

MIPSpro™ Auto-Parallelizing Option Programmer's Guide

Document Number 007-3572-002

CONTRIBUTORS

Written by Don Moccia

Illustrated by Martha Levine

Production by Carmela Leckie

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.

© 1997, 1998, Silicon Graphics, Inc.— All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights are reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

Silicon Graphics and the Silicon Graphics logo are registered trademarks of Silicon Graphics, Inc., and Developer Magic, IRIS Power C, S2MP, Origin2000, POWER Fortran Accelerator, and POWER Series are trademarks of Silicon Graphics, Inc. MIPSpro is a trademark of MIPS Technologies, Inc.

Contents

List of Examples	v
List of Figures	vii
List of Tables	ix
About This Guide	xi
What This Guide Contains	xi
Conventions Used in This Guide	xii
Additional Reading	xiii
Manual Parallelization References	xiii
References on Optimization Techniques	xiii
Parallelization for O32 Compilers	xiv
1. Automatic Parallelization for MIPSpro Compilers	1
Understanding Automatic Parallelization	2
About the MIPSpro Auto-Parallelizing Option	2
Using the MIPSpro Auto-Parallelizing Option	4
Invoking the Auto-Parallelizing Option	4
MIPSpro Compiler Command-Line Options	7
Optimization Options	7
Interprocedural Analysis	7
Loop Nest Optimizer Options	8
Miscellaneous Optimization Options	9
Understanding the Auto-Parallelizing Option Output Files	11
About the .l File	11
OpenMP and the Auto-Parallelizing Fortran 77 Output Files	12
About the .w2f.f and .w2c.c Files	13
About the .m and .anl Files	15
Running Your Program	15

- 2. Understanding Incomplete Optimization 17**
 - Failing to Parallelize Safe Loops 17
 - Function Calls in Loops 18
 - GO TO Statements in Loops 18
 - Problematic Array Subscripts 19
 - Indirect Array References 19
 - Unanalyzable Subscripts 19
 - Hidden Knowledge 20
 - Conditionally Assigned Temporary Nonlocal Variables in Loops 20
 - Unanalyzable Pointer Usage in C and C++ 21
 - Arbitrary Pointer Dereferences 22
 - Arrays of Arrays 22
 - Loops Bounded by Pointer Comparisons 22
 - Aliased Parameter Information 23
 - Parallelizing the Wrong Loop 24
 - Inner Loops 24
 - Small Trip Counts 25
 - Poor Data Locality 25
 - Incurring Unnecessary Parallelization Overhead 27
 - Unknown Trip Counts 27
 - Nested Parallelism 28
- 3. Assisting the MIPSpro Auto-Parallelizing Option 29**
 - Strategies for Assisting the Auto-Parallelizing Option 29
 - Compiler Directives for Automatic Parallelization 30
 - C*\$* NO CONCURRENTIZE and #pragma no concurrentize 32
 - C*\$* CONCURRENTIZE and #pragma concurrentize 32
 - C*\$* ASSERT DO (CONCURRENT) and #pragma concurrent 32
 - C*\$* ASSERT DO (SERIAL) and #pragma serial 33
 - C*\$* ASSERT CONCURRENT CALL and #pragma concurrent call 33
 - C*\$* ASSERT PERMUTATION and #pragma permutation 35
 - C*\$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent 35
 - C*\$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial 36
- Index 39**

List of Examples

Example 1-1	Subroutine in File testl.f	11
Example 1-2	Listing in File testl.l	12
Example 1-3	Subroutine in File testw2.f	14
Example 1-4	Listing in File testw2.w2f.f	14
Example 2-1	Distribution of Iterations	26
Example 2-2	Two Nests in Sequence	26

List of Figures

Figure 1-1 Files Generated by the MIPSpro Auto-Parallelizing Option 3

List of Tables

Table 1-1	MIPSpro 7.2 (and Later) Compilers and the Auto-Parallelizing Option 1
Table 1-2	Files Generated by -apo keep 5
Table 3-1	Auto-Parallelizing Option Directives, Assertions, and Pragmas 31

About This Guide

This document describes how to use the MIPSpro Auto-Parallelizing Option (APO) to automatically detect and exploit parallelism in Fortran 77, Fortran 90, C, and C++ programs. The MIPSpro APO, an optional extension available since the 7.2 release of the MIPSpro compilers, supports N32 and N64 application binary interfaces (ABIs). For information about parallelization for the O32 ABI, refer to the guides listed under “Parallelization for O32 Compilers” on page xiv. See the ABI(5) reference page for explanations of the N32, N64, and O32 ABIs.

In its original release, this document was called the *MIPSpro Automatic Parallelizer Programmer's Guide*. The major change in its content is information about the OpenMP proposed standard for parallel computing, and the new **-apo** command-line option. See the MIPSpro APO release notes for other changes.

What This Guide Contains

This document contains the following chapters:

Chapter 1, “Automatic Parallelization for MIPSpro Compilers,” describes the Auto-Parallelizing Option, how to invoke it, and how to select its output files.

Chapter 2, “Understanding Incomplete Optimization,” explains how to recognize and avoid problems that may arise using the MIPSpro APO.

Chapter 3, “Assisting the MIPSpro Auto-Parallelizing Option,” explains the compiler directives recognized by the MIPSpro APO.

Conventions Used in This Guide

This document uses the following conventions and symbols:

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

Here is an example illustrating the syntax conventions:

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

The previous syntax statement indicates that

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

The following statements are valid examples of the syntax described above:

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

```
f77 -O3 -n32 -mips4 -apo list test.f -c.
```

Additional Reading

This guide describes using the MIPSpro Auto-Parallelizing Option to automatically parallelize programs written for the N32 and N64 ABIs. Related readings on manual parallelization, other types of optimization, and the parallelization of programs for the O32 ABI are listed in the following sections.

Manual Parallelization References

For details about using manual parallelization directives to label parallel loops and code regions for multiprocessing, refer to the appropriate chapters in the following:

- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*
- *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, N64, 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 tools 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 principles and techniques of parallel compiler optimization.

Parallelization for O32 Compilers

The Auto-Parallelizing Option can be used with 7.2 and newer releases of the MIPSpro compilers. It does not support earlier MIPSpro releases or the O32 ABI. If you are using O32 compilers, you can use the following references for parallelization information:

- *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 existing Fortran 77 programs to run efficiently on Silicon Graphics POWER Series multiprocessor systems.

Automatic Parallelization for MIPSpro Compilers

This chapter discusses automatic parallelization for the 7.2 and later releases of the Silicon Graphics MIPSpro compilers. You can achieve automatic parallelization with these compilers using the MIPSpro Auto-Parallelizing Option (APO), an optional software product for programs written for the N32 and N64 application binary interfaces. See the ABI(5) reference page for information on the N32 and N64 ABIs.

The MIPSpro APO is an extension integrated into the four compilers listed in the left column of Table 1-1. It is not a source-to-source preprocessor as was used prior to the 7.2 release. If the Auto-Parallelizing Option is installed, the compilers are known as *auto-parallelizing compilers* and are referred to by the names in the right column.

Table 1-1 MIPSpro 7.2 (and Later) Compilers and the Auto-Parallelizing Option

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

This chapter contains these sections:

- “Understanding Automatic Parallelization” on page 2 introduces parallelization in the MIPSpro compilers.
- “About the MIPSpro Auto-Parallelizing Option” on page 2 describes the MIPSpro APO’s features.
- “Using the MIPSpro Auto-Parallelizing Option” on page 4 tells how to use the MIPSpro APO 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 Auto-Parallelizing Option extension of the compiler analyzes and restructures the program with little or no intervention by you. The MIPSpro APO 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 documents listed under “Manual Parallelization References” on page xiii. The introduction also contains useful optimization references under “References on Optimization Techniques” on page xiii.

About the MIPSpro Auto-Parallelizing Option

The Auto-Parallelizing Option helps you exploit parallelism in programs to enhance their performance on multiprocessor systems. It is a compiler extension controlled with flags in the command lines that invoke the MIPSpro auto-parallelizing compilers. Although their runtime performance suffers slightly on single-processor systems, parallelized programs can be created and debugged with the MIPSpro auto-parallelizing compilers on any Silicon Graphics system that uses a MIPS processor.

Starting with the 7.2 release, the auto-parallelizing compilers integrate automatic parallelization, provided by the MIPSpro APO, with other compiler optimizations, such as interprocedural analysis (IPA) and loop nest optimization (LNO). Whereas releases prior to 7.2 relied on source-to-source preprocessors, the 7.2 and later versions internalize automatic parallelization into the optimizer of the MIPSpro compilers. As seen in Figure 1-1, the MIPSpro APO works on an intermediate representation generated during the compiling process. This provides several benefits:

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

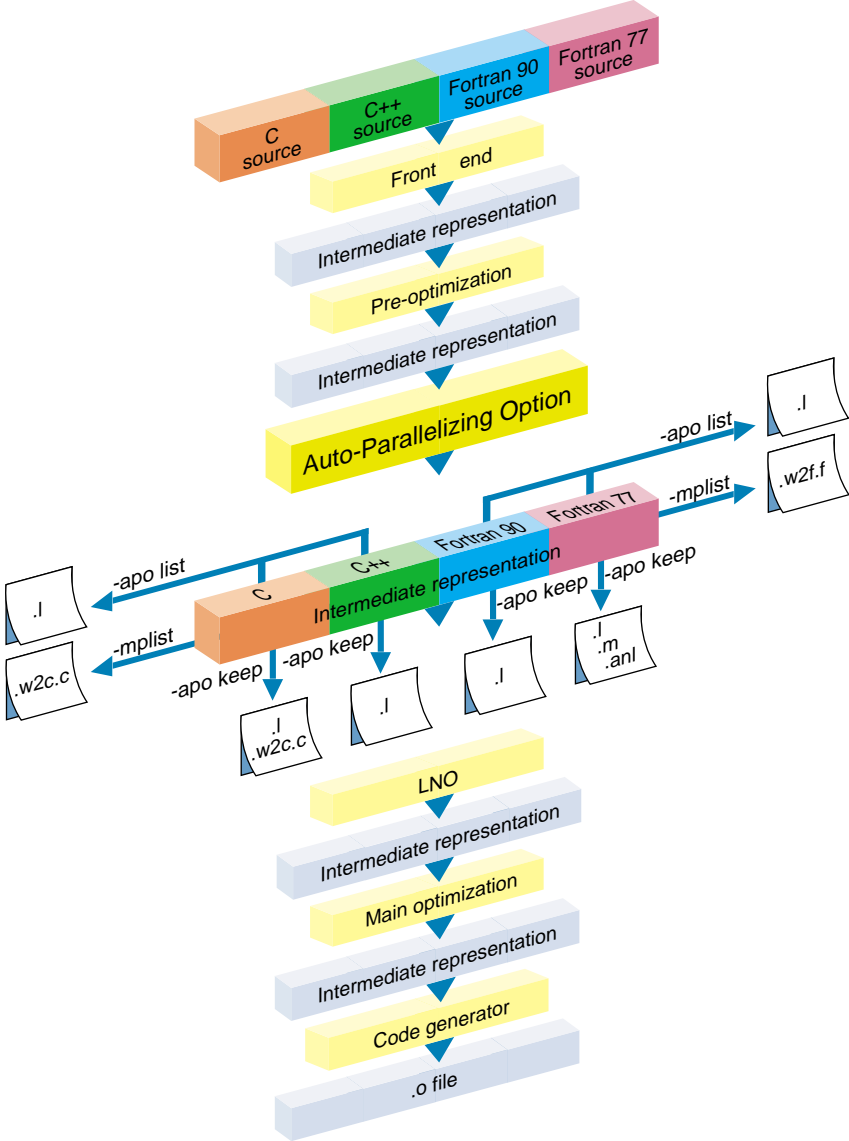


Figure 1-1 Files Generated by the MIPSpro Auto-Parallelizing Option

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.

Using the MIPSpro Auto-Parallelizing Option

This section describes how to use the MIPSpro Auto-Parallelizing Option to compile and run parallelized programs.

- “Invoking the Auto-Parallelizing Option” on page 4 demonstrates how to compile programs with the auto-parallelizing versions of Fortran 77, Fortran 90, C, and C++.
- “MIPSpro Compiler Command-Line Options” on page 7 describes options relevant to the MIPSpro APO.
- “Understanding the Auto-Parallelizing Option Output Files” on page 11 explains how to use the listing mechanisms of the MIPSpro APO to see what transformations were performed and what regions of codes were parallelized.
- “Running Your Program” on page 15 shows how to run your parallelized program.

Invoking the Auto-Parallelizing Option

You invoke the Auto-Parallelizing Option by including the **-apo** flag on the command line that starts a MIPSpro auto-parallelizing compiler. Additional flags allow you to generate reports to aid in debugging. The syntax for compiling programs with the MIPSpro APO is as follows:

- Auto-Parallelizing Fortran 77, Auto-Parallelizing Fortran 90, Auto-Parallelizing C, and Auto-Parallelizing C++ are invoked using **-apo**:

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

```
f90 options -apo[{list|keep}] filename
```

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

```
CC options -apo[{list|keep}] filename
```

- Alternatively, the auto-parallelizing compilers may be invoked using the **-pfa** or **-pca** flags instead of **-apo**. These options are provided for backward compatibility and their use is deprecated.

The command-line entries are defined as follows:

<i>options</i>	The MIPSpro compiler command-line options. -O3 is recommended for using the MIPSpro APO. For details, see “MIPSpro Compiler Command-Line Options” on page 7, and the documentation for your MIPSpro compiler.
-apo	Invoke the Auto-Parallelizing Option.
-apo list	Invoke the MIPSpro APO and produce a <i>.l</i> 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 <i>.l</i> File” on page 11.
-apo keep	Invoke the MIPSpro APO and generate <i>.l</i> , <i>.w2c.c</i> , <i>.m</i> , and <i>.anl</i> files as 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 equivalent parallelized program for Fortran 77, in a <i>.w2f.f</i> file, or for C, in a <i>.w2c.c</i> file. These files are discussed in the section “About the <i>.w2f.f</i> and <i>.w2c.c</i> Files” on page 13. Do not use with -apo keep , -FLIST , or -CLIST .
<i>filename</i>	The name of the file containing the source code.

Note: Starting with the 7.2.1 release of the MIPSpro compilers, the **-apo keep** and **-mplist** flags cause Auto-Parallelizing Fortran 77 to generate *.m* and *w2f.f* files based on OpenMP directives (see “OpenMP and the Auto-Parallelizing Fortran 77 Output Files” on page 12). To have the compiler emit the pre-OpenMP directives, use the Fortran 77 option **-FLIST:emit_pcf** instead of these flags.

The files generated by **-apo keep** with the various compilers are shown in Table 1-2. The *.l* file is the same as that generated using **-apo list**, and the *.w2c.c* file is the same as that generated using **-mplist**. The other two files are for Fortran 77 only. They are the *.m* file, a parallelized equivalent program based on OpenMP, and the *.anl* file, a file for use with WorkShop Pro MPF. These files are explained in the section “About the *.m* and *.anl* Files” on page 15.

Table 1-2 Files Generated by **-apo keep**

File Suffix	f77	f90	cc	CC
<i>.l</i>	Yes	Yes	Yes	Yes
<i>.w2c.c</i>	No	No	Yes	No

Table 1-2 (continued) Files Generated by -apo keep

File Suffix	f77	f90	cc	CC
.m	Yes	No	No	No
.anl	Yes	No	No	No

Consider a typical command line:

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

This command uses Auto-Parallelizing Fortran 77 (**f77 ... -apo**) to compile (**-c**) the file *myProg.f* with the MIPSpro compiler options **-O3**, **-n32**, and **-mips4**. Using **-O3**, which requests aggressive optimization, is recommended for using the MIPSpro APO. It is covered in “Optimization Options” on page 7. 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 **-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 use **-apo keep** instead of **-mplist** to get a *.anl* file.

To use the Auto-Parallelizing Option correctly, remember these points:

- The MIPSpro APO can be used only with **-n32** or **-64** compiles. With **-o32** compiles, the **-pfa** and the **-pca** flags invoke the older, Power parallelizers, and the **-apo** flag is not supported.
- If you link separately, you must have one of the following in the link line:
 - the **-apo** 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:
 - **-apo 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. Starting with MIPSpro 7.2, the Auto-Parallelizing Option 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-Parallelizing 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 compiler command-line options most commonly needed with the Auto-Parallelizing Option:

- “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 its use is recommended to run the MIPSpro APO. The optimization at this level maximizes code quality even if it requires extensive compile time or relaxing language rules. The **-O3** option 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 xiii.

As of the MIPSpro 7.2.1 release, the Auto-Parallelizing Option with IPA is able to optimize only those loops whose function calls were inlined by the MIPSpro APO. This is further described in “Function Calls in Loops” on page 18.

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 compiled with **-apo**, none of its inlined subroutines are parallelized. This is true even if the subroutines are compiled separately with **-apo**, because with IPA automatic parallelization is deferred until link time.

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 is true with the Auto-Parallelizing Option, the same optimizations and control options can be used with Fortran, C, or C++ programs. The *MIPSpro Compiling and Performance Tuning Guide* describes three LNO options of particular interest to users of the MIPSpro APO:

- **-LNO:parallel_overhead=*n***. This option controls the auto-parallelizing 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 MIPSpro APO 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 MIPSpro APO to ignore all of the directives, assertions, and pragmas covered in "Compiler Directives for Automatic Parallelization" on page 30. This includes the directive **C*\$* NO CONCURRENTIZE**.

If you are using Fortran 77 or C, you can view the transformed code in the original source language after the 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*. Because it is generated at a later stage of the compilation, this *.w2f.f* file differs somewhat from the *.w2f.f* file generated by the **-mplist** option (see "About the *.w2f.f* and *.w2c.c* Files" on page 13). You can read the *.w2f.f* file, which is a compilable Fortran representation of the original program after the LNO phase. Because the 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 MIPSpro APO:

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

The **-OPT:alias=*name*** option has several variations. One of interest to users of the MIPSpro APO 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 produces 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, the compiler assumes memory operations dereferencing different named pointers do not alias with each other, and different dereferencing depths of the same pointer do not alias with each other. For example, 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 MIPSpro APO 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 MIPSpro APO 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 Auto-Parallelizing Option Output Files

Processing a program with the Auto-Parallelizing Option often results in excellent parallelization with no further effort. But, as described in Chapter 2, "Understanding Incomplete Optimization," there are cases that cannot be effectively parallelized automatically. To help analyze these cases, the MIPSpro APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. By understanding these listings, you may be able to identify small problems that prevent a loop from being made parallel. With a little work, you can often remove these data dependences, dramatically improving 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 such as 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 15.

About the .l File

The **-apo list** and **-apo keep** options generate files, whose names end with *.l*, that list 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 -apo list testl.f -c.
```

the Auto-Parallelizing Option 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 indicates that the name of the subroutine is `__mpdo_sub_1`. For more information about interpreting this file, you can refer to Chapter 2, “Understanding Incomplete Optimization,” or “Additional Reading” on page xiii.

OpenMP and the Auto-Parallelizing Fortran 77 Output Files

The 7.2.1 release of the MIPSpro compilers is the first to incorporate OpenMP, a cross-vendor API for shared-memory parallel programming in Fortran and, eventually, C and C++. OpenMP— a collection of directives, library routines, and environment variables—is used to specify shared-memory parallelism in source code. Additionally, OpenMP is intended to enhance your ability to implement the coarse-grained parallelism of large code sections. On Silicon Graphics platforms, OpenMP replaces the older Parallel Computing Forum (PCF) and SGI DOACROSS directives for Fortran. More information about the specification can be found at the OpenMP Web site: <http://www.openmp.org/>.

The MIPSpro APO interoperates with OpenMP as well as with the older directives. This means that an Auto-Parallelizing Fortran 77 or Auto-Parallelizing Fortran 90 file may use a mixture of directives from each source. As of the 7.2.1 release, the only OpenMP-related

changes that most MIPSpro APO users see are in the Auto-Parallelizing Fortran 77 *w2f.f* and *.m* files, generated using the **-mplist** and **-apo keep** flags, respectively. The parallelized source programs contained in these files now contain OpenMP directives. None of the other MIPSpro auto-parallelizing compilers generate source programs based on OpenMP.

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

The *.w2f.f* and *.w2c.c* files contain Fortran 77 and C code, respectively, that mimics the behavior of programs after they undergo automatic parallelization. The representations in these files are designed to be readable so that you can see what portions of the original code were not parallelized. You can use the information in these files to change the original programs to aid their parallelization.

The MIPSpro auto-parallelizing compilers create the *.w2f.f* and *.w2c.c* files by invoking the appropriate translator to turn the compilers' internal representations into Fortran 77 or C. In most cases, the files contain valid code that can be recompiled, although compiling a *.w2f.f* or *.w2c.c* file with a standard MIPSpro compiler does not produce object code that is exactly the same as that generated by an auto-parallelizing compiler processing the original source. This is because the MIPSpro APO is an internal phase of the MIPSpro auto-parallelizing compilers, not a source-to-source preprocessor, and does not use a *.w2f.f* or *.w2c.c* source file to generate the object file.

The **-mplist** option tells Auto-Parallelizing Fortran 77 to compile a program and generate a *.w2f.f* file. Because it is generated at an earlier stage of the compilation, this *.w2f.f* file is much more easily understood than the *.w2f.f* file generated using the **-FLIST:=on** option (see "Loop Nest Optimizer Options" on page 8). The parallelized program in the *.w2f.f* file uses OpenMP directives (see "OpenMP and the Auto-Parallelizing Fortran 77 Output Files" on page 12). A *.w2f.f* program that uses PCF instead of OpenMP can be generated by adding the option **-FLIST:emit_pcf** to the *f77* command line. There is no *.w2f.f* file for Auto-Parallelizing Fortran 90.

The **-mplist** and **-apo keep** options instruct Auto-Parallelizing C to generate a *.w2c.c* file. The *.w2c.c* file contains a parallelized program based on directives similar to the Fortran ones of PCF. There is no *.w2c.c* file for the auto-parallelizing version of C++.

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 -apo -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.

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

```
C *****
C Fortran file translated from WHIRL Sun Dec 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      PARALLEL DO will be converted to SUBROUTINE __mpdo_trivial_1
C$OMP PARALLEL DO private(i), shared(a)
      DO i = 1, 10000, 1
        a(i) = 0.0
      END DO
      RETURN
      END ! trivial
```

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

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`. **C\$OMP PARALLEL DO** is an OpenMP directive that specifies a parallel region containing a single DO directive. See “OpenMP and the Auto-Parallelizing Fortran 77 Output Files” on page 12 for more information on OpenMP.

About the .m and .anl Files

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

- A .m file, which is similar to the .w2f.f file. It is based on OpenMP (see “OpenMP and the Auto-Parallelizing Fortran 77 Output Files” on page 12) and mimics the behavior of the program after automatic parallelization. It is also annotated with information that is used by Workshop ProMPF. A parallelized Fortran 77 program based on the PCF directives instead of OpenMP can be created by using the option **-FLIST:emit_pcf** in addition to the **-apo keep** flag.
- A .anl file, which is used by Workshop ProMPF.

Silicon Graphics provides a separate product, WorkShop Pro MPF, that provides a graphical interface to aid in both automatic and manual parallelization 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.

SpeedShop, another Silicon Graphics product, allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop consists of an API, a set of commands that can be run in a shell, 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 various numbers of processors. The default is to have the run-time environment select the number of processors to use based on how many are available.

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

```
setenv OMP_NUM_THREADS 2
```

causes the program to create two threads regardless of the number of processors available. Using `OMP_NUM_THREADS` is preferable to using `MP_SET_NUMTHREADS` and its older synonym `NUM_THREADS`, which preceded the release of the MIPSpro APO with OpenMP.

The environment variable `OMP_DYNAMIC` allows you to control whether the run-time environment should dynamically adjust the number of threads available for executing parallel regions to optimize system resources. The default value is `TRUE`. If `OMP_DYNAMIC` is set to `FALSE`,

```
setenv OMP_DYNAMIC FALSE
```

dynamic adjustment is disabled.

Understanding Incomplete Optimization

This chapter discusses programming practices that prevent your code from being fully optimized by the Auto-Parallelizing Option. The MIPSpro APO cannot always parallelize programs effectively. Sometimes this is unavoidable, but at other times you can help it improve its effectiveness. There are three general categories of incomplete optimization:

- The MIPSpro APO does not detect that a loop is safe to parallelize, as described in “Failing to Parallelize Safe Loops” on page 17.
- The MIPSpro APO parallelizes the wrong loop, usually in a nest, as covered in “Parallelizing the Wrong Loop” on page 24.
- The MIPSpro APO unnecessarily parallelizes a loop, as discussed in “Incurring Unnecessary Parallelization Overhead” on page 27.

Failing to Parallelize Safe Loops

A program’s performance may be severely constrained by the Auto-Parallelizing Option’s not recognizing 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 MIPSpro APO analyzes every loop in a sequential program. If it cannot prove a loop is safe, it does not parallelize that loop.

The MIPSpro APO often does not parallelize loops containing any of the following constructs described in this section:

- “Function Calls in Loops” on page 18
- “GO TO Statements in Loops” on page 18
- “Problematic 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 "About the *.l* File" on page 11, can show you if any of these constructs are in your code.

Function Calls in Loops

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

- Interprocedural analysis (IPA), specified by the **-IPA** command-line option, can provide the MIPSpro APO with enough information to parallelize some loops containing subroutine calls by inlining those calls. For more information on IPA, see "Interprocedural Analysis" on page 7 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 MIPSpro APO to ignore the dependences of function calls when analyzing the specified loops by using the following:
 - the assertion **C*\$* ASSERT CONCURRENT CALL** for Auto-Parallelizing Fortran 77 and Auto-Parallelizing Fortran 90
 - the **#pragma concurrent call** for Auto-Parallelizing C and Auto-Parallelizing C++

GO TO Statements in Loops

GO TO statements are unstructured control flows. The Auto-Parallelizing Option converts most unstructured control flows in loops into structured flows that can be parallelized. However, **GO TO** statements in loops can still cause two problems:

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

Problematic 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 MIPSpro APO 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*$* ASSERT PERMUTATION` assertion, discussed in “`C*$* ASSERT PERMUTATION` and `#pragma permutation`” on page 35, is appropriate.

Unanalyzable Subscripts

The MIPSpro APO 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 or difference of any combination of the first three items, such as $N*21+K-251$

In the following case, the MIPSpro APO cannot 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 does not overlap. However, the MIPSpro APO does not know the value of the variables and therefore cannot make the loop parallel. Using the `C*$* ASSERT DO (CONCURRENT)` assertion, explained in “`C*$* ASSERT DO (CONCURRENT)` and `#pragma concurrent`” on page 32, lets the MIPSpro APO parallelize this loop. You can also use manual parallelization.

Conditionally Assigned Temporary Nonlocal Variables in Loops

When parallelizing a loop, the Auto-Parallelizing Option often localizes (privatizes) temporary scalar and array variables by giving each processor its own non-shared copy of them. 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, the MIPSpro APO must give each processor a distinct, private TMP array. In this example, it is able to localize TMP and, thereby, to parallelize the loop.

The MIPSpro APO 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 is the processor that executed iteration N . Because T is conditionally assigned, the MIPSpro APO cannot determine which copy to use.

The solution comes with the realization that the loop is inherently parallel if the conditionally assigned variable T is localized. If the value of T is not used outside the loop, replace T with a local variable. Unless T is a local variable, the MIPSpro APO 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 and not as true arrays. The following practices involving pointers interfere with the Auto-Parallelizing Option's effectiveness:

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

Arbitrary Pointer Dereferences

The MIPSpro APO 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 MIPSpro APO 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 ANSIC did not allow variable-sized multidimensional arrays when this document was written, MIPSpro 7.2 C and C++ implemented this proposal. See the *C Language Reference Manual* for details.

Loops Bounded by Pointer Comparisons

The MIPSpro APO 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 such as 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++ programs 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 MIPSpro APO 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 used to guarantee that *a* and *b* provide exclusive access to their arrays. This assurance permits the MIPSpro APO 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];
}
```

There are two more ways of dealing with aliased parameter information:

- The keyword `restrict` behaves similarly to `__restrict`, but might 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 that `-LANG:restrict` 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 may have unforeseen effects.

Parallelizing the Wrong Loop

The Auto-Parallelizing Option parallelizes a loop by distributing its iterations among the available processors. When parallelizing nested loops, such as *I*, *J*, and *K* in the example below, the MIPSpro APO 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 MIPSpro APO chooses. In fact, the loop the MIPSpro APO parallelizes may be an inferior choice for any of three reasons:

- It is an inner loop, as discussed in “Inner Loops” on page 24.
- It has a small trip count, as covered in “Small Trip Counts” on page 25.
- It exhibits poor data locality, as described in “Poor Data Locality” on page 25.

The MIPSpro APO’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

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 Auto-Parallelizing Option tries to parallelize the outermost loop, after possibly interchanging loops to make a more promising one outermost. If the outermost loop attempt fails, the MIPSpro APO parallelizes an inner loop if possible.

Checking the `.l` file, described in “About the `.l` File” on page 11, tells you which loop in a nest was parallelized. If the loop was not the outermost, the reason is probably one of

those mentioned in “Failing 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 “Incurring Unnecessary Parallelization Overhead,” loops with small trip counts generally run faster when they are not parallelized. Consider how this fact affects this Fortran example:

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

The Auto-Parallelizing Option 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 MIPSpro APO often cannot know that *M* is small, you can do one of the following:

- Use one of the C*\$* **ASSERT DO PREFER** assertions to tell the MIPSpro APO that it is better to parallelize the *J* loop. They are discussed in
 - “C*\$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent” on page 35
 - “C*\$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial” on page 36
- Use the manual parallelization directives contained in the “Manual Parallelization References” on page xiii.

Poor Data Locality

Computer memory has a hierarchical organization. Higher 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 further down. In multiprocessor systems, each processor has its own cache memory. Because it is time consuming for one processor to access another processor’s cache, 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 to be 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 Auto-Parallelizing Option may choose to parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor accesses 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 instruct the MIPSpro APO to do this with the **C*\$* ASSERT DO PREFER** assertions described in

- “C*\$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent” on page 35
- “C*\$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial” on page 36

Incurring Unnecessary Parallelization Overhead

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 Counts

If the trip count is not known (and sometimes even if it is), the Auto-Parallelizing Option parallelizes the loop conditionally. It generates code for both a parallel and a sequential version. By generating two versions, the MIPSpro APO can avoid running in parallel a loop that turns out to have a small trip count. The MIPSpro APO 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 run-time 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 run-time overhead of choosing between them. You can avoid this overhead by using the **prefer** pragmas or **C*\$* ASSERT DO PREFER** assertions, discussed in

- “C*\$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent” on page 35
- “C*\$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial” on page 36

These compiler directives ensure that the MIPSpro APO knows in advance whether or not to parallelize the loop.

Nested Parallelism

Nested parallelism is not supported by the Auto-Parallelizing Option. Thus, for every loop that could be parallelized, the MIPSpro APO must generate a test that determines if the loop is being called from within either another parallel loop or a 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, the test can be avoided. 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()**. For more information on these compiler directives, see

- “C*\$* ASSERT DO (SERIAL) and #pragma serial” on page 33
- “C*\$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial” on page 36

Assisting the MIPSpro Auto-Parallelizing Option

This chapter discusses actions you can take to enhance the performance of the MIPSpro Auto-Parallelizing Option.

- “Strategies for Assisting the Auto-Parallelizing Option” on page 29 gives strategies for optimizing your code for the MIPSpro APO.
- “Compiler Directives for Automatic Parallelization” on page 30 discusses the compiler directives the MIPSpro APO recognizes.

Strategies for Assisting the Auto-Parallelizing Option

There are circumstances that interfere with the Auto-Parallelizing Option’s ability to optimize programs. As shown in Chapter 2, “Understanding Incomplete Optimization,” problems are sometimes caused by coding practices. Other times, the MIPSpro APO does not have enough information to make good parallelization decisions. You can pursue three strategies to attack these problems and to achieve better results with the MIPSpro APO.

- The first approach is to modify your code to avoid coding practices that the MIPSpro APO cannot analyze well. Specific problems and solutions are discussed in Chapter 2, “Understanding Incomplete Optimization.”
- The second strategy is to assist the MIPSpro APO with the manual parallelization directives. They are described in the *MIPSpro Compiling and Performance Tuning Guide*, and require the **-mp** compiler option. The MIPSpro APO is designed to recognize and coexist with manual parallelism. You can use manual directives with some loop nests, while leaving others to the MIPSpro APO. 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 is 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 compiler directives to give the MIPSpro APO more information about your code. The automatic directives are described in “Compiler Directives 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: The automatic directives 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.

Compiler Directives for Automatic Parallelization

The Auto-Parallelizing Option recognizes three types of compiler directives:

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

In practice, the MIPSpro APO makes little distinction between Fortran assertions and Fortran directives. The automatic parallelization compiler directives do not impose parallelism; they give hints and assertions to the MIPSpro APO to assist it in choosing the right loops. The section “Invoking the Auto-Parallelizing Option” on page 4 gives

more details on compiling with the MIPSpro APO. Table 3-1 lists the directives, assertions, and pragmas that the MIPSpro APO recognizes.

Table 3-1 Auto-Parallelizing Option Directives, Assertions, and Pragmas

Compiler Directive	Meaning and Notes
C*\$* NO CONCURRENTIZE #pragma no concurrentize	Varies with 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.
C*\$* ASSERT DO (CONCURRENT) #pragma concurrent	Do not let perceived dependences between two references to the same array inhibit parallelizing. Does not require -apo .
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 -apo .
C*\$* ASSERT PERMUTATION (<i>array_name</i>) #pragma permutation (<i>array_name</i>)	Array <i>array_name</i> is a permutation array. Does not require -apo .
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 compiler directives affect the compiling process even if **-apo** is not specified.
 - **C*\$* ASSERT DO (CONCURRENT)** and **#pragma concurrent** may affect optimizations such as loop interchange.
 - **C*\$* ASSERT CONCURRENT CALL** and **#pragma concurrent call** also may affect optimizations such as loop interchange.
 - **C*\$* ASSERT PERMUTATION** and **#pragma permutation** may affect any optimization that requires permutation arrays.

- The general compiler option `-LNO:ignore_pragmas` causes the MIPSpro APO 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)` instructs the MIPSpro APO, when analyzing the loop immediately following this assertion, to 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 MIPSpro APO 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 six facts to be aware of when using this assertion:

- If multiple loops in a nest can be parallelized, **C*\$* ASSERT DO (CONCURRENT)** causes the MIPSpro APO 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 MIPSpro APO's loop interchange operations.
- The assertion does not affect how the MIPSpro APO analyzes **CALL** statements. See "**C*\$* ASSERT CONCURRENT CALL** and **#pragma concurrent call**" on page 33.
- The assertion does not affect how the MIPSpro APO 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 **-apo** is not specified.
- The compiler may find some obvious real dependences. If it does so, it ignores this assertion.

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

C*\$* ASSERT DO (SERIAL) instructs the Auto-Parallelizing Option not to parallelize the loop following the assertion. However, the MIPSpro APO may parallelize another loop in the same nest. The parallelized loop may be either inside or outside the designated sequential loop.

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

The **C*\$* ASSERT CONCURRENT CALL** assertion instructs the MIPSpro APO to ignore the dependences of subroutine and function calls contained in the loop that follows the assertion. 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.
- The assertion affects the compilation even when **-apo** is not specified.

The MIPSpro APO 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, 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 can manually parallelize the above example safely. But the MIPSpro APO does not know to localize T , and it illegally parallelizes the loop because of the assertion.

C*\$* ASSERT PERMUTATION and #pragma permutation

When placed inside a subroutine, **C*\$* ASSERT PERMUTATION** (*array_name*) tells the MIPSpro APO 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 **SUB10** 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 three 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 MIPSpro APO cannot determine that the array elements used as indexes are distinct.
- **C*\$* ASSERT PERMUTATION** (*array_name*) affects every loop in a subroutine, even those that appear ahead it.
- The assertion affects compilation even when **-apo** is not specified.

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

C*\$* ASSERT DO PREFER (CONCURRENT) instructs the Auto-Parallelizing Option to 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 MIPSpro APO does not parallelize a loop because of this assertion.

The following code encourages the MIPSpro APO 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 Auto-Parallelizing Option follows these guidelines:

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

Symbols

#pragma concurrent, 32
#pragma concurrent call, 33
 function calls, 18
#pragma concurrentize, 32
#pragma no concurrentize, 32
#pragma permutation, 35
#pragma prefer concurrent, 35
#pragma prefer serial, 36
#pragma serial, 33
__restrict type qualifier, 23
 aliased parameter information, 23

Numbers

-64 compiler option, 6

A

ABI

N32, xi, 1, 6
N64, xi, 1, 6
O32, xi, 6

aliased parameter information, 23

.anl file, 15

-apo flag

-apo keep, 5
 .i file, 11

w2c.c file, 13

-apo list, 5

 .i file, 11

invoking automatic parallelization, 4

 not specified, 31

arbitrary pointer dereferences, 22

arrays

 of arrays, 22

 permutation, 35

 variable-sized multidimensional, 22

assertion

 ASSERT CONCURRENT CALL, 18, 33

 ASSERT DO (CONCURRENT), 20, 32

 ASSERT DO (SERIAL), 33

 nested parallelism, 28

 ASSERT DO PREFER (CONCURRENT), 35

 ASSERT DO PREFER (SERIAL), 36

 nested parallelism, 28

 ASSERT PERMUTATION, 19, 35

assertions, Fortran, 30

automatic parallelization, 2

 directives, 30

 invoking, 4

C

C, 1

 Auto-Parallelizing, 1

 invoking automatic parallelization, 4

C*\$* ASSERT CONCURRENT CALL, 33

 function calls, 18

C*\$* ASSERT DO (CONCURRENT), 32
 hidden knowledge, 20
C*\$* ASSERT DO (SERIAL), 33
 nested parallelism, 28
C*\$* ASSERT DO PREFER (CONCURRENT), 35
C*\$* ASSERT DO PREFER (SERIAL), 36
 nested parallelism, 28
C*\$* ASSERT PERMUTATION, 35
 indirect array references, 19
C*\$* CONCURRENTIZE, 32
C*\$* NO CONCURRENTIZE, 32
C++, 1
 Auto-Parallelizing, 1
 invoking automatic parallelization, 4
-CLIST option, 9
 conflict with -apo keep, 5
 conflict with -mplist, 5
compiler directives, 30
conditionally assigned temporary variable, 21

D

data locality, 25
directive
 CONCURRENTIZE, 32
 NO CONCURRENTIZE, 32
directives, Fortran, 30
DOACROSS directives, 12

F

-FLIST option, 9
 conflict with -apo keep, 5
 conflict with -mplist, 5
Fortran 77, 1
 Auto-Parallelizing, 1
 invoking automatic parallelization, 4

Fortran 90, 1
 Auto-Parallelizing, 1
 invoking automatic parallelization, 4

G

GO TO statements, 18

H

hidden knowledge, 20

I

indirect array references, 19
interprocedural analysis
 function calls, 18
 -IPA, 7

K

keep option, -apo keep, 5

L

-LANG option, -LANG:restrict, 23
.l file, 11
list option, -apo list, 5
loop nest optimizer
 -LNO, 8
 -LNO:auto_dist, 9
 -LNO:ignore_pragmas, 9, 32
 -LNO:parallel_overhead, 9
 unknown trip count, 27
loops

bounded by pointer comparisons, 22
inner, 24

M

manual parallelization, 2
 directives, 29
.m file, 15
MIPSpro 7.2 compilers, 1
 invoking automatic parallelization, 4
MIPSpro compilers
 Auto-Parallelizing C, 1
 Auto-Parallelizing C++, 1
 Auto-Parallelizing Fortran 77, 1
 Auto-Parallelizing Fortran 90, 1
 C, 1
 C++, 1
 Fortran 77, 1
 Fortran 90, 1
miscellaneous options
 -OPT, 9
 -OPT:alias, 9
 -OPT:alias=disjoint, 10
 -OPT:alias=restrict, 9
 -OPT:roundoff, 10
MP_SET_NUMTHREADS, 16
-mplist flag, 5
 .w2c.c file, 13
 .w2f.f file, 13
-mp option, 29

N

N32 ABI, xi, 1, 6
-n32 compiler option, 6
N64 ABI, xi, 1, 6
nested parallelism, 28

NUM_THREADS, 16

O

O32 ABI, xi, 6
-o32 compiler option, 6
-O3 optimization, 7
 recommended, 5
object code
 N32, xi
 N64, xi
 O32, xi
OMP_DYNAMIC, 16
OMP_NUM_THREADS, 16
OpenMP
 -apo keep flag, 13
 Auto-Parallelizing Fortran 77, 12
 -mplist flag, 13
-OPT, 9
 :alias, 9
 :alias=disjoint, 10
 :alias=restrict, 9
 aliased parameter information, 24
 :roundoff, 10

P

parallelization, 2
 automatic, 2
 manual, 2
parallelization problems
 aliased parameter information, 23
 arbitrary pointer dereferences, 22
 arrays of arrays, 22
 conditionally assigned temporary variable, 21
 data locality, 25
 function calls, 18
 GO TO statements, 18

- hidden knowledge, 20
- indirect array references, 19
- inner loops, 24
- loops bounded by pointer comparisons, 22
- nested parallelism, 28
- trip count, 25
- unanalyzable subscripts, 19
- unknown trip count, 27
- pca flag (deprecated), invoking automatic parallelization, 4
- PCF (Parallel Computing Forum), 12
 - FLIST:emit_pcf option, 13
- permutation array, 35
- pfa flag (deprecated), invoking automatic parallelization, 4
- pragma
 - concurrent, 32
 - concurrent call, 18, 33
 - concurrentize, 32
 - no concurrentize, 32
 - permutation, 35
 - prefer concurrent, 35
 - prefer serial, 36
 - serial, 33
- pragmas, C and C++, 30

R

- restrict
 - aliased parameter information, 23
 - keyword, 23
 - type qualifier, 23

T

- trip count, 25
 - unknown, 27

U

- unanalyzable subscripts, 19

W

- .w2c.c file, 13
- .w2f.f file, 13
- WHIRL, 14
- ws, 5

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 007-3572-002.

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: 650-932-0801
- 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