MIPSpro™ Automatic Parallelizer Programmer's Guide

Document Number 007-3572-001

CONTRIBUTORS

Written by Don Moccia
Illustrated by Martha Levine
Production by Julie Sheikman
Engineering contributions by Dror Maydan, Robert Cox, and Marty Itzkowitz.
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

MIPSpro™ Automatic Parallelizer Programmer's Guide
Document Number 007-3572-001

Contents

List of Examples

List of Figures

List of Tables

About This Guide

This guide describes how to use the MIPSpro™ Automatic Parallelizer to automatically detect and exploit parallelism in Fortran 77, Fortran 90, C, and C++ programs. The automatic parallelizer (AP) is an optional extension to MIPSpro 7.2 and newer compilers generating objects for the N32 and N64 ABIs. For information about parallelization for O32 objects, refer to the guides listed under "Parallelization for O32 Compilers" on page xvi. See the ABI(5) reference pages for information about N32, N64, and O32 objects.

What This Guide Contains

This guide contains the following chapters:

Chapter 1, "MIPSpro Automatic Parallelization," describes the automatic parallelizer, how to invoke it, and how to control its output.

Chapter 2, "Understanding Incomplete Optimization," explains how to recognize and avoid problems using the AP.

Chapter 3, "Assisting the MIPSpro Automatic Parallelizer," explains the directives, assertions, and pragmas recognized by the AP.

Conventions Used in This Guide

This guide uses the following conventions and symbols:

Bold	Indicates literal command-line options, filenames, keywords, function and subroutine names, pathnames, and directory names.
Italics	Represents user-defined values. Replace the item in italics with a legal value. Italics are also used for command names and manual titles.

`Courier`	Indicates command syntax, program listings, computer output, and error messages.
[ ]	Enclose optional command arguments.
( )	Surround arguments or are empty if the function has no arguments following function or subroutine names. Surround reference page section in which the command is described.
{ }	Enclose two or more items from which you must specify exactly one.
|	Separates two or more optional items.
...	Indicates that the preceding optional items can appear more than once in succession.
`%`	Shell prompt for users other than the superuser.

Here is an example illustrating the syntax conventions:

```
f77 options -pfa[{list|keep}] [-mplist] filename
```

The previous syntax statement indicates that

- you must use the command *f77*
- you may use one or more *options*, each separated by a space
- you must use the option **-pfa**
- you can specify **list** or **keep**
- you can specify **-mplist**
- you must use a *filename*

The following statements are valid examples of the described syntax:

```
f77 -O3 -n32 -mips4 -c -pfa -mplist myProg.f
f77 -O3 -n32 -mips4 -pfa list test.f -c.
```

Suggestions for Further Reading

This document describes using the MIPSpro compilers to achieve automatic parallelization for N32 and N64 objects. Related readings on other types of optimization are listed in the following sections.

Manual Parallelization References

For details about adding manual parallelization directives to label parallel loops and code regions, refer to the following:

- Chapters 5-7 of the *MIPSpro Fortran 77 Programmer's Guide*
- Chapters 4 and 5 of the *MIPSpro 7 Fortran 90 Command and Directives Reference Manual*
- Chapter 11, "Multiprocessing C/C++ Compiler Directives," of the *C Language Reference Manual*

References on Optimization Techniques

For details about different kinds of optimization, refer to the following:

- *MIPSpro Compiling and Performance Tuning Guide.* This guide details the components of the MIPSpro compiler system, other programming tools and interfaces, and dynamic shared objects. It also explains ways to improve program performance when using N32, 64-bit, or O32 object code.
- *Developer Magic: WorkShop Pro MPF User's Guide*. This guide describes how to use Developer Magic™: WorkShop Pro MPF, a graphical tool for analyzing the structure and parallelization of multiprocessing Fortran 77 applications.
- *Developer Magic: Performance Analyzer User's Guide*. This guide describes how to use the Performance Analyzer, a group of tool for analyzing program performance.
- *SpeedShop User's Guide*. This guide describes how to use the SpeedShop performance tools to analyze a program's performance.
- Wolfe, Michael. *High Performance Compilers for Parallel Computing*. Redwood City: Addison-Wesley Publishing Company, 1996. This text covers the principles and techniques of parallel compiler optimization.

Parallelization for O32 Compilers

The automatic parallelizer is available starting with the 7.2 release of the MIPSpro compilers. It does not cover O32 object code. If you are using O32, you should refer to the following references:

- *IRIS Power C User's Guide*. This guide describes how to use IRIS Power C ™, a C compiler that analyzes sequential code to determine where loops can run in parallel and generates object code that can use multiple processors.
- *POWER Fortran Accelerator User's Guide*. This guide describes how to use the POWER Fortran Accelerator ™ (PFA), a source-to-source preprocessor that enables you to efficiently run existing Fortran 77 programs on Silicon Graphics POWER Series ™ multiprocessor systems.

Chapter 1

MIPSpro Automatic Parallelization

This chapter discusses automatic parallelization and how to achieve it with the Silicon Graphics® MIPSpro Automatic Parallelizer (AP). The automatic parallelizer, new with the release of the MIPSpro 7.2 compilers, is an optional software product for use with N32 and N64 objects. It is an extension to the four compilers listed in the left column of Table 1-1, not a source-to-source preprocessor as was used prior to the 7.2 release. If the AP is installed, the compilers are known as *auto-parallel compilers* and are referred to by the names in the right column.

Table 1-1 MIPSpro 7.2 Compilers

Standard Compilers	Compilers With the AP Option
MIPSpro Fortran 77	MIPSpro Auto-Parallel Fortran 77
MIPSpro Fortran 90	MIPSpro Auto-Parallel Fortran 90
MIPSpro C	MIPSpro Auto-Parallel C
MIPSpro C++	MIPSpro Auto-Parallel C++

This chapter contains these sections:

- "Understanding Automatic Parallelization" on page 1 introduces parallelization in the MIPSpro compilers.
- "About the MIPSpro Automatic Parallelizer" on page 2 describes the new automatic parallelizer's features.
- "Using the MIPSpro Automatic Parallelizer" on page 4 tells how to use the AP to achieve parallelization.

Understanding Automatic Parallelization

Parallelization is the process of analyzing sequential programs for parallelism and restructuring them to run efficiently on multiprocessor systems. The goal is to minimize the overall computation time by distributing the computational workload among the available processors. Parallelization can be automatic or manual.

During *automatic parallelization*, the AP part of the compiler analyzes and restructures the program with little or no intervention by you. The AP automatically generates code that splits the processing of loops among multiple processors. An alternative is *manual parallelization*, in which you perform the parallelization using compiler directives and other programming techniques. Manual parallelization is discussed in the references listed under “Manual Parallelization References” on page xv. The introduction also contains useful optimization references under “References on Optimization Techniques” on page xv.

About the MIPSpro Automatic Parallelizer

The MIPSpro Automatic Parallelizer helps exploit the parallelism in programs to provide better performance on multiprocessor systems. It is a compiler extension controlled with flags in the command lines that invoke the MIPSpro auto-parallel compilers. The AP can be used on single processor or multiprocessor systems. Although they suffer a slight runtime performance penalty on single processor systems, parallelized programs can be created and debugged on any Silicon Graphics system with the AP and 7.2 compilers.

Starting with the 7.2 release, the MIPSpro auto-parallel compilers integrated automatic parallelization, provided by the AP, with the other compiler optimizations, such as interprocedural analysis (IPA) and loop nest optimization (LNO). Whereas previous releases relied on source-to-source preprocessors, the new version internalizes automatic parallelization into the optimizer of the MIPSpro compilers. As seen in Figure 1-1, the AP works on an intermediate representation generated during the compiling process. This integration provides several benefits:

- Automatic parallelization is integrated with the optimizations for single processors.
- The options and pragmas of the AP are consistent with the MIPSpro compilers.
- Support for C++ is now possible.
- The runtime and compile-time performance is improved.

These benefits were not possible with the earlier MIPSpro compilers, which achieved parallelization by relying on the Power Fortran and Power C preprocessors to provide source-to-source conversions before compilation.

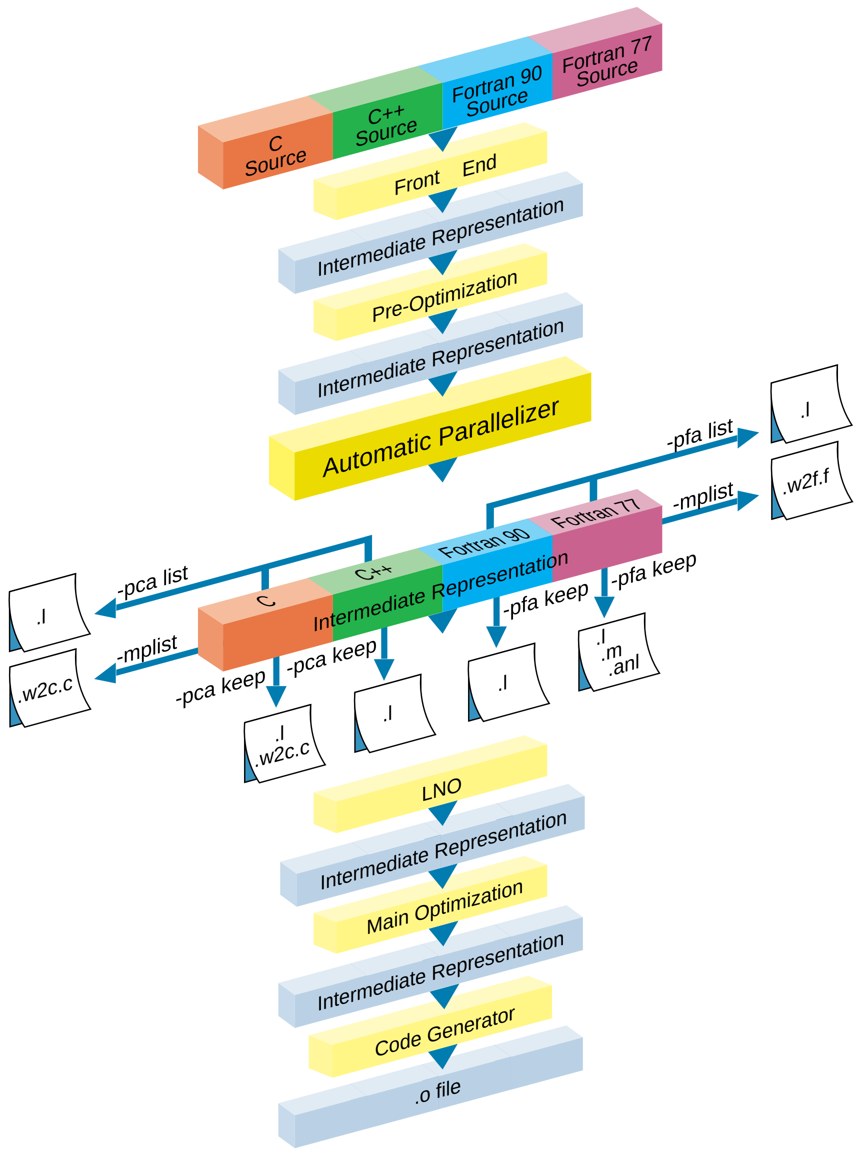

Figure 1-1 Files Generated by the Automatic Parallelizer

Using the MIPSpro Automatic Parallelizer

This section describes how to use the MIPSpro Automatic Parallelizer to compile and run parallelized programs.

- “Invoking the Automatic Parallelizer” on page 4 demonstrates how to compile parallelized programs in the auto-parallel versions of Fortran 77, Fortran 90, C, and C++.
- “MIPSpro Compiler Command-Line Options” on page 6 describes options relevant to the AP.
- “Understanding The Automatic Parallelizer Output Files” on page 10 explains how to use the listing mechanisms of the AP to see what transformations were performed and what regions of codes were parallelized.
- “Running Your Program” on page 14 shows how to run your parallelized program.

Invoking the Automatic Parallelizer

You invoke the automatic parallelizer by including the **-pfa** or **-pca** flag on the command line that starts a MIPSpro auto-parallel compiler. Additional flags allow you to generate reports to aid in debugging. The syntax for compiling programs with the AP is as follows:

- For Auto Parallel Fortran 77 and Auto Parallel Fortran 90 use **-pfa**:

```
f77 options -pfa[{list|keep}] [-mplist] filename
f90 options -pfa[{list|keep}]           filename
```

- For Auto Parallel C and Auto Parallel C++ use **-pca**:

```
cc options -pca[{list|keep}] [-mplist] filename
CC options -pca[{list|keep}]           filename
```

The command-line entries are defined as follows:

*option*s	The MIPSpro compiler command-line options. -O3 is recommended for using the AP. For details, see “MIPSpro Compiler Command-Line Options” on page 6, and the documentation for your MIPSpro compiler.
-pfa or **-pca**	Invoke the automatic parallelizer.

list	Produce a *.l* file, a listing of those parts of the program that can run in parallel and those that cannot. This file is discussed in "About the .l File" on page 11.
keep	Generate the listings shown in Table 1-2. Because of data conflicts, do not use with **-mplist**, or the LNO options **-FLIST** and **-CLIST**. See "Loop Nest Optimizer Options" on page 8.
-mplist	Generate the parallelized equivalent program in Fortran 77 (a *.w2f.f* file), or in C (a *.w2c.c* file). These files are discussed in the section "About the .w2f.f and .w2c.c Files" on page 12. Do not use with **-pfa keep** and **-pca keep**, or **-FLIST** and **-CLIST**.
filename	The name of the file containing the source code.

The files generated by **-pfa keep** and **-pca keep** are shown in Table 1-2. The *.l* file is the same as that generated using **-pfa list** or **-pca list**, and the *.w2c.c* is the same as that generated using **-mplist**. The other two files are the *.m* file, a parallelized equivalent program, and a *.anl* file, a file to be used with WorkShop Pro MPF. These files are explained in the section "About the .m and .anl Files" on page 14.

Table 1-2 Files Generated by -pfa keep and -pca keep

File Suffix	f77	f90	cc	CC
.l	Yes	Yes	Yes	Yes
.m	Yes	No	No	No
.anl	Yes	No	No	No
.w2c.c	No	No	Yes	No

Consider a typical command line:

```
f77 -O3 -n32 -mips4 -c -pfa -mplist myProg.f
```

This command uses Auto-Parallel Fortran 77 to compile (**-c**) the file *myProg.f* with the MIPSpro compiler options **-O3**, **-n32**, and **-mips4**. The option **-n32** requests an object with an N32 ABI; **-mips4** requests that the code be generated with the MIPS IV instruction set. You can find out more about these options in the *MIPSpro Compiling and Performance Tuning Guide*. Using **-O3**, which requests aggressive optimization, is recommended for using the AP. It is covered in "Optimization Options" on page 7. Using **-mplist** requests that a parallelized Fortran 77 program be created in the file *myProg.w2f.f*. If you are using

WorkShop Pro MPF, you may want to substitute **-keep** for **-mplist** because of the *.anl* file that would result.

To use the automatic parallelizer correctly, remember these points:

- The AP can only be used with -n32 or -64 compiles. The **-pfa** and the **-pca** flags invoke the older automatic parallelizers with -o32 compiles.
- If you link separately, you must have one of the following in the link line:
  - the **-pfa** flag
  - the **-pca** flag
  - the **-mp** option (See the *MIPSpro Fortran 77 Programmer's Guide.*)
- Because of data set conflicts, you can use only one of the following in a compilation:
  - **-pfa keep** or **-pca keep**
  - **-mplist**
  - **-FLIST** or **-CLIST** (See "Loop Nest Optimizer Options" on page 8.)

MIPSpro Compiler Command-Line Options

Prior to MIPSpro 7.2, parallelization was done by the Power Fortran and Power C preprocessors, which had their own set of flags. With MIPSpro 7.2, the automatic parallelizer does the parallelization and recognizes the same options as the compilers. This has reduced the number of options you need to know and has simplified their use. For example, suppose you are using Auto-Parallel Fortran 77 and want to turn off round-off changing transformations in all phases of compiling. In MIPSpro 7.2, specifying **-OPT:roundoff=0** does this. Previously, you also needed to add the option **-pfa,-r=0** to turn off round-off changing transformations in the Power Fortran preprocessor.

The next sections cover the MIPSpro compiler command-line options most commonly needed with the AP:

- "Optimization Options" on page 7
- "Interprocedural Analysis" on page 7
- "Loop Nest Optimizer Options" on page 8
- "Miscellaneous Optimization Options" on page 9

For more extensive information about compiler command-line options, see the *MIPSpro Compiling and Performance Tuning Guide* and the guide for your compiler.

Optimization Options

Optimization option **-O3** performs aggressive optimization and is recommended to run the AP. The optimization at this level maximizes code quality even if it requires extensive compile time or relaxing language rules. **-O3** uses transformations that are usually beneficial but can hurt performance in pathological cases. This level may cause noticeable changes in floating-point results due to the relaxation of operation-ordering rules. Floating-point optimization is discussed further in “Miscellaneous Optimization Options” on page 9.

Interprocedural Analysis

Interprocedural analysis (IPA), invoked by the **-IPA** command-line option, performs program optimizations that can be done only with knowledge of the whole program. Typical IPA optimizations are

- procedure inlining
- identification of global constants
- dead function elimination
- dead variable elimination
- dead call elimination
- interprocedural alias analysis
- interprocedural constant propagation

More information about these optimizations can be found in the books listed under “References on Optimization Techniques” on page xv.

Using IPA, the automatic parallelizer is able to optimize some loops that contain function calls, as described in “Function Calls in Loops” on page 18. In the MIPSpro 7.2 release, using the AP with the **-IPA** option to parallelize loops containing function calls is only modestly successful.

Note: If IPA expands subroutines inline in a calling routine, the subroutines are compiled with the options of the calling routine. If the calling routine is not being compiled with **-pfa** or **-pca**, no inlined subroutine is parallelized. This is true even if the subroutines are compiled separately with **-pfa** or **-pca**, because automatic parallelization does not occur until after IPA finishes.

Loop Nest Optimizer Options

The loop nest optimizer (LNO) performs loop optimizations that better exploit caches and instruction-level parallelism. Some of the optimizations of the LNO are

- loop interchange
- loop fusion
- loop fission
- cache blocking and outer loop unrolling

The LNO runs when you use the **-O3** option. It is an integrated part of the compiler back end, not a preprocessor. As with the AP, the same optimizations and the same control options apply to Fortran, C, and C++ programs. As described in the *MIPSpro Compiling and Performance Tuning Guide*, there are three LNO options of particular interest to users of the AP:

- **-LNO:parallel_overhead=***n*. This option controls the auto-parallel compiler's estimate of the overhead incurred by invoking parallel loops. The default value for *n* varies on different systems, but is typically in the low thousands of processor cycles.
- **-LNO:auto_dist=on**. This option requests that the AP insert data distribution directives to provide the best memory utilization on the S2MP™ (Scalable Shared-Memory Parallel) architecture of the Origin2000™ platform. There is more information about this option in your compiler's reference pages.
- **-LNO:ignore_pragmas**. This option causes the AP to ignore all of the directives, assertions, and pragmas covered in "Directives, Assertions, and Pragmas for Automatic Parallelization" on page 30. This includes the **C*$* NO CONCURRENTIZE** directive.

If you are using Fortran 77 or C, you can view the transformed code in the original source language after LNO performs its transformations. Two translators, integrated into the back end, convert the compiler's internal representation into the original source language. You can invoke the desired translator by using the *f77* option **-FLIST:=on** or the *cc* option **-CLIST:=on**. For example,

```
f77 -O3 -FLIST:=on test.f
```

creates an *a.out* object file and the Fortran file *test.w2f.f*. This *.w2f.f* file differs somewhat from a *.w2f.f* file generated by the **-mplist** option because it is generated at a later stage of the compilation. You can read the *.w2f.f* file, which is a compilable Fortran representation of the original program after the LNO phase. Because LNO is not a preprocessor, recompiling the *.w2f.f* file may result in an executable that differs from the original compilation of the *.f* file.

Miscellaneous Optimization Options

Miscellaneous optimizations, controlled by the **-OPT** command-line option, are those not associated with a distinct compiler phase. Two of these optimizations are particularly relevant to the AP:

- **-OPT:alias=***name*
- **-OPT:roundoff=***n*

The **-OPT:alias=***name* option has several variations. One of interest to a user of the AP is **-OPT:alias=restrict**. Under this option, the compiler assumes a very restrictive model of aliasing: Memory operations dereferencing different named pointers are assumed neither to alias with each other, nor to alias with any named scalar variable. Even explicit assignments, such as the one below, are forbidden:

```
int *i, *j;
i = j;
```

Consider this code:

```
void dbl (int *i, int *j){
   *i = *i + *i;
   *j = *j + *j; }
```

The compiler assumes that *i* and *j* point to different memory locations, and can produce an overlapped schedule for the two calculations. See also “Aliased Parameter Information” in Chapter 2 about using **__restrict** as an alternative approach to aliasing.

A related option is **-OPT:alias=disjoint**. Under this option, memory operations dereferencing different named pointers are assumed not to alias with each other, and different dereferencing depths of the same pointer are assumed not to alias with each other. Thus, if *p* and *q* are pointers, **p* does not alias with **q*, ***p*, or ***q*.

The **-OPT:roundoff=***n* option controls floating-point accuracy and the behavior of overflow and underflow exceptions relative to the source language rules. The default for **-O3** optimization is **-OPT:roundoff=2**. This setting allows transformations with extensive effects on floating-point results. It allows associative rearrangement across loop iterations, and the distribution of multiplication over addition and subtraction. It disallows only transformations known to cause overflow, underflow, or cumulative round-off errors for a wide range of floating-point operands.

At the **-OPT:roundoff=2** or **3** level of optimization, the AP may change the sequence of a loop's floating-point operations in order to parallelize it. Because floating-point operations have finite precision, this change may cause slightly different results. If you want to avoid these differences by not having such loops parallelized, you must compile with the **-OPT:roundoff=0** or **1** command-line option. Consider this example:

```
REAL A, B(100)
DO I = 1, 100
  A = A + B(I)
END DO
```

At the default setting of **-OPT:roundoff=2** for the **-O3** level of optimization, the AP parallelizes this loop. At the start of the loop, each processor gets a private copy of *A* in which to hold a partial sum. At the end of the loop, the partial sum in each processor's copy is added to the total in the original, global copy. This value of *A* may be different from the value generated by a version of the loop that is not parallelized.

Understanding The Automatic Parallelizer Output Files

Merely processing a program with the AP often results in excellent parallelization. But, as described in Chapter 2, "Understanding Incomplete Optimization," there are cases that cannot be effectively parallelized automatically. To help analyze these cases, the AP provides a number of options to generate listings that describe where parallelization failed and where it succeeded. By understanding these listings, you may identify small problems that prevent a loop from being made parallel. With a little work, you can often remove these data dependences and dramatically improve the program's performance.

Tip: When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. Optimizing a routine that uses only one percent of the execution time cannot significantly improve the overall performance of your program. To determine where the program spends its execution time, you can use tools like SpeedShop and the WorkShop Pro MPF Parallel Analyzer View. More information about these tools can be found in “About the .m and .anl Files” on page 14.

About the .l File

The **-pfa list**, **-pca list**, **-pfa keep**, and **-pca keep** options generate a listing whose filename ends with *.l*. This *.l* file lists the original loops in the program along with messages telling whether or not the loops were parallelized. For loops that were not parallelized, an explanation is given.

Example 1-1 shows a simple Fortran 77 program. The subroutine is contained in a file named *testl.f*.

Example 1-1 Subroutine in File testl.f

```
SUBROUTINE sub(arr, n)
  REAL*8 arr(n)
  DO i = 1, n
    arr(i) = arr(i) + arr(i-1)
  END DO
  DO i = 1, n
    arr(i) = arr(i) + 7.0
    CALL foo(a)
  END DO
  DO i = 1, n
    arr(i) = arr(i) + 7.0
  END DO
END
```

When *testl.f* is compiled with

```
f77 -O3 -n32 -mips4 -pfa list testl.f -c.
```

the automatic parallelizer produces the file *testl.l*, shown in Example 1-2.

Example 1-2 Listing in File testl.l

```
Parallelization Log for Subprogram sub_
3: Not Parallel
         Array dependence from arr on line 4 to arr on line 4.
6: Not Parallel
         Call foo on line 8.
10: PARALLEL (Auto) __mpdo_sub_1
```

The last line (10) is important to understand. Whenever a loop is run in parallel, the parallel version of the loop is put in its own subroutine. The MIPSpro profiling tools attribute all the time spent in the loop to this subroutine. The last line tells us that the name of the subroutine is **__mpdo_sub_1**. For more information, you can refer to Chapter 2, "Understanding Incomplete Optimization," or "Suggestions for Further Reading" on page xiv.

About the .w2f.f and .w2c.c Files

The **-mplist** option compiles a program and generates a *.w2f.f* file for auto-parallel Fortran 77, or a *.w2c.c* file for auto-parallel C. Because it is generated at a earlier stage of the compilation, this *.w2f.f* file is much more easily understood than the *.w2f.f* file generated by the **-FLIST:=on** option. The **-pca keep** option also create a *.w2c.c* file. These files do not exist for the auto-parallel versions of Fortran 90 and C++. The files mimic the behavior of the program after automatic parallelization. The compilers create the files by invoking the appropriate translator to turn the compiler's internal representation into C or Fortran 77. The representations are designed to be readable so that you can see what portions of the code were not parallelized. You can then use this information to change the original program. In most cases, the files contain valid code that can be recompiled.

Note: Compiling a *.w2f.f* or *.w2c.c* file with a standard MIPSpro compiler does not produce exactly the same code that is generated by a MIPSpro auto-parallel compiler processing the original source. Because it is an internal phase of the MIPSpro compilers, not a source-to-source preprocessor, the AP does not normally use a *.w2f.f* or *.w2c.c* file to generate the object file.

Consider the subroutine in Example 1-3, contained in a file named *testw2.f*.

Example 1-3 Subroutine in File testw2.f

```
SUBROUTINE trivial(a)
  REAL a(10000)
  DO i = 1,10000
    a(i) = 0.0
  END DO
END
```

After compiling *testw2.f* using

```
f77 -O3 -n32 -mips4 -c -pfa -mplist testw2.f
```

you get an object file, *testw2.o*, and a file, *testw2.w2f.f*, that contains the code shown in Example 1-4.

Note: WHIRL is the name for the compiler's intermediate representation.

Example 1-4 Listing in File testw2.w2f.f

```
C ***********************************************************
C Fortran file translated from WHIRL Mon Jul  7 16:53:44 1997
C ***********************************************************


        SUBROUTINE trivial(a)
        IMPLICIT NONE
        REAL*4 a(10000_8)
C
C       **** Variables and functions ****
C
        INTEGER*4 i
C
C       **** statements ****
C
C       DOACROSS will be converted to SUBROUTINE __mpdo_trivial_1
C$DOACROSS local(i), shared(a)
        DO i = 1, 10000, 1
          a(i) = 0.0
        END DO
        RETURN
        END ! trivial
```

As explained in “About the .l File” on page 11, parallel versions of loops are put in their own subroutines. In this example, that subroutine is **__mpdo_trivial_1**. The **C$DOACROSS** directive is described with all the other manual parallelization directives in Chapter 5, “Fortran Enhancements for Multiprocessors,” of the *MIPSpro Fortran 77 Programmer’s Guide*, or Chapter 11, “Multiprocessing C/C++ Compiler Directives,” of the *C Language Reference Manual*. Also, refer to the references listed under “Manual Parallelization References” on page xv.

About the .m and .anl Files

For Auto-Parallel Fortran 77, the **-pfa keep** option generates two files in addition to a *.l* file:

- A *.m* file that is similar to the *.w2f.f* and *.w2c.c* files. It mimics the behavior of the program after automatic parallelization, and is also annotated with information that is used by Workshop ProMPF.
- A *.anl* file that is used by the Workshop ProMPF tool.

Silicon Graphics provides a separate product, WorkShop Pro MPF, that provides a graphical interface to aid in both automatic and manual parallelism for Fortran 77. In particular, the WorkShop Pro MPF Parallel Analyzer View helps you understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. Refer to the *Developer Magic: WorkShop Pro MPF User’s Guide* and the *Developer Magic: Performance Analyzer User’s Guide* for details.

Another Silicon Graphics product is SpeedShop, which allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop consists of a set of commands that can be run in a shell, an API, and a number of libraries to support the commands. For more information, see the *SpeedShop User’s Guide*.

Running Your Program

Running a parallelized version of your program is no different from running a sequential one. The same binary can be executed on different numbers of processors. The default is to have the runtime environment select how many processors to use based on the number available.

You can change the default behavior by setting the environmental variable MP_SET_NUMTHREADS, which tells the system to use an explicit number of processors. The statement

```
% setenv MP_SET_NUMTHREADS 2
```

causes the program to create two threads regardless of the number of processors available. For compatibility with older releases, the environment variable NUM_THREADS is supported as a synonym for MP_SET_NUMTHREADS.

By setting the environment variable MP_SUGNUMTHD,

```
% setenv MP_SUGNUMTHD
```

you can dynamically vary the number of processors used from loop to loop based on the system load. Setting MP_SUGNUMTHD causes the runtime library to create an additional, asynchronous process that periodically wakes up and monitors the system load. When idle processors exist, this process increases the number of threads, up to a maximum of MP_SET_NUMTHREADS. When the system load increases, the process decreases the number of threads, possibly to 1. When MP_SUGNUMTHD is not set, this feature is disabled and multithreading works as before. Refer to the *MIPSpro Fortran 77 Programmer's Guide* for details.

Chapter 2

Understanding Incomplete Optimization

This chapter discusses programming practices that prevent the MIPSpro Automatic Parallelizer (AP) from fully optimizing your code. The AP is not always able to parallelize programs effectively. Sometimes this is unavoidable, but other times you can help it improve its effectiveness. There are three general categories of incomplete optimization:

- The automatic parallelizer does not detect that a loop is safe to parallelize, as described in “Failure to Parallelize Safe Loops” on page 17.
- The automatic parallelizer chooses the wrong nested loop to make parallel, as covered in “Parallelizing the Wrong Loop” on page 24.
- The automatic parallelizer parallelizes a loop, incurring unnecessary overhead, as discussed in “Unnecessary Parallelization Overhead” on page 27.

Failure to Parallelize Safe Loops

A program’s performance may be severely constrained when the automatic parallelizer does not recognize that a loop is safe to parallelize. A loop is safe if there isn’t a data dependence, such as a variable being assigned in one iteration of a loop and used in another. The AP analyzes every loop in a sequential program. If it cannot prove a loop is safe, it does not parallelize that loop.

The AP often does not parallelize loops containing any of the constructs described in this section. These constructs result from the following practices:

- “Function Calls in Loops” on page 18
- “GO TO Statements in Loops” on page 18
- “Complicated Array Subscripts” on page 19
- “Conditionally Assigned Temporary Nonlocal Variables in Loops” on page 20
- “Unanalyzable Pointer Usage in C and C++” on page 21

However, in many instances such loops can be automatically parallelized after minor changes. Reviewing your program's *.l* file, described in "Understanding The Automatic Parallelizer Output Files" on page 10, will show you if any of these constructs are in your code.

Function Calls in Loops

By default, the automatic parallelizer does not parallelize a loop that contains a function call because the function in one iteration may modify or depend on data in other iterations of the loop. However, a couple of tools can help with this problem.

- Interprocedural analysis (IPA), specified by the **-IPA** command-line option, can provide the automatic parallelizer with enough information to parallelize some loops containing subroutine calls. Because it adds a lot of compiling time, using interprocedural analysis with the AP may not be worthwhile if your code does not have many subroutine calls within loops. Although of limited value in the MIPSpro 7.2 release, the combined performance of IPA and the automatic parallelizer will be improved in future releases. For more information on IPA, see "Interprocedural Analysis" in Chapter 1 and the *MIPSpro Compiling and Performance Tuning Guide*.
- As discussed in "C*$* ASSERT CONCURRENT CALL and #pragma concurrent call" on page 33, you can tell the AP to ignore the dependences of function calls when analyzing the specified loops by using the following:
  - the assertion **C*$* ASSERT CONCURRENT CALL** for Auto-Parallel Fortran 77 and Auto-Parallel Fortran 90
  - the **#pragma concurrent call** for Auto-Parallel C and Auto-Parallel C++

GO TO Statements in Loops

GO TO statements cause unstructured control flows. The AP converts most unstructured control flows in loops into structured flows that can be parallelized. However, using **GO TO** in loops can still cause two problems:

- Unstructured control flows the AP cannot restructure. You must either restructure or manually parallelize the loops containing flows.
- Early exits from loops. Loops with early exits cannot be parallelized, either automatically or manually.

Complicated Array Subscripts

There are three cases where array subscripts are too complicated to permit parallelization:

- The subscripts are indirect array references.
- The subscripts are unanalyzable.
- The subscripts rely on hidden knowledge.

Indirect Array References

The automatic parallelizer is not able to analyze indirect array references. Consider the following Fortran example:

```
DO I = 1, N
  A(IB(I)) = ...
END DO
```

This loop cannot be run safely in parallel if the indirect reference *IB(I)* is equal to the same value for different iterations of *I*. If every element of array *IB* is unique, the loop can safely be made parallel. To achieve parallelism in such cases, you can use either manual or automatic methods to achieve parallelism. For automatic parallelization, the **C*$* ASSERT PERMUTATION** directive and the **#pragma permutation** (discussed in “C*$* ASSERT PERMUTATION (array_name) and #pragma permutation (array_name)” on page 35) would be appropriate.

Unanalyzable Subscripts

The automatic parallelizer cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain four elements:

- literal constants: 1, 2, 3, …
- variables: I, J, K, …
- the product of a literal constant and a variable, such as N*5 or K*32
- a sum of any combination of the first three items, such as N*21 +K +251

In the following case, the AP is not able to analyze the division operator (/) in the array subscript and cannot reorder the loop:

```
DO I = 2, N, 2
  A(I/2) = ...
END DO
```

Hidden Knowledge

In the following example there may be hidden knowledge about the relationship between the variables *M* and *N*:

```
DO I = 1, N
  A(I) = A(I+M)
END DO
```

The loop can be run in parallel if $M > N$, because the array reference will not overlap. However, because the AP does not know the value of the variables, it cannot make the loop parallel. To get around this problem, you can use the **C*$* ASSERT DO (CONCURRENT)** assertion or **#pragma concurrent** (documented in "C*$* ASSERT DO (CONCURRENT) and #pragma concurrent" on page 32). You can also use manual parallelization.

Conditionally Assigned Temporary Nonlocal Variables in Loops

When parallelizing a loop, the AP often localizes (privatizes) temporary scalar and array variables. Consider the following example:

```
DO I = 1, N
  DO J = 1, N
    TMP(J) = ...
  END DO
  DO J = 1, N
    A(J,I) = A(J,I) + TMP(J)
  END DO
END DO
```

The array *TMP* is used for local scratch space. To successfully parallelize the outer (*I*) loop, each processor must be given a distinct, private *TMP* array. In this example, the AP is able to localize *TMP* and parallelize the loop.

The AP runs into trouble when a conditionally assigned temporary variable might be used outside of the loop, as in the following example.

```
SUBROUTINE S1(A, B)
  COMMON T
  ...
  DO I = 1, N
    IF (B(I)) THEN
      T = ...
      A(I) = A(I) + T
    END IF
  END DO
  CALL S2()
END
```

If the loop were to be run in parallel, a problem would arise if the value of *T* were used inside subroutine **S2()**. Which processor's private copy of *T* should **S2()** use? If *T* were not conditionally assigned, the answer would be the processor that executed iteration *N*. But *T* is conditionally assigned and the AP cannot determine which copy to use.

The loop is inherently parallel if the conditionally assigned variable *T* is localized. If the value of *T* is not used outside the loop, you should replace *T* with a local variable. Unless *T* is a local variable, the AP must assume that **S2()** might use it.

Unanalyzable Pointer Usage in C and C++

The C and C++ languages have features that make them more difficult than Fortran to automatically parallelize. These features are often related to the use of pointers, such as implementing multidimensional arrays as pointer accesses, not as true arrays. The following practices involving pointers interfere with the automatic parallelizer's effectiveness:

- arbitrary pointer dereferences
- arrays of arrays
- loops bounded by pointer comparisons
- aliased parameter information

Arbitrary Pointer Dereferences

The automatic parallelizer does not analyze arbitrary pointer dereferences. It cannot parallelize a list traversal, for example. The only pointers it analyzes are array references and pointer dereferences that can be converted into array references.

Arrays of Arrays

Multidimensional arrays are sometimes implemented as arrays of arrays. Consider this example:

```
double **p;
for (int i = 0; i < n; i++)
  for (int j = 0; J < n; j++)
    p[i][j] = ...
```

If *p* is a true multidimensional array, the outer loop can be run safely in parallel. However, if two of the array pointers, *p*[2] and *p*[3] for example, reference the same array, the loop must not be run in parallel. Although this duplicate reference is unlikely, the AP cannot prove it doesn’t exist. You can avoid this problem by always using variable length arrays. To parallelize the code fragment above, rewrite it as follows:

```
double p[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    p[i][j] = ...
```

Note: Although ANSI C does not yet allow variable-sized multidimensional arrays, MIPSpro 7.2 C and C++ already implement this proposal. See the *C Language Reference Manual* for details.

Loops Bounded by Pointer Comparisons

The AP parallelizes only those loops for which the number of iterations can be exactly determined. In Fortran programs this is rarely a problem, but in C and C++, issues related to overflow and unsigned arithmetic can come to play. For example, pointers are unsigned data types and cannot have negative values. One consequence is that loops to be parallelized should not be bounded by pointer comparisons like this:

```
int *pl, *pu;
...
for (int *p = pl; p < pu; p++)
```

In this case, it appears that the number of loop iterations is either the quantity *pu*-*p* or zero, whichever is greater. However, because they are unsigned types, *pu*-*p* must always be greater than zero, even if *p* > *pu*. Compiling this loop results in a *.l* file entry stating that the bound cannot be standardized. To avoid this result, restructure the loop to be of this form:

```
int lb, ub;
...
for (int i = lb; i < ub; i++)
```

Aliased Parameter Information

Perhaps the most frequent impediment to parallelizing C and C++ are aliases. Although Fortran guarantees that multiple parameters to a subroutine are not aliased to each other, C and C++ do not. Consider the following example:

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

This loop can be parallelized only if arrays *a* and *b* do not overlap. With the **__restrict** (two underscores) type qualifier keyword, available with the MIPSpro 7.2 C and C++ compilers, you can inform the AP that the arrays do not overlap. This keyword tells the compiler to assume that dereferencing the qualified pointer is the only way the program can access the memory pointed to by that pointer. Thus loads and stores through that pointer are not aliased with loads and stores through any other pointers.

The next example shows the **__restrict** type qualifier being used to guarantee that *a* and *b* provide exclusive access to their arrays. This assurance permits the AP to proceed with the parallelization.

```
void sub(double * __restrict a, double __restrict *b, n) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

Two more problematical ways of dealing with aliased parameter information follow:

- The keyword **restrict** behaves similarly to **__restrict**, but could clash with other meanings of **restrict** in older programs. To enable the **restrict** keyword, you must specify **-LANG:restrict** in the command line. The type qualifier **__restrict** does not require **-LANG:restrict** to be specified.

- The **-OPT:alias=***restrict* compiler option, covered in “Miscellaneous Optimization Options” on page 9, guarantees no aliasing by pointers. It is less desirable to use because it is global in scope and cannot be limited to specific variables.

Parallelizing the Wrong Loop

The AP parallelizes a loop by distributing its iterations among the available processors. When parallelizing nested loops, like *I*, *J*, and *K* in the example below, the AP distributes only one of the loops:

```
DO I = 1, L
  ...
  DO J = 1, M
    ...
    DO K = 1, N
      ...
```

Because of this restriction, the effectiveness of the parallelization of the nest depends on the loop that the automatic parallelizer chooses. In fact, the loop the AP parallelizes may be an inferior choice for any of three reasons:

- It is an inner loop.
- It has a small trip count.
- It exhibits poor data locality.

The automatic parallelizer’s heuristic methods are designed to avoid these problems. The next three sections show you how to increase the effectiveness of these methods:

- “Inner Loops” on page 24
- “Small Trip Counts” on page 25
- “Poor Data Locality” on page 25

Inner Loops

With nested loops, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from more processors processing larger sections of the program, saving synchronization and other overhead costs. Therefore, the

AP always tries to make the loop it parallelizes become the outermost loop. Failing a successful interchange, the AP will parallelize an inner loop if possible.

Checking the *.l* file, described in “About the .l File” on page 11, will tell you which loop from a nest was parallelized. If the loop was not the outermost, the reason is probably one of those mentioned in “Failure to Parallelize Safe Loops” on page 17. Because of the potential for improved performance, it is probably advantageous for you to modify your code so that the outermost loop is the one parallelized.

Small Trip Counts

The *trip count* is the number of times a loop is executed. As discussed in “Unnecessary Parallelization Overhead,” loops with small trip counts generally run faster when they are not parallelized. Consider how this fact affects the following example:

```
DO I = 1, M
  DO J = 1, N
```

The AP may try to parallelize the *I* loop because it is outermost. If *M* is very small, it would be better to interchange the loops and make the *J* loop outermost before parallelization. Because the AP often cannot know that *M* is small, you can do one of the following:

- Use one of the **DO PREFER** assertions or **prefer** pragmas, discussed in “Directives, Assertions, and Pragmas for Automatic Parallelization” on page 30, to tell the AP that it is better to parallelize the *J* loop.
- Use the manual parallelization directives contained in the “Manual Parallelization References” on page xv.

Poor Data Locality

Computer memory has a hierarchical organization. As one travels up the hierarchy, memory becomes closer to the CPU, faster, more expensive, and more limited in size. Cache memory is at the top of the hierarchy, and main memory is farther down. In multiprocessor systems, each processor has its own cache memory. It is time consuming for one processor to access another processor’s cache. Therefore, a program’s performance is best when each processor has the data it needs in its own cache.

Programs, especially those that include extensive looping, often exhibit *locality of reference*: If a memory location is referenced, there is a good chance that it or a nearby location will be referenced in the near future. Loops designed to take advantage of locality do a better job of concentrating data in memory, increasing the probability that a processor will find the data it needs in its own cache.

To see the effect of locality on parallelization, consider Example 2-1 and Example 2-2. In each case assume that the loops are parallelized and that there are *p* processors.

Example 2-1 Distribution of Iterations

```
DO I = 1, N
  ...A(I)
END DO
DO I = N, 1, -1
  ...A(I)...
END DO
```

In the first loop of Example 2-1, the first processor accesses the first *N/p* elements of *A*, the second processor accesses the next *N/p* elements, and so on. In the second loop, the distribution of iterations is reversed: The first processor accesses the last *N/p* elements of *A*, and so on. Most elements are not in the cache of the processor needing them during the second loop. This example should run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 2-2 Two Nests in Sequence

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = B(J,I) + ...
  END DO
END DO

DO I = 1, N
  DO J = 1, N
    B(I,J) = A(J,I) + ...
  END DO
END DO
```

In Example 2-2, the AP may chose to parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor would access the first *N/p* rows of *A* and the first *N/p* columns of *B*. In the second nest, the first processor accesses the first *N/p* columns of *A* and the first *N/p* rows of *B*. This example runs much more efficiently if you parallelize the *I* loop in one nest and the *J* loop in the other. You can use the **DO PREFER** assertions or the **prefer** pragmas described in “Directives, Assertions, and Pragmas for Automatic Parallelization” on page 30 to achieve this.

Unnecessary Parallelization Overhead

The parallelization of loops does not come without cost. There is overhead associated with distributing the iterations among the processors and synchronizing the results of the parallel computations. There can also be memory-related costs, such as the cache interference that occurs when different processors try to access the same data. One consequence of this overhead is that not all loops should be parallelized. As discussed in “Small Trip Counts” on page 25, loops that have a small number of iterations run faster sequentially than in parallel. Two other cases of unnecessary overhead involve

- unknown trip counts
- nested parallelism

Unknown Trip Count

When the trip count is unknown (and sometimes when it is known), the AP parallelizes the loop conditionally: Code is generated for both a parallel and a sequential version. By generating two versions, the AP can avoid running in parallel a loop that turns out to have a small trip count. The AP chooses the version to use based on the trip count, the code inside the loop’s body, the number of processors available, and an estimate of the cost to invoke a parallel loop in that runtime environment.

Note: You can control this cost estimate by using the option **-LNO:parallel_overhead=***n*. The default value of *n* varies on different systems, but a typical value is several thousand machine cycles.

Having a sequential and parallel version of the loop incurs the runtime overhead of choosing between them. You can avoid this overhead by using the **DO PREFER** assertions or **prefer** pragmas, discussed in “Directives, Assertions, and Pragmas for Automatic Parallelization” in Chapter 3, to insure that the AP knows in advance whether or not to parallelize the loop.

Nested Parallelism

Nested parallelism is not supported by the AP. Thus, for every loop that can be parallelized, the AP must generate a test that checks for the loop being called from within another parallel loop or parallel region. While this check is not very expensive, it can add overhead. The following example demonstrates nested parallelism:

```
SUBROUTINE CALLER
  DO I = 1, N
    CALL SUB
  END DO
  ...
END
SUBROUTINE SUB
  ...
  DO I = 1, N
    ...
  END DO
END
```

If the loop inside **CALLER()** is parallelized, the loop inside **SUB()** cannot be run in parallel when **CALLER()** calls **SUB()**. In this case, there is one way to avoid the test. If **SUB()** is always called from **CALLER()**, you can use the **C*$* ASSERT DO (SERIAL)** or the **C*$* ASSERT DO PREFER (SERIAL)** assertion to force the sequential execution of the loop in **SUB()**. The C and C++ equivalents are **#pragma serial** and **#pragma prefer serial**.

Chapter 3

Assisting the MIPSpro Automatic Parallelizer

This chapter discusses actions you can take to enhance the performance of the automatic parallelizer.

- "Assisting the Automatic Parallelizer" on page 29 gives strategies for optimizing your code for the automatic parallelizer.
- "Directives, Assertions, and Pragmas for Automatic Parallelization" on page 30 discusses the assertions and directives the automatic parallelizer recognizes.

Assisting the Automatic Parallelizer

There are circumstances that interfere with the automatic parallelizer's ability to optimize programs. As shown in Chapter 2, "Understanding Incomplete Optimization," problems are sometimes caused by coding practices. Other times, the AP does not have enough information to make good parallelization decisions. You can pursue three strategies to attack these problems and achieve better results with the AP.

- The first approach is to modify your code to avoid coding practices that the AP cannot analyze well. Specific problems and solutions are discussed in Chapter 2.
- The second strategy is to assist the AP with the manual parallelization directives. They are described in the *MIPSpro Compiling and Performance Tuning Guide*, and require the **-mp** compiler option. The AP is designed to recognize and coexist with manual parallelism. You can use manual directives with some loop nests, while leaving others to the AP. This approach has both positive and negative aspects.

Positive:	The manual parallelization directives are well defined and deterministic. If you use a manual directive, the specified loop will be run in parallel. **Note:** This last statement assumes that the trip count is greater than one and that the specified loop is not nested in another parallel loop.
Negative:	You must carefully analyze the code to determine that parallelism is safe. Also, you must mark all variables that need to be localized.

- The third alternative is to use the automatic parallelization directives to give the AP more information about your code. The automatic directives are described in "Directives, Assertions, and Pragmas for Automatic Parallelization" on page 30. Like the manual directives, they have positive and negative features.

Positive:	The automatic directives are easier to use: They allow you to express the information you know without your having to be certain that all the conditions for parallelization are met.
Negative:	They are hints and thus do not impose parallelism. In addition, as with the manual directives, you must ensure that you are using them safely. Because they require less information than the manual directives, automatic directives can have subtle meanings.

Directives, Assertions, and Pragmas for Automatic Parallelization

The automatic parallelizer recognizes three types of compiler directives:

- Fortran directives enable, disable, or modify features of the automatic parallelizer.
- Fortran assertions assist the automatic parallelizer by providing it with additional information about the source program.
- Pragmas are the C and C++ counterparts to Fortran directives and assertions.

In practice, the AP makes little distinction between Fortran assertions and Fortran directives. The automatic compiler directives do not impose parallelism; they give hints and assertions to the AP in order to assist it in paralleling the right loops. The section "Invoking the Automatic Parallelizer" on page 4 gives more details on compiling with the AP. Table 3-1 lists the directives, assertions, and pragmas that the automatic parallelizer recognizes.

Table 3-1 Automatic Parallelizer Directives, Assertions, and Pragmas

Compiler Directive	Meaning
C*$* NO CONCURRENTIZE #pragma no concurrentize	Depends on placement. Either do not parallelize any loops in a subroutine, or do not parallelize any loops in a file.
C*$* CONCURRENTIZE #pragma concurrentize	Override C*$* NO CONCURRENTIZE.

Table 3-1 (continued) Automatic Parallelizer Directives, Assertions, and Pragmas

Compiler Directive	**Meaning**
C*$* ASSERT DO (CONCURRENT) #pragma concurrent	Do not let perceived dependences between two references to the same array inhibit parallelizing. Does not require **-pfa** or **-pca**.
C*$* ASSERT DO (SERIAL) #pragma serial	Do not parallelize the following loop.
C*$* ASSERT CONCURRENT CALL #pragma concurrent call	Ignore dependences in subroutine calls that would inhibit parallelizing. Does not require **-pfa** or **-pca**.
C*$* ASSERT PERMUTATION (*array_name*) #pragma permutation (*array_name*)	Array *array_name* is a permutation array. Does not require **-pfa** or **-pca**.
C*$* ASSERT DO PREFER (CONCURRENT) #pragma prefer concurrent	Parallelize the following loop if it is safe.
C*$* ASSERT DO PREFER (SERIAL) #pragma prefer serial	Do not parallelize the following loop.

There are two important points to remember regarding the compiler directives:

- Three of the compiler directives affect the compiling process even when **-pfa** and **-pca** are not specified.
  - **C*$* ASSERT DO (CONCURRENT)** and **#pragma concurrent**
  - **C*$* ASSERT CONCURRENT CALL** and **#pragma concurrent call**
  - **C*$* ASSERT PERMUTATION (***array_name***)** and **#pragma permutation (***array_name***)**

  C*$* ASSERT DO (CONCURRENT) and **C*$* ASSERT CONCURRENT CALL** can still affect optimizations such as loop interchange. **C*$* ASSERT PERMUTATION** can affect any optimization that requires permutation arrays.
- The general compiler option **-LNO:ignore_pragmas** causes the automatic parallelizer to ignore all of the directives, assertions, and pragmas in this section.

C*$* NO CONCURRENTIZE and #pragma no concurrentize

The **C*$* NO CONCURRENTIZE** directive prevents parallelization. Its effect depends on its placement.

- When placed inside subroutines and functions, the directive prevents their parallelization. In the following example, no loops inside **SUB1()** are parallelized.

```
      SUBROUTINE SUB1
C*$* NO CONCURRENTIZE
        ...
      END
```

- When placed outside of a subroutine, **C*$* NO CONCURRENTIZE** prevents the parallelization of all subroutines in the file, even those that appear ahead of it in the file. Loops inside subroutines **SUB2()** and **SUB3()** are not parallelized in this example:

```
      SUBROUTINE SUB2
        ...
      END
C*$* NO CONCURRENTIZE
      SUBROUTINE SUB3
        ...
      END
```

C*$* CONCURRENTIZE and #pragma concurrentize

Placing the **C*$* CONCURRENTIZE** directive inside a subroutine overrides a **C*$* NO CONCURRENTIZE** directive placed outside it. In other words, this directive allows you to selectively parallelize subroutines in a file that has been made sequential with **C*$* NO CONCURRENTIZE**.

C*$* ASSERT DO (CONCURRENT) and #pragma concurrent

C*$* ASSERT DO (CONCURRENT) says that when analyzing the loop immediately following this assertion, the AP should ignore all dependences between two references to the same array. Be aware that if there are real dependences between array references, **C*$* ASSERT DO (CONCURRENT)** may cause the AP to generate incorrect code.

The following example is a correct use of the assertion when *M* > *N*:

```
C*$* ASSERT DO (CONCURRENT)
       DO I = 1, N
         A(I) = A(I+M)
```

There are five facts to be aware of when using this assertion:

- If multiple loops in a nest can be parallelized, **C*$* ASSERT DO (CONCURRENT)** causes the AP to prefer the loop immediately following the assertion.
- Applying this directive to an inner loop may cause the loop to be made outermost by the AP's loop interchange operations.
- The assertion does not affect how the AP analyzes **CALL** statements. See "C*$* ASSERT CONCURRENT CALL and #pragma concurrent call" on page 33.
- The assertion does not affect how the AP analyzes dependences between two potentially aliased pointers. See "Aliased Parameter Information" on page 23 for a discussion of aliased pointers.
- This assertion affects the compilation even when **-pfa** and **-pca** are not specified.

C*$* ASSERT DO (SERIAL) and #pragma serial

C*$* ASSERT DO (SERIAL) instructs the AP not to parallelize the loop following the assertion. However, the automatic parallelizer may parallelize another loop in the nest. The parallelized loop may be either inside or outside the sequential loop.

C*$* ASSERT CONCURRENT CALL and #pragma concurrent call

The **C*$* ASSERT CONCURRENT CALL** assertion tells the AP to ignore the dependences of subroutine and function calls contained in a loop. Other points to be aware of are the following:

- The assertion applies to the loop that immediately follows it and to all loops nested inside that loop.
- It affects the compilation even when **-pfa** and **-pca** are not specified.

The automatic parallelizer ignores the dependences in subroutine **FRED()** when it analyzes the following loop:

```
C*$* ASSERT CONCURRENT CALL
       DO I = 1, N
         CALL FRED
         ...
       END DO
       SUBROUTINE FRED
         ...
       END
```

To prevent incorrect parallelization, you must make sure the following conditions are met when using **C*$* ASSERT CONCURRENT CALL**:

- A subroutine inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the subroutine.
- A subroutine inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the subroutine.

The following code shows an illegal use of the assertion. Subroutine **FRED()** writes to variable *T* which is also read from by **WILMA()** during other iterations.

```
C*$* ASSERT CONCURRENT CALL
       DO I = 1,M
         CALL FRED(B, I, T)
         CALL WILMA(A, I, T)
       END DO
       SUBROUTINE FRED(B, I, T)
         REAL B(*)
         T = B(I)
       END
       SUBROUTINE WILMA(A, I, T)
         REAL A(*)
         A(I) = T
       END
```

By localizing the variable *T*, you could manually parallelize the above example safely. But the AP does not know to localize *T*, and it illegally parallelizes the loop because of the assertion.

C*$* ASSERT PERMUTATION (array_name) and #pragma permutation (array_name)

When placed inside a subroutine, **C*$* ASSERT PERMUTATION** tells the AP that *array_name* is a *permutation* array: Every element of the array has a distinct value. The assertion does not require the permutation array to be *dense*. In other words, while every *IB(I)* must have a distinct value, there can be gaps between those values, such as *IB(1)* = 1, *IB(2)* = 4, *IB(3)* = 9, and so on.

Array *IB* is asserted to be a permutation array for both loops in **SUB1()** in this example.

```
      SUBROUTINE SUB1
        DO I = 1, N
          A(IB(I)) = ...
        END DO
C*$* ASSERT PERMUTATION (IB)
        DO I = 1, N
          A(IB(I)) = ...
        END DO
      END
```

There are a couple of points to be made about this assertion:

- As shown in the example, you can use this assertion to parallelize loops that use arrays for indirect addressing. Without this assertion, the AP is not able to determine that the array elements used as indexes are distinct.
- **C*$* ASSERT PERMUTATION** affects every loop in a subroutine, even those that appear ahead it.
- The assertion affects compilation even when **-pfa** and **-pca** are not specified.

C*$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent

C*$* ASSERT DO PREFER (CONCURRENT) says that the AP should parallelize the loop immediately following the assertion, if it is safe to do so. This assertion is always safe to use. Unless it can determine the loop is safe, the AP will not parallelize a loop because of this assertion.

The following code encourages the AP to run the *I* loop in parallel:

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

When dealing with nested loops, the automatic parallelizer follows these guidelines:

- If the loop specified by this assertion is safe to parallelize, the AP chooses it to parallelize, even if other loops in the nest are safe.
- If the specified loop is not safe to parallelize, the AP uses its heuristics to choose among loops that are safe.
- If this directive is applied to an inner loop, the AP may make the loop outermost.
- If this assertion is applied to more than one loop in a nest, the AP uses its heuristics to choose one of the specified loops.

C*$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial

The **C*$* ASSERT DO PREFER (SERIAL)** assertion tells the AP not to parallelize the loop that immediately follows. It is essentially the same as **C*$* ASSERT DO (SERIAL)**. In the following case, the assertion requests that the *J* loop be run serially:

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

The assertion applies only to the loop directly after the assertion. For example, the AP still tries to parallelize the *I* loop in the code shown above. The assertion is used in cases like this when the value of *N* is very small.

Index

Tell Us About This Manual

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

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

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

Please send the title and part number of the document with your comments. The part number for this document is 007-3572-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-965-0964
- To send your comments by **traditional mail**, use this address:

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