (A,B,C) local (I)
CSGI$ startloop 3
C$PAR pdo
       do 2 I=1,1000
        A(I) = B(I) + C(I)
    2   continue
C$PAR end pdo nowait
CSGI$ endloop 3
C$PAR end parallel
      end
```

The transformed file contains numerous generated directives. These directives, such as C$PAR, direct the parallel execution of the compiled program. Many of these directives are documented in the *MIPSpro Fortran 90 Programmer's Guide* (for example, C$PAR is a PCF directive that marks the start and end of a parallel work unit). However, some directives are not intended for general use and are not documented (for example, C$SGI is not documented).

You use the transformed source file for information only. The transformed source uses fixed format and loops in the Fortran 77 style, instead of the Fortran 90 syntax. As a result, it might not compile correctly if your were to change its suffix to *.f90* and compile it.

Using the Workshop Pro MPF File

The WorkShop Pro MPF Parallel Analyzer View *cvpav* helps you understand the structure and operation of parallelized applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. The Parallel Analyzer View reads the *.anl* analysis files generated by Power Fortran and displays editable parameters for each DO loop. These parameters are easily customized and explored with the help of the Parallel Analyzer View's user-friendly graphical interface.

A sample of a *.anl* file is shown in Example 2-3.

Example 2-3 Workshop Pro MPF Analysis File

```
product "KAP/SGI_F90 MP                          "
version "00.00 k240000 190195"
routine "SAMPLE" line 2 - 7
begin_orig_loop_section "SAMPLE"
orig_loop 1 "SAMPLE" do nest 1 range 4 - 6
 index "I"
loop_option 1 optimize 5
loop_option 1 roundoff 0
loop_option 1 scalar_optimize 3
loop_option 1 limit 20000
loop_option 1 arc_limit 5000
loop_option 1 minconcurrent 1000
loop_option 1 unroll 4
loop_option 1 unroll_weight 100
loop_option 1 max_invariant_if_growth 500
loop_option 1 each_invariant_if_growth 20
variable_list 1 "C":r , "B":r , "A": w, "I":rw
end_orig_loop 1
end_orig_loop_section "SAMPLE"
begin_transformed_loop_section "SAMPLE"
transformed_loop 3 1 "SAMPLE" ordinary
workload 5
schedtype simple "Default"
mode parallel
nest 1
line_range 4 - 6
end_transformed_loop 3 "SAMPLE"
end_transformed_loop_section "SAMPLE"
begin_vln
file_name 1 "pfa1.f90"
```

```
vln_translation 1  1  1
vln_translation 1  2  1
end_vln
```

Controlling the Power Fortran Listing File

The Power Fortran listing file shows the input source statements and how Power Fortran treated them. It is your primary tool for judging the effect of parallelization, and for discovering how to make Power Fortran more effective. The listing file produced by Example 2-1 is shown in Example 2-4.

Example 2-4 Generated Listing File

```
Footnotes     Actions        DO Loops    Line

              DIR                           1   # 1 "pfa1.f90"
                                            1   subroutine sample (a,b,c)
                                            2   dimension a(1000),b(1000),c(1000
              C              +---------     3   do i = 1, 1000
                             *              4     a(i) = b(i) + c(i)
                             *_________     5   end do
                                            6   end
Abbreviations Used
 DIR      directive
 C        concurrentized

    Loop Summary
             From    To      Loop       Loop       at
     Loop#   line    line    label      index      nest    Status
     1       3       5       Do         I          1       concurrentized
```

The page-heading lines and some blank lines have been deleted from Example 2-4. The output in Example 2-4 is the result of using default listing options. A listing with more details is shown under “Interpreting the Listing” on page 12.

Note: The first line 1, containing a number directive, was actually added by *cpp*. The source file passes through *cpp* prior to Power Fortran.

Paginating the Listing

The driver option *–lines=n* (or *–ln=n*) paginates the listing at *n* lines per page. When you specify *–lines=0*, the listing is paginated at subroutine boundaries.

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

Specifying Optional Information

The driver option *–listoptions=list* option (or *–lo=list*) specifies the information to include in the listing file. The *list* is any combination of the option letters in Table 2-2. The default is *–listoptions=ol*.

Table 2-2 Listing File Include Options

Letter	Controls This Information
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 *f90* –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.f90* with Power Fortran and includes an annotated listing of the original program and a summary of the optimizations performed in the listing file:

```
% f90 -pfa -lo=ls source.f90
```

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. These messages range from syntax warnings and error messages to messages about the optimizations performed. *list* is any combination of the option letters summarized in Table 2-3.

Table 2-3 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.

Viewing the Listing File

The listing file is in 132-column format. To view the file, open a window with 132 columns and the number of rows in a page by entering an *xwsh* command with specific geometry.

```
% xwsh -geometry 132x55
```

Optionally you can include a font specification in order to reduce the size of the window while retaining readability.

```
% xwsh -fn "-*-screen-medium-r-normal--12" -geometry 132x40
```

When you want to display the listing file from a shell script or makefile, you can consider other options described in the *xwsh*(1) reference page. The command fragments in Example 2-5 might appear in a shell script. This *xwsh* starts up quickly because it does not open a shell. The user browses the listing using the *xwsh* scroll buffer, whose size is set to 5000 lines in order to accommodate large listings.

Example 2-5 Use of *xwsh* From a Shell Script

```
f90 ... -pfalist -lines=40 ... -c $FILENAME.f90
xwsh -fn "-*-screen-medium-r-normal--12" \
     -geometry 132x40 \
     -sl 5000 \
     -title "$FILENAME listing" \
     -nokeyboard -nomenu -hold \
     -e cat $FILENAME.L
```

Interpreting the Listing

The listing file generated by Power Fortran lists the changes Power Fortran made to the code, and shows where and why the code was not changed. 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. When you understand the information in the Power Fortran listing, you can recognize where small changes to the source code will make a difference in which code is run in parallel.

Default Listing Fields

Any Power Fortran listing that includes the includes the default list options *o* and *l* (as described under "Specifying Optional Information" on page 10) contains the following fields:

- source lines with line numbers
- DO loop annotations

- action summary
- footnotes
- syntax errors/warning messages

The listing fragment in Example 2-6 illustrates these fields.

Example 2-6 Listing Fragment Showing Common Fields

```
Footnotes  Actions       DO Loops    Line
                                       16
                                        17   integer,parameter::adim = 1000
                                        18   real ra(adim),rb(adim),rc(adim)
                                        19
                                        20   ! load the arrays -- suboptimal i/o for example
                                        21      open (unit=11,action="READ",file="pfa1.data")
 1        NO NCS         +---------     22      do i = 1,adim
 2        NO NCS         !              23         read (unit=11,fmt="2E15.8") ra(i),rb(i)
                         !_________     24      end do
                                        25
                                        26   ! swizzle the contents
                                        27   ! function has no side-effects
          DIR                           28   !*$* ASSERT CONCURRENT CALL
          C              +---------     29      do i = 1,adim
                         *              30         rc(i) = swizzle(ra(i),rb(i))
                         *_________     31      end do
                                        32
                                        33   ! un-swizzle in subroutine
                                        34      call subber(ra,rb,rc)
                                        35
                                        36   end
 Abbreviations Used
  NO       not optimized
  DIR      directive
  NCS      non-concurrent-stmt
  C        concurrentized
 Footnote List
   1: not optimized   This loop contains unoptimizable input or output statements.
   2: not optimized   This input or output statement inhibits loop optimization.
```

Line Numbers

A statement in the listing with a line number, such as 21, is either that line from the original source file, or else it is derived from that line. 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 generated line is labeled with the same number as the line of the original source from which it was generated.

DO Loop Marking

The listing displays the boundaries of DO loops graphically in the column headed "DO Loops." Power Fortran surrounds each loop (up to a nesting level of 10) with a loop delimiter character. The delimiters form brackets around each loop nest level. The delimiter characters are listed in Table 2-4.

Table 2-4 Listing File DO Loop Delimiters

Character	Denotes
|	Generic DO loop
*	Power Fortran can run loop in parallel
!	Problem preventing optimization

A statement contained within *n* **DO** loops has *n* of these loop delimiters on that line. You can use the delimiters to trace the extent of each loop, and to quickly learn which loops were parallelized.

Action Summary

When Power Fortran translates or modifies a statement, it places abbreviations in the "Actions" column of the listing to explain what was done to that statement. The abbreviations used in this column are expanded at the bottom of each page. Table 2-5 lists and explains the values that can appear in the Actions column.

Table 2-5 Power Fortran Action Abbreviations

Value	Meaning
C	Concurrentized: This statement will execute on multiple CPUs.
DD	Data Dependence: Data dependence prevents Power Fortran from running this statement in parallel.
DEL	Deleted: ?
DIR	Directive: This statement recognized as a directive. If you code a directive and that line does not have the DIR abbreviation, Power Fortran did not recognize the directive. Check the setting of the *–directives* driver option and the syntax of the directive.
E	Error: Error found in this statement. The message can refer to missing or extra characters, illegal keywords, or text placed in the wrong column. The transformed source file (*.m*) contains an unmodified copy of this line.
EX	Extension: A construct in the original program is a Fortran language extension not supported in the language that Power Fortran generates..
I	Inserted statement.
INF	Information message. See the related listing footnote..
INL	Inlined: Statement inserted in the inlining process.
IPA	Interprocedural analysis: Statement inserted or modified during inlining or IPA.
LM	Label modification: Loop target label generated or modified.
LR	Loop Reordering: This DO statement has been moved in an nterchange with a nested or containing loop.
MIS	Miscellaneous: Unspecific message. This message does not always mean that something is wrong with the program.
NC NCS	Not Concurrentized: this statement could not be made concurrent for the reason shown in another code.

Table 2-5 (Continued) Power Fortran Action Abbreviations

Value	Meaning
NO	Not Optimized: some Power Fortran internal data structure or table has overflowed. To process this source, you must split the file into smaller sections.
NT NTS	Not Tiled: This statement could not be optimized for cache performance.
NV NVS	Not Vectorized: This statement could not be optimized for vector performance.
OE	Option Error: An error detected in Power Fortran driver option. This error does not stop processing of a program unit.
OPT	Optimized: This statement modified by scalar optimization.
OT OTF	Output Translation Failure: This statement cannot be represented in the Fortran language generated by Power Fortran.
OW	Option Warning: A questionable condition detected in a Power Fortran driver option.
Q	Question: This statement cannot be optimized unless more information is given. The question is given in a footnote line. You can usually answer this question with an appropriate assertion.
SC SO	Scalar Optimization: This statement modified by scalar optimization..
STD	Standardized: This statement was transformed into a standard Fortran form in order to improve the changes of being able to optimize it.
T	Tiled: This statement transformed to improve cache memory performance.
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 message.
TL	Too Large: The source program is too large; Power Fortran can only process it in smaller sections.
V	Vectorized: This statement modified for vector performance.
W	(Warning) Contains syntax warnings.

Footnotes

The “Footnotes” column gives details concerning the actions or nonactions taken. Power Fortran numbers and prints the footnotes at the bottom of each program unit under the Footnote List heading. For example, the following line from a listing contains the footnote number 13.

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

In the Footnote List displayed at the end of the program unit, the footnote appears.

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

In this example, 13 is the footnote number, DD (data dependence) is the explanation for the action (in this case, nonaction).

Syntax Errors/Warning Messages

When a program has errors, the listing contains the error text following the line in error. The same messages are printed to *stderr* during the compile. The following example displays a listing with an error.

```
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
```

Sample Listings

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

Indirect Indexing

Power Fortran cannot determine if it can run a loop in parallel when the code uses indirect indexing. A loop is indirectly indexed when it uses the value from some auxiliary array as the index value rather than the **DO** loop variable. The code in Example 2-7 illustrates indirect indexing.

Example 2-7 Loop With Indirect Indexing

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

When Example 2-7 is submitted to Power Fortran, it produces the listing shown in Example 2-8.

Example 2-8 Listing of Loop With Indirect Indexing

```
Footnotes     Actions        DO Loops    Line

               DIR                           1   # 1 "pfa3.f90"
                                             1         subroutine INDIREX ( A, B, INDEX, N )
               I                             1          integer N, I
               I                             1          real A(N), B(N)
               I                             1          integer INDEX(N)
               I                             4   CSGI$ startloop 2
 1             Q              +---------     4          do   I=1,N
 2             DD             !              5           A(INDEX(I)) = A(INDEX(I)) + B(I)
                              !_________     6          end do
               I                             6   CSGI$ endloop 2
                                             7         end
```

```
Abbreviations Used
 DD       data dependence
 Q        question
 I        inserted
 DIR      directive

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

DD in the Actions column for statement 6 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 in *index* are not unique, different iterations might write to the same element of *A*.

The question given in footnote 1 asks if index is a permutation of the integers 1..*n*. (Since all elements *index*(*i*) are used as subscripts of *a*, Power Fortran assumes they are all taken from the set 1..*n*; however only a permutation of 1..*n* contains no duplicate values.)

If you are sure that *index* is a permutation vector, there can be no data dependence between elements of *A*. You inform Power Fortran of this by adding an assertion, as shown in Example 2-9.

Example 2-9 Loop Asserting Indirect Indexing Through a Permutation

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

The listing in Example 2-10 shows that Power Fortran finds the loop safe to run in parallel.

Example 2-10 Listing of Loop With Asserted Permutation

```
 Footnotes     Actions        DO Loops    Line
               DIR                           1   # 1 "pfa3.f90"
                                             1         subroutine INDIREX ( A, B, INDEX, N )
               I                             1          integer N, I
               I                             1          real A(N), B(N)
               I                             1          integer INDEX(N)
               DIR                           4   C*$* assertpermutation ( INDEX )
               I                             5   C$PAR parallel if (N .gt. 142) shared (N,IN
DEX,A,B) local (I)
               I                             5   CSGI$ startloop 3
               I                             5   C$PAR pdo
               LM I C         +---------     5          do 2 I=1,N
                              *              6           A(INDEX(I)) = A(INDEX(I)) + B(I)
               I              *_________     7       2   continue
               I                             7   C$PAR end pdo nowait
               I                             7   CSGI$ endloop 3
               I                             7   C$PAR end parallel
                                             8         end

 Abbreviations Used
  LM       label modification
  I        inserted
  DIR      directive
  C        concurrentized
```

Note: Power Fortran cannot verify the truth of this, or any, assertion. When you make an assertion, it is up to you to be certain that it is true for all possible input data.

Function Call

The code in Example 2-11 contains a call to an external function.

Example 2-11 Loop With Call to External Function

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

```
      a(i) = force(b(i),c(i))
   end do
end
```

The Power Fortran listing for this code is shown in Example 2-12.

Example 2-12 Listing of Loop With External Call

```
Footnotes     Actions       DO Loops    Line
              DIR                          1   # 1 "pfa4.f90"
                                           1   subroutine caller(a,b,c,n)
                                           2      real a(n), b(n), c(n)
                                           3      external force
1             NO NCS        +---------     4      do i = 1,n
2             NO NCS        !              5         a(i) = force(b(i),c(i))
                            !_________     6      end do
                                           7   end
Abbreviations Used
 NO       not optimized
 DIR      directive
 NCS      non-concurrent-stmt
Footnote List
  1: not optimized               This loop contains an unoptimizable call to "FORCE".
  2: not optimized               This statement contains an unoptimizable call to "FORCE".
```

Since Power Fortran cannot be sure that function **force** has no side-effects, it cannot parallelize this loop. If the function has a side-effect—that is, if it assigns data to variables other than its arguments and result—errors would occur when it was called concurrently from multiple CPUs.

If you know that a procedure has no side-effects, you can insert an assertion to tell Power Fortran, as shown in Example 2-13.

Example 2-13 Loop With Asserted Concurrent Call

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

With this modification, Power Fortran parallelizes the loop as shown in Example 2-14.

Example 2-14 Listing of Loop With Asserted Concurrent Call

```
Footnotes     Actions       DO Loops    Line
              DIR                          1   # 1 "pfa4.f90"
                                           1   subroutine caller(a,b,c,n)
                                           2      real a(n), b(n), c(n)
                                           3      external force
              DIR                          4   !*$*assert concurrent call
              C             +---------     5      do i = 1,n
                            *              6         a(i) = force(b(i),c(i))
                            *_________     7      end do
                                           8   end

Abbreviations Used
 DIR      directive
 C        concurrentized
```

Procedure Parameters

There is a further subtlety in using procedure calls in parallelized loops. Power Fortran does not examine the code of the external function (unless you use the optimization of procedure inlining, discussed in the *MIPSpro Fortran 90 Programmer's Guide*). As a result, Power Fortran cannot know whether the external procedure modifies any of its arguments. Code with this ambiguity is shown in Example 2-15.

Example 2-15 Loop With Ambiguous Parameter

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

The question is, does **my_function** modify its argument *m*? If it only reads *m*, there can be a single copy of m used by all concurrent instances of the loop. If the function assigns to *m*, each loop instance needs its own copy of the variable. These two modes, called SHARE and LOCAL respectively, are discussed in Chapter 7 of the *MIPSpro Fortran 90 Programmer's Guide*.

Given Example 2-15, Power Fortran assumes that m is read-only, and gives it SHARE status. This can be seen when the transformed program is included in the listing (*-listoptions=loi*), as shown in Example 2-16.

Example 2-16 Listing of Loop With Ambiguous Parameter

```
DIR                              4   C*$*  assertconcurrentcall
I                                5   C$PAR parallel if (N .gt. 16) shared (N,A,B,M) local (I)
I                                5   CSGI$ startloop 3
I                                5   C$PAR pdo
LM I C         +---------        5          do 2 I=1,N
               *                 6            A(I) = real (MY_FUNCTION (B(I),M))
               *                 7            B(I) = A(I) + real (M)
I              *_________        8        2   continue
I                                8   C$PAR end pdo nowait
I                                8   CSGI$ endloop 3
I                                8   C$PAR end parallel
```

The C$PAR clause shared(N,A,B,M) in Example 2-16 specifies that single copies of those variables will be accessed concurrently from all instances of the concurrent loop. If in fact **my_function** changes the value of *m*, then this is incorrect. If this is the case, you can give Power Fortran the hint it needs by adding a visible assignment to *m* prior to the procedure call, as shown in Example 2-17.

Example 2-17 Loop With Apparent Assignment to Parameter

```
subroutine tricky (a,b,c,n,m)
   real a(*), b(*)
   external my_function
!*$*assert concurrent call
   do i = 1, n
      m = 0 ! force M to be LOCAL
      a(i) = my_function (b(i), m)
      b(i) = a(i) + m
   end do
end
```

Since Power Fortran sees an assignment to *m* within the loop, it declares *m* to be LOCAL (without regard to its use as a parameter). This is an example of how it is sometimes necessary to put extra statements in a loop in order to get the best speed from a parallel program.

Note: If **my_function** both reads the input value of *m* and writes a new value, the trick in Example 2-17 will not work. You would have to find an assignment to *m* that preserves its value in each iteration of the loop. Unfortunately, most such non-assignments, such as *m*=*m**1, are likely to be deleted as unnecessary by the scalar optimization phase of Power Fortran!

Reductions

A reduction produces a single value from a set of values. The subroutine in Example 2-18 calculates the sum of products of two arrays.

Example 2-18 Sum of Products Reduction

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

With Example 2-18 as input, Power Fortran produces the listing in Example 2-19.

Example 2-19 Listing of Sum of Products Procedure

```
                                             1   subroutine sum_prods(a,b,n,sum)
                                             2      real a(n), b(n), sum
                                             3      sum = 0.0
                             +---------      4      do i = 1,n
1             DD             !               5         sum = sum + a(i)*b(i)
                             !_________      6      end do
                                             7   end

Abbreviations Used
 DD       data dependence
```

Because different iterations of the loop read and write the variable *sum*, Power Fortran detects a dependence. However, reduction is a special kind of data dependence. Because *sum* only accumulates a total, you can accumulate subtotals in parallel and then combine the subtotals at the end. (This general technique for dealing with reduction is discussed in Chapter 7 of the *MIPSpro Fortran 90 Programmer's Guide*.)

In fact, Power Fortran can recognize the four most common reductions—sum, product, min and max—and parallelize them. It does not do this automatically because, since the parallel version of the code adds the elements together in a different order than the single-process version, the round-off errors can accumulate differently in a parallelized reduction, producing a different answer depending on the number of CPUs that execute the code. In fact, if you use the dynamic scheduling option, 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 your application is not affected, you can direct Power Fortran to use parallel reduction using either the C*$* ROUNDOFF(2) directive or the driver option *–WK,–roundoff=2*. The resulting listing is shown in Example 2-20.

Example 2-20 Listing of Reduction With Roundoff Directive

```
              DIR                             1   # 1 "pfa6.f90"
                                              1   subroutine sum_prods(a,b,n,sum)
                                              2      real a(n), b(n), sum
                                              3      sum = 0.0
              DIR                             4   !*$*roundoff(2)
              C              +---------       5      do i = 1,n
              SO             *                6         sum = sum + a(i)*b(i)
                             *_________       7      end do
                                              8   end
Abbreviations Used
 SO       scalar optimization
 DIR      directive
 C        concurrentized
```

The round-off error produced by a parallel reduction operation is not necessarily worse than the round-off error seen in the serial version. It is simply different. If your original application was not affected by round-off error, there is no reason to worry about it in the parallel version. However, if 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.

Table 2-6 shows the types of reductions Power Fortran supports.

Table 2-6 Power Fortran Reductions

Type	Operator	Example
Sum	+	sum = sum + *expression*
Product	*	prod = prod * *expression*
Min	min( )	least = min(least, *expression*)
Max	max( )	most = max(most, *expression*)

All these reductions are under the control of the *–roundoff* driver option and C*$*ROUNDOFF directive, even though technically the min and max reductions do not involve round-off problems.

Chapter 3

Working With Power Fortran 90

This chapter documents your control of Power Fortran using driver options, directives, and assertions.

Using Driver Options

You use *f90* driver options to control the operations of Power Fortran. This topic discusses the options that are unique to Power Fortran. (For other driver options, see the *MIPSpro Fortran 90 Programmer's Guide*.) This chapter discusses the driver options in functional groups. For a summary ordered alphabetically, see Appendix C, "Power Fortran Driver Options."

Enabling Parallelization of Loops

The *–concurrentize* option (or *–conc*) directs Power Fortran to convert eligible loops to run in parallel. This option is assumed when the *-pfa* option is used to enable Power Fortran.

Use *–noconcurrentize* option (or *–nconc*) to prevent Power Fortran from converting loops, while still applying scalar optimizations.

You can use the C*$*NOCONCURRENTIZE directive to prevent transformation of a particular loop (see "C*$*[NO]CONCURRENTIZE" on page 35).

Specifying a Work Threshold

The *–minconcurrent=n* option (or *–mc=n*) specifies the minimum amount of work that must be found inside any loop in order for it to be profitable to execute the loop in parallel. 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 default value is 500 or more operation. The higher the value for *n*, the larger (more iterations, more statements, or both) the loop body must be, to be run in parallel. You can set the threshold for a particular loop using the C*$* MINCONCURRENT(*n*) directive.

When the loop bounds are known at compile time (that is, when they are constants), Power Fortran can compute the exact iteration count and decide whether to run the loop in parallel. Loops that are too small are not parallelized.

When the loop bounds are undefined at compile time, Power Fortran adds an IF clause to the generated parallel directive. This clause tests at run-time if sufficient work exists. If not, the loop runs serially.

To disable testing the work threshold throughout the program, specify *–minconcurrent=0*. You can disable the threshold for a specific loop using the C*$* MINCONCURRENT(0) directive.

The loop in Example 3-1 executes an unpredictable amount of work.

Example 3-1 Loop With Unknown Amount of Work

```
subroutine minc(x,y,z,n)
integer n
real x(n),y(n),z(n)
   do i = 1,n
      x(i) = y(i)*z(i)
   end do
end
```

Power Fortran generates the transformed loop shown in Example 3-2.

Example 3-2 Transformed Loop With Unknown Amount of Work

```
DIR                             1   # 1 "pfa7.f90"
                                1         subroutine MINC ( X, Y, Z, N )
I                               1          integer N, I
I                               1          real X(N), Y(N), Z(N)
I                               4   C$PAR parallel if (N .gt. 200) shared (N,X,Y,Z) local (I)
I                               4   CSGI$ startloop 3
```

```
I                             4   C$PAR pdo
LM I C         +---------     4          do 2 I=1,N
               *              5            X(I) = Y(I) * Z(I)
I              *_________     6       2    continue
I                             6   C$PAR end pdo nowait
 I                            6   CSGI$ endloop 3
I                             6   C$PAR end parallel
                              7         end
```

The IF clause in the C$PAR directive at line 4 is executed at runtime. It tests whether there are the default 500 operations in the loop body (apparently Power Fortran counts 2.5 operations in statement 5). Each time this subroutine is entered the test is made and, when *n*>200, the loop executes in parallel. Otherwise, it runs serially

Enabling Parallel I/O

The *–parallelio* option (or *–pio*) enables the parallelization of loops that contain I/O statements. The *-parallelio=no* is the default. Use this option only on systems with parallel I/O capabilities or where you are sure that I/O statements in a parallelized loop are not executed.

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. The limit *n* does not correspond to the **DO** loop nest level. It is an estimate of the number of loop orderings that Power Fortran could generate from a loop nest.

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

Larger limits can 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.

You can set the same limit either globally or for a particular part of the source module using the C*$* LIMIT(*n*) directive.

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 compilation takes Valid values for *n* are:

0	Disables optimization; no loops are converted.
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 (see “Reductions” on page 24 for discussion of *–roundoff*.)
3	Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wraparound variables.
4	Generates two versions of a loop, if necessary, to break a data dependence arc. This level also implements more-exact data dependence tests and allows special index sets (called wraparound variables) to convert more code to run in parallel.
5	Fuses two adjacent loops if it is legal to do so (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.

The same option is supported when Power Fortran is not invoked, to control scalar optimizations (refer to Chapter 4 of the *MIPSpro Fortran 90 Programmer's Guide*). You can set the optimization level with global use of the C*$* OPTIMIZE(*n*) directive.

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 as a result of the different order of execution between parallel and serial loops. Valid values for *n* are the following integers:

0, 1	Suppresses any round-off transformations. This is the default.
2	Allows reductions to be performed in parallel.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 90 Programmer's Guide* for details.)

See "Reductions" on page 24 for an example.

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" on page 39).
- 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. 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 90 Programmer's Guide*.

Preventing Use of Directives and Assertions

Power Fortran does not check the correctness of assertions. Directives inserted into a source module and then forgotten can cause unexpected results that contradict the driver options you employ. If you specify that an untrue assertion or unwanted directive is causing problems, you can control the interpretation of directives and assertions in three ways.

- Use the *–nodirectives* driver option to cause all directives to be treated as comments.
- Use the *-directives=list* driver option to selectively prevent or allow the interpretation of Cray, VAST, and SGI directives (refer to the *pfa*(1) reference page).
- Place the C*$* NO ASSERTIONS directive to tell Power Fortran to ignore all following assertions.

Using Directives and Assertions

After you run a Fortran source program through Power Fortran once, you can use directives and assertions to fine-tune program execution. (Driver options apply to the program as a whole.) 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 use directives and assertions to force Power Fortran to process portions of code in various ways. 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.

Preventing Parallelization

Often you know of parts of your program that should not be parallelized, typically because you know that these parts contribute little to the execution time of the program. You can insert directives or assertions (called assertions, actually they have the effect of directives) to prevent Power Fortran from parallelizing specific loops.

C*$* ASSERT DO (SERIAL)

The assertion C*$* ASSERT DO (SERIAL) 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 code shown in Example 3-3.

Example 3-3 Nested Loop

```
subroutine nest(x,y,z,n)
real x(n,n,n), y(n,n), z(n,n)
   do i=1,n
      do j = 1,n
         do k = 1,n
            x(i,j,k) = x(i,j,k) * y(i,j)
         end do
         do k = 1,n
            X(i,j,k) = X(i,j,k) + Z(i,k)
         end do
      end do
   end do
end
```

As written, Power Fortran will interchange loop variables to invert the order of the three loops, and parallelizes the outermost loop. Adding an assertion as shown in Example 3-4 makes a dramatic change.

Example 3-4 Nested Loop With Serial Assertion

```
subroutine nest(x,y,z,n)
real x(n,n,n), y(n,n), z(n,n)
   do i=1,n
      do j = 1,n
!*$* ASSERT DO(SERIAL)
         do k = 1,n
            x(i,j,k) = x(i,j,k) * y(i,j)
         end do
         do k = 1,n
            X(i,j,k) = X(i,j,k) + Z(i,k)
         end do
      end do
   end do
end
```

The assertion forces the loop on *I*, the loop on *J*, and the first loop on *K* to be serial. Power Fortran still executes the second loop on *K* in parallel.

CDIR$ NEXT SCALAR

Silicon Graphics Power Fortran supports the Cray directive, CDIR$ NEXT SCALAR as a synonym for the C*$* ASSERT DO (SERIAL) assertion.

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 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 code fragment in Example 3-5 is an example of how to use the assertion.

Example 3-5 Inner Loop Preferred Serial

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

The assertion requests that the loop on *J* be serial. In this construction, Power Fortran tries to run the loop on *I* in parallel. This capability is useful when you know the value of *M* to be very small or less than *N*.

If the assertion in Example 3-4 is changed from DO (SERIAL) to DO PREFER (SERIAL), Power Fortran makes remarkable alterations in the transformed program, producing two copies of the outer loop, each containing just one of the inner loops on *K*.

Explicit Parallelization

Sometimes you might need to hand-tune a loop so that it will run in parallel. You can use the C$ DOACROSS directive, or the PCF directives, to explictly mark a section of code for parallel execution. These directives are discussed in detail in Chapter 7 of the *MIPSpro Fortran 90 Programmer's Guide*.

C*$*[NO]CONCURRENTIZE

The C*$*[NO]CONCURRENTIZE directive enables or disables Power Fortran from transforming eligible loops to run in parallel to the end of the current program unit.

Used locally, these directives override the *–[no]concurrentize* driver option. Used globally, these directives have the same effect as the *–[no]concurrentize* driver option. (See "Enabling Parallelization of Loops" on page 27)

CVD$ CONCUR

Power Fortran supports the VAST directive CVD$CONCUR. Power Fortran interprets this directive as if it were the C*$*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 loop immediately after it.

Consider the code in Example 3-6.

Example 3-6 Nested Loop With Preferred Loop

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

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

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 analyzes a loop, it categorizes the loop into one of three groups:

- yes (loop is safe to run in parallel)
- no (definite data dependency)
- not sure (assumed data dependency)

Normally, Power Fortran does not run in parallel a loop with an assumed data dependency. C*$* ASSERT DO (CONCURRENT) tells Power Fortran to go ahead and run “not sure” loops in parallel.

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.

Clarifying Dependencies

With assertions, you can give Power Fortran information about the use of data in the program. This allows Power Fortran to parallelize more loops or to do so more efficiently.

C*$* ASSERT RELATION

The C*$* ASSERT RELATION(*name.xx.name*) assertion specifies that a particular logical relation always holds between two variables, or between a variable and a constant. *name* is the variable or constant, and *xx* is any of the logical relations GT, GE, EQ, NE, LT, or LE. Used locally, this assertion applies only to the following DO statement.

C*$* ASSERT RELATION can be specified globally 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.

The code in Example 3-7 illustrates the use of C*$* ASSERT RELATION.

Example 3-7 Loop With Implied Relation

```
subroutine sumdown(a,b,m,n)
   integer m,n,j
   real a,b
   dimension a(:),b(:)
   do j=1,n
      a(j) = a(j+m) + b(j)
   end do
end
```

When Power Fortran analyzes this code, it detects that a data dependence would exist if *M* were less than or equal to *N*, because in that case different instances of a concurrent loop could store into the same elements of *A*. Power Fortran parallelizes the loop by generating this IF statement:

```
if (M .ge. 1 .and. N .ge. M .or. M .le. (-1)) then
```

The THEN branch of the IF contains a serial version of the loop as originally written. The ELSE branch, executed when no data dependency can exist, contains a parallel version of the loop.

When you know that M is always greater than N, there is no need for this duplication of code. You can use C*$* ASSERT RELATION to inform Power Fortran, as shown in Example 3-8. When you do this, Power Fortran generates only the parallel form of the loop.

Example 3-8 Loop With Asserted Relation

```
subroutine sumdown(a,b,m,n)
   integer m,n,j
   real a,b
   dimension a(:),b(:)
!*$* assert relation (m.gt.n)
   do j=1,n
      a(j) = a(j+m) + b(j)
   end do
end
```

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 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:

```
!*$* assert no recurrence (x)
   do i= 1,m,5
      x(k) = x(k) + x(i)
   end do
```

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 following 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 a detailed example of the use of this assertion, see “Indirect Indexing” on page 18.

C*$* ASSERT CONCURRENT CALL

C*$* ASSERT CONCURRENT CALL tells Power Fortran to ignore assumed dependencies that are caused by a subroutine call or a function reference. You must ensure that the subroutines and referenced functions have no side effects and are safe for parallel execution. This assertion applies to all subroutine and function references in the following loop. For detailed examples of the use of this assertions, see “Function Call” on page 20 and “Procedure Parameters” on page 22.

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.

Appendix A

Power Fortran Assertions

This appendix summarizes the Power Fortran assertions in alphabetical order. When viewing this Guide online, you can use the cross-references in the following table as hypertext links to the descriptive paragraphs.

Table A-1 Power Fortran Assertions

Assertion	Summary	Discussion
C*$* ASSERT CONCURRENT CALL	page 42	page 20
C*$* ASSERT DO (CONCURRENT)	page 42	page 36
C*$* ASSERT DO (SERIAL)	page 42	page 33
C*$* ASSERT DO PREFER (CONCURRENT)	page 42	page 35
C*$* ASSERT DO PREFER (SERIAL)	page 43	page 34
C*$* ASSERT [NO] LAST VALUE NEEDED	page 43	
C*$* ASSERT NO RECURRENCE	page 43	page 38
C*$* ASSERT NO SYNC	page 43	
C*$* ASSERT PERMUTATION	page 44	page 18
C*$* ASSERT RELATION	page 44	page 37

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)

C*$* ASSERT DO (CONCURRENT) tells Power Fortran to ignore assumed data dependencies. Normally, does not run loops containing assumed data dependencies in parallel. C*$* ASSERT DO (CONCURRENT) tells Power Fortran to go ahead and run such 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)

C*$* ASSERT DO (SERIAL) tells Power Fortran to run the immediately following 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 following loop to run in parallel.

C*$* ASSERT DO PREFER (CONCURRENT)

C*$* ASSERT DO PREFER (CONCURRENT) runs a the immediately following 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.

Power Fortran does not generate parallel code if you use the *–noconcurrentize* driver option or the C*$* NOCONCURRENTIZE directive.

C*$* ASSERT DO PREFER (SERIAL)

C*$* ASSERT DO PREFER (SERIAL) indicates that you want to execute the immediately following loop in serial mode. This assertion directs Power Fortran to leave the 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

Power Fortran gives each instance of the parallel loop its own, temporary copy of the iteration variable to use. Power Fortran generates code so that the instance of the loop that represents the "last" iteration (which may or may not execute last in actual time sequence) assigns its final value for the iteration variable to the actual variable, so that this "last" value will be available to the serial code that follows the loop.

C*$* ASSERT NO LAST VALUE NEEDED specifies that the final values from loops are not used, so last-value assignments are unnecessary. This assertion is active until reset or until the end of the program.

C*$* ASSERT NO RECURRENCE

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

C*$* ASSERT NO SYNC

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

C*$* ASSERT PERMUTATION

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

C*$* ASSERT RELATION

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

Appendix B

Power Fortran Directives

This appendix summarizes the Power Fortran directives in alphabetical order. When viewing this Guide online, you can use the cross-references in the following table as hypertext links to the descriptive paragraphs.

Table B-1 Power Fortran Directives

Directive	Class	Summary	Discussion
C$&	SGI	page 48	
C*$* CONCURRENTIZE	SGI	page 46	page 35
C*$* LIMIT	SGI	page 46	page 29
C*$* MINCONCURRENT	SGI	page 46	page 27
C*$* NO ASSERTIONS	SGI	page 47	page 32
C*$* NOCONCURRENTIZE	SGI	page 47	page 35
C*$* OPTIMIZE	SGI	page 47	page 30
C*$* ROUNDOFF	SGI	page 48	page 24
CDIR$ IVDEP	Cray	page 49	page 36
CDIR$ NEXT SCALAR	Cray	page 49	page 34
C$ DOACROSS	SGI	page 48	page 27
CVD$ CNCALL	VAST	page 49	page 39
CVD$ CONCUR	VAST	page 49	page 35

C*$* CONCURRENTIZE

C*$*CONCURRENTIZE tells Power Fortran to convert eligible loops to run in parallel. This directive, when specified globally, has the same effect as the *–concurrentize* command line option. See also "C*$* NOCONCURRENTIZE" on page 47.

C*$* LIMIT

C*$*LIMIT(*n*) 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 value of *n* does not correspond to the 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* driver option; see "limit" on page 52.

C*$* MINCONCURRENT

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 is added to the generated parallelizing directive to test at run time if sufficient work exists.

C*$* NO ASSERTIONS

C*$* NO ASSERTIONS directs Power Fortran to ignore all following assertions to end of file.

C*$* NOCONCURRENTIZE

C*$*NONCONCURRENTIZE prevents Power Fortran from converting loops to run in parallel. Used globally, it has the same effect as the *-noconcurrentize* driver option. See also "C*$* CONCURRENTIZE" on page 46.

C*$*OPTIMIZE

C*$*OPTIMIZE(*n*) 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	Disables optimization; no loops are converted.
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 (see "Reductions" on page 24 for discussion of *–roundoff*.)
3	Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wraparound variables.

4	Generates two versions of a loop, if necessary, to break a data dependence arc. This level also implements more-exact data dependence tests and allows special index sets (called wraparound variables) to convert more code to run in parallel.
5	Fuses two adjacent loops if it is legal to do so (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.

C*$*ROUNDOFF

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

0, 1	Suppresses any round-off transformations. This is the default.
2	Allows reductions to be performed in parallel.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 90 Programmer's Guide* for details.)

C$ DOACROSS

C$ DOACROSS tells the MIPSpro Fortran 90 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. The use of C$ DOACROSS is covered in detail in the *MIPSpro Fortran 90 Programmer's Guide*.

Note: MIPSpro Power Fortran automatically generates C$PAR directives from the PCF directive set. Older versions of Power Fortran generated C$ DOACROSS instead.

C$&

The C$& directive continues the preceding parallel directive onto a continuation lines.

CDIR$ IVDEP

Power Fortran interprets the Cray CDIR$ IVDEP directive as if it were a C*$* ASSERT DO (CONCURRENT) assertion (see “C*$* ASSERT DO (CONCURRENT)” on page 42). Cray directives are disabled by default; see “[no]directives” on page 52.

CDIR$ NEXT SCALAR

Power Fortran interprets the CDIR$ NEXT SCALAR directive as if it were a C*$* ASSERT DO(SERIAL) assertion (see “C*$* ASSERT DO (SERIAL)” on page 42). Cray directives are disabled by default; see “[no]directives” on page 52.

CVD$ CNCALL

Power Fortran interprets the CVD$ CNCALL directive as if it were the C*$* ASSERT CONCURRENT CALL assertion (see “C*$* ASSERT CONCURRENT CALL” on page 42).

CVD$ CONCUR

Power Fortran interprets the CVD$ CONCUR directive as if it were the C*$*CONCURRENTIZE directive (see “C*$* CONCURRENTIZE” on page 46).

Power Fortran Driver Options

This appendix summarizes the driver options that relate specifically to automatic parallelization in Power Fortran. Many additional options relating to optimization are documented in the *MIPSpro Fortran 90 Programmer's Guide*. When viewing this Guide online, you can use the cross-references in the following table as hypertext links to the descriptive paragraphs.

Table C-1 Power Fortran Driver Options

Option (long)	Option (short)	Default (with -pfa)	Summary	Discussion
–[no]concurrentize	-[no]conc	-conc	page 52	page 27
-[no]directives[=list]	-[n]dr[=list]	–dr=ackpv	page 52	page 32
-limit=n	-lm	-lm=20000	page 52	page 29
-lines=n	-ln	-lines=55	page 53	page 10
-listoptions=list	-lo=list	-lo=li	page 53	page 10
-minconcurrent	-mc	-mc=500	page 54	page 27
-optimize=n	-o=n	depends on On	page 55	page 30
-[no]parallelio	-[no]pio	-nopio	page 55	page 29
-roundoff=n	-r=n	depends on On	page 52	page 24
-suppress=list	-su=list	(none)	page 56	page 11

[no]concurrentize

The *–concurrentize* option allows Power Fortran to convert eligible loops to run in parallel throughout the source file, subject to use of directives within the source file (see "C*$* NOCONCURRENTIZE" on page 47).

The *-noconcurrentize* option disables conversion of eligible loops throughout the source file, subject to the use of the directives within the source file (see "C*$* CONCURRENTIZE" on page 46).

[no]directives

The option *-directives=list* enables interpretation of directives in the source file. The letters in *list* specify different classes of directives to be recognized:

A	Unique Power Fortran directives
C	Cray (CDIR$) directives
K	Power Fortran C*$* form directives
S	Sequent C$ directives
V	VAST (CVD$) directives

The default is to enable all but Cray directives, since there are cases where the Cray CDIR$ IVDEP directive is unsafe in SGI Fortran (see "CDIR$ IVDEP" on page 36).

The options *-nodirectives* disables processing of all directives in the source file.

limit

The *–limit=n* option 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. 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 value of *n* does not correspond to the loop nest level. It is an estimate of the number of loop orderings that Power Fortran can generate from a loop nest.

The same processing limit can be specified within the source file using a directive; see “C*$* LIMIT” on page 46.

lines

The *–lines* option sets the page size for the listing produced by Power Fortran. Specifying *–lines=0* paginates at subroutine boundaries only.

listoptions

The *–listoptions=list* option specifies the information to include in the listing file (.L). The letters in *list* include any combination of the letters in Table C-2. The default is *–listoptions=ol*.

Table C-2 Listing File Include Options

Letter	Controls This Information
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 *f90* –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.

minconcurrent

The *–minconcurrent=n* option establishes the minimum amount of work needed inside a loop to make executing a loop in parallel worthwhile. If the loop does not contain at least this much work, the loop will not be run in parallel.

When the loop bounds are constants, Power Fortran can decide whether or not to transform the loop at compile time. If the loop bounds are not constants, Power Fortran transforms the loop to parallel form but includes an IF clause in the parallelizing 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.

optimize

The *–optimize=n* option sets the optimization level. The higher you set the optimization level, the more code is optimized and the longer compilation takes Valid values for *n* are:

0	Disables optimization; no loops are converted.
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 (see "Reductions" on page 24 for discussion of *–roundoff*.)
3	Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wraparound variables.
4	Generates two versions of a loop, if necessary, to break a data dependence arc. This level also implements more-exact data dependence tests and allows special index sets (called wraparound variables) to convert more code to run in parallel.
5	Fuses two adjacent loops if it is legal to do so (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.

parallelio

The *–parallelio* option enables the parallelization of loops that contain I/O statements. Use this option only on systems with parallel I/O capabilities or where I/O statements in loops are not executed.

The *-noparallelio* option (the default) disables transformation of any loop with an I/O statement in it.

suppress

The *–suppress* option lets you 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 option letters summarized in Table C-3.

Table C-3 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 *f90*)

Index

Tell Us About This Manual

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

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

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

Please send the title and part number of the document with your comments. The part number for this document is 007-2760-001.

Thank you!

Three Ways to Reach Us

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

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