IRIS Power C User's Guide

Introduction

About This Guide

This guide describes how to use IRIS Power C™. It is written for software programmers/developers who want to make efficient use of the IRIX™ multiprocessors to execute code in parallel. This guide assumes that you understand how multiprocessors work and have a working knowledge of parallel programming.

For information on parallel programming, see "Reference Material," in this introduction.

What Is IRIS Power C?

IRIS Power C is a C compiler that automatically analyzes sequential code to determine where loops can run in parallel and then generates object code that can use multiple processors. It enables you to recompile existing serial C programs so that they run efficiently on multiprocessor computers without time−consuming hand recoding. Consequently, your original programs remain portable; you don't have to be concerned about the specifics of the system.

What Is the IRIS Power C Analyzer?

The IRIS Power C Analyzer (PCA) is a C code optimization preprocessor that detects potential parallelism in C code. PCA can insert explicit compiler directives that allow the data−independent code to run in parallel. You can use PCA with the C compiler or as a standalone product. Preparing code to run parallel (concurrentization) is not the only way PCA can improve the performance of your code. PCA optimizations include:

- Concurrentization
- Local variable identification
- Sum, dot−product, MAX and MIN reduction recognition
- Loop reordering
- Loop unrolling
- Induction variable recognition
- Global forward substitution
- Variable lifetime analysis
- Dead−code elimination
- Procedure in−lining
- Memory management

PCA's conversion process is designed to operate effectively without your intervention. You can send PCA−generated code directly to the compiler. In this sense, PCA is more a part of the compilation process than a standalone tool.

Document Overview

This guide contains the following chapters and appendices:

Chapter 1, "Compiling with IRIS Power C," explains the command−line options to use at analysis time to alter PCA defaults, such as instructing PCA to set optimization levels or enable/disable a feature.

Chapter 2, "How to Use IRIS Power C," describes how to compile programs with Power C, interpret the PCA listing files, and use Power C to run programs in parallel.

Chapter 3, "PCA Command−Line Options," details the options to use to customize output, such as limiting the depth of an in−line expansion.

Chapter 4, "Power C Analyzer Directives," explains directives that give PCA additional information (about the input program) that PCA cannot determine.

Chapter 5, "Multiprocessing C Compiler Directives," describes how to use the multiprocessor C compiler to produce code that can run concurrently.

Chapter 6, "PCA Transformations," offers examples of PCA transformations of ordinary C code into explicit parallel syntax.

Chapter 7, "In–lining and Interprocedural Analysis," explains command–line options and in–line pragmas you can use to in–line functions or to perform Inter–procedural Analysis.

Chapter 8, "The PCA Listing," describes PCA's listings and messages to users.

Appendix A, "Improving PCA Performance," lists suggestions to use when customizing PCA for a particular program.

Appendix B, "Data–Dependence Analysis," describes data dependence analysis and explains how PCA uses this information to determine if a given loop can run safely in parallel.

Appendix C, "Run Time Environment Variables," describes the run time options available to control execution of your code.

Related Documentation

The following Silicon Graphics documents contain information relevant to Power C:

- *IRIX User's Reference Manual*, Mountain View, California, Silicon Graphics, Inc., 1989, document number 007–0606–052 (Documents user commands and utilities.)

Reference Material

Parallel Programming

- *Introduction to Parallel Programming*, Steven Brawer, San Diego, California, Academic Press, Inc., 1989 (ISBN 0–12–128470–0), describes general parallel programming concepts. Although the examples are in FORTRAN, most of the concepts described also relate to Power C.
- *Parallel Programming on Silicon Graphics Computer Systems*, Mountain View, California, Silicon Graphics, Inc., 1989, (part number 007–0770–010) documents parallel programming as implemented on Silicon Graphics multiprocessor computer systems.

Data Dependence

- *Advanced Compiler Optimization for Supercomputers*, David Padua and Michael Wolfe, Communications of the ACM, Volume 29, No. 12, December 1986.
- *Data Dependence and Its Application to Parallel Processing*, Michael Wolfe and Utpal Banerjee, International Journal of Parallel Programming, Volume 16, No. 2, April 1988.
- *Optimizing Supercompilers for Supercomputers*, Michael Wolfe, Ph.D. Thesis, Department of Computer Science, Report No. 82–1009, University of Illinois, Urbana, Illinois, October, 1982.

Style Conventions

The conventions used in the guide help make information easy to access and understand. The following list defines the notation and syntax conventions:

italic	Indicates arguments in a command line that you must replace with a valid value. In text it is used to indicate commands, document titles, file names, glossary items, new terms, and variables.
`courier`	Indicates computer output and program listings.
`courier bold`	Indicates computer input and non–printing keys.
[ ]	Brackets enclose optional command arguments. Do not enter the brackets.
. . .	An ellipsis indicates that the preceding optional items can appear more than once in succession.

( )	Parentheses enclose items. Enter them exactly as shown.
{ }	Braces enclose items from which you must select exactly one. Do not enter the braces.
|	The vertical bar separates items from which you can choose one.

For example,

```
-optimize=integer
```

means that for the *optimize* option, you must substitute an integer representing the level of optimization you want, such as

```
-optimize=2
```

This guide uses lowercase letters for PCA command–line options; command–line options are not case sensitive. File name parameters, however, are case sensitive.

Chapter 1

Compiling with IRIS Power C

You can use IRIS Power C (PCA) in two ways:

- As a standalone analysis tool, *pca*(1) - Passing options directly to PCA and using it as a standalone analyzer are explained in **Chapter 3, "PCA Command–Line Options."**
- As a phase of the C Compiler, *cc*(1) - This chapter describes how to use Power C with the C compiler (*cc*) to produce code that runs in parallel. When you use Power C this way, you simply pass *pca* options to *cc*.

Table 1–1 lists the options you can pass to *cc*.

Option to cc	Function
-pca	Run Power C Analyzer (*pca*) and the multiprocessing compiler (*mpc*)
-pca list	All of the above plus generate a listing file
-pca keep	All of the above plus keep the *mp* source file
-WK,*options*	Pass options to *pca*
-mp	Run the multiprocessing compiler only

Table 1–1 Power C Options to *cc*

In many cases, you will invoke *pca* as an option to *cc*. You can also change the PCA default settings and pass options to *pca* as part of a *cc* compilation. To do this, just pass these options via the *-WK* mechanism. For more information on the *cc* compiler and the *-WK* options, refer to *cc*(1) in the *IRIX User's Reference Manual*.

1.1 Compiler Command–Line Syntax

The C compiler command–line syntax is:

```
cc [argument] –pca [list|keep] [–WK,-option[=value]
[,-option[=value]]...] filename.c
```

Each option and its description appears in **Table 1–2** .

Option	Description
cc	Invokes the C compiler that compiles, optimizes, assembles, and link edits the program.
argument	Passes an argument to *cc* (see *cc*(1) for the arguments and their descriptions).
-pca	Invokes the *pca* optimizer that concurrentizes C by restructuring certain parts of code and adding parallel programming directives where possible.
list	Produces the annotated *pca* listing file, *file.l*. The content of this file varies depending on the value of the *-lo* command–line option (see Section 3.5.4, "listoptions"). The default is to produce no listing file.
keep	Produces the file containing the concurrentized C source code, *file.m*, that is accepted by the compiler.

-WK,	Allows you to pass *pca* options on the command line. Do not insert a space between -*WK*, and the option(s). If you specify multiple options, separate each with a comma (,).
-*option*	Allows you to specify a *pca* command−line option listed in Table 1−3 and explained in Chapter 3, "PCA Command−Line Options".
value	Sets a value for a command−line option (see Chapter 3, "PCA Command−Line Options" for values).
filename.c	Uses the specified file as the C source file. The file name must have a *.c* suffix.

Table 1−2 Power C Command−Line Options to *cc*

For example, to compile a C program with *-pca* and the option *-lo=l*, enter:

```
cc −pca −WK,−lo=l prog.c
```

The *-lo* (*listoptions*) option is described in **Chapter 3, "PCA Command−Line Options"** .

1.2 Generating a Listing File

You can generate several types of files when you compile your program. To generate a PCA listing file, use the *list* option. This option produces *file.l*, which contains the annotated *pca* listing of the parts of the program that can (and cannot) run in parallel on multiprocessors. **Chapter 2, "How to Use IRIS Power C,"** has an example of a listing file. The content of this file varies depending on the value of the *-lo* command−line option (see **Section 3.5.4, "listoptions"**). The default is to produce no listing file.

1.3 Generating mp Source and Listing Files

To generate a multiprocessing source file as well as a listing file, use the *keep* option. This option produces an intermediate file, *file.M*, that is accepted by the compiler. **Chapter 2, "How to Use IRIS Power C"** has an example of an intermediate multiprocessing source file.

1.4 Passing Options to PCA

To pass options to PCA, send the -*WK*, option to *cc*. The options are listed in **Table 1−3** .

Purpose	Long Name	Short Name	Default Value
Run code in parallel	concurrentize	conc	concurrentize
	noconcurrentize	nconc	concurrentize
	minconcurrent=*n*	mc=*n*	minconcurrent=1000
Optimize code	arclimit=*n*	arclm=*n*	arclimit=2000
	address_resolution_level=*n*	arl=*n*	arl=1
	limit=*n*	lm=*n*	limit=5000
	machine=*list*	ma=*list*	machine=s
	nomachine	nma	machine=s
	optimize=*n*	o=*n*	optimize=5
	roundoff=*n*	r=*n*	roundoff=0
	scalaropt=*n*	so=*n*	scalaropt=4
	syntax=[a|k]	sy=[a|k]	syntax=a
	unroll=*n*	ur=*n*	unroll=4
	unroll2=*n*	ur2=*n*	unroll2=100
In−lining and Inter−procedure Analysis	inline[=*names*]	inl[=*names*]	(off)
	ipa[=*names*]	ipa[=*names*]	(off)
	inline_create=*file*	incr=*file*	(off)
	ipa_create=*file*	ipacr=*file*	(off)
	inline_from_files=*list*	inff=*list*	current source file
	inline_from_libraries=*list*	infl=*list*	
	ipa_from_files=*list*	ipaff=*list*	(off)
	ipa_from_libraries=*list*	ipafl=*list*	current source

	inline_depth[=*n*]	ind[=*n*]	file
	inline_looplevel[=*n*]	inll[=*n*]	(off)
	ipa_looplevel[=*n*]	ipall[=*n*]	ind=2
	inline_manual	inm	inll=2
	ipa_manual	ipam	ipall=2
			(off)
			(off)
Input/Output	cmp=*file*	cmp=*file*	see text
	nocmp	ncmp	see text
	input[=*file*]	i[=*file*]	see text
	list=*file*	l=*file*	nolist
	nolist	nl	nolist
Listing	cmpoptions=*list*	cp=*list*	nocmpoptions
	nocmpoptions	ncp	nocmpoptions
	lines=*n*	ln=*n*	lines=55
	listoptions=*list*	lo=*list*	(no listing)
	listingwidth=<80|132>	lw=<80,132>	80
Memory Management	cacheline=*n*	chl=*n*	chl=64
	cachesize=*n*	chs=*n*	chs=64
	dpregisters=*n*	dpr=*n*	dpr=12
	fpregisters=*n*	fpr=*n*	fpr=12
	setassociativity=*n*	sasc=*n*	sasc=1
Invariant IF Floating	each_invariant_if_growth=*n*	eiifg=*n*	eiifg=20
	max_invariant_if_growth=*n*	miifg=*n*	miifg=500
Command Line Options for Portability	DOLLAR	P	off
	FLOAT	INLC	off
	SIGNED	STDIO	off
	VOLATILE		off
	PROCESSORS		P=0
	INLINE_AND_COPY		off
	STDIO		off

Table 1–3 PCA Command_line Options

For more information on the command–line options, see **Chapter 3, "PCA Command–Line Options"**. For in–lining and interprocedural analysis (IPA), see **Chapter 7, "In–lining and Interprocedural Analysis."**

1.5 Summary

This chapter described the *pca* options that you can pass to the *cc* compiler. You can also pass these options directly to PCA, and use it as a standalone analyzer. **Chapter 3, "PCA Command–Line Options"** explains each PCA option in detail and describes how to use the Analyzer.

Before turning to **Chapter 3, "PCA Command–Line Options"** however, continue with **Chapter 2, "How to Use IRIS Power C,"** which tells you how to interpret the *.L* and *.M* files, and explains how to use Power C to get more of a program to run in parallel.

Chapter 2

How to Use IRIS Power C

This chapter describes how to use IRIS Power C to get the maximum amount of code to run in parallel. You can use the Power C Analyzer either in conjunction with the *cc* compiler (as **Chapter 1, "Compiling with IRIS Power C"** explained) or as a standalone analyzer (as described in **Chapter 3, "PCA Command–Line Options"**).

What does PCA do to your code? It inserts directives into the code that tells the compiler to run loops in parallel. It does not change the logic of your program although, if requested, it will rewrite portions of the program to run faster. This chapter explains what happens to the code when you use *pca* and explains how to use *pca* as an analyzer.

The Power C Analyzer is a powerful tool that can:

- direct C code to run in parallel
- determine data dependencies
- distribute well−behaved loops and certain other code across multiprocessors
- optimize source code

Power C is neither an extension to the C language nor a way to concurrentize any arbitrary C program. Furthermore, if the Power C directives are removed or ignored, then the code becomes a valid serial program. This allows full source code compatibility with other non−multiprocessing systems.

2.1 Compiling Programs with PCA

This section explains some of the ways to use PCA and what happens during the stages of compilation. **Figure 2−1** shows the program flow through the C compiler. When you compile a program you have the option of specifying *-pca* (the Power C Analyzer and multiprocessing compiler) or *-mp* (the multiprocessing compiler only).

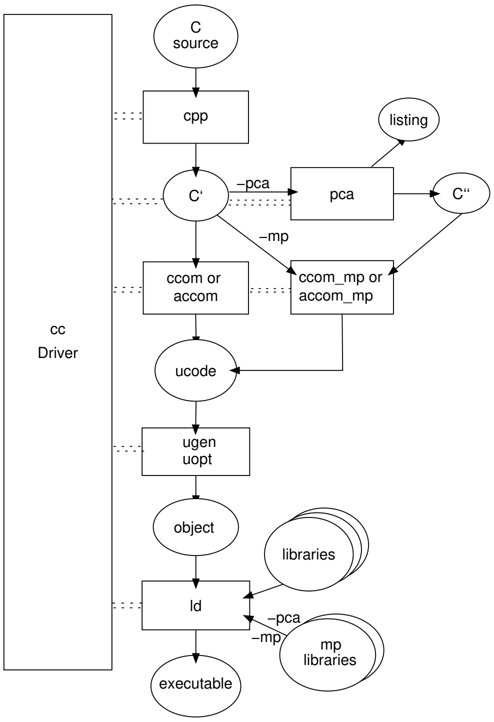


Figure 2–1 Power C Process Flow

The first stage invokes *cpp* to handle *cpp* directives. (For more information, see *cpp*(1) in the *IRIX User's Reference Manual*.) If you specify *-pca*, PCA puts directives into the *cpp* output that allows data–independent loops to run in parallel. As explained in **Chapter 1, "Compiling with IRIS Power C"** you can use the *list* and *keep* options to direct PCA to generate a listing file (with the suffix *.L*) and/or an intermediate multiprocessing source file (with the suffix *.M*). Also, you can hand–insert directives into the code prior to compilation. Finally, the multiprocessing C compiler compiles the transformed PCA–generated file to produce an executable object file. You can run this executable on any Silicon Graphics IRIS system (single or multiprocessor); the executable will adapt to the number of processors present at execution.

Figure 2–2 shows how PCA internally processes the C source code. PCA parses the source into an internal representation, performs data dependency analysis and transformations, generates C source code from the internal representation, and produces intermediate C code.

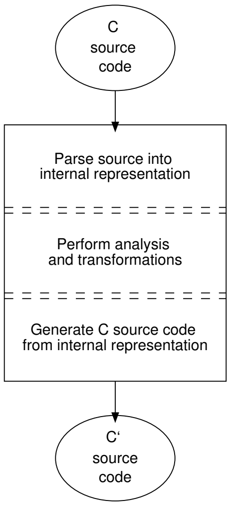


Figure 2–2 PCA Processing Flow

If you specify the multiprocessing compiler, with the *-mp* or *-pca* options, it identifies parallel directives, rewrites the parallel code with explicit runtime calls, processes the C code, and generates the executable object. **Figure 2–3** shows this code flow.

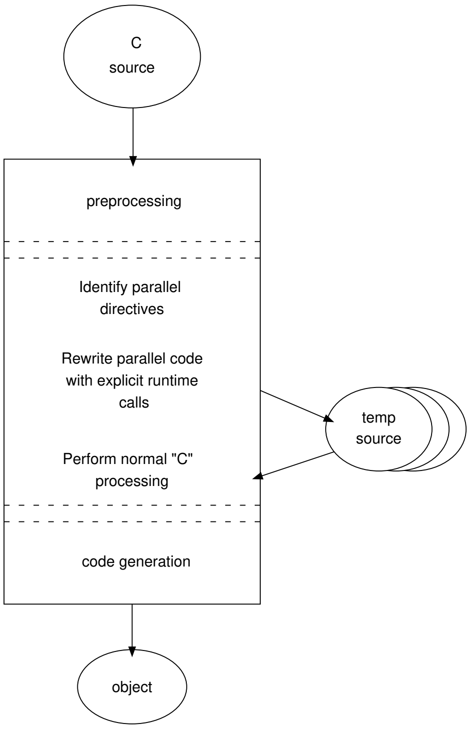


Figure 2–3 Multiprocessing Compiler Flow

2.1.1 Different Ways of Concurrentizing Code

You can use Power C in different ways to produce concurrentized code: automatically, manually, and iteratively. If you use Power C automatically as a phase of compilation, PCA concurrentizes parts of the code that it determines will safely run in parallel and then compiles the code. You will probably invoke *pca* this way when you first begin using Power C. **Figure 2–4** shows the process flow.

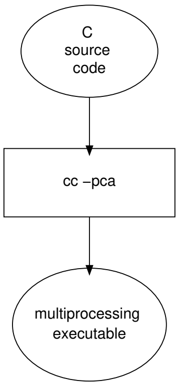


Figure 2−4 Using -pca

Rather than using the Power C Analyzer, you can enter directives manually to concurrentize the code. This method is shown in **Section 2.3, "Examples,"** toward the end of this chapter. **Figure 2−5** shows the program flow when you hand insert the multiprocessing directives.

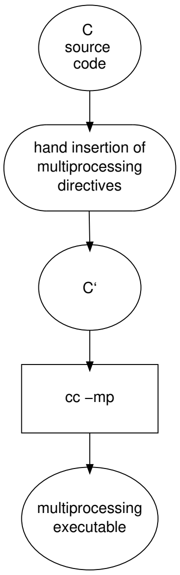


Figure 2–5 Inserting Directives Manually

In addition to the automatic and manual methods of compilation, you can also use Power C iteratively, as shown in **Figure 2–6** . This method is best for getting the maximum amount of code to run in parallel.

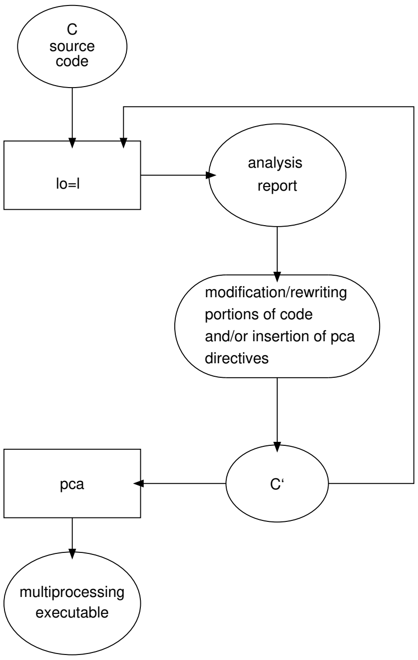


Figure 2–6 Using PCA Iteratively

Now that you have seen the different ways you can use Power C, the next step is to figure out how further to concurrentize the maximum amount of code. The sections that follow illustrate the iterative method outlined in **Figure 2–6** , and describe how Power C can help to optimize a program and run more of it in parallel.

2.1.2 Using the Listing File

First, use the listing file to see exactly which loops were concurrentized. To generate a listing file, use the *list* option. (Refer to **Chapter 1, "Compiling with IRIS Power C"** for more information on the *list* option.) The listing file contains the annotated pca listing of the parts of the program that can (and cannot) run in parallel on multiprocessors. The content of this file varies depending on the value of the *-lo* command−line option (see **Section 3.5.4, "listoptions"**).

For example, suppose you have a simple C program named *test.c* that looks like this:

```
int a[1000], sum;
void example_3_2_2 ()
{
      int i;
      sum = 0;
      for (i=0; i<1000; i++)
          a[i] = i;
      for (i=0; i<1000; i++)
          sum += a[i];
      printf ("The sum is %d\n", sum);
}
```

Now, let PCA do the work for you. Compile the program by passing the *-pca* option to *cc*, and generate a listing file by using the *list* option.

To compile the program, enter:

```
cc −c −pca list test.c
```

which produces a PCA listing file, *test.L*.

The listing file looks similar to this:

```
---------------------------Loop Table--------------------------
                                  Nest
Loop Message                    Level  Contains Lines
===============================================================
for i                          1    6−7 "test.c"
     Original Loop Split Into Sub−Loops
     1. Concurrent & Enhanced Scalar
                                 1    6−7, 9 "test.c"
     Line:6 Loop unrolled 4 times to improve scalar performance.
     Line:6 Loop has been fused with others to reduce overhead.
     2. Enhanced Scalar         1    6, 9 "test.c"
     Line:6 Loop unrolled 4 times to improve scalar performance.
     Line:9 Data dependence involving this line
     due to variable"sum".
     Line:9 Loop has been fused with others to reduce overhead.
for i                          1    8−9 "test.c"
               2 loops total
               1 loops concurrentized
               1 this loop has been fused with other loops
```

PCA was able to concurrentize the first *for* loop. In this case, initialization of array a was executed in parallel on the available processors. PCA was unable to concurrentize the second loop, hence the "preferred scalar mode" message. However, in subsequent sections, you will see how to get PCA to concurrentize the other *for* loop. (For more information on the listing file, see **Chapter 8, "The PCA Listing"** .)

2.1.3 Using the Listing and mp Source Files

To generate a multiprocessing source file as well as a listing file, use the *keep* option. (Refer to **Chapter 1, "Compiling with IRIS Power C"** for more information on the *keep* option.) This option produces a *test.L* file and an intermediate file, *test.M*.

To generate these files, enter:

```
cc −c −pca keep test.c
```

which yields a listing file, *test.L*, and an intermediate file containing concurrentized C source code, *test.M*. The content of the file, *test.M*, appears in the example that follows:

```
int a[1000];
int sum;
void example_3_2_3( )
{
      int i;
      int _Kii1;
      sum = 0;
#pragma parallel shared(a) local(_Kii1)
#pragma pfor iterate(_Kii1=0;1000;1)
      for ( _Kii1 = 0; _Kii1<=999; _Kii1++ ) {
          a[_Kii1] = _Kii1;
      }
      for ( _Kii1 = 0; _Kii1<=999; _Kii1++ ) {
          sum +=  a[_Kii1];
      }
      printf( "The sum is %d\n", sum );
}
```

The *test.M* file shows that PCA finds it can run a *for* loop in parallel. PCA inserts the directives:

```
#pragma parallel shared(a) local(_Kii1)
```

and

```
#pragma pfor iterate(_Kii1=0;1000;1)
```

into the code. The *#pragma parallel* tells the multiprocessing C compiler to start a parallel region. The phrase *shared(a)* tells the compiler that all processes that execute the *for* loop share the array a. The phrase *local(i)* indicates that every process executing the loop has a local variable *i*.

The *#pragma pfor* tells the compiler that the next *for* loop can run in parallel, and the loop statements are executed on the processors available. The phrase *iterate* gives the multiprocessing C compiler the information it needs to uniquely identify the iterations of the loop and partition them to particular threads of execution.

2.1.4 Getting More Code to Run in Parallel

As you can see in the previous examples, using PCA without options has some drawbacks. If roundoff error is not critical, the second *for* loop should also be able to safely run in parallel. All that's needed is to pass an option to *pca*, specifically the *-roundoff* (*-r*) option. To do this, enter:

```
cc –pca keep –WK,–r=2 test.c
```

The *-r=2* option tells PCA to mark reductions to run concurrently. See **Chapter 3, "PCA Command–Line Options"** for details on the *-roundoff* option.

After execution, both loops were concurrentized. The listing file looks similar to this:

```
---------------------------Loop Table---------------------------
                                 Nest
Loop Message                    Level   Contains Lines
================================================================
for i                              1     6–7 "test.c"
    1. Concurrent & Enhanced Scalar  1     6–7, 9 "test.c"
    Line:6 Loop has been fused with others to reduce overhead.
for i                              1     8–9 "test.c"
               2 loops total
               1 loops concurrentized
               1 this loop has been fused with other loops
```

The intermediate file, *test.M*, now looks like this:

```
int a[1000];
int sum;
void example_3_2_4(  )
{
      int i;
      int _Kii1;
      int sum1;
      sum = 0;
#pragma parallel shared(a, sum) local(_Kii1, sum1)
      {
          sum1 = 0;
#pragma pfor iterate(_Kii1=0;1000;1)
          for ( _Kii1 = 0; _Kii1<=999; _Kii1++ ) {
              a[_Kii1] = _Kii1;
              sum1 +=  a[_Kii1];
          }
#pragma critical
          {
              sum +=  sum1;
          }
      }
      printf( "The sum is %d\n", sum );
}
```

It's easy to see where PCA found it could run more code in parallel. PCA expanded the previously inserted directives:

```
#pragma parallel shared(a) local(i)
```

to

```
#pragma parallel shared(a, sum) local(_Kii1, sum1)
```

and inserted a new directive:

```
#pragma critical
```

into the code. The *#pragma parallel* is expanded to include *sum* and *sum1*. The *#pragma critical* instructs each thread to add its partial sum, *sum1* to *sum*, one at a time.

2.2 Using PCA to Run Programs in Parallel

As you learned in the previous section, the Power C Analyzer automatically identifies loops in a C program that can run safely in parallel. For a few programs, you may be able to use PCA automatically to make a significant part of the code run in parallel. And to make the program more efficient, you can pass PCA options to *cc*, such as the *–roundoff* option that was used in the previous example. However, for most programs, you will also want to make small code changes that let PCA run more of the code in parallel. Often, the changes are easy to make.

The PCA listing indirectly supplies information about when and where to modify your code. The more you know about where your program spends much of its time, and the better you understand the PCA listings, the easier it is to recognize where small changes to the source code can produce more parallel code.

2.2.1 How Does the Power C Analyzer Work?

When you use the Power C Analyzer, PCA does a data–dependency analysis on the code. During this analysis, PCA looks for *for* loops with the property that each iteration of the loop is independent of all other iterations. Because each iteration of the loop is self–contained, the system can execute the iterations in any order (or even simultaneously on separate processors) and get the same answer after running all iterations as it would running the iterations serially.

When PCA finds a loop that has the property of data independence, it knows it can safely run the loop in parallel. When PCA finds a loop that has iterations that could depend on other iterations, it cannot run the loop in parallel, but PCA can tell you what is causing the problem. You can use directives to get around the problem, if you know that there are no real data dependencies.

Finally, if PCA is unable to run the loop in parallel, you can often modify your program to make it safe to run in parallel.

2.2.2 Getting Started

When trying to find loops to run in parallel, review the listing file and focus your efforts on the areas of the code that use the bulk of the run time. You cannot significantly improve the performance of your program by spending a lot of time trying to parallelize a routine that used only 1% of its run time.

2.2.3 Profiling the Code

To find out where your code spends its time, take an execution profile of the program. Use either pc−sample profiling by using *cc*(1) with the *-p* option, or basic block profiling by using *pixie*(1). The application program is then run to gather performance data, and *prof*(1) is used to generate a profile report from the collected data. For details on these commands, see the *IRIX User's Man Pages* and the *IRIS−4D Series Compiler Guide.*

You can do the profiling in two ways. The first is the conservative approach, where you take a profile of the original (nonparallel) job. Then you run in parallel only the loops that account for most of the run time. This approach reduces the chances that something may go wrong because it makes fewer changes to the code. It also focuses most of the effort into the smallest number of lines of code.

The second approach is more aggressive. After running the program through PCA, profile the resultant multiprocessed job. Use this approach if you think that PCA does a good job with the existing program (for example, a converted program that already runs well on traditional vector architectures). Many such programs run in parallel very well without additional effort. You can then focus on the routines with which PCA has a problem.

After you decide on an approach, use the profile data to direct your efforts to the most time−consuming routines. Once you find such a routine, submit it alone to PCA. If the routine is in the middle of a large file, consider using *csplit*(1) to isolate that routine. Compile it with the *-pca list* option, and examine the listing file. The PCA listing identifies which loops PCA can and cannot run in parallel. For the latter, the listing also tells why PCA could not convert the loop for parallel execution.

2.2.4 Interpreting the PCA Listing

At times, PCA may not run a loop in parallel. PCA generates messages in the listing that complain about the code. One message is:

```
dependencies prevent parallelism
```

This message means that PCA believes data dependencies exist in the loop. The listing also names the variables PCA believes are involved in the dependence. Dependencies can be simple to complex, so deal with each dependency on an individual basis.

PCA may generate another message:

```
unoptimizable function/subroutine call
```

This message means that the loop includes a call to an external procedure or function. Because PCA cannot see into that other routine, it must assume that the routine contains data dependencies. To change this assumption, either use in−lining or interprocedural analysis (described in **Chapter 7, "In−lining and Interprocedural Analysis"**) or, if you know the routine is safe to run in parallel, insert the *concurrent*, *concurrent call*, or *no side effects* directives.

2.3 Examples

The following examples show how inserting directives into the C source code produces code that runs in parallel.

2.3.1 Permutation Index

This example uses the file *permute.c*, which contains the following C code:

```
void example_3_4_1 (double a[], double b[], long index[])
{
      long i;
#pragma distinct (a[], b[], index[])
#pragma concurrent
      for (i=0; i<10000; i++)
          a[index[i]] += b[i];
}
```

Two directives have been inserted into the code; without these directives, PCA cannot produce concurrentized code. The *#pragma distinct* tells PCA that no data dependencies occur between a, b, and index; that is, these objects do not overlap. The *#pragma concurrent* tells PCA to ignore assumed dependencies in the next loop. (See **Chapter 4, "Power C Analyzer Directives,"** for a complete description of these pragmas.)

Compiling *permute.c*:

```
cc -pca keep permute.c
```

produces the listing file, *permute.L*, which looks similar to this:

```
-----------------------  Loop Table  --------------------------
                                        Nest
Loop          Message                Level  Contains Lines
===============================================================
for i                                1      6-7 "permute.c"
     1. Concurrent & Enhanced Scalar   1      6-7 "permute.c"
              1 loops total
              1 loops concurrentized
```

The intermediate multiprocessing file, *permute.M*, looks like this:

```
void permute( double a[], double b[], long index[] )
{
      long i;
#pragma parallel shared(index, a, b) local(i)
#pragma pfor iterate(i=0;10000;1)
      for ( i = 0; i<=9999; i++ ) {
          a[index[i]] +=  b[i];
      }
}
```

PCA was able to safely run the loop in parallel. PCA inserted *#pragma parallel shared* and *#pragma pfor iterate*. 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 would get a different location in *a*). A permutation vector is a list of numbers, each of which is different from all the others. If you know that a is a permutation vector, then data independence exists. An example of a permutation index is a list of objects in which each object appears exactly once.

Note: PCA cannot check the truth of directives. When you use a directive, you must be certain that the directive is always true for all possible input data.

2.3.2 Function Call

The next example has *#pragma concurrent call* inserted into the code. This pragma tells PCA that the function calls in the next loop are safe to run in parallel. The file *sub.c* contains:

```
double a[10000], b[10000];
void sub (double [], double [], long);
void example_3_4_2 ()
{
      long i;
#pragma concurrent call
      for (i=0; i<10000; i++)
          sub (a, b, i);
}
```

The listing file, *sub.L*, looks similar this:

```
----------------------  Loop Table  ---------------------------
                                      Nest
Loop        Message                Level  Contains Lines
===============================================================
for i                              1      7-8 "sub.c"
      1. Concurrent & Enhanced Scalar 1      7-8 "sub.c"
               1 loops total
               1 loops concurrentized
```

After compilation, the intermediate file, *sub.M*, looks like this:

```
double a[10000];
double b[10000];
void sub( double [], double [], long  );
void sub(  )
{
      long i;
#pragma parallel shared(a, b) local(i)
#pragma pfor iterate(i=0;10000;1)
      for ( i = 0; i<=9999; i++ ) {
          sub( a, b, i );
      }
   }
```

PCA was able to safely run the loop in parallel. PCA inserted *#pragma parallel shared* and *#pragma pfor iterate* into the code.

2.4 Summary

PCA provides a great deal of information about the dependencies of loops in a C program. Often, PCA can use the information to run loops automatically in parallel. However, when PCA is unable to convert the code automatically for parallel execution, you can often tell it more information that allows it to mark the code to run in parallel. Often, you'll have to make only small changes to the code or add a PCA command−line option to transform a program into an efficient parallel version.

To learn more about PCA command−line options and how to use PCA as a standalone analyzer, turn to **Chapter 3, "PCA Command−Line Options."**

Chapter 3

PCA Command−Line Options

This chapter explains how to use the Power C Analyzer and describes the *pca* command−line options.

Chapter 2 described how to use PCA by passing *-pca* and its options to the C compiler. In this mode, PCA is run as a phase of compilation. PCA analyzes the code, adds parallel directives, and then compiles the program.

This chapter explains how to run PCA as a standalone analysis tool. By using *pca* and reviewing its analysis report, you can try different options and/or code modifications to see their effect on your program. Once you have reached optimum parallelism, you can do a final compilation with the *-pca* option.

3.1 pca Command–Line Syntax

The *pca* command–line syntax is:

```
/usr/lib/pca [ options ] ... filename.c
```

When you specify a command–line option, you can use the long name, short name, or any portion of the name that uniquely identifies the command (for example, *-roundoff* or *-r*). If a command–line option appears more than once on the command line, PCA uses the last occurrence—except for input/output options. PCA does not accept multiple occurrences of input/output options.

The command–line options described in this chapter appear in lowercase letters; however, PCA is not case sensitive.

Table 3–1 lists the *pca* command–line options. The first column defines the functional category of the option: concurrentization, general optimization, in–lining and interprocedural analysis, input/output, and listing. The next three columns list the long name, short name, and default values of each option.

Purpose	Long Name	Short Name	Default Value
Run code in parallel	concurrentize	conc	concurrentize
	noconcurrentize	nconc	concurrentize
	minconcurrent=*n*	mc=*n*	minconcurrent =1000
Optimize code	arclimit=*n*	arclm=*n*	arclimit=2000
	address_resolution_level=*n*	arl=*n*	arl=1
	limit=*n*	lm=*n*	limit=5000
	machine=*list*	ma=*list*	machine=s
	nomachine	nma	machine=s
	optimize=*n*	o=*n*	optimize=5
	roundoff=*n*	r=*n*	roundoff=0
	scalaropt=*n*	so=*n*	scalaropt=3
	syntax=[a|k]	sy=[a|k]	syntax=a
	unroll=*n*	ur=*n*	unroll=4
	unroll2=*n*	ur2=*n*	unroll2=100
In–lining and Inter–procedural Analysis	inline[=*names*]	inl[=*names*]	(off)
	ipa[=*names*]	ipa[=*names*]	(off)
	inline_create=*file*	incr=*file*	(off)
	ipa_create=*file*	ipacr=*file*	(off)
	inline_from_files=*list*	inff=*list*	current source
	inline_from_libraries=*list*	infl=*list*	file
	ipa_from_files=*list*	ipaff=*list*	(off)
	ipa_from_libraries=*list*	ipafl=*list*	current source
	inline_depth[=*n*]	ind[=*n*]	file
	inline_looplevel[=*n*]	inll[=*n*]	(off)
	ipa_looplevel[=*n*]	ipall[=*n*]	ind=2
	inline_manual	inm	inll=2
	ipa_manual	ipam	ipall=2
			(off)
			(off)
Input/Output	cmp=*file*	cmp=*file*	see text
	nocmp	ncmp	see text
	input[=*file*]	i[=*file*]	see text
	list=*file*	l=*file*	nolist

	nolist	nl	nolist
Listing	cmpoptions=*list*	cp=*list*	nocmpoptions
	nocmpoptions	ncp	nocmpoptions
	lines=*n*	ln=*n*	lines=55
	listoptions=*list*	lo=*list*	(no listing)
	listingwidth=<80|132>	lw=<80,132>	80
Memory Management	cacheline=*n*	chl=*n*	chl=64
	cachesize=*n*	chs=*n*	chs=64
	dpregisters=*n*	dpr=*n*	dpr=6
	fpregisters=*n*	fpr=*n*	fpr=12
	setassociativity=*n*	sasc=*n*	sasc=1
Invariant IF Floating	each_invariant_if_growth=*n*	eiifg=*n*	eiifg=20
	max_invariant_if_growth=*n*	miifg=*n*	miifg=500
Command Line Options for Portability	DOLLAR	P	off
	FLOAT	INLC	off
	SIGNED	STDIO	off
	VOLATILE		off
	PROCESSORS		P=0
	INLINE_AND_COPY		off
	STDIO		off

Table 3–1 *pca* Command–Line Options

The following pages explain each option giving its short name, long name, and default, and whether or not you can disable it by using the *no* ([*n*]) notation. In–lining and Interprocedural Analysis are explained in **Chapter 7, "In–lining and Interprocedural Analysis."**

PCA runs after the standard *C preprocessor*. The code examples in this chapter show the original code (before the preprocessor) and the PCA–transformed code (with some of the C preprocessor additions stripped off for clarity).

3.2 Concurrentization Options

Concurrentization is the process by which PCA converts code to execute concurrently (in parallel) on multiple processors.

3.2.1 concurrentize

The syntax for this option is:

```
–[n]conc
–[no]concurrentize  (long name)
–conc               (default value)
```

The *-concurrentize* option tells PCA to mark eligible loops to run concurrently (in parallel). The *-noconcurrentize* option tells PCA not to mark loops to run in parallel but does not prohibit any of the other optimizations that PCA can make.

3.2.2 minconcurrent

The syntax for this option is:

```
–mc=n
–minconcurrent=n   (long name)
–mc=1000           (default value)
```

Executing a loop in parallel incurs overhead that varies with different loops. If a loop has little work, parallel execution might be slower than serial execution because of the overhead. However, beyond a certain level, you can improve performance through parallel execution. This level is passed to PCA with the *-minconcurrent* option.

The range of values for the *-minconcurrent* option is:

```
>=0
```

The higher the *-minconcurrent* value, the larger (more iterations, more statements, or both) the loop body must be in order to run concurrently. To disable this feature and run all possible code in parallel, use the command−line option *-minconcurrent=0*.

At analysis time, PCA estimates the amount of computation inside a loop. You can see this estimate in the Loop Summary (see **Section 8.1.4, "Loop Table (I)"** in Chapter 9, "The PCA Listing") in the "iteration workload" column. This estimate is roughly the number of operators plus the number of operands, excluding the loop index. The product of the workload in each iteration times the number of iterations is considered to be the amount of work of the loop, and this is the value that is compared with the *-minconcurrent* value. If the loop bounds are constant and the estimated amount of work is greater than the *-minconcurrent* value, PCA generates concurrent code for the loop. Otherwise, it leaves the loop serial. However, if the *for* loop bounds are not known at compilation time, PCA generates an *if* expression in the *parallel* pragma. The compiler interprets this expression as a request to generate two loops, one concurrentized and one left serial, which are checked at runtime to decide whether or not to execute the loop in parallel.

The following loop illustrates this feature with the *-minconcurrent* default:

```
int a[], b[], c[], n;
void example_4_2_2 ()
{
      int i;
      for (i=0; i<n; i++) {
          a[i] = b[i] + c[i];
      }
}
```

becomes:

```
int a[];
int b[];
int c[];
int n;
void example_4_2_2(  )

{
      int i;
#pragma parallel if(n > 201) byvalue(n) shared(a, b, c) local(i)
#pragma pfor iterate(i=0;n;1)
      for ( i = 0; i<n; i++ ) {
          a[i] = b[i] + c[i];
      }
}
```

The Loop Summary (from the listing file) shows what PCA concurrentized.

```
----------------------------Loop Table-------------------------
                                Nest
Loop         Message             Level     Contains Lines
================================================================
for i                           1        5-7 "example_4_2_2.c"
     1. Concurrent                1        5-7 "example_4_2_2.c"
```

PCA calculates that the amount of "work" being done by each iteration is 5 units. At run time, if the iteration count *n* is less than or equal to 200 (1000/5), the concurrent loop is executed serially; otherwise it is executed in parallel.

If you specify *-minconcurrent=0* on the command line, the if(n > 201) clause will be left out of the *# pragma parallel*, and the loop will always execute in parallel.

3.3 Optimization Options

The following sections explain each optimization command−line option.

3.3.1 syntax

The syntax for the syntax option is:

```
-sy=[a|k]
-syntax=[a|k]              (long name)
-syntax=a                  (default value)
```

The *syntax* option allows you to select the dialect of C that PCA expects. The default dialect is ANSI C (*-syntax=a*). Specifying *-syntax=k* instructs PCA to accept traditional, K&R C.

If you don’t specify a dialect, PCA will adjust to the actual dialect used in your source.

3.3.2 address_resolution_level

The syntax for this option is:

```
-arl=n

-address_resolution_level=n         (long name)
-arl=1                              (default value)
```

The *-address_resolution_level* option lets you control the assumptions that PCA makes about memory aliases. **Table 3−2** lists the levels of control.

An associated directive, *#pragma arl=n*, has the same meaning as the *-arl* command−line option (see **Chapter 4, "Power C Analyzer Directives,"** for details).

Each of the levels described in **Table 3−2** is cumulative; that is, specifying *arl=3* includes all the actions of *arl=1*, *arl=2*, as well as *arl=3*.

Value **Description**

0 Make no assumptions about memory aliases.

1 Assume that there are no pointer self−references (the default); that is, a pointer will not contain its own address. Self−referencing pointers are not common, and this level avoids the problem in loops such as:

```
int *p
...
for ( i=0; i<n; i++ ) }
    p[i] = a[i];
    }
```

In the example, there could be dependencies from the first iteration to the other iterations since *p[0]* might be *&p*.

2 Assume that none of the objects represented by the parameters overlap in memory; that is, each argument is distinct from the other. This is equivalent to *#pragma distinct* for all parameters (see Chapter 4, "Power C Analyzer Directives," for a description of *#pragma distinct*).

This is not true for most C functions, and PCA will assume there is (or could be) parameter aliasing unless you specify *arl=2* or greater.

3 Assume globals, parameters, and locals form distinct groups. The memory locations referred to using local variables will be different from the memory locations referred to using global variables, and both of these will be different from the memory locations referred to through parameters. For example:

```
float *a;
f(x)
float x[1000];
{
   int i;
```

```
float f[1000];
for ( i=0; i<1000; i++ ) {
   a[i] = x[i] + f[i];
     }
}
```

pca will not concurrentize this loop unless you specify *arl=3* (or greater), which indicates that the arrays *a*, *f*, and *x* are distinct.

4	Assume that there are no aliases for objects; that is, all pointers/arrays are distinct from each other. If pointers are used, only one name is used to reference an object.

Table 3−2 Address Resolution Levels, *arl*

3.3.3 scalaropt

The syntax for this option is:

```
-so=n

-scalaropt=n         (long name)
-scalaropt=2         (default value)
```

The *-scalaropt* option sets the level of scalar optimization PCA will perform. Scalar optimizations include *dusty−deck transformations*, *dead code elimination*, and *loop unrolling*.

The parameter sets the optimization level as described in **Table 3−3** .

Scalar optimizations are discussed in detail in **Section 6.3, "Scalar Optimizations."**

Value	Description
0	Perform no scalar optimizations.
1	Perform only simple scalar optimizations, such as dead−code elimination, global forward substitution, and dusty−deck IF transformations. Perform code floating if *-roundoff >=1*.
2	Perform the full range of scalar optimizations. Remove floating invariant IFs from loops. Recognize induction variables. Reroll loops, expand arrays, peel loops, perform loop fusion.
3	Enable memory management if *-roundoff=3*. Allow dead code elimination of unnecessary program fragments during output conversion. Other optimizations might expose more dead code.

Table 3−3 Saclar Optimization Levels

3.3.4 limit

The syntax for this option is:

```
-lm=n

-limit=n             (long name)
-lm=5000             (default value)
```

PCA estimates how much time it would need to analyze each loop−nest construct. If a nest of loops is too deep, PCA ignores the outer loop and recursively visits the inner loops until it finds a nest of loops that is not too deep. The *-limit* option is the upper threshold of the amount of work that controls what PCA thinks is "too deep."

Larger loop−nest limits might allow PCA to convert the outer loops of a deeply nested loop structure to run in parallel. (Running the outermost loop in parallel usually results in the best performance increase.) But larger loop−nest limits can increase the analysis time. The limit does not correspond to the *for* loop−nest level. It is an estimate of the number of loop orderings that PCA can generate from a loop−nest. The *-limit* option resets this internal limit.

Note: This limit is adequate for most programs. If your program is extremely complex, you might want to increase this limit.

3.3.5 arclimit

The syntax for this option is:

```
-arclm=n

-arclimit=n          (long name)
-arclm=2000          (default value)
```

The *-arclimit* option sets the size of the "dependence arc data structure" that PCA uses to perform data–dependence analysis. (See **Appendix B, "Data–Dependence Analysis,"** for a description of data–dependence analysis.) This data structure is dynamically allocated on a loop–nest by loop–nest basis.

The formula PCA uses to estimate the number of dependence arcs for a given loop–nest is:

```
array_size = max (#_of_statements * 4, arclimit value)
```

PCA assumes that each statement will have four dependence arcs (a worst–case estimate).

When you include the Loop Summary in the listing file (*-listoptions=l*), PCA marks any loop that was too complex for the dependence data structure to hold the information. The following example shows the Loop Summary (from the listing file for a PCA run with the value *-arclimit=200*). In this example, PCA detected that the given loop, which contained 123 statements, had too many dependence arcs for the data structure as allocated. The storage that was allocated for the dependence arc array had been:

```
max(123 * 4 , 200) = 492
The Loop Summary looks like this:
---------------------------Loop Table--------------------------
                                 Nest
Loop         Message          Level    Contains Lines
===============================================================
for i                         1      5-129 "example_4_3_5a.c"
      1. Scalar                 1      5-129 "example_4_3_5a.c"
              Line:5  Data dependence analysis aborted due to insufficient
 storage for graph arcs.
```

Suppose for the above example, you change the *-arclimit* value to something greater than 492. PCA might be able to optimize the given loop provided that there are no data–dependence violations. The next example shows the Loop Summary after setting *-arclimit=2000*.

```
--------------------------Loop Table---------------------------

                                 Nest
Loop         Message          Level    Contains Lines
===============================================================
for i                           1      5-129 "example_4_3_5b.c"
      1. Concurrent               1      5-129 "example_4_3_5b.c"
```

The maximum valid *-arclimit* value is 2000. If you specify a value greater than 2000, PCA defaults to allocating 2000 for the data–dependence array. PCA gives no warning when it does this.

3.3.6 machine

The syntax for this option is:

```
-[n]ma=list

-[no]machine=list    (long name)
-ma=s                (default value)
```

The *-machine* option is list–valued. It has three valid values: *n*, *o*, and *s*. **Table 3–4** defines these values.

Value **Description**

n	Prefer *nonstride–1* array access over *stride–1* array access. For some arrays, *nonstride–1* array access provides the best performance.
o	Do not consider innermost loops for parallel execution. If a loop does not do very much, running the loop in parallel might take longer than running the loop serially because of the overhead. PCA makes decisions concerning the overhead:benefit ratio when it evaluates a loop for parallel execution. If the loop bounds are unknown at analysis time, PCA might generate concurrent code for innermost loops (depending on the *minconcurrent* value), a practice that might be inefficient for the actual loop bounds.
s	Prefer a *for* loop that generates *stride–1* (contiguous) references over one that generates *nonstride–1* operands when PCA must choose only one to mark for parallel execution. This option typically generates the most efficient code, and is the default.

Table 3–4 *machine* Values

If you change the machine option to include choices other than the default value (*-machine=s*), you must also include the default value *s* if it is still to be in effect. For instance, if you want to tell PCA not to try to run inner loops concurrently (option value *o*) but to consider all other eligible loops for parallel execution (option value *s*) you must specify

```
–machine=os
```

If you specify *-machine=o*, you enable *NO–INNER–LOOPS*, but disable the default (prefer *stride–1*) option. You can use any combination of the three choices, except for the self–contradicting combination of *s* (prefer *stride–1*) and *n* (prefer *nonstride–1*).

To disable the options, on the command line, enter:

```
–nomachine
```

3.3.7 optimize

The syntax for this option is:

```
–o=n

–optimize=n         (long name)
–o=5                (default value)
```

The *-optimize* option sets the optimization level, ranging from the integer 0 (minimum optimization) to the integer 5 (maximum optimization). Each optimization level is cumulative: level 5 performs all optimizations made by the previous levels. **Table 3–5** describes optimization levels.

A higher optimization level results in more optimization along with increased analysis time. Many programs written for a parallel processing environment do not need advanced transformations; with these programs, a lower optimization level is enough.

Value	Description
0	Do not mark code for parallel execution.
1	Mark eligible code for parallel execution.
2	Apply *for* loop interchanging techniques and recognize sum reductions as safe for parallel execution. (PCA doesn't mark sum reduction loops for parallelization unless *roundoff=2*.) Use lifetime analysis to determine when the code needs last– value assignment of scalars to make a loop safe to run in parallel. Use more powerful data–dependence tests to find more loops that can run safely in parallel.
3	Recognize linear recurrences as safe for parallel execution. Use loop interchanging, when possible, to improve memory referencing. This level also allows loop interchanging for triangular loops. Use special case data– dependence tests to find more loops that can run safely in parallel. Recognize special index sets, (wrap–around variables) as safe for parallel execution.
4	Split a loop in two, if necessary, to break a data–dependence arc. Use exact data–dependence tests to find more loops that are safe to run in parallel. Enable loop unrolling.

5	Transform two adjacent loops into a single loop. Use data−dependent tests to allow fusion of more loops than possible with standard techniques.

Table 3−5 Optimization Levels

3.3.8 roundoff

The syntax for this option is:

```
-r=n

-roundoff=n          (long name)
-r=0                 (default value)
```

The *-roundoff* option allows control of whether or not PCA runs reductions (for example, the summing of an array of values) in parallel. When a reduction runs serially, all operations occur in the same order, so the roundoff error is the same from one execution of the code to the next. But when a reduction runs in parallel, the separate threads of execution do not do all the operations in the same order as the serial version. Thus, the roundoff error can differ from that of the serial version. Furthermore, the roundoff error of the multiprocess version can vary from one run to the next. Often, roundoff error is not important.

Unfortunately, some algorithms (for example, branching on an exact match) are sensitive to even small differences in roundoff error. If your code is sensitive to roundoff error, you can tell PCA not to allow reductions in the code it converts to run in parallel. This guarantees that the results of the multiprocess code is always the same as the serial version. In fact, that is the reason that the default value of *roundoff* is 0 (no arithmetic reductions).

Each *-roundoff* level is cumulative (level 3 performs everything up to and including this level). **Table 3−6** describes the roundoff levels.

Value	Description
0	Do not convert reductions to run in parallel (the default). In particular, PCA does not convert arithmetic recurrences and arithmetic reductions (such as SUM and PRODUCT) to run in parallel. PCA can still convert nonarithmetic reductions to run in parallel (such as MAX of a vector).
1	Allow PCA to simplify expressions with operands that are between binary and unary operators. Allow expression simplification due to forward substitution. Allow code floating, if the *scalaropt* switch is ³ 1. The same as *0* for reductions.
2	Allow PCA to mark reductions to run concurrently. Allow loop interchanging around arithmetic reductions. Perform concurrent reductions with pre−scheduled concurrent loops and local accumulation of reduction results. Thus, the answers can vary from one execution to the next.
3	Recognize real (float) induction variables. Enable memory management if *scalaropt=3*.

Table 3−6 *roundoff* Levels

3.3.9 unroll and unroll2

The syntax is:

```
-ur=n
-unroll=n             (long name)
-ur=4                 (default value)
-ur2=n-unroll2=n      (long name)
-ur2=100              (default value)
```

The *-unroll* and *-unroll2* options control how PCA unrolls scalar inner loops. In most cases, when PCA cannot convert loops to execute concurrently, PCA can unroll the loop to improve performance. (More work per iteration with fewer iterations gives less overhead.) Set *-optimize=4* to enable the *-unroll* and *-unroll2* options. **Table 3−7** describes *unroll* values.

Value	Description
0	Use the default values to unroll.
1	Do no unrolling.
n>=2	At most, unroll *n* iterations.

Table 3–7 *unroll* Values

For example, the default (4,100) means at most four iterations, and a maximum work per unrolled iteration of 100.

You can control unrolling in two ways. The first is to use the number of iterations, and the second is to use the "work per unrolled iteration" factor. To use the "work per unrolled iteration" factor, PCA analyzes a given loop by computing an estimate of the computational work that is inside the loop for ONE iteration. This rough estimate is based on the following criteria:

number of assignments +

number of if statements +

number of subscripts +

number of arithmetic operations

The following example assumes *unroll=8* and *unroll2=100*.

```
int a[], b[], n;
void example_4_3_9 ()
{
      int i;
      for (i=0; i<n; i++)
          a[i] = b[i] / a[i-1];
}
```

This example has:

1 assignment

0 ifs

3 subscripts

1 arithmetic operator

––––––––––––––––––––––––––

5 is the weighted sum (the work for 1 iteration)

PCA then divides this into 100 to give an unroll factor of 20. But eight was specified for the maximum number of unrolled iterations. PCA takes the minimum of the two values (8) and unrolls that many iterations. The maximum number of iterations that PCA can unroll is 100. If you request more than that number, PCA gives no warning of its inability to comply.

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

For example:

```
int a[];
int b[];
int n;
void example_4_3_9(  )
{
      int i;
      int _Kii1;
      _Kii1 = (n)%(8);
      for ( i = 0; i<_Kii1; i++ ) {
          a[i] = b[i] / a[i-1];
      }
      for ( i = _Kii1; i<n; i+=8 ) {
          a[i] = b[i] / a[i-1];
          a[i+1] = b[i+1] / a[i];
          a[i+2] = b[i+2] / a[i+1];
          a[i+3] = b[i+3] / a[i+2];
```

```
            a[i+4] = b[i+4] / a[i+3];
            a[i+5] = b[i+5] / a[i+4];
            a[i+6] = b[i+6] / a[i+5];
            a[i+7] = b[i+7] / a[i+6];
        }
}
```

3.4 Input−Output Options

The following sections explain the function of each option that affects PCA's input−output file selection.

3.4.1 cmp

The syntax for this option is:

```
-[n]cmp=file

-[no]cmp=file              (long name)
standard output            (the default)
```

The *-cmp* (compile file) option tells PCA to write the optimized C program to a file. If you specify *-cmp=file* , PCA writes the transformed C to the specified file. The default file for the transformed code is *standard output*.

If you use *-cmp* without a file name, PCA writes the transformed code to *file.M*, where *file* is the input file name from the command line with the trailing *.c* (if any) stripped off. (See the following description of the *-input* option for a special case.)

To tell PCA not to generate a C output file, enter

`-nocmp`

on the command line.

3.4.2 input

The syntax for this option is:

```
-i=file

-input=file     (long name)
no default
```

Usually, you will simply include the input file name on the command line. The *-input=file* option is an alternative way of specifying the input file.

Specifying *-input* without a file name tells PCA to read the source file from *standard input*. Then PCA writes the transformed code and (optional) listing file to *standard output* unless you use the *-cmp* and *-list* options to give explicit file names.

3.4.3 list

The syntax for this option is:

```
-[n]l=file

-[no]list=file    (long name)
-nolist           (the default)
```

The *-list* option tells PCA where to write the listing you request when you use the *-listoptions* option. If you specify *-list=file*, PCA writes the listing to the specified *file*. To explicitly disable generation of the listing file, enter

`-nolist on`

on the command line.

If you specify *-list* without a file name, PCA writes the listing file to *file.L*, where *file* is the input file name with the trailing *.c* (if any) stripped off. (See the previous description of the *-input* option for a special case.)

If you do not use the *-list* option, but do use *-listoptions=list*, PCA writes the listing file to *standard output*. PCA writes all diagnostic messages, syntax errors, and so forth, to *standard error*.

3.5 Listing Options

The following sections explain the function of each listing option. You must use these options in conjunction with the *-list* option.

3.5.1 listingwidth

This option sets the maximum line length for the listing file produced by PCA. The syntax for this option is:

```
-lw=[132|80]

-listingwidth=[132|80]   (long name)
-lw=80                   (the default)
```

The line length affects the format of the loops summary table (produced by *-lo=l*) and the PCA options table (*-lo=k*). The default line length is 80, convenient for use on most terminals. The 132 column width is optimal for most line printers. No other values are allowed at present.

3.5.2 cmpoptions

The syntax for this option is:

```
-[n]cp=i

-[no]cmpoptions=i      (long version)
-ncp                   (default value)
```

The *cmpoptions* flag specifies additional information for inclusion in the transformed (*.cmp*) file. PCA currently supports only the *i* value for cmpoptions, which directs PCA to include special line−number directives.

Special line numbers are *# line* directives which can appear in the transformed program file to reference line numbers of the original source code. The line in the transformed code immediately following a "# line" comment is either the transformed version of the referenced line, or a line inserted by PCA just before the referenced line. PCA includes the name of the source file in the form it appeared in on the command line.

In the unrolled loop below, the for in the original source code was on line 7, and the assignment on line 8:

```
# line 7 "../csource/unr5.c"
     for ( i = il + 1; i<=n; i+=3) {
        a[i] = b[i] / a[i−1]
# line 8 "../csource/unr5.c"
        a[i+1] = b[i+1] / a[i];
# line 8 "../csource/unr5.c"
        a[i+2] = b[i+2] / a[i+1];
# line 8 "../csource/unr5.c"
     }
```

3.5.3 lines

The syntax for this option is:

```
-ln=n

-lines=n              (long name)
-ln=55                (default value)
```

The *-lines* option tells PCA to paginate the listing file for printing. Use the *-lines* option to change the number of lines printed per page. The *-lines=0* option tells PCA to paginate only at subroutine boundaries.

3.5.4 listoptions

The syntax for this option is:

```
-lo=list

-listoptions=list      (long name)
no listing             (the default)
```

The *-listoptions* option tells PCA what information to include in the listing file.

Table 3–8 describes the *-listoptions* values.

Value	Description
c	Print the Calling Tree of the entire program.
i	Insert line numbers into transformed code referencing line numbers of the original.
k	Print PCA options used at the end of the listing.
l	Print the loop–by–loop optimization table.
n	Print program unit names, as processed, in the error file.
p	Print the analysis performance statistics.
s	Summarize loop optimizations.

Table 3–8 *listoptions* Values

The transformed code is always recorded in the transformed code file, whether or not you request a listing file.

3.6 Memory Management Options

These options set parameters which PCA uses to optimize memory hierarchy usage. You can obtain better optimization of memory reference patterns if you know how much data can be kept in fast memory, such as cache or arithmetic registers, and the costs of moving data in the memory hierarchy. To enable memory management, you must set *-scalaropt=3* and *-roundoff=3*.

3.6.1 cacheline

The syntax of this option is:

```
-chl=n

-cacheline=n         (long version)
-chl=16              (default value)
```

Use the *cacheline* option to inform PCA of the width in bytes of the memory channel between cache and main memory.

3.6.2 cachesize

The syntax of this option is:

```
-chs=n
```

```
-cachesize=n        (long version)
-chs=64             (default value)
```

Use the *cachesize* option to inform PCA of the size in kilobytes of the cache memory.

3.6.3 dpregisters

The syntax of this option is:

```
-dpr=n

-dpregisters=n       (long version)
-dpr=6               (default value)
```

The *dpregisters* option specifies the number of double–precision floating point registers each processor has.

3.6.4 spregisters

The syntax of this option is:

```
-spr=n

-spregisters=n      (long version)
-spr=12             (default value)
```

The *spregisters* option specifies the number of single–precision floating point registers each processor has.

3.6.5 setassociativity

The syntax of this option is:

```
-sasc=n

-setassociativity=n  (long version)
-sasc=1              (default value)
```

The *setassociativity* option provides information on the mapping of physical addresses in main memory to cache pages. The default, *1*, specifies that a datum in main memory can be placed in only one place in cache. If this cache page is in use, its current contents must be dropped in order to copy the new page into cache.

3.7 Invariant IF Floating Options

You can use two options to control how much code expansion PCA will allow when expanding invariant–IF loops. The options are *each_invariant_if_growth* and *max_invariant_if_growth*. Use these options to control the code growth of a program unit, that is, a subroutine, function, or main procedure. Each option has a product–specific default.

The syntax of these options is given in **Table 3–9** .

Long Form	Short Form	Valid Range	Default Value
each_invariant_if_growth=	eiifg=	0-100	50
max_invariant_if_growth=	miifg=	0-1000	500

Table 3–9 Invariant–IF Options

```
for (i= ...) {
     section–1
     if ()
        section–2
```

```
     else
        section-3
     section-4
}
```

The *each_invariant_if_growth* option controls the allowed sizes of sections 1 and 4, where size is the number of user−visible executable statements. If sections 1 and 4 are smaller than the value of *each_invariant_if_growth*, then the invariant IF will be floated as shown below:

```
if () then
     for (i= ...) {
        section-1
        section-2
        section-4
     }
else
     for (i= ...) {
        section-1
        section-3
        section-4
     }
```

The *max_invariant_if_growth* option sets a threshhold that acts as a regulatory mechanism for the invariant−IF transformation. Whenever code growth (measured in user−visible executable statements) in a program unit has exceeded this threshhold, PCA will only perform invariant−IF floating in that program unit if there is no code replication. In the example above, no code replication would be necessary in the original loop nest if sections 1 and 4 were absent.

3.8 Command Line Options for Portability

These options are provided for the sake of easy portability among compilers. Note that there are currently no short versions of these options names.

3.8.1 DOLLAR, (no short name), (off)

The *dollar* command line option allows dollar signs to be used as identifiers under both ANSI C and Kernighan and Ritchie C.

For example, the following program will work correctly under either ANSI mode or Kernighan and Ritchie mode if the *dollar* option is enabled.

```
int $i=121961;
main(){
printf("$i is %d.\\n",$i);
}
```

3.8.2 FLOAT, (no short name), (off)

Under Kernighan and Ritchie C, all variables declared as type *float* are promoted to type *double* before arithmetic operations are performed on them.

The *float* option prevents this promotion to *double*, that is, all variables declared as type *float* remain type *float*.

This option is ignored under ANSI C, since the default behavior of ANSI C treats *float* variables as *float* with no promotion to *double*.

3.8.3 SIGNED, (no short name), (off)

By default, a variable declared as type *char* is interpreted as an *unsigned char*. The *signed* option causes variables declared as type *char* to be interpreted as type *signed char*.

This option is sometimes necessary when porting code from other platforms whose C compiler defaults *char* to *signed char*.

3.8.4 VOLATILE, (no short name), (off)

The volatile option indicates that all variables are implicitly volatile.

Use of this option severely limits the optimization that can be done.

3.8.5 PROCESSORS, P, P=0

The *kap* option optimizes for an unknown number of processors.

Certain of the concurrency optimizations require knowing the number of processors that are available. If this number is known at compile time, the generated code is more efficient.

If *integer* is 1, *kap* turns off concurrency.

3.8.6 INLINE_AND_COPY, INLC, (off)

The *inline_and_copy* option functions like the *inline* option except that if all CALLs or references to a subprogram are inlined, the text of the routine is not optimized but is rather copied unchanged to the transformed code file. This option is intended for use when inlining routines from the same file as the call and has no special effect when the routines being inlined are taken from a library or another source file.

After a subprogram has been inlined everywhere it is used, leaving it unoptimized saves compilation time. When a program involves multiple source files, the unoptimized routine will still be available in case one of the other source files contains a reference to it, so no errors will result.

Note: The *inline_and_copy* algorithm assumes that all CALLs and references to the routine precede it in the source file.

If the routine is referenced after the text of the routine and that particular call site cannot be inlined, the unoptimized version of the routine will be invoked.

3.8.7 STDIO, STDIO, (off)

The *stdio* qualifier instructs *kap* to perform *strength* reduction on calls to certain functions in the standard I/O library.

Programs which use functions such as *printf* heavily will generally have improved I/O performance when this is done.

The *−scalaropt=3* option is required to enable this transformation.

3.9 Summary

This chapter described the details of the *pca* command−line options and explained how to use PCA as a standalone analyzer to mark code to run on multiple processors. The next four chapters present additional ways of obtaining concurrentized code. These chapters describe:

- PCA directives that you can insert into the code
- Compiler directives that the multiprocessing C compiler recognizes
- PCA transformations that optimize concurrentization of a loop
- In−lining and interprocedural analysis that streamline function calls

---

Chapter 4

Power C Analyzer Directives

You can use directives to provide additional information about a program that PCA cannot derive from its analysis of the program. Although you can use PCA without directives, they improve the optimization results. Directives provide information only. However, PCA notes the information in a directive and takes that information into consideration when trying to identify data dependencies. **Table 4–1** lists PCA directives and their durations.

PCA Directive	Duration
#pragma serial	next loop
#pragma concurrent	next loop
#pragma concurrent call	next loop
#pragma set chunksize (*n*)	next loop
#pragma set numthreads (n)	next loop
#pragma set schedtype (type)	next loop
#pragma no side effects (name[name...])	program unit
#pragma distinct (name,name[name...])	program unit
#pragma arl(*n)*	selectable
#pragma inline [here][routine][global] [(name[name..])]	selectable
#pragma ipa [here][routine][global] [(name[name..])]	selectable
#pragma padding (variable list)	program unit
#pragma storage order (variable list)	program unit

Table 4–1 PCA Directives

To understand how PCA interprets directives, first consider "assumed" data dependences. For example, consider the loop:

```
for (i=0; i<n; i++) X[i] = X[i-1] + X[m];
```

In this loop, *X* is an array, *n* and *m* are scalars, and nothing is known about the relationship between *n* and *m*. Two types of data dependencies occur. Between *X[i]* and *X[i–1]* there is a forward dependence, and the distance is known to be 1. Between *X[i]* and *X[m]*, PCA tries to find a relation but cannot, because it does not know the value of *m* in relation to *n*. The second dependence is called an assumed dependence, because it is assumed to exist but cannot be proven to exist.

If you know that the assumed data dependency was incorrect, you can tell PCA so by using a directive. If no definite data dependencies exist, PCA can convert the loop to run in parallel.

Use caution when using a directive because PCA cannot check the truth of an assertion implied by the directive. If you make an untrue assertion, PCA may run a data–dependent loop in parallel. This situation is very dangerous, because such code can intermittently produce the wrong answer.

The following sections describe each of these directives.

4.1 #pragma serial

This directive forces the loop immediately following it to be serial, and restricts optimization by forcing all enclosing loops to also be serial. The syntax for this directive is:

```
#pragma serial
```

PCA can still optimize loops that are inside the serial loop, but not enclosing the serial loop. Consider the code:

```
    for (i=0; i<N; i++)
      for (j=0; j<N; j++) {
#pragma serial
        for (k=0; k<N; k++)
           x[i][j][k] = x[i][j][k] * y[i][j];
        for (k=0; k<N; k++)
           x[i][j][k] = x[i][j][k] + z[i][k];
       }
```

The directive forces the *i* and *j* loops, and the first *k* loop to be serial. PCA can still optimize the second *k* loop, but it does not distribute (interchange) the *i* or *j* loops to try to get an optimizable loop. PCA always honors the *#pragma serial* directive. This directive is in effect only for the next loop.

4.2 #pragma concurrent

Use the *#pragma concurrent* directive to tell PCA to ignore assumed dependences in the following loop. The syntax for this directive is:

```
#pragma concurrent
```

If the loop contains definite dependencies in addition to the assumed dependencies, PCA does not convert the loop to run in parallel. In this case, the example on the previous example would be left serial, because it has a known dependence.

Note: PCA does not generate code that executes in parallel (concurrently) if you use the *-noconcurrentize* command line option.

This directive is in effect only for the next loop.

4.3 #pragma concurrent call

Use the *#pragma concurrent call* directive to tell PCA that the function calls in the following loop are safe to execute in parallel.

The syntax for this directive is:

```
#pragma concurrent call
```

PCA ignores all potential data dependences due to the function argument(s). This directive applies only to the immediately following loop and not to any nested or surrounding loops. Put a *#pragma concurrent call* directive before each concurrentizable loop with function references. Be sure that the functions called do not introduce data dependencies.

A better way to concurrentize a loop with function calls is to use either *#pragma no side effects* or interprocedural analysis. IPA directs PCA to determine the true data dependencies and not rely on user assessment. IPA is explained in **Chapter 7, "In-lining and Interprocedural Analysis."**

4.4 #pragma set chunksize, #pragma set numthreads, and #pragma set schedtype

These pragmas tell PCA which values to use for *chunksize*, *numthreads*, and *schedtype*.

The syntax for each of these directives is:

```
#pragma set chunksize (n)
#pragma set numthreads (n)
#pragma set schedtype (type)
```

For *chunksize*, the range of values for n is 1 to 1,000,000. For *numthreads*, the range of values for *n* is 1 to 255. If PCA sees values larger than these, it will assume the maximum and generate a warning message. If PCA sees values smaller that 1, it will generate a warning message and ignore the pragma.

The *schedtype* *types* are:

- simple
- dynamic
- interleave
- gss
- runtime

Refer to **Section 5.2.1, "#pragma parallel"** for a complete description of *num-threads*, and **Section 5.2.2, "#pragma pfor"** for descriptions of *chunksize* and *schedtype*.

4.5 #pragma no side effects

C functions frequently produce more information than just the returned value. Changing values of arguments via pointers or arrays, changing global data, and I/O can make a function unsafe to run concurrently.

The *#pragma no side effects* directive tells PCA to assume that all of the named functions are safe to execute concurrently. This means that the functions perform no I/O and that they modify only local variables.

The syntax for this directive is:

```
#pragma no side effects ( name [,name...] )
```

If you pass pointers or array names to the function and use this directive, PCA assumes that the memory locations they represent are not modified. The functions named must be declared before the directive.

4.6 #pragma arl

Use *#pragma arl* (address resolution level) to control the assumptions PCA makes about memory aliases. The syntax for this directive is:

```
#pragma arl(n)
```

where n is the level of control. **Table 4−2** describes the levels of control.

Value	Description
0	Make no assumptions about memory aliases.
1	Assume there are no pointer self−references (the default).
2	Assume function arguments are distinct from each other.
3	Assume local pointer/arrays are distinct from global pointers/arrays.
4	Assume all pointers/arrays are distinct from each other.

Table 4−2 Address Resolution Levels, *#pragma arl*

The directive has the same meaning as the *-arl* command−line option. See **Chapter 3, "PCA Command−Line Options"** for more information on this option and the levels of control.

When this directive appears inside a function (between the outer { and } of a function definition), it applies only to that function. If the directive appears outside a function, it sets the default value to be used for all functions that follow.

The command−line option is equivalent to a pragma at the beginning of the file and is thus overridden by other *#pragma arl* directives in the file.

4.7 #pragma distinct

Use *#pragma distinct* to indicate that two objects do not overlap.

The syntax for this directive is:

```
#pragma distinct (expr1,expr2[,expr3,expr4...])
```

where

expr1, *expr2*... represent objects.

The form of the expressions allowed is:

id	a variable
**id*	what a pointer variable points to
id []	the array whose name is *id*

All variables involved must be previously declared. For example, for pointer *p* and array *a*, you can assert:

```
#pragma distinct (*p, a[])
```

if **p* never overlaps with *a[i]* for any *i* used in the program.

The range of the directive is the function where it was made and all succeeding functions. If the assertion is made about local variables or parameters, it will have no effect beyond the immediate function. These variables cannot be used outside the immediate function.

4.8 #pragma inline and #pragma ipa

Use the *inline* and *ipa* directives to select manually which function(s) to in−line or perform interprocedural analysis on and at which call sites. The syntax is:

```
#pragma [no]inline [here][routine][global] [( name[,name...])]
#pragma [no]ipa [here][routine][global] [( name[,name...])]
```

If either of these directives appears with a name list, all occurrences of the named functions will be in−lined/analyzed, if possible, in all references within the scope of the directive. If the directive appears without a list of functions, all function references are eligible. (See **Chapter 7, "In−lining and Interprocedural Analysis"** for more information about these pragmas.)

The *no* forms turn off in−lining and IPA of the named function(s). The *scope* keywords are interpreted as:

here	applies only to the next statement
routine	applies to the rest of the program unit
global	applies to the rest of the input file

You can terminate the *routine* and *global* scopes by the corresponding *no* directives. (Or terminate a *noinline* directive with an appropriate *inline* directive.)

These pragmas can override the *-inline*, *-ipa*, *-inline_looplevel*, and *-ipa_looplevel* command−line options. You can use *#pragma inline* and *#pragma ipa* in addition to, or in place of, command−line controlled in−lining/interprocedural analysis.

Note: The *inline_man* or *ipa_man* command−line option must be specified for the corresponding directive to be enabled (see **Chapter 7, "In−lining and Interprocedural Analysis"** for more information).

4.9 Memory Management pragmas

PCA supports two *memory management* directives, *#pragma padding* and *#pragma storage order*. PCA uses these output directives to pass information on data layout to the compiler or to itself (if you are using PCA to process a program interactively). If PCA processes a program more than once, it will use the information in the directives inserted in previous runs to direct its cache usage optimizations.

4.9.1 #pragma padding

Use the *padding* directive to identify the listed arrays and scalar variables as objects which PCA created for the purpose of data alignment. PCA uses this directive when it reprocesses a program; the compiler will ignore this directive. The syntax of *#pragma padding* is:

```
#pragma padding (variable1 [, variable2 ...])
```

The following rules govern the use of the *padding* directive.

- You can use more than one *padding* directive within a single program unit.
- The *padding* directive will be placed immediately after the declarations section of the program unit (the *main* function or called function).
- A *padding* object can be routine−local or external.
- A *padding* object can not be a dummy argument to the procedure or function.

4.9.2 #pragma storage order

The *storage order* directive specifies the relative order in which storage should be allocated for the listed routine–local variables and arrays. PCA can reduce cache collisions by positioning the arrays correctly. The C compiler currently ignores the *storage order* directive.

The syntax of *#pragma storage order* is:

```
#pragma storage order (variable1 [, variable2 …])
```

The rules governing the use of *#pragma storage order* are:
- You can use more than one *storage order* directive per program unit. Each directive can be interpreted separately.
- The *storage order* directives will be placed directly after the declaration section of the program unit.
- An object listed in a *storage order* must be local to the program unit.
- An object listed in a *storage order* must not be:
  - mentioned in another *storage order* directive
  - an external variable or array
  - a dummy argument to the procedure or function

PCA can generate as many *#pragma storage order* directives as it considers useful.

To interpret a storage order directive, the compiler must place the named objects in memory in the order listed. For example:

```
     float a1[100], a2[3], a3[200]
#pragma storage order (a1,a2,a3)
```

On a machine with 4 bytes per float variable, the compiler would place the variables as follows:
- *a1* would be placed at some address *X*.
- *a2* would be placed at *X* + 100*4.
- *a3* would be placed at *X* + 100*4 + 3*4

Note that both static and automatic storage schemes are allowed, so as long as all of the objects in a single *storage order* are placed in the same scheme.

The padding and storage order directives often appear together, as in the following example.

```
     double _Kdd13[770];
     double _Kdd14[770];
#pragma padding(_Kdd14, _Kdd13)
#pragma storage order(c, _Kdd13, b, _Kdd14, a)
```

4.10 Parallelizing Loops that Deal with Linked Lists

When dealing with elements of linked lists, PCA allows you to parallelize:
- loops in which each iteration processes a different member of the list and the computations for each element are independent of each other, that is, they can be computed in any order.
- loops in which each iteration processes a different member of the list and the computations for each element are to a large extent independent of each other, but have a small portion of the code which has to be processed in the order in which the elements appear in the list.

Two pragmas support these two cases: *#pragma plist*, and *#pragma ordered*.

4.10.1 #pragma plist

Syntax:

```
#pragma plist unordered (list vars.; initialize shared;  initialize
local; condition; increment) for (initialize shared, initialize
local; condition; increment)
{
     ... /* loop body */
}
```

4.10.2 #pragma ordered

Syntax:

```
#pragma plist ordered (list vars.; initialize shared;   initialize
local; condition; increment) for (initialize shared, initialize
local; condition; increment)
{
     ... /* unordered loop body 1 */
#pragma ordered
     {
        ... /* ordered loop body */
     }
     ... /* unordered loop body 2 */
}
```

Note: For both the above cases, the increment operation can be performed anywhere without any side effect other than that of modification of the loop variable.

The following example shows an unordered loop, which uses *#pragma plist* unordered, and an ordered loop, which uses *#pragma ordered*.

```
#define N 100 #define LOOP 10
#define ERROR 5
typedef struct st_1 *sptr;
struct st_1 {
     sptr next;
     int data;
};
sptr head;
main ()
{
     sptr list = 0;
     double sum2;
     int cnt;
     int i, j, k, t;
     double sum1;
     double psum;
     int pcnt;
     int error = 0;
     head = (sptr) malloc (sizeof (struct st_1));
     head->data = N;
     for (list = head, i = 1; i < N; i++) {
        list->next = (sptr) malloc (sizeof (struct st_1));
        list = list->next;
        list->data = (N - i);
     }
     list->next = 0;
     sum1 = 0;
     for (list = head, i = 0; list; i++, list = list->next) {
        if (list->data != N - i) {
           printf ("Mismatch: i = %d, data = %d\n", i, list->data);
           break;
        }
        sum1 += list->data;
     }
     printf ("SUM1 = %le\n", sum1);
     for (i = 0; (i < LOOP) && (error < ERROR); i++) {
        sum2 = 0;
        cnt = 0;
```

```
#pragma parallel shared (list, head, sum2, cnt) local (psum, pcnt, t)
        {
#pragma plist ordered (list; list=head, sum2 = 0, cnt = 0; psum = 0, pcnt =
 0; list; list = list->next;)
           for (list= head, sum2 = 0, cnt = 0, psum = 0, pcnt = 0;
               list;
               list = list->next)
           {
#pragma ordered
              {
                 pcnt++;
                 psum += list->data;
              }
           }
#pragma critical
           {
              sum2 += psum;
              cnt += pcnt;
              printf ("sum2 = %le, psum = %le, pcnt = %d\n",
                       sum2, psum, pcnt);
           }
        }
        if (sum2 != sum1) {
           error++;
           printf ("ERROR: i = %d, count = %d, SUM2 = %le\n",i,
                   cnt, sum2);
        }
        printf ("\n");
     }
}
```

Chapter 5

Multiprocessing C Compiler Directives

In addition to the usual interpretation performed by any other C compiler, the multiprocessing C compiler can process explicit multiprocessing directives to produce code that can run concurrently on multiple processors. **Table 5–1** lists the multiprocessing directives used when processing code in parallel regions. The multiprocessing C compiler does not know whether you or PCA (or a combination of the two) put the directives in the code. The multiprocessing C compiler does not check for or warn against data dependencies that have been violated. That kind of analysis is left to PCA.

Pragma	Description
#pragma parallel	Start a parallel region
#pragma pfor	Mark a *for* loop to run in parallel
#pragma one processor	Execute statement on only one processor
#pragma critical	Protect access to critical statement(s)
#pragma independent	Start independent code section that executes in parallel with other code in the parallel region
#pragma synchronize	Stop threads until all threads reach here
#pragma enter gate	Note threads that have reached here
#pragma exit gate	Stop threads until all threads have passed the matching *#pragma enter gate*
#pragma plist	
#pragma ordered	

Table 5–1 Multiprocessing C Compiler Directives

After the multiprocessing directives are inserted (either by the multiprocessing C compiler or by you), you can pass the code through PCA. The directives and their associated code will remain unchanged and pass directly through to the *.out* file, but unrelated sections of code will be optimized

5.1 Why Use Parallel Regions?

To understand many of the multiprocessing C compiler directives, consider the concept of a parallel region. On some systems, a parallel region is merely a single loop that runs in parallel. However, with Power C, a parallel region can include several loops and/or independent code segments that execute in parallel.

Using large parallel regions can improve the performance of your code in ways not possible merely by executing a series of isolated loops in parallel. For example, parallel regions save some of the processing overhead associated with preparing each region to run in parallel. In addition, parallel regions do not force synchronization at the end of each of the contained loops.

Thus, if a thread finishes its work early, it can go on to execute the next section of code—providing that the next section of code is not dependent on the completion of the previous section. However, when creating parallel regions, you need more sophisticated synchronization methods than you need for isolated parallel loops.

5.1.1 New Multiprocessing Compiler Directives

PCA does not recognize or generate directives that were only recently added to the multiprocessing C compiler. If PCA finds one of these new multiprocessing C compiler directives in your code, it prints a warning message and discards it. This guide clearly notes the new directives that are not processed by PCA. In future releases, PCA will recognize (and where appropriate, generate) these new directives. Thus, you should feel free to use these new directives in your code, but add them only after you have finished with PCA.

5.1.2 Coding Rules of Pragmas

Power C pragmas are modeled after the Parallel Computing Forum (PCF) directives for parallel FORTRAN. The PCF directives define a broad range of parallel execution modes and provide a framework for defining C pragmas.

Some changes have been made to make the pragmas more C−like:

- Each pragma starts with *#pragma* and follows the conventions of ANSI−C for compiler directives. You may use white space before and after the *#*, and you must sometimes use white space to separate the words in a pragma, as with C syntax. A line that contains a pragma can contain nothing else (code or comments).
- Pragmas apply to only one succeeding statement. If a pragma applies to more than one statement, you must make a compound statement. C syntax lets you use curly braces, { }, to do this. Because of the differences between C syntax and FORTRAN, C can omit the PCF directives that indicate the end of a range (for example, *END PSECTIONS*).
- If you put a variable on a *local* list, it is as if you declared a variable of the same type and name inside the parallel statement.
- The *pfor* pragma replaces the *PARALLEL DO* directive because the *for* statement in C is more loosely defined than the *FORTRAN DO* statement.

To make it easier to use pragmas, you can put several keywords on a single pragma line, or spread the keywords over several lines. In either case, you must put the key words in the correct order, and each pragma must contain an initial keyword.

For example:

```
#pragma parallel shared(a,b,c, n) local(i) pfor
#pragma iterate(i=0;n;1)
for (i=0; i<n; i++) a[i]=b[i]+c[i];
```

does the same thing as:

```
#pragma parallel
#pragma shared( a )
```

```
#pragma shared( b, c, n )
#pragma local( i )
#pragma pfor
#pragma iterate(i=0;n;1)
     for (i=0; i<n; i++) a[i]=b[i]+c[i];
```

5.2 Parallel Regions

A parallel region consists of a number of work–sharing constructs. Currently, Power C supports the following work–sharing constructs:

- a loop executed in parallel
- an independent code section executed in parallel with the rest of the code in the parallel region
- "local" code run (identically) by all threads
- code executed by only one thread
- code run in "protected mode" by all threads

In addition, Power C supports two types of explicit synchronization:

- synchronize
- enter/exit gate

A simple parallel region consists of only one work–sharing construct, usually a loop. (A parallel region consisting of only a serial section or independent code is a waste of time.)

A parallel region of code can contain sections that execute sequentially as well as sections that execute concurrently. A single large parallel region has a number of advantages over a series of isolated parallel regions: each isolated region executes a single loop in parallel. At the very least, the single large parallel region can help reduce the overhead associated with moving from serial execution to parallel execution.

Large mixed parallel regions also let you avoid the forced synchronization that occurs at the end of each parallel region. The large mixed parallel region also allows you to use pragmas that execute independent code sections that run concurrently.

To start a parallel region, use the *parallel* pragma. To mark a *for* loop to run in parallel, use the *pfor* pragma. To start an independent code section that executes in parallel with the rest of the code in the parallel region, use the *independent* pragma.

Figure 5–1 shows the execution of a typical parallel program with parts running in sequential and parallel mode.

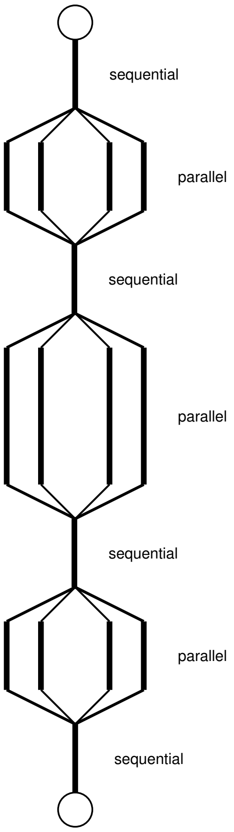


Figure 5–1 Program Execution

When you or PCA start a program, nothing actually runs in parallel until it reaches a parallel region. Then multiple threads begin (or continue, if this isn't the first parallel region), and the program runs in parallel mode. When the program exits a parallel region, only a single thread continues (sequential mode) until the program again enters a parallel region and the process repeats.

The synchronization needs within a simple parallel region are simple; you can use the *critical* or *one processor* pragma to handle them.

The following subsections describe these directives.

5.2.1 #pragma parallel

To start a parallel region, use the *parallel* pragma. This pragma has a number of modifiers, but to run a single loop in parallel, the only modifiers you usually use are *shared*, *byvalue*, and *local*. These options tell the multiprocessing C compiler which variables to share between all threads of execution and which variables should be treated as local.

The code that comprises the parallel region is delimited by curly braces ({ }) and immediately follows the parallel pragma and its modifiers.

The syntax for this pragma is:

```
#pragma parallel shared (variables) byvalue (variables)
#pragma local (variables) optional modifiers
{ code }
```

The *parallel* pragma has six modifiers: *shared*, *byvalue*, *local*, *if*, *ifinline*, and *numthreads*.

Their syntax is:

```
shared ( variable names )
byvalue ( variable   names )
local ( variable   names )
if ( integer   valued   expr )
[no]ifinline
numthreads ( expr )
numthreads (percent=expr)
numthreads (expr)
```

Where:

shared	Tells the multiprocessing C compiler the names of all the variables that the threads must share. (If PCA creates a parallel region, it does this for you.)
byvalue	Puts a variable in the *variable names* list after this option to tell the multiprocessing C compiler that it can pass those shared variables as values rather than by reference. This fine−tuning option helps the multiprocessing C compiler optimize code. PCA will generate this variable as appropriate. However, used incorrectly, this option can generate erroneous code. Be careful what you put in this variable list or you may generate incorrect code. You can put a variable in this list only if the variable is: ● a scalar ● not already in the shared list ● read only
local	Tells the multiprocessing C compiler the names of all the variables that must be private to each thread. (When PCA sets up a parallel region, it does this for you.)
if	Lets you set up a condition that is evaluated at run time to determine whether or not to run the statement(s) serially or in parallel. At compile time, it is not always possible to judge how much work a parallel region does (for example, loop indices are often calculated from data supplied at run time). Avoid running trivial amounts of code in parallel because you cannot make up the overhead associated with running code in parallel. PCA will also generate this condition as appropriate. If the *if* condition is false (equal to zero), then the statement(s) runs serially. Otherwise, the statement(s) run in parallel.
[no]ifinline	Helps the multiprocessing C compiler optimize code when you also use the *if* option. This option is a fine−tuning option. Using the *ifinline* option (which is the default unless you use *noifinline*) causes a slight increase in code size but faster execution. This feature is turned off if you use *noifinline* (that is, the code is smaller but a little slower).
numthreads	(min=*expr*, max=*expr*)
numthreads	(percent=*expr*)
numthreads	(*expr*) Tells the multiprocessing C compiler the number of available threads to use when running this region in parallel. (The default is all the available threads.) The *min* clause instructs the compiler that this section is not to run in parallel unless at

least *expr* threads are available.
The *max* clause indicates that at most *expr* threads out of the available threads should be used. The actual number used is the smaller of *expr* and the number of threads available.
The *percent* clause instructs the compiler to use *expr* percent of the available threads.
In general, you should never have more threads of execution than you have processors, and you should specify *numthreads* with the MPC_NUM_THREADS environmental variable at run time (see **Appendix C, "Run Time Environment Variables"**). If you want to run a loop in parallel while you run some other code, you can use this option to tell the multiprocessing C compiler to use only some of the available threads.
The usage *#pragma numthreads* (*expr*) is equivalent to *#pragma numthreads* (max= *expr*).
expr should evaluate to a positive integer.

For example, to start a parallel region in which to run the following code in parallel:

```
for (idx=n; idx; idx--) {
     a[idx] = b[idx] + c[idx];
}
```

you or PCA must enter:

```
#pragma parallel shared( a, b, c ) byvalue(n) local( idx )
```

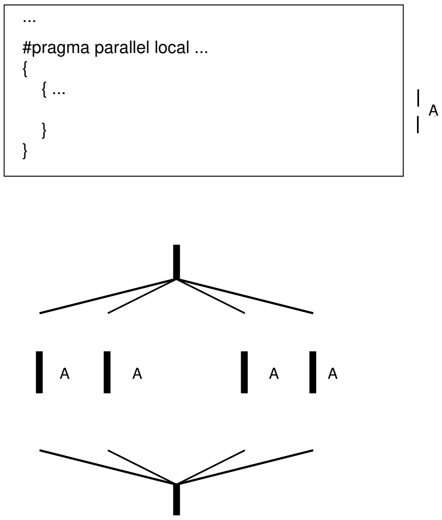
or:

```
#pragma parallel
#pragma shared( a, b, c )
#pragma byvalue(n)
#pragma local(idx)
```

before the statement or compound statement (code in curly braces, { }) that comprises the parallel region.

Any code within a parallel region but not within any of the explicit parallel constructs (pfor, independent, one processor, and critical) is termed local code. Local code typically modifies only local data and is run by all threads.

Figure 5–2 shows local code execution.



```
...
#pragma parallel local ...
{
   { ...

   }
}
```

Figure 5–2 Execution of Local Code Segments

5.2.2 #pragma pfor

Use *#pragma pfor* to run a *for* loop in parallel only if the loop meets all of these conditions:

- All the values of the index variable can be computed independently of the iterations.
- All iterations are independent of each other—that is, data used in one iteration does not depend on data created by another iteration. A quick test for independence: if the loop can be run backwards, then chances are good the iterations are independent.
- The number of iterations is known (no infinite or data–dependent loops) at execution time.
- The *pfor* is contained within a parallel region.

If the code after a *pfor* is not dependent on the calculations made in the *pfor* loop, there is no reason to synchronize the threads of execution before they continue. So, if one thread from the *pfor* finishes early, it can go on to execute the serial code without waiting for the other threads to finish their part of the loop.

The *#pragma pfor* directive takes several modifiers; the only one that is required is *iterate*. **Figure 5–3** shows *#pragma parallel*, which starts a parallel region and tells the multiprocessing C compiler that the i variable must be local (private) to each processor. *#pragma pfor* tells the compiler that each iteration of the loop is unique and to partition the iterations among the threads for execution.

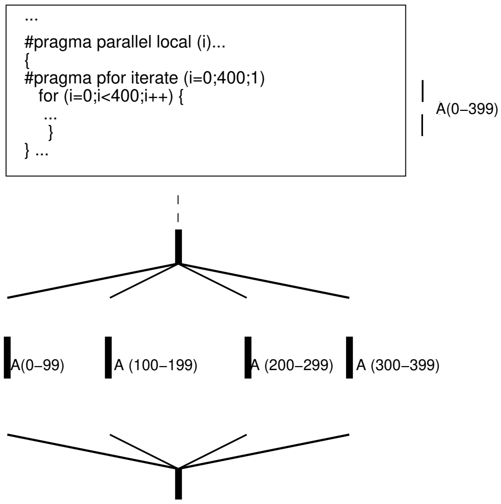
The syntax for *#pragma pfor* is:

```
#pragma pfor iterate ( ) optional modifiers
for ...
    { code ... }
```

The *pfor* pragma has three modifiers. Their syntax is:

```
iterate( index variable=expr1; expr2; expr3 )
schedtype ( type )
chunksize  ( expr )
```

Figure 5–3 shows parallel code segments using *#pragma pfor*.



```
...
#pragma parallel local (i)...
{
#pragma pfor iterate (i=0;400;1)
   for (i=0;i<400;i++) {
     ...
      }
} ...
```

Figure 5–3 Parallel Code Segments Using *#pragma pfor*

Where:

iterate Gives the multiprocessing C compiler the information it needs to identify the unique iterations of the loop and partition them to particular threads of execution.
index variable is the index variable of the *for* loop you want to run in parallel.
expr1 is the starting value for the loop index.
expr2 is the number of iterations for the loop you want to run in parallel.
expr3 is the increment of the *for* loop you want to run in parallel.
For example, for the *for* loop

```
for (idx=n; idx; idx--) {
```

```
for (idx=n; idx; idx--) {
      a[idx] = b[idx] + c[idx];
}
```

the *iterate* modifier to *pfor* should be:

```
iterate(idx=n;n;-1)
```

This loop counts down from the value of *n*, so the starting value is the current value of *n*. The number of trips through the loop is *n*, and the increment is –1.

schedtype (*type*)

Tells the multiprocessing C compiler how to share the loop iterations among the processors. The *schedtype* chosen depends on the type of system you are using and the number of programs executing (see **Table 5–2**).

Single–User System *	**Multiuser System**
simple (iterations take same amount of time)	*gss* (data–sensitive iterations vary slightly)
gss (data–sensitive iterations vary slightly)	*dynamic* (data–sensitive iterations vary greatly)
dynamic (data–sensitive iterations vary greatly)	

* If you are on a single–user system but are executing multiple programs, select the scheduling from the Multiuser column.

Table 5–2 Choosing a schedtype

Figure 5–4 shows how loop iterations can vary.

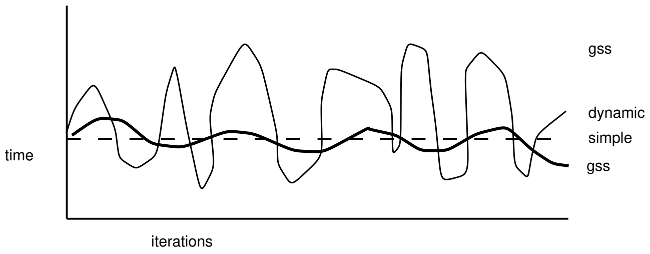


Figure 5–4 Variance of Loop Iterations

You can use the following valid types to modify *schedtype*:

simple	(the default) tells the run time scheduler to partition the iterations evenly among all the available threads.
runtime	tells the compiler that the *real* schedule type will be specified at run time.
dynamic	tells the run time scheduler to give each thread *chunksize* iterations of the loop. *chunksize* should be smaller than (*number of total iterations*)/(*number of threads*). The advantage of *dynamic* over *simple* is that *dynamic* helps distribute the work more evenly than *simple*. Depending on the data, some iterations of a loop can take longer to compute than others, so some threads may finish long before the others. In this situation, if the

	iterations are distributed by *simple*, then the thread waits for the others. But if the iterations are distributed by *dynamic*, the thread does not wait, but goes back to get another *chunksize* iteration until the threads of execution have run all the iterations of the loop.
interleave	tells the run time scheduler to give each thread *chunksize* iterations (described below) of the loop, which are then assigned to the threads in an interleaved way.
gss	(guided self–scheduling) tells the run time scheduler to give each processor a varied number of iterations of the loop. This is like *dynamic*, but instead of a fixed *chunksize*, the chunk size iterations begin with big pieces and end with small pieces. If I iterations remain and P threads are working on them, the piece size is roughly: `I/(2P) + 1` Programs with triangular matrices should use *gss*.

Figure 5–5 shows the effects of the different types of loop scheduling.

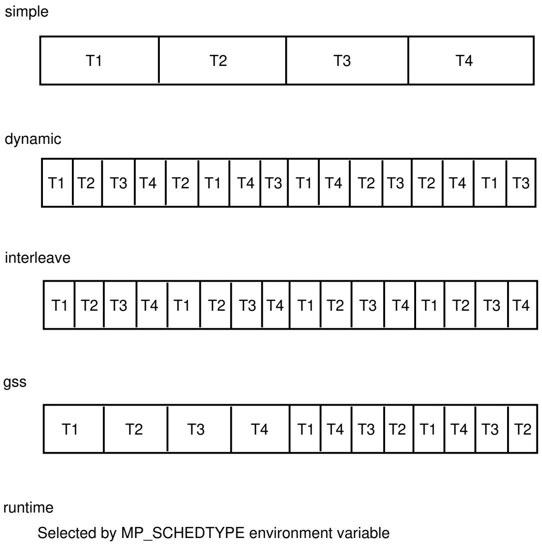


Figure 5–5 Loop Scheduling Types

chunksize (*expr*)

Tells the multiprocessing C compiler how many iterations to define as a chunk when you use the *dynamic* or *interleave* modifier (described above).
expr should be positive integer, and should evaluate to the following formula:

```
number of iterations
--------------------
            X
```

where X^{2} 2* – 10* the number of threads. Select 2* the number of threads when iterations vary slightly and 10* the number of threads when iterations vary greatly.

Performance gain may diminish after 10*.

To run the example:

```
for (idx=n; idx; idx--){
     a[idx] = b[idx] + c[idx];
}
```

in parallel, PCA or you must enter the pragmas:

```
#pragma parallel
#pragma shared( a, b, c )
#pragma byvalue(n)
#pragma local(idx)
#pragma pfor iterate(idx=n;n;-1)
for (idx=n; idx; idx--){
     a[idx] = b[idx] + c[idx];
}
```

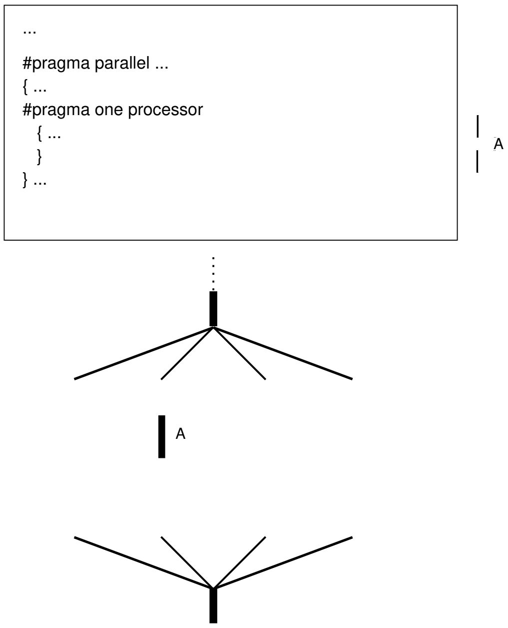
5.2.3 #pragma one processor

A *#pragma one processor* directive causes the statement that follows it to be executed by exactly one thread.

The syntax of this pragma is:

```
#pragma one processor
{ code }
```

Figure 5–6 shows code executed by only one thread. No thread may proceed past this code until it has been executed.



```
...

#pragma parallel ...
{ ...
#pragma one processor
   { ...
   }
} ...
```

Figure 5–6 One Processor Segment

If a thread is executing the statement following this pragma, then other threads that encounter this statement must wait until the statement has been executed by the first thread, then skip the statement and continue on.

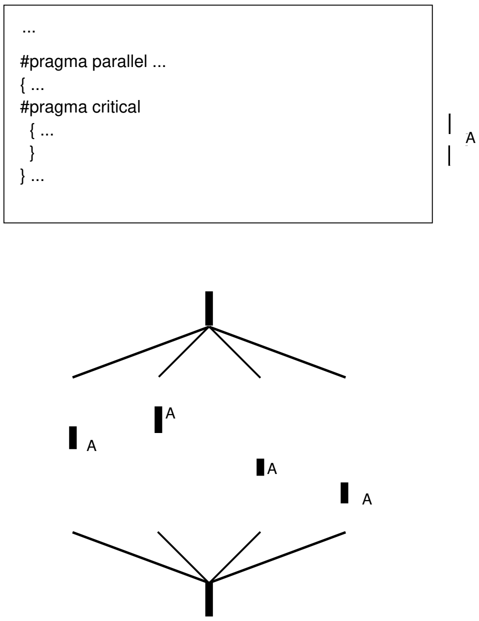
If a thread has completed execution of the statement preceded by this pragma, then all threads encountering this statement skip the statement and continue without pause.

5.2.4 #pragma critical

Sometimes the bulk of the work done by a loop can be done in parallel, but the entire loop cannot run in parallel because of a single data–dependent statement. Often, you can move such a statement out of the parallel because of a single data−dependent statement. Often, you can move such a statement out of the parallel region. When that is not possible, you can sometimes use a lock on the statement to preserve the integrity of the data.

In Power C, use the *critical* pragma to put a lock on a critical statement (or compound statement using { }). When you put a lock on a statement, only one thread at a time can execute that statement. If one thread is already working on a *critical* protected statement, any other thread that wants to execute that statement must wait until the other thread has finished executing it. **Figure 5−7** shows critical segment execution.

Note: The current release of Power C allocates one global lock that is shared among all *#pragma critical* directives by default. Most uses of the *critical* pragma are to protect access to a very limited set of data, data that is usually referenced in many places in the program. By sharing one lock, all references by all guarded statements are properly protected. See **Section 5.4, "The lock Clause"** for more information.



```
...
#pragma parallel ...
{ ...
#pragma critical
  { ...
  }
} ...
```

Figure 5–7 Critical Segment Execution

The syntax of the critical pragma is:

```
#pragma critical
{ code }
```

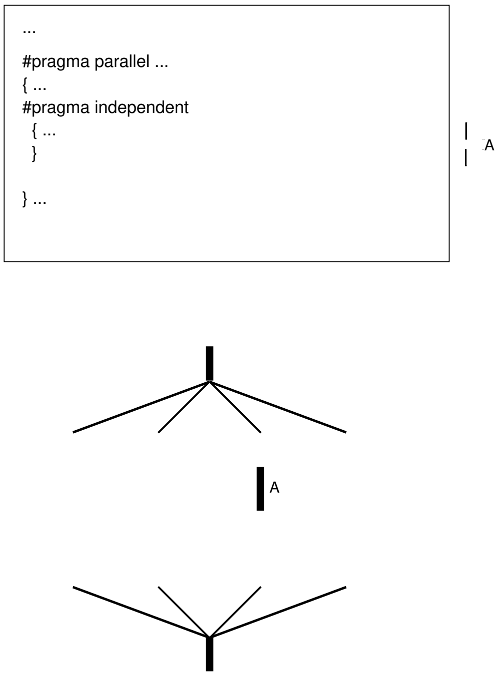
The statement(s) after the critical pragma will be executed by all threads, but only by one at a time.

5.2.5 #pragma independent

Running a loop in parallel is a class of parallelism sometimes called "fine–grained parallelism" or "homogeneous parallelism." It is called homogeneous because all the threads execute the same code on different data. Another class of parallelism is called "coarse−grained parallelism" or "heterogeneous parallelism." As the name suggests, the code in each thread of execution is different.

Ensuring data independence for heterogeneous code executed in parallel is not always as easy as it is for homogeneous code executed in parallel. (And assuring data independence for homogeneous code is not a trivial task.)

The *independent* pragma has no modifiers. Use this pragma to tell the multiprocessing C compiler to run code in parallel with the rest of the code in the parallel region. **Figure 5−8** shows an independent segment with execution by only one thread. However, other threads may proceed past this code as soon as it starts execution.



```
...
#pragma parallel ...
{ ...
#pragma independent
  { ...
  }

} ...
```

Figure 5−8 Independent Segment Execution

The syntax for #pragma independent is:

```
#pragma independent
{ code }
```

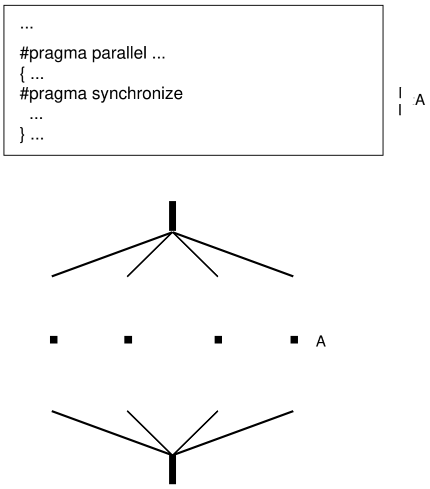
Note: The Power C Analyzer does not yet know how to generate this new pragma. Do not include it in code that you intend to pass through PCA. Insert this pragma only after you are finished with PCA.

5.3 Synchronization

To account for data dependencies, it is sometimes necessary for threads to wait for all other threads to complete executing an earlier section of code. Two sets of directives implement this coordination: *#pragma synchronize* and *#pragma enter/exit gate*.

5.3.1 #pragma synchronize

A *#pragma synchronize* tells the multiprocessing C compiler that within a parallel region, no thread can execute the statements that follows this pragma until all threads have reached it. This directive is a classic barrier construct. **Figure 5–9** shows this synchronization.



```
...
#pragma parallel ...
{ ...
#pragma synchronize
  ...
} ...
```

Figure 5–9 Synchronization

The syntax for this pragma is:

```
#pragma synchronize
```

5.3.2 #pragma enter gate and #pragma exit gate

You can use two additional pragmas to coordinate the processing of code within a parallel region. These additional pragmas work as a matched set. They are *#pragma enter gate* and *#pragma exit gate*.

A gate is a special barrier. No thread may exit the gate until all threads have entered it. **Figure 5–10** shows execution using gates.

Figure 5–10 Execution Using Gates

This construct gives you more flexibility when managing dependencies between the work–sharing constructs within a parallel region.

For example, suppose you have a parallel region consisting of the work–sharing constructs A, B, C, D, E, and so forth. A dependency might exist between B and E such that you could not execute E until all the work on B was completed, as shown below:

```
#pragma parallel ...
{
..A..
..B..
..C..
..D..
..E.. (depends on B)
}
```

One way to handle this would be to put a *synchronize* before E. But this directive is wasteful if all the threads have cleared B and are already in C or D. All the faster threads would pause before E until the slowest thread completed C and D:

```
#pragma parallel ...
{
..A..
..B..
..C..
..D..
#pragma synchronize
..E..
}
```

To reflect this dependency, put a *#pragma enter gate (name*) after B and a *#pragma exit gate* (*name*) before E. Putting the *enter gate* after B tells the system to note which threads have completed the B work–sharing construct. Putting the *exit gate* pragma prior to the E work sharing construct tells the system to allow no thread into E until all threads have cleared B.

```
#pragma parallel ...
{
..A..
..B..
#pragma enter gate (foo)
..C..
..D..
#pragma exit gate (foo)
..E..
}
```

#pragma enter gate

The syntax of this pragma is:

```
#pragma enter gate ( name )
```

name	is a name you create to uniquely identify the work construct controlled by this pragma.

For example, construct D might be dependent on construct A, and construct F might be dependent on construct B, but you would not want to stop at construct D because all the threads had not cleared B. By using *enter/exit gate* pairs, you can make subtle distinctions about which construct is dependent on which other construct.

Put this pragma after the work–sharing construct that all threads must clear before the *#pragma exit* *gate* of the same name.

Note: The Power C Analyzer does not yet know how to generate this new pragma. Do not include it in code that you intend to pass through PCA. Insert this pragma only after you are finished with PCA.

#pragma exit gate

The syntax of this pragma is:

```
#pragma exit gate( name )
```

Put this pragma before the work−sharing construct that is dependent on the *#pragma enter gate* of the same name. No thread enters this work−sharing construct until all threads have cleared the work−sharing construct controlled by the corresponding *#pragma enter gate* of the same name.

Note: The Power C Analyzer does not yet know how to generate this new pragma. Do not include it in code that you intend to pass through PCA. Insert this pragma only after you are finished with PCA.

5.4 The lock Clause

The pragma *lock* clause lets you control the lock to be used during the execution of the various parallel code segments.

The syntax of this pragma is:

```
lock (locktype)
```

where *locktype* can have one of the following values:

block	use a lock exclusively for the block representing this parallel code segment
region	use a lock that is unique to the parallel region
global	use a lock that is unique to the parallel runtimes
others	use a lock that is provided by you. The name, in this case, should correspond to a user−defined variable. It is your responsibility to acquire and dispose of the lock.

For a critical region outside of a parallel region, *region* and *block* are *not* valid lock types. If you don't specify the lock directive, the default values are:

- For a critical segment, a *global* lock is assumed.
- For other segments, a *block* lock is assumed.

Chapter 6

PCA Transformations

Source transformations applied by PCA convert ordinary C code into explicit concurrent syntax. This chapter describes some of the rules and conditions that must be met for a loop to be successfully optimized. This chapter also describes the possible transformations in this process:

- Loop Concurrentization
- Loop Reordering
- Scalar Optimizations
- Loop Rerolling
- Loop Unrolling
- Loop Fusion
- Memory Management for Data Locality

6.1 Loop Concurrentization

The following subs discuss how PCA treats reductions and local variables when it converts the loop to run concurrently (in parallel).

PCA searches the submitted code for *for* loops that are safe to run in parallel and inserts a directive before eligible loops. The simplest cases occur when the statements of the loop have no loop−carried dependence. For example:

```
int a[], b[], c[], d[], n;
void example_7_1 ()
{
      int i;
      for (i=0; i<n; i++) {
          a[i] = b[i] + c[i];
          if (d[i])
              a[i] = a[i] / d[i];
      }
}
becomes:
int a[];
int b[];
int c[];
int d[];
int n;
void example_7_1(  )
{
      int i;
#pragma parallel if(n > 84) byvalue(n) shared(a, b, c, d) local(i)
#pragma pfor iterate(i=0;n;1)
      for ( i = 0; i<n; i++ ) {
          a[i] = b[i] + c[i];
          if (d[i]) {
              a[i] /=  d[i];
              }
      }
}
```

6.1.1 Reductions

Loops that involve reductions can run in parallel only if you synchronize access to the reduced value. For example, if a reduction performed some calculation on the elements of an array and then summed the results, you need to synchronize access to the sum. Only one thread of execution at a time should access the sum. This type of synchronization is perhaps the most common. The following example illustrates how PCA typically transforms a sum reduction to run in parallel. You must set *-roundoff=3* to enable sum reductions. For example:

```
int b[], c[], n, sum;
void example_7_1_1 ()
{
      int i;
      for (i=0; i<n; i++) {
          sum += b[i] + c[i];
      }
}
becomes:
int b[];
int c[];
int n;
int sum;
void example_7_1_1(  )
{
```

```
      int i;
      int sum1;
#pragma parallel if(n > 126) byvalue(n) shared(c, b, sum) local(i)
 reduction(sum1)
      {
          sum1 = 0;
#pragma pfor iterate(i=0;n;1)
          for ( i = 0; i<n; i++ ) {
              sum1 +=  b[i] + c[i];
          }
#pragma critical
          {
              sum +=  sum1;
          }
      }
}
```

To produce the sum, PCA produces a local partial sum (*sum1*) for each thread of execution and then sums these partial sums in a controlled manner.

6.1.2 Local Variables

To generate concurrent code for a loop, PCA sometimes needs to use temporary variables. Each independent thread of execution needs its own set of temporary variables. PCA allocates local variables (temporaries) to the separate threads of execution by putting variable names in the *local* clause of the *#pragma parallel* that starts the parallel region. For example:

```
int a[], b[], c[], d[], n;
void example_7_1_2 ()
{
      int i, t;
      for (i=0; i<n; i++) {
          t = a[i] + b[i];
          c[i] = t * 2.335;
          d[i] = t * 3.221;
      }
}
becomes:
int a[], b[], c[], d[], n;
void example_7_1_2(  )
{
      int i;
      int t;
      int _Kii1;
      _Kii1 = (n)%(4);
      for ( i = 0; i<_Kii1; i++ ) {
          t = a[i] + b[i];
          c[i] = t * 2.335;
          d[i] = t * 3.221;
      }
#pragma parallel if(n>38) byvalue(_Kii1,n) shared(a,b,c,d) local(t,i)
#pragma pfor iterate(i=_Kii1;(n-_Kii1+3)/4;4)
      for ( i = _Kii1; i<n; i+=4 ) {
          t = a[i] + b[i];
          c[i] = t * 2.335;
          d[i] = t * 3.221;
          t = a[i+1] + b[i+1];
          c[i+1] = t * 2.335;
          d[i+1] = t * 3.221;
          t = a[i+2] + b[i+2];
```

```
          c[i+2] = t * 2.335;
          d[i+2] = t * 3.221;
          t = a[i+3] + b[i+3];
          c[i+3] = t * 2.335;
          d[i+3] = t * 3.221;
      }
}
```

6.2 Loop Reordering

Sometimes, PCA cannot convert an outer *for* loop to execute concurrently, but it can convert an inner loop. In these cases, PCA attempts to interchange the loops in the *for* nest, which increases the amount of work that each thread of execution does and improves the efficiency of the code. For example:

```
int a[][1000], b[][1000], n;
void example_7_2 ()
{
      int i, j;
      for (j=0; j<n; j++)
          for (i=0; i<n; i++)
              a[j][i] = a[j-1][i] + b[j][i];
}
becomes:
int a[][1000];
int b[][1000];
int n;
void example_7_2(  )
{
      int i;
      int j;
      int _Kii1;
#pragma parallel byvalue(n) shared(a, b) local(_Kii1, i, j)
#pragma pfor iterate(i=0;n;1)
      for ( i = 0; i<n; i++ ) {
          _Kii1 = (n)%(4);
          for ( j = 0; j<_Kii1; j++ ) {
              a[j][i] = a[j-1][i] + b[j][i];
          }
          for ( j = _Kii1; j<n; j+=4 ) {
              a[j][i] = a[j-1][i] + b[j][i];
              a[j+1][i] = a[j][i] + b[j+1][i];
              a[j+2][i] = a[j+1][i] + b[j+2][i];
              a[j+3][i] = a[j+2][i] + b[j+3][i];
          }
      }
   }
```

6.3 Scalar Optimizations

PCA uses the following standard optimizations to enhance performance of the concurrent code it generates.

- Induction Variable Recognition
- Global Forward Substitution
- Loop Peeling
- Lifetime Analysis
- Invariant *if* Floating

- Derived Assertions
- Dead Code Elimination

6.3.1 Induction Variable Recognition

Induction variables are integers that are incremented or decremented by the same amount for each *for* loop iteration. Auxiliary loop induction variables, such as *j* and *k* shown in the examples that follow, are recognized. For example:

```
int a[], b[], n;
void example_7_3_1 ()
{
      int i, j, k;
      for (i=0; i<n; i++) {
          a[j] = b[k];
          j++;
          k += 2;
      }
}
```

becomes:

```
int a[];
int b[];
int n;
void example_7_3_1(  )
{
      int i;
      int j;
      int k;
      int _Kii2;
      _Kii2 = k;
#pragma parallel if(n > 53) byvalue(n, j, _Kii2) shared(a, b) local(i)
#pragma pfor iterate(i=0;n;1)
      for ( i = 0; i<n; i++ ) {
          a[j+i] = b[_Kii2+i*2];
      }
}
```

6.3.2 Global Forward Substitution

A *global forward* substitution pass finds relationships between variables that do not necessarily depend on the loop index variables. Consider *n* and *npl* in this example:

```
int a[][1000], m, n;
void example_7_3_2 ()
{
      int i, np1;
      np1 = n + 1;
      for (i=0; i<m; i++) {
          a[i][n] = a[i-1][np1];
      }
}
```

PCA determines that *npl* depends only on *n*, and not on the loop index variable. It breaks the apparent data dependence, and makes the loop parallel:

```
int a[][1000];
int m;
int n;
void example_7_3_2(  )
```

```
{
      int i;
      int np1;
   #pragma parallel if(m > 91) byvalue(m, n) shared(a) local(i)
#pragma pfor iterate(i=0;m;1)
      for ( i = 0; i<m; i++ ) {
          a[i][n] = a[i-1][(n+1)];
      }
}
```

6.3.3 Loop Peeling

There are occasions when you will want to use a trick known as a "wrap−around" variable. For example, you might be using an array to simulate a cylindrical coordinate system (where the left edge of the array is adjacent to the right edge). In the following example, the variable *jml* wraps around partway through the loop.

```
nt a[], b[], n;
void example_7_3_3 ()
{
      int j, jm1;
      jm1 = n;
      for (j=0; j<n; j++) {
          b[j] = (a[j] + a[jm1]) / 2;
          jm1 = j;
      }
}
```

In the first iteration, *jm1* is *n*. In all iterations except for *j=1*, the value of *jm1* is *j−1*. Thus, *jm1* is an induction variable for the loop after the first iteration. By peeling off the first iteration of the loop, the*jm1* induction variable can be exploited.

For example:

```
int a[][1000];
int m;
int n;
void example_7_3_2(  )
{
      int i;
      int np1;
   #pragma parallel if(m > 91) byvalue(m, n) shared(a) local(i)
#pragma pfor iterate(i=0;m;1)
      for ( i = 0; i<m; i++ ) {
          a[i][n] = a[i-1][(n+1)];
      }
}
```

PCA can peel off several iterations where multiple wrap−around variables exist.

6.3.4 Lifetime Analysis

In general, when PCA inserts a temporary local variable in a thread of execution, PCA must assign the last value of the temporary variable to the original scalar. When you set *optimize* to 2 or higher (see **Chapter 3, "PCA Command−Line Options"**). PCA does a *lifetime analysis* to determine if the value of the original scalar is used outside the loop. It also eliminates dead code, as described in the section on dead code removal. (See **Section 6.3.7, "Dead Code Elimination"** .)

If the variable is reused within the compilation unit, the last value of the scalar is assigned as shown in the following code segment. For example:

```
int a[], b[], c[], n;
void example_7_3_4 ()
{
      int i, x, y;
      for (i=0; i<n; i++) {
          x = a[i];
          y = b[i];
          c[i] = x + y;
      }
      printf (" x=%d \\n", x);
}
becomes:
int a[],  b[],  c[];
int n;
void example_7_3_4(  )

{
      int i;
      int x;
      int y;
      int _Kii1;
      _Kii1 = ((n)>(0) ? (n) : (0));
#pragma parallel if(n > 101) byvalue(n) shared(c, a, b) local(i)
#pragma pfor iterate(i=0;n;1)
      for ( i = 0; i<n; i++ ) {
          c[i] = a[i] + b[i];
      }
      if (_Kii1 > 0)
          x = a[_Kii1-1];
      printf( " x=%d \\n", x );
}
```

6.3.5 Invariant if Floating

When an *if* condition is invariant in the loop, PCA often can move the *if* statement outside the loop, further improving the performance of the code. You need to set *-optimize=4* or higher for this optimization to occur.

For example:

```
int a[], b[], c[], n, x;
void example_7_3_5 ()
{
      int i;
      for (i=0; i<n; i++) {
          a[i] = b[i] + c[i];
          if (x > 0)
              a[i] *= x;
      }
}
```

becomes:

```
int a[],  b[],  c[];
int n,  x;
void example_7_3_5(  )
{
      int i;
      if (x > 0) {
#pragma parallel if(n > 101) byvalue(n, x) shared(a, b, c) local(i)
```

```
#pragma pfor iterate(i=0;n;1)
          for ( i = 0; i<n; i++ ) {
              a[i] = b[i] + c[i];
              a[i] *=  x;
          }
      } else {
#pragma parallel if(n > 201) byvalue(n) shared(a, b, c) local(i)
#pragma pfor iterate(i=0;n;1)
          for ( i = 0; i<n; i++ ) {
              a[i] = b[i] + c[i];
          }
          }
}
```

6.3.6 Derived Assertions

PCA can derive some information about the relative values of scalar integers from the *if* and assignment statements in the program. For example:

```
int a[], b[], m, n;
void example_7_3_6 ()
{
      int i;
      if (m > n) {
          for (i=0; i<n; i++) {
              a[i] = a[i+m] + b[i];
          }
      }
}
```

becomes:

```
int a[], b[], m, n;
void example_7_3_6(  )
{
      int i;
      if (m > n) {
#pragma parallel if(n > 143) byvalue(n, m) shared(a, b) local(i)
#pragma pfor iterate(i=0;n;1)
          for ( i = 0; i<n; i++ ) {
              a[i] = a[i+m] + b[i];
          }
          }
}
```

The transformation is legal because the loop can be executed only when the value of *m* is greater than *n*.

6.3.7 Dead Code Elimination

When you set the -*scalaropt* option to 1 or higher, PCA performs dead code elimination. To get the full benefit of dead code elimination, combine it with optimizations such as subprogram in−lining (function call expansion) and forward substitution. These optimizations expose useless or unreachable code that PCA cannot otherwise see. PCA will also perform the following optimizations:

- Removal of unreachable code. For example:

```
float x, y;
void example_7_3_7a ()
{
      goto hop;
      x = 2.0;
```

```
hop:
      y = 13.0;
}
```

becomes:

```
float x;
float y;
void example_7_3_7a(  )
{
hop:
      y = 13.0;
}
```

- Removal of zero–trip and empty *for* loops. For example, the following code is deleted completely:

```
float x, y;
void example_7_3_7b ()
{
      int i;
      for (i=10; i<2; i++)
          x = x + y;
}
```

- Elimination of resolved conditionals. For example:

```
float x;
void example_7_3_7c ()
{
      if (12 > 10)
          x = 1.0;
      else
          x = 2.0;
}
```

Because the true branch is always taken, this code becomes:

```
float x;
void example_7_3_7c(  )
{
      x = (float)(1.0);
}
```

- Removal of unnecessary or unprofitable code. (*scalaropt* must be ≥ 2 and *optimize* ≥ *2* for this optimization.) PCA performs *lifetime analysis* to determine the reaching definitions of variables and removes unused definitions.

```
void example_7_3_7d ()
{
      float x, y;
      y = 5.0;     /* no subsequent use of local variable y */
      x = 3.0;     /* variable x redefined */
      x = 4.0;
      printf ("%g \n", x);
}
```

becomes:

```
void example_7_3_7d(  )
{
     float x;
     float y;
     x = (float )(4.0);
     printf( "%g \n", x );
}
```

6.4 Loop Rerolling

Many programs have loops that were unrolled manually over several iterations to amortize the cost of the test and branch at each iteration of the *for* loop. Before PCA can evaluate these unrolled loops for parallel execution, the loops must be rerolled to a simpler form. See the following dusty−deck transformations.

```
int a[], b[], c[], n;
void example_7_4a ()
{
      int i;
      for (i=0; i<n; i+=2) {
          a[i] = b[i] + c[i];
          a[i+1] = b[i+1] + c[i+1];
      }
}
```

PCA recognizes this iteration as an unrolled loop and rerolls it before evaluating it for parallel execution, as follows:

```
int a[], b[], c[], n;
void example_7_4a(  )

{
      int i;
#pragma parallel if(n > 199) byvalue(n) shared(a, b, c) local(i)
#pragma pfor iterate(i=0;(n+1)/2*2;1)
      for ( i = 0; i<=(n + 1) / 2 * 2 − 1; i++ ) {
          a[i] = b[i] + c[i];
      }
}
```

PCA can also recognize unrolled summations (with *-roundoff=3*):

```
int b[], c[], n, sum;
void example_7_4b ()
{
      int i;
      for (i=0; i<n; i+=2) {
          sum += b[i] + c[i];
          sum += b[i+1] + c[i+1];
      }
}
and reroll them:
int b[], c[], n, sum;
void example_7_4b(  )
{
      int i;
      int sum1;
#pragma parallel if(n > 125) byvalue(n) shared(c, b, sum) local(i)
 reduction(sum1)
      {
          sum1 = 0;
#pragma pfor iterate(i=0;(n+1)/2*2;1)
          for ( i = 0; i<=(n + 1) / 2 * 2 − 1; i++ ) {
              sum1 +=  b[i] + c[i];
          }
#pragma critical
```

```
          {
              sum +=  sum1;
          }
      }
}
```

6.5 Loop Unrolling

Unrolling of *for* loops is the standard manual optimization technique that creates more statements in a small loop by repeating the original statement. PCA can automatically unroll both serial and parallel loops to speed execution. Unrolling a loop involves duplicating the loop body one or more times within the loop, adding an increment (or changing the increment that was already in the loop), and possibly inserting code before the loop to execute the excess iterations of the loop (the "cleanup code").

If the loop bounds are constant and the iteration count of the loop is small, PCA can eliminate the loop entirely and replace it with copies of the loop body, or PCA can omit the cleanup code. To enable loop unrolling, set the *-scalaropt* command−line option to 2.

The following example was run with *-unroll=8*, and *-unroll2=1000*. (See **Chapter 3, "PCA Command−Line Options"** for more information on these command−line options.) If the loop bounds are unknown at compilation time, PCA might analyze a loop.

For example:

```
int a[], b[], c[], n;
void example_7_5a ()
{
      int i;
      for (i=0; i<n; i++) {
          a[i] = b[i] + c[i];
      }
}
```

is unrolled as:

```
int a[], b[], c[], n;
void example_7_5a(  )
{
      int i;
      int _Kii1;
      _Kii1 = (n)%(8);
      for ( i = 0; i<_Kii1; i++ ) {
          a[i] = b[i] + c[i];
      }
#pragma parallel if(n > 201) byvalue(_Kii1, n) shared(a, b, c) local(i)
#pragma pfor iterate(i=_Kii1;(n-_Kii1+7)/8;8)
      for ( i = _Kii1; i<n; i+=8 ) {
          a[i] = b[i] + c[i];
          a[i+1] = b[i+1] + c[i+1];
          a[i+2] = b[i+2] + c[i+2];
          a[i+3] = b[i+3] + c[i+3];
          a[i+4] = b[i+4] + c[i+4];
          a[i+5] = b[i+5] + c[i+5];
          a[i+6] = b[i+6] + c[i+6];
          a[i+7] = b[i+7] + c[i+7];
      }
   }
```

But if the loop iteration count is constant and small, PCA can remove the loop entirely. For example:

```
int a[], b[], c[];
```

```
void example_7_5b ()
{
      int i;
      for (i=0; i<5; i++) {
          a[i] = b[i] + c[i];
      }
}
```

becomes:

```
int a[], b[], c[];
void example_7_5b(  )

{
      int i;
      a[0] = b[0] + c[0];
      a[1] = b[1] + c[1];
      a[2] = b[2] + c[2];
      a[3] = b[3] + c[3];
      a[4] = b[4] + c[4];
}
```

6.6 Loop Fusion

Loop fusion is a conventional compiler optimization that transforms two adjacent loops into a single loop. The use of data−dependence tests allows fusion of more loops than is possible with standard techniques. You must use *-scalaropt=2* to enable loop fusion.

In the following example, the first two loops are fused and concurrentized together. Fusing these loops reduces *for*–loop overhead and the amount of synchronization required. PCA recognizes that the third loop must execute after the first two and does not fuse it with the others.

For example:

```
int a[], b[], c[], d[], n;
void example_7_6 ()
{
      int i;
      for (i=0; i<n; i++) {
          a[i] = b[i] + c[i];
      }
      for (i=0; i<n; i++) {
          a[i] = a[i] + d[i];
      }
      for (i=0; i<n; i++) {
          d[i] = a[i+1];
      }
}
```

becomes:

```
int a[], b[], c[], d[], n;
void example_7_6(  )
{
      int i, _Kii1;
#pragma parallel if(n > 50) byvalue(n) shared(a, b, c, d) local(_Kii1, i)
      {
#pragma pfor iterate(_Kii1=0;n;1)
          for ( _Kii1 = 0; _Kii1<n; _Kii1++ ) {
              a[_Kii1] = b[_Kii1] + c[_Kii1];
```

```
              a[_Kii1] +=  d[_Kii1];
          }
#pragma synchronize
#pragma pfor iterate(i=0;n;1)
          for ( i = 0; i<n; i++ ) {
              d[i] = a[i+1];
          }
      }
}
```

6.7 Memory Management for Data Locality

The following sections describe PCA's memory management options and how it uses them. The options available are:

- *cachesize* - use this option to pick block sizes, that is, the sizes of the sections in the cache.
- *cacheline* - use this option to inform PCA of the width in bytes of the memory channel between cache and main memory.
- *fpregisters* and *spregisters* - use these options to pick unrolling factors, and to make sure registers are not overflowed when unrolling.
- *setassociativity* - use this option to decide which memory management algorithm to use.

6.7.1 Memory Management Techniques

PCA can enhance your program's performance on machines that use cache memory. It uses a combination of memory *padding*, loop *blocking*, loop *interchanging*, and outer loop *unrolling* to optimize re−use of operands in memory. You can enable memory management by setting *scalaropt*^{3} *3* and *roundoff*^{3} *3*. (The default settings are *scalaropt=3* and *roundoff=0*.)

PCA improves memory access patterns in loops processing arrays by working on small sections of the arrays that fit into cache and give large cache hit ratios. You can control the sizes of these array sections by using the memory management command−line options. The default settings for these options are the standard characteristics of the machine the code is targeted for. The default settings will produce the best results for most programs. However, in some specialized cases you might want to modify the settings to adjust the array section sizes.

PCA determines at runtime which of two memory management algorithms to use, depending on the settings for *cacheline* and *setassociativity*. The default algorithm keeps square blocks of data in cache, while the other keeps long, narrow blocks of data in cache.

The example below shows how loop blocking, loop interchanging, and loop unrolling can be used to improve the performances of this matrix multiplication code. The PCA command−line options used to analyze it were *o=3*, *so=2*, *r=3*, and *arl=2*. The *arl* option is set because the arrays are function arguments and PCA assumes by default that they might overlap.

```
double matm(n,a,b,c)
int n;
double a[200][200],b[200][200],c[200][200];
{
     int i,j,k;
     for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
           a[i][j] = 0.0;
           for (k=0; k<n; k++)
              a[i][j] = a[i][j] + b[i][k]*c[k][j];
        }
     return (a[3][5]);
}
```

becomes:

```
double matm( n, a, b, c )
```

```
      int n;
      double  (*a)[200];
      double  (*b)[200];
      double  (*c)[200];
{
      int i;
      int j;
      int k;
      int _Kii1;
      int _Kii2;
      int _Kii3;
      int _Kii4;
      int _Kii5;
      int _Kii6;
      int _Kii9;
      double _Kdd1;
      int _Kii10;
      int _Kii11;
      double _Kdd2;
      int _Kii12;
      double _Kdd3;
      int _Kii13;
      double _Kdd4;
      int _Kii14;
      double _Kdd5;
      int _Kii15;
      double _Kdd6;
      int _Kii16;
      double _Kdd7;
      int _Kii17;
      double _Kdd8;
      int _Kii18;
      double _Kdd9;
      int _Kii19;
      double _Kdd10;
      int _Kii20;
      double _Kdd11;
      int _Kii21;
      int _Kii22;
      double _Kdd12;
      int _Kii23;
      int _Kii24;
      int _Kii25;
      int _Kii26;
      int _Kii27;
      int _Kii28;
      _Kii10 = n - 1;
      _Kii27 = n / 10;
#pragma parallel byvalue(n, _Kii10) shared(a) local(i, j)
#pragma pfor iterate(i=0;n/10;10)
      for ( i = 0; i<=n - 10; i+=10 ) {
          for ( j = 0; j<=_Kii10; j++ ) {
              a[i][j] = 0.0;
              a[i+1][j] = 0.0;
              a[i+2][j] = 0.0;
              a[i+3][j] = 0.0;
              a[i+4][j] = 0.0;
              a[i+5][j] = 0.0;
```

```
              a[i+6][j] = 0.0;
              a[i+7][j] = 0.0;
              a[i+8][j] = 0.0;
              a[i+9][j] = 0.0;
          }
      }
      _Kii1 = _Kii27 * 10;
      _Kii11 = n - 1;
      for ( i = _Kii1; i<=_Kii11; i++ ) {
          _Kii26 = (_Kii11 + 1)%(3);
          for ( j = 0; j<_Kii26; j++ ) {
              a[i][j] = 0.0;
          }
          for ( j = _Kii26; j<=_Kii11; j+=3 ) {
              a[i][j] = 0.0;
              a[i][j+1] = 0.0;
              a[i][j+2] = 0.0;
          }
      }
      _Kii2 = n;
      _Kii5 = 0;
      _Kii3 = (_Kii2 - 1)%(546) + 1;
      _Kii4 = _Kii3;
      _Kii22 = n - 1;
      _Kii25 = n - 10;
      _Kii24 = n - 1;
      _Kii28 = (_Kii25 + 10) / 10;
      for ( _Kii6 = 1; _Kii6>=_Kii2; _Kii6+=546 ) {
          _Kii21 = _Kii5 + _Kii4 - 1;
          for ( k = 0; k<=_Kii25; k+=10 ) {
              _Kii12 = k + 1;
              _Kii13 = k + 2;
              _Kii14 = k + 3;
              _Kii15 = k + 4;
              _Kii16 = k + 5;
              _Kii17 = k + 6;
              _Kii18 = k + 7;
              _Kii19 = k + 8;
              _Kii20 = k + 9;
#pragma parallel byvalue(_Kii22, k, _Kii12, _Kii13, _Kii14, _Kii15, _Kii16,
 _Kii17, _Kii18, _Kii19, _Kii20, _Kii5, _Kii21)
#pragma         shared(b, a, c) local(_Kdd1, _Kdd2, _Kdd3, _Kdd4, _Kdd5,
 _Kdd6, _Kdd7, _Kdd8, _Kdd9, _Kdd10, _Kdd11, i, j)
#pragma pfor iterate(i=0;_Kii22+1;1)
              for ( i = 0; i<=_Kii22; i++ ) {
                  _Kdd2 = b[i][k];
                  _Kdd3 = b[i][_Kii12];
                  _Kdd4 = b[i][_Kii13];
                  _Kdd5 = b[i][_Kii14];
                  _Kdd6 = b[i][_Kii15];
                  _Kdd7 = b[i][_Kii16];
                  _Kdd8 = b[i][_Kii17];
                  _Kdd9 = b[i][_Kii18];
                  _Kdd10 = b[i][_Kii19];
                  _Kdd11 = b[i][_Kii20];
                  for ( j = _Kii5; j<=_Kii21; j++ ) {
                      _Kdd1 = a[i][j];
                      _Kdd1 +=  _Kdd2 * c[k][j];
                      _Kdd1 +=  _Kdd3 * c[_Kii12][j];
```

```
                      _Kdd1 +=  _Kdd4 * c[_Kii13][j];
                      _Kdd1 +=  _Kdd5 * c[_Kii14][j];
                      _Kdd1 +=  _Kdd6 * c[_Kii15][j];
                      _Kdd1 +=  _Kdd7 * c[_Kii16][j];
                      _Kdd1 +=  _Kdd8 * c[_Kii17][j];
                      _Kdd1 +=  _Kdd9 * c[_Kii18][j];
                      _Kdd1 +=  _Kdd10 * c[_Kii19][j];
                      _Kdd1 +=  _Kdd11 * c[_Kii20][j];
                      a[i][j] = _Kdd1;
                  }
              }
          }
          _Kii9 = _Kii28 * 10;
          _Kii23 = _Kii5 + _Kii4 - 1;
#pragma parallel byvalue(_Kii24, _Kii5, _Kii23, _Kii9) shared(b, a, c)
 local(_Kdd12, i, j, k)
          {
              for ( k = _Kii9; k<=_Kii24; k++ ) {
#pragma pfor iterate(i=0;_Kii24+1;1)
                  for ( i = 0; i<=_Kii24; i++ ) {
                      _Kdd12 = b[i][k];
                      for ( j = _Kii5; j<=_Kii23; j++ ) {
                          a[i][j] +=  _Kdd12 * c[k][j];
                      }
                  }
#pragma synchronize
              }
          }
          _Kii5 +=  _Kii4;
          _Kii4 = 546;
      }
      return a[3][5];
}
```

Chapter 7
In–lining and Interprocedural Analysis

This chapter provides additional information about the PCA command–line options and in–line pragmas that you can use to *inline* functions and perform *interprocedural analysis*.

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

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

Table 7–1 lists the in–lining options.

In–lining-Purpose	**Long Name**	**Short Name**	**Default Value**
Specify routine to in–line	inline[=*name*[,*name*...]]	inl[=*names*]	off
Create preprocessed library	inline_create=*lib*.klib	incr=*lib*.klib	off
Define inlinable routines	inline_from_files=*list*	inff=*list*	current source file
Specify library from **which to in–line**	inline_from_libs=*list*	infl=*list*	off
Specify call nest level	inline_depth[=*n*]	ind[=*n*]	ind=2

Specify *for* loop–nest level	inline_looplevel[=*n*]	inll[=*n*]	inll=2
Specify manual control	inline_manual	inm	off

Table 7–1 In–lining Options

Table 7–2 lists the IPA options.

IPA-Purpose	**Long Name**	**Short Name**	**Default Value**
Specify routine to analyze	ipa[=*name*[,*name*...]]	ipa[=*names*]	off
Create preprocessed library	ipa_create=*lib*.klib	ipacr=*lib*.klib	off
Define routines for IPA	ipa_from_files=*list*	ipaff=*list*	current source file
Specify library from **which to do IPA**	ipa_from_libs=*list*	ipafl=*list*	off
Specify *for* loop–nest level for IPA	ipa_looplevel[=*n*]	ipall[=*n*]	ipall=2
Specify manual control	ipa_manual	ipam	off

Table 7–2 Interprocedural Analysis Options

The rest of this chapter covers the in–lining and interprocedural analysis command–line options and pragmas, related command–line options, examples of their use, and information on program constructs that inhibit in–lining. In–lining and interprocedural analysis are symmetrical from the command–line standpoint-you use related sets of commands and pragmas for them. (Many places that say *in–lining* apply to both *in–lining* and *interprocedural analysis*.)

7.1 In–lining and IPA Command–Line Options

In–lining has two phases:

1. Define the universe of in–linable routines.
2. Select which routines in that universe to in–line or analyze.

The *from_files* and *from_libs* options define the universe of in–linable routines. The *inline*, *ipa*, and *looplevel* options select which of the available routines are to be in–lined/analyzed. The *create* options set up collections of routines for inclusion in later PCA runs.

The subsections that follow define the syntax for in–lining and interprocedural analysis command–line options. The short forms of their names appear in square brackets ([ ]).

7.1.1 The inline_from and ipa_from Options

The *inline_from* and *ipa_from* options take the following form:

```
-inline_from_files=list          [-inff=list]
-inline_from_libraries=list      [-infl=list]
-ipa_from_files=list             [-ipaff=list]
-ipa_from_libraries=list         [-ipafl=list]
```

where *list* is one or more of the following:

- source file name
- library file name
- directory

Separate each item in the list by commas. Do not use shell wild card characters in the list of files and directories. The default is current source file. Different types of files are distinguished by their extensions. For example:

```
-inline_from_files=xj.c,yy.c,../mrtn
```

looks for routines in the C source files *xj.c* and *yy.c*, and in C source files in the directory *../mrtn*. (Including the directory *../mrtn* is equivalent to the UNIX® notation *../mrtn/*.c*). All source files that contain

C preprocessor directives must be preprocessed by the cc compiler before being in–lined or analyzed.

The *from_libraries* versions of these options take as their arguments lists of function libraries and directories containing such libraries.

PCA recognizes the type of file by its extension, or lack of one (see **Table 7–3** for the file types).

File Extension	Type of File
.c	C source file
.klib	Library from *inline/ipa_create*
other	Directory

Table 7–3 File Types

Two special abbreviations are:

dash (-)	A dash specifies the current source file (as listed on the command line, or specified in a *-input=file* command–line option).
period (.)	A period specifies the current working directory.

Specifying a nonexistent file or directory is a command–line error.

If you specify multiple *from_files* and *from_libraries* options, their lists are concatenated to get a bigger universe.

Routine name references are resolved by a search in the order that files appear in *from_files* and *from_libraries* options on the command line. Libraries are searched in their original lexical order. Multiple *from_files* and *from_libraries* lists are searched in the order in which they appear on the command line.

7.1.2 Creating and Using Libraries

To create a preprocessed library, use the following syntax:

```
–inline_create=library_name.klib       [–incr=lib_name.klib]
–ipa_create=library_name.klib          [–ipacr=lib_name.klib]
```

To specify a library file to in–line from, use:

```
–inline_from_libraries=list            [–infl=list]
–ipa_from_libraries=list               [–ipafl=list]
```

The default source for routines to put into the library is the current source file. If you specify *inline_from (ipa_from)*, the routines in the listed files are the ones put into the library. This provides a method to combine or expand libraries. Just include the old library(ies) and any new file(s) in an *inline_from (ipa_from)* option.

Routines are included in libraries in the order in which they appear in the input file(s). This order guarantees that if multiple routines with the same name are in the same source file, the one chosen for in–lining will be the one you expect from the algorithm under *inline_from*, described previously.

A library created with *inline_create* will work for in–lining or IPA, since it is just partially reduced source code. However, a library made with *ipa_create* may not appear in an *-inline_from=list*. Such use is flagged with a warning message.

If no library name is given, the name used is *file.klib*, where *file* is the input file name with any trailing *.c* stripped off.

When creating a library, only one *create* option may be given. That is, only one library may be created per PCA run. If the library file existed prior to running PCA, it is overwritten. When you specify this option on the command line, no transformed code file will be generated. See the previous description of the *from_libraries* options for information on using libraries created with these options.

If you don't specify an *inline (ipa)* option, the default is to include all the functions in the source file in the library, if possible. See **Section 7.5, "Conditions That Inhibit In–lining"** later in this chapter for a list of conditions that can prevent a function from being in–lined.

An example of in–lining from the library created above is included in the section of examples later in this chapter.

7.1.3 Naming Specific Routines

To specify the names of particular routines to in-line, use:

```
-inline[=name[,name...]]          [-inl=name,...]
-ipa[=name[,name...]]             [-ipa=name,...]
```

The default is all routines in the function universe. You can specify this by any *inline_from* (*ipa_from*) option, subject to the *looplevel* setting.

In-lining and IPA are *off* by default, that is, if no in-lining (IPA) options are specified and no in-lining (IPA) directives are found in the source code, no in-lining (IPA) is performed.

If you omit *inline* (*ipa*) from the command line, automatic selection of routines to in-line is disabled. You can manually select functions to in-line (analyze) with the *-inline_manual* (*-ipa_manual*) options and the *inline* and *ipa* pragmas.

If you specify *inline* (*ipa)* on the command line without a list of routine names, then all routines in the in-lining (IPA) universe are eligible, subject to the *looplevel* value.

If you specify *inline* (*ipa*) on the command line with a list of routine names, then only the listed routines are eligible, subject to the *looplevel* value.

7.1.4 for Loop Level

To set a minimum *for* loop nest level for function call expansion, use:

```
-inline_looplevel[=n]          [-inll[=n]]
-ipa_looplevel[=n]             [-ipall[=n]]
```

Use the *looplevel* option to limit in-lining and interprocedural analysis to just functions that are referenced in nested loops, where the reduced function call overhead or enhanced optimization will be multiplied.

The argument is defined from the most deeply nested leaf of the *call tree*.

The default, 2, restricts in-lining (interprocedural analysis) to the best-seeming candidate routines.

For example:

```
main
{
    ...
     a();  ------>  a() {...}
}
    ..
   for (..) {
     for (..) {
      b();  --------->  b() {
     }                    for (..) {
   }                        for (..) {
                              c();  -------> c() {...}
                            }
                          }
                        }
```

The call to *b* is inside a doubly nested loop and is more profitable to expand than the call to *a*. The call to c is quadruply nested, so in-lining *c* yields the biggest gain of the three.

The argument is defined from the most deeply nested function reference:

-inline_looplevel=1
Only the functions referenced in the most deeply nested call site(s) may be expanded (function *c* in the previous example). If more than one function call is at the same loop-nest level, all of them are selected when that level is included.

-inline_looplevel=2
Only function calls at the most deeply nested level and one loop less deeply nested

-inline_looplevel=3
: Level 3 is required to in−line function *b*, since its call is two loops less nested than the call to function *c*.
A value of 3 or greater causes *c* to be in−lined into *b*, then the new *b* to be in−lined into the main program.

-inline_looplevel (or *-inline_looplevel=large number*)
: A large number permits in−lining at any nesting level. The calling tree written to the listing file with *-listoptions=c* includes the nesting depth level of each call in each program unit and the aggregate nesting depth (the sum of the nesting depths for each call site, starting from the main program). Use this information to identify the best functions for in−lining.

A function that passes the *looplevel* test is in−lined everywhere it is used, even places that are not in deeply nested loops. If some, but not all, invocations of a function are to be expanded, use the *inline* and *ipa* pragmas just before each function call that is to be expanded (see the next section).

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

7.1.5 Manual Control

To instruct PCA to recognize the *#pragma* [*no*]*inline* and *#pragma* [*no*]*ipa* directives, use these options:

```
−inline_manual              [−inm]
−ipa_manual                 [−ipam]
```

This allows manual control over which functions are in−lined/analyzed at which call sites (see the following section, **Section 7.2, "In−lining Pragmas"**).

The default is to ignore these pragmas. To enable these pragmas, include *-inline_manual* (*-ipa_manual*) on the command line.

Since *#pragma [no]inline* and *#pragma [no]ipa* are not affected by the *looplevel* command−line options, you can use them either with or without the command−line control.

7.2 In−lining Pragmas

The *inline*/*ipa* pragmas tell PCA to in−line (or perform interprocedural analysis on) the named functions. The syntax is:

```
#pragma [no]inline [here][routine][global] [(name[,name...])]
#pragma [no]ipa [here][routine][global] [(name[,name...])]
```

These pragmas tell PCA whether or not to in−line/analyze the named functions. These pragmas combine next−line, entire routine, and global (entire program) scope. If you omit these optional elements, all functions referenced on the next line of code that are in the in−lining/analyzing universe are in−lined on that one line.

These pragmas are disabled by default. Enable them with the *-inline_manual* and *-ipa_manual* command−line options. They are independent of the other in−lining and IPA command−line options, and you can use them instead of, or in addition to, command−line controlled in−lining.

Keywords: here, routine, and global

The keywords, *here*, *routine*, and *global* are described below.

here	If you include the scope keyword *here*, or if you don’t specify any scope, the pragma applies only to the next statement.
routine	If you include the scope keyword *routine*, the pragma applies to the rest of the routine, or until a corresponding *no* appears. (Or, if the first pragma was a *noinline* (*noipa*), until the corresponding *inline* (*ipa*) pragma.)

global	If you include the scope keyword *global*, or if the pragma appears before any lines of source code, the pragma applies to the entire file, or until toggled with the corresponding *no* pragma. (Or, if the first pragma was a *noinline* (*noipa*), until the corresponding *inline* (*ipa*) pragma.) Typically, *global* pragmas appear only at the top of the source file. The same routine name may not appear in both global in−lining and global IPA lists, either by pragmas or the *inline* (*ipa*) options.

These keywords must appear in lowercase, as function names are case sensitive. The optional *names* are function names. If any functions are named in the directive, it applies only to them. If *no* function names are given, the pragma applies to *all* functions. The parentheses around the function names are not required if the list of function names is empty.

If a *#pragma inline* or *#pragma ipa* names a routine not in the universe, a warning message is issued, and the pragma is ignored.

7.3 Listing File Additions

You can print the calling tree with the *-listoptions=c* option.

7.3.1 -listoptions=c

The optional calling tree includes the loop−nest depth level of each function call. The metric uses the convention of the *-inline_looplevel* and *-ipa_looplevel* options. The farthest−out leaf is 1, and higher values trace back to the main program.

7.4 In−lining/IPA Examples

The following code examples demonstrate a few of the possibilities for using the features described in this chapter. Because PCA undergoes constant enhancement, the code that your version of PCA produces may not be identical to the code in these examples. The temporary variable names, in particular, can change without substantially altering the transformed code.

Unless otherwise noted, the following examples were run with the *-optimize* and *-scalaropt* options set to:

```
−o=0 −so=0
```

to show the in−lining more clearly. If you specify nonzero values, the functions are first in−lined or analyzed, and then the concurrentization/ dusty−deck transformations (see **Chapter 3, "PCA Command−Line Options"**) are applied. In some cases, C preprocessor additions or code modifications were removed to make the examples simpler.

7.4.1 In−lining Example-Same Source File

The following example demonstrates in−lining both with *-inline=matm* (only the function *matm* will be in−lined), and with *-inline* (both functions are in−lined). The PCA output includes optimized versions of both functions, in addition to the expanded main program. An example source file follows:

```
void example_8_4_1 ()
{
      int i, n;
      double a[200][200], b[200][200], c[200][200];
      double cksum, matm();
      setup (b, 200);
      setup (c, 200);
      for (n=25; n<200; n=n+25) {
          cksum = matm (n, a, b, c);
          printf ("For N=  %d   checksum= %q \\n", n, cksum);
      }
}
void setup (double e[200][200], int n)
```

```
{
      int i, j;
      for (i=0; i<n; i++)
          for (j=0; j<n; j++)
              e[i][j] = ((i + 7*j) % 10) / 10.0;
      return;
}
double matm (int n, double a[200][200], double b[200][200], double
 c[200][200])
{
      int i, j, k;
      for (i=0; i<n; i++)
          for (j=0; j<n; j++) {
              a[i][j] = 0.0;
              for (k=0; k<n; k++)
                  a[i][j] = a[i][j] + b[i][k]*c[k][j];
          }
      return (a[3][5]);
}
```

This is the main program generated by *-inline=matm*:

```
void example_8_4_1(  )
{
      int i, n;
      double a[200][200];
      double b[200][200];
      double c[200][200];
      double cksum;
      double matm( );
      setup( b, 200 );
      setup( c, 200 );
      for ( n = 25; n<=199; n+=25 ) {
          cksum = matm( n, a, b, c );
          printf( "For N=  %d   checksum= %q \\n", n, cksum );
      }
}
void setup( double e[][200], int n )
{
      int i;
      int j;
      for ( i = 0; i<n; i++ ) {
          for ( j = 0; j<n; j++ ) {
              e[i][j] = ((i + j * 7) % 10) / 10.0;
          }
      }
      return ;
}
double matm( int n, double a[][200], double b[][200], double c[][200] )
{
      int i;
      int j;
      int k;
      double _Kdd1;
      for ( i = 0; i<n; i++ ) {
          for ( j = 0; j<n; j++ ) {
              a[i][j] = 0.0;
              _Kdd1 = a[i][j];
              for ( k = 0; k<n; k++ ) {
```

```
                  _Kdd1 +=  b[i][k] * c[k][j];
              }
              a[i][j] = _Kdd1;
          }
      }
      return a[3][5];
}
```

This is the output generated by *-inline*:

```
void example_8_4_1(  )
{
      int i;
      int n;
      double a[200][200];
      double b[200][200];
      double c[200][200];
      double cksum;
      double matm( );
      setup( b, 200 );
      setup( c, 200 );
      for ( n = 25; n<=199; n+=25 ) {
          cksum = matm( n, a, b, c );
          printf( "For N=  %d   checksum= %q \\n", n, cksum );
      }
}
void setup( double e[][200], int n )
{
      int i;
      int j;
      for ( i = 0; i<n; i++ ) {
          for ( j = 0; j<n; j++ ) {
              e[i][j] = ((i + j * 7) % 10) / 10.0;
          }
      }
      return ;
}
double matm( int n, double a[][200], double b[][200], double c[][200] )
{
      int i;
      int j;
      int k;
      double _Kdd1;
      for ( i = 0; i<n; i++ ) {
          for ( j = 0; j<n; j++ ) {
              a[i][j] = 0.0;
              _Kdd1 = a[i][j];
              for ( k = 0; k<n; k++ ) {
                  _Kdd1 +=  b[i][k] * c[k][j];
              }
              a[i][j] = _Kdd1;
          }
      }
      return a[3][5];
}
```

7.4.2 In−lining Example with a Library

The next example demonstrates the creation of a library and in−lining functions from it, a two−step process.

First step: Create the library.

The file *subfil.c* contains these two functions:

```
extern double sin (double);
#pragma no side effects (sin)
void mkcoef (double coef[], int n)
{
      int i;
      for (i=0; i<n; i++)
          coef[i] = 1.0 / (i + 1);
}
double yval (double x, double coef[], int n)
{
      double sum;
      int i;
      sum = 0.0;
      for (i=0; i<n; i++)
          sum = sum + coef[i] * sin ((i + 1) * x);
      return (sum);
}
```

Run the file through the C preprocessor to create the file *subfil.cpp*:

```
/usr/lib/cpp subfil.c > subfil.cpp
```

Then execute the PCA command:

```
/usr/lib/pca  -inline_create=subfil.klib   -list=subfil.L subfil.cpp
```

This creates a library file with the two functions, and a listing file *subfil.L*, which contains only a list of routines and whether or not each was saved in the library:

```
function mkcoef -- saved
function yval -- saved
```

Second step: Inline the functions into a calling program.

The file *sqwv.c* contains the main program:

```
void example_8_4_2 ()
{
      double coef[15], y[2000], yval();
      int i;
      mkcoef (coef, 15);
      for (i=0; i<2000; i++)
          y[i] = yval ((i + 1) * 0.001 * 3.14159, coef, 15);
      for (i=0; i<2000; i=i+10)
         printf ("%f %f %f %f %f %f %f %f %f %f \\n",y[i],y[i+1],
y[i+2], y[i+3], y[i+4], y[i+5], y[i+6], y[i+7], y[i+8],
y[i+9]);
}
```

Run the commands:

```
/usr/lib/cpp sqwv.c > sqwv.cpp
/usr/lib/pca -infl=subfil.klib -o=0 -d=0 sqwv.cpp \
-cmp=sqwv.cmp
```

This puts the following into the file sqwv.cmp:

```
void example_8_4_2(  )
```

```
{
      double coef[15];
      double y[2000];
      double yval( );
      int i;
      double _Kdd1[129];
#pragma padding(_Kdd1)
#pragma storage order(y, _Kdd1, coef)
      mkcoef( coef, 15 );
      for ( i = 0; i<=1999; i++ ) {
          y[i] = yval( (i + 1) * 0.001 * 3.14159, coef, 15 );
      }
      for ( i = 0; i<=1999; i+=10 ) {
          printf( "%f %f %f %f %f %f %f %f %f %f \\n", y[i], y[i+1], y[i+2],
 y[i+3], y[i+4], y[i+5], y[i+6], y[i+7], y[i+8], y[i+9]
               );
      }
}
```

In the previous example, all other optimizations were turned off to show the expansion more clearly. If you specify non−zero values for the *-optimize*, *-scalaropt*, and *-roundoff* options, PCA first in−lines the routines, then performs the optimizations in the usual manner.

7.4.3 IPA Example

In the following example, the variables *n* and *np1* have a simple relationship. This relationship is hidden behind a function call, however, so PCA normally will not try to concurrentize the loop in the main program.

When you specify the *-ipa=rxgfs* command−line option, PCA will inspect the named function for information on the relationship of its arguments and returned value and the surrounding code. The *assumed dependence* is lifted, and the loop can be safely concurrentized.

If a function cannot be in−lined (this simple one can be), or if you don't want to in−line it, it can often still be analyzed for its effects on the calling routine.

The next example looks like this:

```
void example_8_4_3 ()
{
      int np1, i, m, n;
      int a[100][100];
      np1 = rxgfs(n);
      for (i=0; i<m; i++) {
          a[i][n] = a[i−1][np1];
      }
}
int rxgfs(int n)
{
      return (n+1);
}
```

When run with the default values for *-optimize* and *-scalaropt*, the example becomes (the function is not shown):

```
void example_8_4_3(  )
{
      int np1;
      int i;
      int m;
      int n;
      int a[100][100];
```

```
        int a[100][100];
        np1 = rxgfs( n );
        for ( i = 0; i<m; i++ ) {
            a[i][n] = a[i-1][np1];
        }
}
int rxgfs( int n )
{
        return (n + 1);
}
```

7.4.4 Notes on In–lining and IPA

You may perform *either* in–lining *or* interprocedural analysis in a PCA run. If you want to in–line some routines and use IPA for others, you must do this in *two* PCA runs.

- Routines to be in–lined must pass all the criteria (*-inline=-inline_looplevel*) to be in–lined. (See the following section for the exception to this rule.)
- The *#pragma* [*no*]*inline* and *#pragma* [*no*]*ipa* directives, when enabled, override the in–lining/IPA command–line options.
- A *#pragma inline global* directive without a function name list instructs PCA to in–line every function it can regardless of the *-inline* and *-inline_looplevel* settings.
- A *#pragma noinline global* directive instructs PCA not to in–line anything, regardless of the *-inline* and *-inline_looplevel* settings.

No in–lining or interprocedural analysis will be performed if the primary source file is *stdin*. (See the description of the *-input* command–line option in **Chapter 3, "PCA Command–Line Options"** for more information on specifying the primary source file.)

When you specify a library with *-inline_from_libraries*, routines may be taken from that library for in–lining into the source code. No attempt is made to in–line routines from the source file into routines from the library.

For example, if the main program calls function *bb*, which is in the library, and *bb* calls function *dd*, which is in the source file, then *bb* can be in–lined into the main program, but PCA will not attempt to in–line *dd* into the text from library routine *bb*.

A library created with *-inline_create* will work for in–lining or IPA, since it is just partially reduced source code, but a library made with *-ipa_create* may not appear in a *-inline_from_libs=list*. It is flagged with a warning message.

In–lining and interprocedural analysis are slow, memory–intensive activities. Using *-inline_looplevel* (in–line all available functions everywhere they are used) for a large set of in–linable routines for a large source file can absorb significant system resources. For most programs, specifying a small value for *-inline_looplevel* and/or a small number of routines with *-inline=* will provide most of the benefits of in–lining. (Specifying a small value also applies to the corresponding IPA options.)

7.5 Conditions That Inhibit In–lining

This section lists conditions that inhibit the in–lining of functions, whether from a library or source file. (See the preceding section for notes on the use of the in–lining command–line options and pragmas.) Conditions that inhibit in–lining include:

- unresolved name conflicts (which usually indicate an incorrect program)
- a function that is too long (> 600 lines)

Chapter 8

The PCA Listing

This chapter describes the types of information available in the optional PCA listing, and the classes of messages that PCA produces. To help you determine PCA's status and efficiency, use the *-listoptions=* option to list the optimizations PCA performed. For example, in some cases PCA may tell you that it could have converted any of three loops to concurrent execution but that it converted only the one it considered most profitable.

At times PCA may not convert a loop to concurrent execution because the loop does so little work that it is not worth the small overhead of parallel execution. Because these changes can produce correct but unexpected code, PCA puts a note in the listing to explain its output.

PCA does not produce a listing file unless you request it.

8.1 listoptions

The *-listoptions=* command–line option tells PCA what information to include in the listing and error files. The listing file can contain any combination of options.

Each *listoptions* option is summarized in **Table 8–1** .

Value	Description
c	Print the Calling Tree of the entire program.
i	Insert line numbers in the transformed code referencing line numbers of the original.
k	Print PCA options used at the end of the listing.
l	Print the loop–by–loop optimization table.
n	Print program unit names, as processed, in the error file.
p	Print the analysis performance statistics.
s	Summarize loop optimizations.

Table 8–1 Listing File Options

You can enter multiple arguments on the command line; separate each with a comma. The following subsections describe the arguments to *-listoptions*.

8.1.1 Calling Tree (c)

PCA lists the calling tree after all program units have been compiled. The program unit information for each calling tree consists of the functions it calls and the routines where that program unit itself is called. An example follows.

```
CALLING TREE
line#         routines         at nest    max. aggregate nest
1           function example_9_1_1
6              call mkcoef        0               0
8              call yval          1               0
11             call printf        1               0
18           function mkcoef
26           function yval
34             call sin           1               0
example_9_1_1
      mkcoef
      yval
          sin
      printf
   CODE MODULES
   example_9_1_1 called from
   mkcoef    called from  example_9_1_1
   printf    called from  example_9_1_1
   sin       called from  yval
   yval      called from  example_9_1_1
```

8.1.2 Insert Line Numbers (i)

The *-listoptions=i* switch directs PCA to insert line number references into the transformed code. The line number indicates that the labeled line is either the same as that line in the original code or is derived from it. These make relating constructs in the original and transformed codes easier. In the unrolled loop that follows, the *for* in the original code was on line 7, and the assignment was on line 8.

```
# 1  "example_9_1_2.c"
int a[];
# 1 "example_9_1_2.c"
int b[];
# 1 "example_9_1_2.c"
int n;
# 2  "example_9_1_2.c"
void example_9_1_2(  )
{
# 4 "example_9_1_2.c"
      int i;
      int _Kii1;
# 5  "example_9_1_2.c"
      _Kii1 = (n)%(3);
# 5  "example_9_1_2.c"
      for ( i = 0; i<_Kii1; i++ ) {
          a[i] = b[i] / a[i-1];
      }
# 5  "example_9_1_2.c"
      for ( i = _Kii1; i<n; i+=3 ) {
          a[i] = b[i] / a[i-1];
# 6  "example_9_1_2.c"
          a[i+1] = b[i+1] / a[i];
# 6  "example_9_1_2.c"
          a[i+2] = b[i+2] / a[i+1];
      }
}
```

8.1.3 PCA Options (k)

The PCA Options table lists the settings of the command–line options related to optimization that were used for this program unit.

```
Options Used for this Program Unit
ADDRESSRESOLUTION=1        ARCLIMIT=2000
CACHELINE=64               CACHESIZE=64
CMPOPTIONS=                CONCURRENTIZEACKV
DPREGISTERS=12             EIIFG=20
FPREGISTERS=12             INLINE_DEPTH=2
INLINE_LOOPLEVEL=2         IPA_LOOPLEVEL=2
LIMIT=5000                 LINES=55
LISTINGWIDTH=80            MACHINE=S
MIIFG=500NO INLINE         MINCONCURRENT=1000
NO INLINE_CREATE           NO INLINE_FROM_FILE
NOINLINE_FROM_LIBRARIES    NO INLINE_MANUAL
NO IPA                     NO IPA_CREATE
NO IPA_FROM_FILES          NO IPA_FROM_LIBRARIES
NO IPA_MANUAL              OPTIMIZE=5
ROUNDOFF=0                 SCALAROPTIMIZE=3
SETASSOCIATIVITY=1         SYNTAX=A
UNROLL=4                   UNROLL2=100
```

8.1.4 Loop Table (l)

The loop table shows what PCA did with each *for* loop. If PCA could not optimize the loop, PCA lists the reason.

```
---------------------------Loop Table--------------------------
                                      Nest
Loop    Message                    Level     Contains Lines
===============================================================
   for i                             1   5-6 "example_9_1_4.c"
          Original Loop Split Into Sub-Loops
     1. Enhanced Scalar              1   5-6, 8 "example_9_1_4.c"
          Line:5 Cleanup loop for loop unrolling.
          Line:8 Loop has been fused with others to reduce
                 overhead.
     2. Concurrent & Enhanced Scalar 1   5-6, 8 "example_9_1_4.c"
          Line:5 Loop has been fused with others to reduce
                 overhead.
          Line:5 Loop unrolled 4 timestoimprove scalar performance.
for i                              1      7-8 "example_9_1_4.c"
```

8.1.5 Name (n)

The program unit names, as processed, are printed in the error file.

```
FILE: example_9_1_5.c
      Function: example_9_1_5
      Function: mkcoef
      Function: yval
0 errors in file example_9_1_5.c
```

8.1.6 Analysis Performance Statistics (p)

The analysis performance statistics list the number of lines in the program unit, the analysis time in seconds, and the analysis rate in lines per minute. The listing also summarized this information after all program units have been analyzed.

```
Compilation Statistics For the Routine example_9_1_6
     14  Lines in Program Unit
   0.29  CPU Time
   2896  Lines Per Minute
      0  Symbol Cache File Writes
      0  Symbol Cache File Reads
      0  Source Save File Reads
      0  Source Save File Writes
      0  Source Save File Opens
      0  Name Table File Writes
      0  Name Table File Reads
Cumulative Compilation Statistics
     15  Lines in Source File
      1  Program Units in Source File
   0.30  CPU Time
   3000  Lines Per Minute
      0  Symbol Cache File Writes
      0  Symbol Cache File Reads
      0  Source Save File Reads
      0  Source Save File Writes
      0  Source Save File Opens
      0  Name Table File Writes
      0  Name Table File Reads
```

8.1.7 Summary Table (s)

The summary table shows how many loops appeared in the program unit, how many loops PCA optimized, and how PCA optimized the loops.

```
3 loops total
1 loops concurrentized
1 preferred scalar mode
1 this loop has been fused with other loops
```

8.2 Syntax Error/Warning Messages

PCA tries to match the syntax error and warning messages of the compiler with which it runs. A file that would cause the compiler to issue a syntax error should cause PCA to issue a syntax error.

When PCA finds a syntax error, it stops reading the input file after it finishes reading the current function definition. PCA does not send the problem function to the output file, so only code without syntax errors appears in the transformed code file.

When illegal syntax (or any other error) is found, PCA writes a message to *standard error*. For example, if the code contains an undeclared variable, you get the error:

```
Error: line 13: file example_9_2.c: idx undefined.
   PCA -- Syntax Errors Detected
```

PCA also writes syntax warning messages to *standard error*, but optimization proceeds. PCA issues syntax warning messages for constructs that are illegal but whose intent is clear.

Appendix A

Improving PCA Performance

This appendix is designed to help you improve PCA’s performance for a particular application. **Table A−1** , which follows, lists common goals and offers suggestions for possible improvements.

Goal	Action
Recognize reductions and recurrences as safe to run in parallel.	Turn on *roundoff* option.
Convert more loops to run in parallel.	Turn up *optimize* option. Use *arl* option. Turn on *roundoff* option. Use directives.
Prevent PCA from converting to parallel execution a large number of inner loops containing a small number of iterations.	Use *machine=o*.
Eliminate dusty−deck transformations.	Turn down *scalaropt* option
Create a more informative listing.	Use *-lo=ls* or other listing options under the *listoptions* command−line option. (See the list option description for how to get a listing file.)
Force PCA to ignore assumed data dependences and convert the loop to run in parallel.	Use *#pragma concurrent*
Allow PCA to convert loops to run in parallel even though those loops contain function calls.	Enable in−lining or interprocedural analysis, or use *no side effects* or *concurrent call* directives

Table A−1 Improving PCA Performance

PCA is a tool to optimize C code and, as with any tool, it performs best when you are familiar with its features and the details of how it works. The PCA default settings can usually improve the performance of your code significantly. However, you can sometimes get larger performance improvements if you know when to use directives and alternate option settings.

Appendix B

Data−Dependence Analysis

This appendix provides a brief explanation of *data dependence* - the criterion that PCA uses to determine if a given loop should run in parallel. PCA determines dependencies between variables and arrays in loop iterations, and bases decisions on this information automatically, informing the user (via the listing file) only of those dependencies that prevent optimization.

PCA uses a data−dependence graph that shows where data values are generated and where they are used within a loop or loop nest. The data−dependence graph is processed (with simple graph traversal techniques) to find potential problem areas, which appear as cycles in the graph. Each data−dependence cycle is carefully examined to see if it can be broken and the loop executed in parallel, or if part or all of the loop must be executed serially.

For a list of in−depth studies on data dependence, see the list of reference material in the Introduction.

B.1 Varieties of Data Dependence

The three kinds of data dependence are flow dependence, anti−dependence, and output dependence. The notation *S1*, *S2*, and *SN* denote statement 1, statement 2, and statement *N*, respectively. In each example, *S2* is dependent on *S1*, due to variable *x*.

B.1.1 Flow Dependence

Data dependence from an assignment to a use of a variable is called flow dependence (or true dependence).

For Example:

```
(S1):     x = 3;
(S2):     y = x;
```

B.1.2 Anti−Dependence

Data dependence from use of a variable to a later reassignment of that variable is called anti−dependence.

For Example:

```
(S1):    y = x;
(S2):    x = 3;
```

B.1.3 Output Dependence

Data dependence from an assignment of a variable to a later reassignment of that variable is called output dependence.

For Example:

```
(S1):    x = 3;
(SN):    x = 4;
```

B.2 Input and Output Sets

To determine data dependence, it is necessary to form the input and output sets for the given statements.

The input set, *IN(S1)*, denotes the set of input items of *S1* (items whose values may be read by *S1*). The output set, *OUT(S1)*, denotes the set of output items (scalar variables or array elements) of statement *S1* (items whose values may be changed by *S1*). The *IN* and *OUT* sets for the assignment statement in the loop are:

```
        for (i=1; i<=10; i++
S1:   x[i] = a[i + 1] * b;
        IN[S1] = {a[2], a[3], a[4], ..., a[11], b}
        OUT[S1] = {x[1], x[2], x[3], ..., x[10]}
```

In practice, PCA often approximates the *IN* and *OUT* sets because the actual loop bounds are frequently unknown at compile time.

B.3 Data−Dependence Relations

For any two statements, *S1* and *S2*, one of the three types of data−dependence relations may be true, or the statements may be *data independent*.

If some item *X* is in *OUT(S1)* and *X* is in *IN(S2)* and *S2* is to use the value of *X* computed in *S1*, then *S2* is flow dependent on *S1*, as in example 1 of **Section B.1, "Varieties of Data Dependence."**

If some item *x* is in *IN(S1)* and *x* is in *OUT(S2)*, but *S1* is to use the value of *x* before it is changed by *S2*, then *S2* is anti−dependent on *S1*, as in example 2 of **Section B.1, "Varieties of Data Dependence."**

If some item *x* is in *OUT(S1)* and *x* is in *OUT(S2)* and the value computed by *S2* is to be stored after the value computed by *S1* is stored, *S2* is output dependent on *S1*, as in example 3 of **Section B.1, "Varieties of Data Dependence."**

Anti−dependence and output−dependence relations are sometimes inadvertently caused by programmers' coding practices. These dependencies can often be removed by more careful coding.

B.4 Data−Dependence Direction Vectors

The direction vector is defined as a sequence of direction vector elements (one element for each loop enclosing both statements involved in the dependence arc).

The following symbols are direction vector elements:

```
<     =     >     <=     >=     <>     *
```

The next example shows a loop:

```
      Loops      Line
+---------       10    for ( i=1; i<n ; i++) {
|                11      a[i] = b[i] + c[i];
|                12      c[i] = a[i-1] - 1;
|_________       13    }
```

If line 12 in iteration *I"* depends on line 11 in iteration *I'*, the element of the direction vector for loop I is shown in **Table B−1** .

Direction Vector Element	When
<	I' must be < I"
=	I' must be = I"
>	I' must be > I"
<=	I' must be < or = I"
>=	I' must be > or = I"
<>	I' must not = I"
*	no relation between I' and I" can be proven

Table B−1 Direction Vector Elements

In the previous example, the dependence for variable a has a direction vector of <, because the dependence flows from iteration *I'* to iteration *I'+1* and *I'* < *I'+1*. For example, the dependence on a[1] flows from iteration 1 to iteration 2, and 1 < 2. The data dependence for the variable c has a direction vector of = because the dependence stays in the same iteration of the loop (from iteration *I'* to iteration *I"*).

B.5 Loop−Carried Dependence

A dependence is said to be carried by a loop if the corresponding direction vector element for that loop has a directional component (<, <=, <>, or *). Loop−carried dependence is an important concept for discovering when the iterations of the loop can be executed concurrently. If no loop−carried dependencies exist, all iterations of that loop can be executed in parallel without synchronization.

B.6 Data−Dependence Example

The following loop cannot be vectorized or concurrentized directly.

```
for ( i=1; i<=n; i++) {
     a[i] = b[i] + 2;
     c[i] = a[i+1] + d[i];
}
```

An anti−dependence on a exists from the second assignment statement to the first. If this loop were directly concurrentized, some executions of the first statement would precede those of the second, and the anti−dependence would be violated. (The values *a*[2] through *a*[*N*] would be incorrect.)

Appendix C

Run Time Environment Variables

Table C−1 lists the run time environment variables.

Variable	Default	Description
MPC_BLOCKTIME	1000000	Thread wait time before blocking
MPC_BLOCKTYPE	SLEEP	Thread action after waiting (YIELD or SLEEP)
MPC_CHUNK[SIZE]	none	Size of loop chunks
MPC_GANG	ON	Control of gang scheduling (ON or OFF)
MPC_MAX_BLOCKS	64	Maximum control blocks in parallel region
MPC_NUM_THREADS	# CPUs	Number of parallel threads
MPC_SCHEDTYPE(or INTERLEAVE)	simple	Loop schedule type (SIMPLE, DYNAMIC, GSS,

Table C−1 Run Time Environment Variable