MIPSpro™ Fortran 77 Programmer's Guide

007–2361–008

Record of Revision

Version	Description
7.3	March 1999. Printing to support the MIPSpro 7.3 release.
008	September 2002 Revision to support the MIPSpro 7.4 release which runs on IRIX version 6.5 and later.

Contents

Figures

Tables

Examples

About This Guide

This guide provides information on implementing FORTRAN 77 programs using the MIPSpro Fortran 77 compiler, version 7.4, which runs on the IRIX operating system, version 6.5 and later. This implementation of FORTRAN 77 contains full American National Standards Institute (ANSI) Programming Language Fortran (X3.9–1978) (in June, 1997, ANSI no longer supported this standard). Extensions provide full VMS Fortran compatibility to the extent possible without the VMS operating system or VAX data representation. This implementation of FORTRAN 77 also contains extensions that provide partial compatibility with programs written in SVS Fortran. This book also describes the Auto-Parallelizing Option (APO) which is an optional software product available for purchase.

The **MIPSpro Fortran 77** compiler supports the `-n32` and `-n64` ABI (Application Binary Interface). The Fortran 77 compiler supports only the `-o32` ABI.

Related Compiler Publications

This manual is one of a set of manuals that describes the compiler. The complete set of manuals is as follows:

- The *MIPSpro Fortran 77 Language Reference Manual* provides a description of the FORTRAN 77 language as implemented on SGI systems.
- The *MIPSpro N32/64 Compiling and Performance Tuning Guide* provides information about improving program performance by using the optimization facilities of the compiler system, the dump utilities, archiver, debugger, and the tools used to maintain Fortran programs.
- The *MIPSpro 64-Bit Porting and Transition Guide* provides an overview of the 64-bit compiler system and language implementation differences, porting source code to the 64-bit system, compilation and run-time issues.
- The *MIPSpro Fortran 90 Commands and Directives Reference Manual* provides information about the Fortran 90 and 95 compiler.
- The `f77`(1), `abi`(5), `lno`(5), `o32`(5), `opt`(5), and `pe_environ`(5) man pages

Compiler Messages

You can obtain compiler message explanations by using the online `explain`(1) command.

Compiler Man Pages

In addition to printed and online prose documentation, several online man pages describe aspects of the compiler. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`(1), `col`(1), and `lpr`(1) commands. In the following example, these commands are used to print a copy of the `explain`(1) man page:

```
% man explain | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `egrep` online, you can type `man grep` or `man egrep`. Both commands display the `grep` man page to your terminal.

Obtaining Publications

Silicon Graphics maintains publications information at the following web site:

`http://techpubs.sgi.com/library`

This library contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

To order a printed Silicon Graphics document, call 1–800–627–9307.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	**Meaning**
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

We value your comments and will respond to them promptly.

Chapter 1

Compiling, Linking, and Running Programs

This chapter provides an overview of the MIPSpro F77 compiler and its use. It contains the following major sections:

- "Compiling and Linking", page 1, describes the compilation environment and how to compile and link Fortran programs. This section also contains examples that show how to create separate linkable objects written in Fortran, C, or other languages supported by the compiler system and how to link them into an executable object program.
- "Compiler Options: an Overview", page 6, provides an overview of debugging, profiling, optimizing, and other options provided with the Fortran `f77` command.
- "Specifying the Buffer Size for Direct Unformatted I/O", page 14, describes the environment variable you can use to specify buffer size.
- "Object File Tools", page 14, briefly summarizes the capabilities of the `elfdump`, `dis`, `nm`, `file`, `size` and `strip` programs that provide listing and other information on object files.
- "Archiver", page 15, summarizes the functions of the `ar` program that maintains archive libraries.
- "Run-Time Considerations", page 15, describes how to invoke a Fortran program, how the operating system treats files, and how to handle run-time errors.

Compiling and Linking

This section discusses compilation and linking issues when using the compiler.

The format of the `f77` command is as follows:

`f77 [`*option*`] ...` *filename*`.[`*suffix*`]`

The *options* argument represents the options you can use with this command. See the `f77`(1) man page for a complete description of the options and their use.

The *filename.suffix* argument is the name of the file that contains the Fortran source statements. The filename must always have the suffix `.f`, `.F`, `.for`, `.FOR`, or `.i`. For example, `myprog.f`.

Compilation

The `f77` command can both compile and link a source module. Figure 1-1 shows the primary compilation phases. It also shows the principal inputs and outputs for the source modules `more.f`.

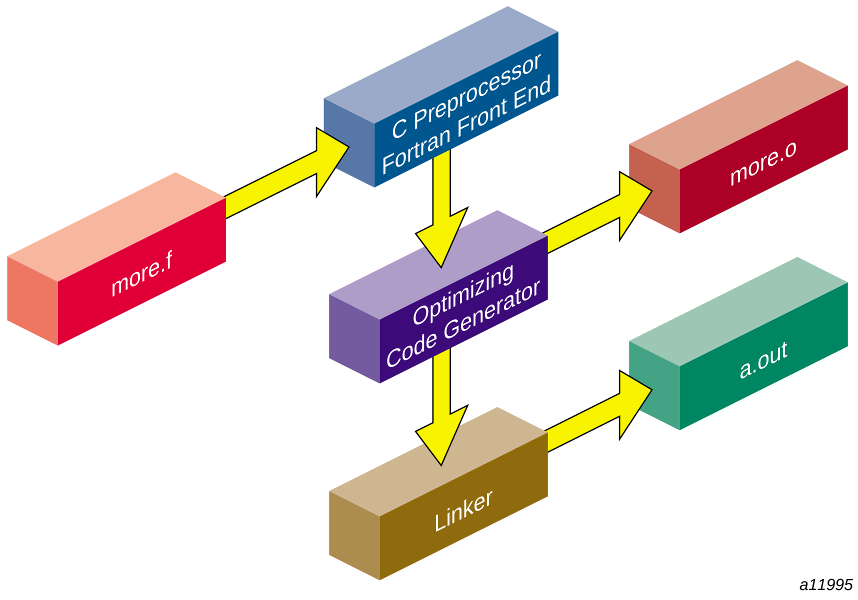

Figure 1-1 Compilation Process

- The source file ends with the required suffixes `.f`, `.F`, `.for`, `.FOR`, or `.i`.
- The Fortran compiler has an integrated C preprocessor that provides full `cpp` capabilities.
- The compiler produces a linkable object file when you specify the `-c` command option. This object file has the same name as the source file, but the suffix is changed to `.o`. For example, the following command line produces the `more.o` file:

  ```
  % f77 more.f -c
  ```

- The default name of the executable object file is `a.out`. For example, the following command line produces the executable object `a.out`:

  ```
  % f77 myprog.f
  ```

- You can specify a name other than `a.out` for the executable object by using the `-o` *name* option, where *name* is the name of the executable object. For example, the following command line links the `myprog.o` object module and produces an executable object named `myprog`:

  ```
  % f77 myprog.o -o myprog
  ```

- The following command line compiles and links the source module `myprog.f` and produces an executable object named `myprog`:

  ```
  % f77 myprog.f -o myprog
  ```

Compiling in C/C++

The compiler system uses drivers that allow you to compile programs for other programming languages, including C and C++. If one of these drivers is installed in your system, you can compile and link your Fortran programs to the language supported by the driver. See the *MIPSpro N32/64 Compiling and Performance Tuning Guide* for a list of available drivers and the commands used with them. See Chapter 3, "Fortran Program Interfaces", page 29, for information about the conventions you must follow when writing Fortran program interfaces to C programs.

When your application has two or more source programs written in different languages, you should compile each program module separately with the appropriate driver and then link them in a separate step. Create objects suitable for linking by specifying the `-c` option, which stops the driver immediately after the assembler phase.

The following two command lines produce linkable objects named `main.o` and `rest.o`, as illustrated in Figure 1-2, page 4.

```
% cc -c main.c
% f77 -c rest.f
```

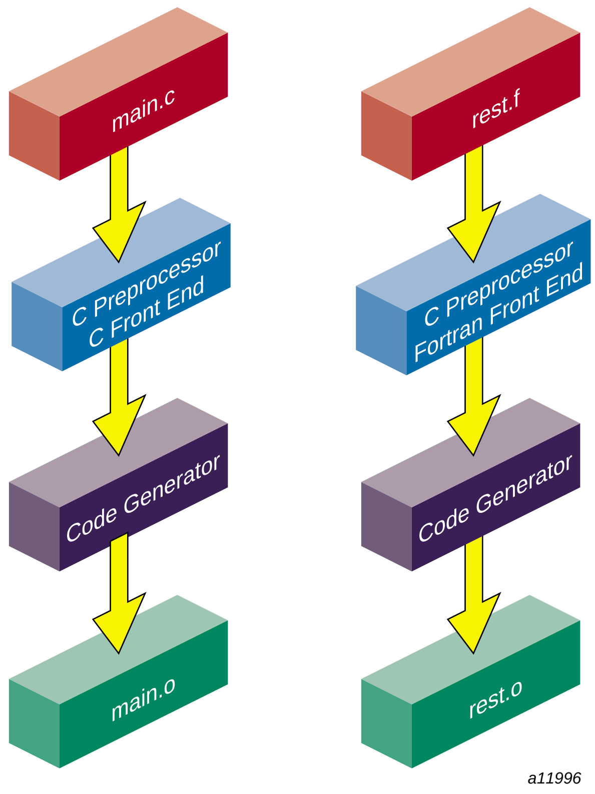

Figure 1-2 Compiling Multilanguage Programs

Linking Objects

You can use the `f77` command to link separate objects into one executable program when any one of the objects is compiled from a Fortran source. The compiler recognizes the `.o` suffix as the name of a file containing object code suitable for linking and it immediately invokes the linker. The following command links the object created in the last example:

```
% f77 -o myprog main.o rest.o
```

You can also use the `cc` command, as shown in this example:

```
% cc -o myprog main.o rest.o -lftn -lm
```

Figure 1-3, page 5 shows the flow of control for this link.

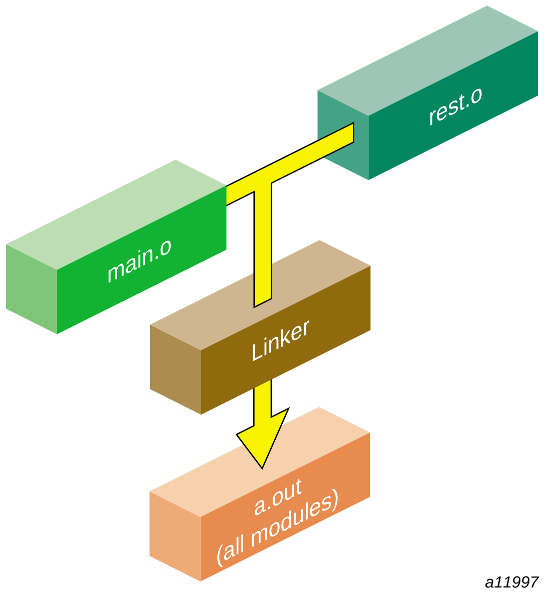

Figure 1-3 Linking

Both `f77` and `cc` use the C link library by default. However, you must specify Fortran libraries explicitly to the linker using the `cc -l` option as shown in the previous example. The characters following `-l` in the previous example are shorthand for link library files described in the following table.

Path	Contents
library `ftn` in `/usr/lib*/nonshared/libftn.a`	Intrinsic function, I/O, multiprocessing, IRIX interface, and indexed sequential access method library for nonshared linking and compiling
library `ftn` in `/usr/lib*/libftn.so`	Same as above, except for shared linking and compiling (this is the default library)

library `m` in `/usr/lib*/libm.so`

Mathematics library

See the FILES section of the `f77`(1) reference page for a complete list of the files used by the Fortran driver. Also see the `ld`(1) man page for information on specifying the `-l` option.

Specifying Link Libraries

You may need to specify libraries when you use IRIX system packages that are not part of a particular language. Most of the man pages for these packages list the required libraries. For example, the `getwd`(3B) subroutine requires the BSD compatibility library `libbsd.a`. Specify this library as follows:

```
% f77 main.o more.o rest.o -lbsd
```

To specify a library created with the archiver, type in the pathname of the library as shown below.

```
% f77 main.o more.o rest.o libfft.a
```

Note: The linker searches libraries in the order you specify. Therefore, if you have a library (for example, `libfft.a`) that uses data or procedures from `-lm`, you **must** specify `libfft.a` first.

Compiler Options: an Overview

This section contains an overview of the Fortran–specific compiler options, such as those that specify input/output files, or specify source file format. For complete information about the compiler options, see the `f77` man page. You can also see the following documents for further information:

- The *MIPSpro N32/64 Compiling and Performance Tuning Guide* for a discussion of the compiler options that are common to all MIPSpro compilers.
- The `apo`(1) man page for options related to the parallel optimizer.
- The `ld`(1) man page for a description of the linker options.

Tip: The `f77 -help` command lists all compiler options for quick reference. Use the `-show` option to direct that the compiler document each phase of execution, showing the exact default and nondefault options passed to each.

- The `lno`(5) man page, for details about the Loop Nest Optimizer.
- The `opt`(5) man page, for details about optimization options to the compiler.

Compiling Simple Programs

You need only a few compiler options when you are compiling a simple program. Examples of simple programs include the following:

- Test cases used to explore algorithms or Fortran language features
- Programs that are mainly interactive
- Programs with performance that is limited by disk I/O
- Programs that will execute under a debugger

In these cases you need only specify the `-g` option for debugging, specify the target machine architecture, and specify the word-length. For example, to compile a single source file to execute under `dbx`, you could use the following commands.

```
f77 -g -mips4 -n32 -o testcase testcase.f
dbx testcase
```

However, a program compiled in this way takes little advantage of the performance features of the machine. In particular, its speed when doing heavy floating-point calculations will be far slower than the machine capacity. For simple programs, however, that is not important.

Using a Defaults Specification File

You can set the Application Binary Interface (ABI), instruction set architecture (ISA), and processor type without explicitly specifying them. Set the `COMPILER_DEFAULTS_PATH` environment variable to a colon-separated list of paths indicating where the compiler should check for a `compiler.defaults` file. If no `compiler.defaults` file is found or if the environment variable is not set, the compiler looks for `/etc/compiler.defaults`. If this file is not found, the compiler uses the built-in defaults.

The `compiler.defaults` file contains a `-DEFAULT:`*option* group command that specifies the default ABI, ISA, and processor. The compiler issues a warning if you specify anything other than `-DEFAULT:` *option* in the `compiler.defaults` file.

The format of the `-DEFAULT:` *option* group specifier is as follows:

```
-DEFAULT:[abi=abitype] [:isa= isatype] [:proc=rtype] [:opt=level]
  [:arith=number]
```

See the `f77`(1) man page for an explanation of the arguments and their allowed values.

Specifying Features to the Compiler

There are several specific features which you may wish to use as you compile your programs. The following tables discuss these features:

- target machine features
- source file formats
- input and output files
- memory allocation and alignment
- debugging and profiling
- optimization levels
- multiprocessing options
- recursion options
- compiler execution options

The following options are used to specify the characteristics of the machine where the compiled program will be used.

Table 1-1 Machine Characteristic Options

Options	Purpose
`-64`, `-n32`, `-o32`	Specifies that the target machine runs 64-bit mode, "new" 32-bit mode available with IRIX 6.1 and above, or old 32-bit mode. The `-64` option is allowed only with the `-mips3` and `-mips4` architecture options.
`-mips3`, `-mips4`	The instruction architecture available in the target machine. Use `-mips3` for MIPS R4000 and R4400® machines; use `-mips4` for MIPS R8000, R10000™ and R12000 machines.
`-TARG:`*option*`,...`	Specify certain details of the target CPU. Many of these options have correct default values based on the preceding options.
`-TENV:`*option*`,...`	Specify certain details of the software environment where the source module will execute. Most of these options have correct default values based on other, more general values.

The following options specify the source file format for files used by the compiler.

Table 1-2 Source File Format Options

Options	Purpose
`-ansi`	Report any nonstandard usages.
`-backslash`	Treat \ in character literals as a character, not as the start of an escape sequence.
`-col72`, `-col120`, `-extend_source`, `-noextend_source`	Specify margin columns of source lines.
`-d_lines`	Compile lines with `D` in column 1.
`-D`*name* `-D`*name=def*, `-U`*name*	Define/undefine names to the integrated C preprocessor.

The following options direct the compiler to use specific input files and to generate specific output file.

Table 1-3 Input/Output File Options

Options	Purpose
`-c`	Generate a single object file for each input file; do not link.
`-E`	Run only the macro preprocessor and write its output to standard output.
`-I`, `-I`*dir*, `-nostdinc`	Specify location of include files.
`-listing`	Request a listing file.
`-MDupdate`	Request Makefile dependency output data.
`-o`	Specify name of output file.
`-S`	Specify only assembly-language source output.

The following options direct the compiler how to allocate memory and how to align variables in memory. These options can affect both program size and program speed.

Table 1-4 Memory Allocation Options

Options	Purpose
`-align`*n*	Align all variables of size *n* on *n*-byte address boundaries. Valid values for *n* are `6`, `8`, `32`, or `64`. When using `32`, objects **larger** than 32 bits can be aligned on 32–bit boundaries. 16–bit objects must be aligned on 16–bit boundaries, and 32–bit objects must be aligned on 32–bit boundaries.
`-d`*n*	Specify the size of `DOUBLE` and `DOUBLE COMPLEX` variables.
`-i`*n*	Specify the size of `INTEGER` and `LOGICAL` variables.
`-r`*n*	Specify the size of `REAL` and `COMPLEX` variables.
`-static`	Allocate all local variables statically, not dynamically on the stack.

The following option directs the compiler to include more or less extra information in the object file for debugging.

Table 1-5 Debugging Option

Option	Purpose
`-g`*level*	Leave more or less symbol-table information in the object file for use with `dbx` or Workshop Pro `cvd`.

The following optimization options are used to communicate to the different optimization phases. The optimizations that are common to all MIPSpro compilers are discussed in the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

Table 1-6 Optimization Options

Options	Purpose
`-O`*level*	Select basic level of optimization, setting defaults for all optimization phases. *level* can be a value from `0` to `3`.
`-INLINE:`*option*`,...`	Standalone inliner option group to control application of intra-file subprogram inlining when interprocedural analysis is not enabled. See the `ipa`(5) man page for more information
`-IPA:`*option*`,...`	Specify Inter-Procedural Analyzer option group to control application of inter-procedural analysis and optimization, including inlining, common block array padding, constant propagation, dead function elimination, alias analysis and others.
`-LNO:`*option*`,...`	Loop nest optimizer (LNO) option control group to control optimizations and transformations performed by LNO. See the `LNO`(5) referece page for more information.
`-OPT:`*option*`,...`	Specify miscellaneous details of optimization. See the `OPT`(5) man page for more information.
`-apo`	Request execution of the parallelizing optimizer.

In addition to optimizing options, the compiler system provides other options that can improve the performance of your programs:

- A linker option, `-G`, controls the size of the global data area, which can produce significant performance improvements. See the *MIPSpro N32/64 Compiling and Performance Tuning Guide*, and the `ld`(1) man page for more information.

- The linker's `-jmpopt` option permits the linker to fill certain instruction delay slots not filled by the compiler front end. This option can improve the performance of smaller programs not requiring extremely large blocks of virtual memory. See the `ld`(1) man page for more information.

The `f77` compiler has several options related to multiprocessing. However, the associated programs and files are not present unless you install Power Fortran 77. The following list summarizes some of the multiprocessing options:

Table 1-7 Multiprocessing Options

Option	**Description**
`-MP:`*options*	The multiprocessing options group enables or disables multiprocessing features. All of the features are enabled by default with the `-mp` option. The following are the individual controls in this group: `dsm=`*flag*. Enables or disables data distribution. *flag* can be either `ON` or `OFF`. Default: `ON`. `clone=`*flag*. Enables or disables auto-cloning. *flag* can be either `ON` or `OFF`. The compiler automatically clones procedures that are called with reshaped arrays as parameters for the incoming distribution. However, if you explicitly specify the distribution on all relevant formal parameters, then you can disable auto-cloning with `-MP:clone=off`. The consistency checking of the distribution between actual and formal parameters is not affected by this flag, and is always enabled. Default: `ON`. `check_reshape=`*flag*`.` Enables or disables generation of the runtime consistency checks across procedure boundaries when passing reshaped arrays (or portions thereof) as parameters. *flag* can be either `ON` or `OFF`. Default: `OFF`. `open_mp=`*flag*`.` Enables or disables compiler to use OpenMP directives. *flag* can be either `ON` or `OFF`. Default: `ON`. `old_mp=`*flag*`.` Enables or disables recognition of the PCF directives. *flag* can be either `ON` or `OFF`.

`-mp`

Enable the multiprocessing and DSM directives. Use this option with either the `-mp` or the `-apo` option. The saved file name has the following form:

`$TMPDIR/P<`*user_subroutine_name*`><`*machine_name*`><pid>`

If the `TMPDIR` environment variable is not set, then the file is in `/tmp`.

`-mp_keep`

Keep the compiler generated temporary file and generate correct line numbers for debugging multiprocessed DO loops.

`-apo`

Run the `apo`(1) preprocessor to automatically discover parallelism in the source code. This also enables the multiprocessing directives. This is an optional software product.

Note: Under `-mp` compilation, the compiler silently generates some bookkeeping information under the `rii_files` directory. This information is used to implement data distribution directives, as well as perform consistency checks of these directives across multiple source files. To disable the processing of the data distribution directives and not generate the `rii_files`, compile your program with the `-MP:dsm=off` option.

You can enable recursion support by using the `-LANG:recursive=ON` option.

In either mode, the compiler supports a recursive stack-based calling sequence. The difference is in the optimization of statically allocated local variables. The following list describes the `-LANG:recursive=` option:

Table 1-8 Recursion Options

Recursive Option	**Purpose**
`=on`	A statically allocated local variable can be referenced or modified by a recursive procedure call. The statically allocated local variable must be stored in memory before making a call and reloaded afterward.

`=off`	The default. The compiler can safely assume a statically allocated local variable will not be referenced or modified by a procedure call and can optimize more aggressively.

The following options control the execution of the compiler phases.

Table 1-9 Compiler Execution Options

Option	**Purpose**
`-E`, `-P`	Execute only the integrated C preprocessor.
`-fe`	Stop compilation immediately after the front-end (syntax analysis) runs.
`-M`	Run only the macro preprocessor.
`-Y`*c*`,`*path*	Load the compiler phase specified by *c* from the specified *path*.
`-W`*c*`,`*option*`,...`	Pass the specified list of options to the compiler phase specified by *c*.

Specifying the Buffer Size for Direct Unformatted I/O

You can use the `FORTRAN_BUFFER_SIZE` environment variable to change the buffer size for direct unformatted I/O. After it is set to 128K (4-byte) words or greater, the I/O on direct unformatted file does not go though the system buffer. No upper limit exists on the number to which you can set `FORTRAN_BUFFER_SIZE`. However, when it exceeds the system maximum I/O limit, then the Fortran I/O library automatically resets it to the system limit.

See the `pe_environ`(5) man page for more information.

Object File Tools

The following tools provide information on object files:

`elfdump`	Lists headers, tables, and other selected parts of an ELF-format object or archive file.
`dis`	Disassembles object files into machine instructions.
`nm`	Prints symbol table information for object and archive files.

`file`	Lists the properties of program source, text, object, and other files. This tool often erroneously recognizes command files as C programs. It does not recognize Pascal or LISP programs.
`size`	Prints information about the text, rdata, data, sdata, bss, and sbss sections of the specified object or archive files. See the `a.out`(4) man page for a description of the contents and format of section data.
`strip`	Removes symbol table and relocation bits.

For more information about these tools, see the *MIPSpro N32/64 Compiling and Performance Tuning Guide* and the `dis`(1), `elfdump`(1), `file`(1), `nm`(1), `size`(1), and `strip`(1) man pages.

Archiver

An archive library is a file that contains one or more routines in object (`.o`) file format. The term *object* refers to a `.o` file that is part of an archive library file. When a program calls an object not explicitly included in the program, the link editor, `ld`, looks for that object in an archive library. The link editor then loads only that object (not the whole library) and links it with the calling program.

The archiver (`ar`) creates and maintains archive libraries and has the following main functions:

- copying new objects into the library
- replacing existing objects in the library
- moving objects about the library
- copying individual objects from the library into individual object files

See the *MIPSpro N32/64 Compiling and Performance Tuning Guide*, and the `ar`(1) man page for additional information about the archiver.

Run-Time Considerations

There are several aspects of compiling that you should consider at run-time. This section discusses some of those aspects:

- invoking a program, "Invoking a Program", page 16

- memory allocation, "Maximum Memory Allocations", page 16
- file formats, "File Formats", page 19
- preconnected files, "Preconnected Files", page 20
- file positions, "File Positions", page 20
- unknown file status, "Unknown File Status", page 20
- quad-precision operations, "Quad-Precision Operations", page 21
- error handling, "Run-Time Error Handling", page 21
- floating point exceptions, "Floating Point Exceptions", page 21

Invoking a Program

To run a Fortran program, invoke the executable object module produced by the `f77` command by entering the name of the module as a command. By default, the name of the executable module is `a.out`. If you included the `-o` *filename* option on the `ld` (or the `f77`) command line, the executable object module has the name that you specified.

Maximum Memory Allocations

The total memory allocation for a program, and individual arrays, can exceed 2 gigabytes (2 GB, or 2,048 MB).

Previous implementations of FORTRAN 77 limited the total program size, as well as the size of any single array, to 2 GB. The current release allows the total memory in use by the program to exceed this. For details about the memory use of individual scalar values, see "Alignment, Size, and Value Ranges", page 23.

Arrays Larger Than 2 Gigabytes

The compiler supports arrays that are larger than 2 gigabytes for programs compiled under the `-64` ABI option. The arrays can be local, global, and dynamically created as the following example demonstrates. Initializers are not provided for the arrays in these examples. Large array support is limited to FORTRAN 77, C, and C++.

```
$cat a2.c

#include <stdlib.h>
```

```
int i[0x100000008];

void foo()
{
int k[0x100000008];
 k[0x100000007] = 9;
 printf(''%d \n'', k[0x100000007]);
}

main()
{
char *j;
j = malloc(0x100000008);
 i[0x100000007] = 7;
 j[0x100000007] = 8;
 printf(''%d \n'', i[0x100000007]);
 printf(''%d \n'', j[0x100000007]);
 foo();
}
```

You must run this program on a 64-bit operating system with IRIX version 6.2 or a higher version. You can verify the system type by using the `uname -a` command. You must have enough swap space to support the working set size and you must have your shell limit datasize, stacksize, and vmemoryuse variables set to values large enough to support the sizes of the arrays. See the `sh`(1) man page for details.

The following example compiles and runs the above code after setting the stacksize to a correct value:

```
$uname -a
IRIX64 cydrome 6.2 03131016 IP19
$cc -64 -mips3 a2.c
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       65536 kbytesn
coredumpsize    unlimited
memoryuse
descriptors     200
vmemoryuse      unlimited
```

```
$limit stacksize unlimited
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       unlimited
coredumpsize    unlimited
memoryuse       754544 kbytes
descriptors     200
vmemoryuse      unlimited
$a.out
7
8
9
```

Local Variable (Stack Frame) Sizes

Arrays that are allocated on the process stack must not exceed 2 GB, but the total of all stack variables can exceed that limit, as in this example:

```
parameter (ndim = 16380)
integer*8 xmat(ndim,ndim), ymat(ndim,ndim), &
    zmat(ndim,ndim)
integer k(1073741824)
integer l(33554432, 256)
```

However, when an array is passed as an argument, it is not limited in size.

```
subroutine abc(k)
integer k(8589934592_8)
```

Static and Common Sizes

When compiling with the `-static` option, global data is allocated as part of the compiled object (`.o`) file. The total size of any `.o` file may not exceed 2 GB. However, the total size of a program linked from multiple `.o` files may exceed 2 GB.

An individual common block may not exceed 2 GB. However, you can declare multiple common blocks, each having that size.

Pointer-based Memory

There is no limit on the size of a pointer-based array, as in this example:

```
integer *8 ndim
parameter (ndim = 20001)
pointer (xptr, xmat), (yptr, ymat), (zptr, zmat), &
    (aptr, amat)
xptr = malloc(ndim*ndim*8)
yptr = malloc(ndim*ndim*8)
zptr = malloc(ndim*ndim*8)
aptr = malloc(ndim*ndim*8)
```

Be sure that `malloc` is called with an `INTEGER*8` value. A count greater than 2 GB would be truncated if assigned to an `INTEGER*4`.

File Formats

The Fortran compiler supports five kinds of external files:

- sequential formatted
- sequential unformatted
- direct formatted
- direct unformatted
- key indexed file

The operating system implements other files as ordinary files and makes no assumptions about their internal structure.

Fortran I/O is based on records. When a program opens a direct file or a key-indexed file, the length of the records must be given. The Fortran I/O system uses the length to make the file appear to be composed of records of the given length. When the record length of a direct unformatted file is 1 byte, the system treats the file as ordinary system files (that is, as byte strings, in which each byte is addressable). A `READ` or `WRITE` request on such files consumes bytes until they are used, rather than restricting the request to a single record.

Because of special requirements, sequential unformatted files are usually read or written only by Fortran I/O statements. Each record is preceded and followed by an integer containing the length of the record in bytes.

During a `READ`, Fortran I/O breaks sequential formatted files into records by using each new line indicator as a record separator. The FORTRAN 77 standard does not define the required result after reading past the end of a record; the I/O system treats the record as being extended by blanks. On output, the I/O system writes a new line indicator at the end of each record. If a user's program also writes a new line indicator, the I/O system treats it as a separate record.

Preconnected Files

The following table shows the standard preconnected files at program start.

Table 1-10 Preconnected Files

Unit#/Unit	**Alternate Unit**
5 (standard input)	(in `READ`)
6 (standard output)	(in `WRITE`)
0 (standard error)	(in `WRITE`)

All other units are also preconnected when execution begins. Unit *n* is connected to a file named `fort.n`. These files need not exist, nor will they be created unless their units are used without first executing an open statement. The default connection is for sequentially formatted I/O.

File Positions

The FORTRAN 77 standard does not specify where `OPEN` should initially position a file that is explicitly opened for sequential I/O. The I/O system positions the file to start of file for both input and output. The execution of an `OPEN` statement followed by a `WRITE` on an existing file causes the file to be overwritten, erasing any data in the file. In a program called from a parent process, units 0, 5, and 6 remain where they were positioned by the parent process.

Unknown File Status

When the `STATUS="UNKNOWN"` parameter is specified in an `OPEN` statement, the following occurs:

- If the file does not exist, it is created and positioned at start of file.

- If the file exists, it is opened and positioned at the beginning of the file.

Quad-Precision Operations

When running programs that contain quad-precision operations, you must run the compiler in round-to-nearest mode. Because this mode is the default, you usually do not have to set it. You will need to set this mode when writing programs that call your own assembly routines. Refer to the `swapRM`(3c) man page for details.

Run-Time Error Handling

When the Fortran run-time system detects an error, the following actions occur:

- A message describing the error is written to the standard error unit (unit 0).
- A core file is produced if the `f77_dump_flag` environment variable is set. You can use `dbx` to inspect this file and determine the state of the program at termination. For more information, see the *dbx User's Guide*.

  To invoke `dbx` using the core file, enter the following command:

  ```
  % dbx executable core
  ```

  where *executable* is the name of the executable file (the default is **a.out**).

Floating Point Exceptions

The `libfpe` library provides two methods for handling floating point exceptions.

The library provides the `handle_sigfpes` subroutine and the `TRAP_FPE` environment variable. Both methods provide mechanisms for handling and classifying floating point exceptions, and for substituting new values. They also provide mechanisms to count, trace, exit, or abort on enabled exceptions. The `-TENV:check_div` compile option inserts checks for divide by zero or overflow. See the `handle_sigfpes`(3f) man page for more information.

Chapter 2

Data Types and Mapping

This chapter describes how the Fortran compiler implements size and value ranges for various data types. In addition, data alignment and accessing misaligned data is also discussed.

For more information about data representation and storage, see the *MIPSpro Fortran Language Reference Manual, Volume 3*.

Alignment, Size, and Value Ranges

Table 2-1 contains information about various Fortran scalar data types. For details on the maximum sizes of arrays, see "Maximum Memory Allocations", page 16.

Table 2-1 Size, Alignment, and Value Ranges of Data Types

Type	Synonym	Size	Alignment	Value Range
`BYTE`	`INTEGER*1`	8 bits	Byte	`-128…127`
`INTEGER*2`		16 bits	Half word	`-32,768…32,767`
`INTEGER`	`INTEGER*4`: When the `-i2` option is used, type `INTEGER` is equivalent to `INTEGER*2`; when the `-i8` option is used, `INTEGER` is equivalent to `INTEGER*8`.	32 bits	Word	$-2^{31} ... 2^{31}-1$
`INTEGER*8`		64 bits	Double word	$-2^{63}…2^{63} -1$
`LOGICAL*1`		8 bits	Byte	`0 1`
`LOGICAL*2`		16 bits	Half word	`0 1`

Type	Synonym	Size	Alignment	Value Range
LOGICAL	LOGICAL*4: When the -i2 option is used, type LOGICAL is equivalent to LOGICAL*2; when the -i8 option is used, type LOGICAL is equivalent to LOGICAL*8.	32 bits	Word	0 1
LOGICAL*8		64 bits	Double word	0 1
REAL	REAL*4: When the -r8 option is used, type REAL is equivalent to REAL*8.	32 bits	Word	See Table 2-2, page 26
DOUBLE PRECISION	REAL*8: When the –d16 option is used, type DOUBLE PRECISION is equivalent to REAL*16.	64 bits	Double word: Byte boundary divisible by eight.	See Table 2-2, page 26
REAL*16		128 bits	Double word	See Table 13
COMPLEX	COMPLEX*8: When the -r8 option is used, type COMPLEX is equivalent to COMPLEX*16.	64 bits	Double word: Byte boundary divisible by four.	See the first bullet item below
DOUBLE COMPLEX	COMPLEX*16: When the -d16 option is used, type DOUBLE COMPLEX is equivalent to COMPLEX*32.	128 bits	Double word: Byte boundary divisible by eight.	See the first bullet item below
COMPLEX*32		256 bits	Double word	See the first bullet item below
CHARACTER		8 bits	Byte	-128…127

When the alignment is **half word**, the byte boundary is divisible by two. When the alignment is **word**, the byte boundary is divisible by four.

The following notes provide details on some of the items in Table 2-1.

- Forcing INTEGER, LOGICAL, REAL, and COMPLEX variables to align on a halfword boundary is not allowed, except as permitted by the -align8, -align16, and -align32 command line options.
- Table 2-1 indicates that REAL*8 (that is, DOUBLE PRECISION) variables always align on a double-word boundary. However, Fortran permits these variables to align on a word boundary if a COMMON statement or equivalencing requires it.
- A COMPLEX data item is an ordered pair of REAL*4 numbers; a DOUBLE COMPLEX data item is an ordered pair of REAL*8 numbers; a COMPLEX*32 data item is an ordered pair of REAL*16 numbers. In each case, the first number represents the real part and the second represents the imaginary part. The following tables list the valid ranges.
- LOGICAL data items denote only the logical values TRUE and FALSE (written as .TRUE. or .FALSE.). However, to provide VMS compatibility, LOGICAL variables can be assigned all integral values of the same size.
- You must explicitly declare an array in a DIMENSION declaration or in a data type declaration. To support DIMENSION, the compiler does the following:
  - allows up to seven dimensions
  - assigns a default of 1 to the lower bound if a lower bound is not explicitly declared in the DIMENSION statement
  - creates an array the size of its element type times the number of elements
  - stores arrays in column-major mode
- The following rules apply to shared blocks of data set up by COMMON statements:
  - The compiler assigns data items in the same sequence as they appear in the common statements defining the block. Data items are padded according to the alignment compiler options or the compiler defaults. See "Access of Misaligned Data", page 27, for more information.
  - You can allocate both character and noncharacter data in the same common block.
  - When a common block appears in multiple program units, the compiler allocates the same size for that block in each unit, even though the size required may differ (due to varying element names, types, and ordering sequences) from unit to unit. The allocated size corresponds to the maximum size required by the block among all the program units except when a common

block is defined by using DATA statements, which initialize one or more of the common block variables. In this case the common block is allocated the same size as when it is defined.

- Table 2-2 lists the approximate valid ranges for REAL*4 and REAL*8 .

Table 2-2 Valid Ranges for REAL*4 and REAL*8 Data Types

Range	REAL*4	REAL*8
Maximum	3.40282356 * 10^{38}	1.7976931348623158 * 10^{308}
Minimum normalized	1.17549424 * 10^{-38}	2.2250738585072012 * 10^{-308}
Minimum denormalized	1.40129846 * 10^{-45}	4.94065645841246544 * 10^{-324}

- REAL*16 constants have the same form as DOUBLE PRECISION constants, except the exponent indicator is Q instead of D. The following table lists the approximate valid range for REAL*16. REAL*16 values have an 11-bit exponent and a 107-bit mantissa; they are represented internally as the sum or difference of two doubles. Therefore, for REAL*16, "normal" means that both high and low parts are normals.

Table 2-3 Valid ranges for REAL*16 Data Types

Range	**Precise Exception Mode w/FS Bit Clear**
Maximum	1.797693134862315807937289714053023 * 10^{308}
Minimum normalized	2.0041683600089730005034939020703004 * 10^{-292}
Minimum denormalized	4.940656458412465441765687928682214 * 10^{-324}
	Fast Mode Precise Exception Mode w/FS Bit Set
Maximum	1.797693134862315807937289714053023 * 10^{308}
Minimum normalized	2.0041683600089730005034939020703004 * 10^{-292}

Minimum denormalized	`2.2250738585072013830902327173324 * 10`$^{-308}$

Access of Misaligned Data

The Fortran compiler allows misalignment of data if specified by special options.

The architecture of the IRIS 4D series assumes a particular alignment of data. ANSI standard FORTRAN 77 cannot violate the rules governing this alignment. Misalignment can occur when using common extensions. This is particularly true for small integer types, which have the following characteristics:

- allow intermixing of character and non-character data in `COMMON` and `EQUIVALENCE` statements
- allow mismatching the types of formal and actual parameters across a subroutine interface
- provide many opportunities for misalignment to occur

Code that use extensions that compile and execute correctly on other systems with less stringent alignment requirements may fail during compilation or execution on the IRIS 4D. This section describes a set of options to the Fortran compiler that allow the compilation and execution of programs whose data may be misaligned. The execution of programs that use these options is significantly slower than the execution of a program with aligned data.

This section describes the two methods that can be used to create an executable object file that accesses misaligned data.

Accessing Small Amounts of Misaligned Data

Use this method if the number of instances of misaligned data access is small or use it to provide information on the occurrence of such accesses so that misalignment problems can be corrected at the source level.

This method catches and corrects bus errors due to misaligned accesses. This ties the extent of program degradation to the frequency of these accesses. This method also includes capabilities for producing a report of these accesses to enable their correction.

To use this method, use one of the following two options to the `f77` command to prevent the compiler from padding data to force alignment:

- Use the `-align8` option if you do not anticipate that your program will have restrictions on alignment.
- Use the `-align16` option if your program must be run on a machine that requires half-word alignment.

You must also use the misalignment trap handler. This requires minor source code changes to initialize the handler and the addition of the handler binary to the link step. See the `fixade`(3f) reference page for details.

Accessing Misaligned Data Without Modifying Source Code

Use this second method for programs with widespread misalignment or whose source code may not be modified.

In this method, a set of special instructions is substituted by the IRIS 4D assembler for data accesses whose alignment cannot be guaranteed. You can choose to have each source file independently substituted.

You can invoke this method by specifying one of the `-align` alignment options (`-align8`, `-align16`) to `f77` when compiling any source file that references misaligned data. If your program passes misaligned data to system libraries, you may also have to link it with the trap handler. See the `f77`(1) reference page and the `fixade`(3f) reference page for more information.

Fortran Program Interfaces

Sometimes it is necessary to create a program that combines modules written in Fortran and another programming language. For example,

- In a Fortran program, you may need access to a facility that is only available as a C function, such as a member of a graphics library.
- In a program in another programming language, you may need access to a computation that has been implemented as a Fortran subprogram (perhaps one of the many BLAS library routines).

This chapter focuses on the interface between Fortran and the C programming language. However other language can be called (for example, C++).

Note: You should be aware that all compilers for a given version of IRIX use identical standard conventions for passing parameters in generated code. These conventions are documented at the machine instruction level in the *MIPSpro Assembly Language Programmer's Guide*, which also details the differences in the conventions used in different releases.

Subprogram Names

The Fortran compiler normally changes the names of subprograms and named common blocks while it translates the source file. When these names appear in the object file for reference by other modules, they are usually changed in two ways:

- Converted to all lowercase letters
- Extended with a final underscore (_) character

The following declarations usually produce the `matrix_`, `mixedcase_`, and `cblk_` identifiers (all in lowercase with appended underscores) in the generated object file:

```
SUBROUTINE MATRIX
function MixedCase()
COMMON /CBLK/a,b,c
```

Note: Fortran intrinsic functions are not named according to these rules. The external names of intrinsic functions as defined in the Fortran library are not directly related to the intrinsic function names as they are written in a program. The use of intrinsic function names is discussed in the *MIPSpro Fortran 77 Language Reference Manual*.

Mixed-Case Names

The Fortran compiler will not generate an external name containing uppercase letters. If you are porting a program that depends on the ability to call such a name, you must write a C function that takes the same arguments but which has a name composed of lowercase letters only. This C function can then call the function whose name contains mixed-case letters.

Note: Previous versions of the FORTRAN 77 compiler for 32-bit systems supported the `-U` compiler option, which directed the compiler to not force all uppercase input to lowercase. As a result, uppercase letters could be preserved in external names in the object file. As now implemented, this option does not affect the case of external names in the object file.

Preventing a Suffix Underscore with $

You can prevent the compiler from appending an underscore to a name by writing the name with a terminal currency symbol (`$`). The `$` is not reproduced in the object file; it is dropped, but it prevents the compiler from appending an underscore. The following declaration produces the name `nounder` (lowercase, but with no trailing underscore) in the object file:

```
EXTERNAL NOUNDER$
```

Note: This meaning of `$` in names applies only to subprogram names. If you end the name of a `COMMON` block with `$`, the name in the object file includes the `$` and ends with an underscore.

Naming Fortran Subprograms from C

In order to call a Fortran subprogram from a C module you must spell the name the way the Fortran compiler spells it, using all lowercase letters and a trailing underscore. A Fortran subprogram declared as follows:

```
SUBROUTINE HYPOT()
```

would typically be declared in the following C function (lowercase with trailing underscore):

```
extern int hypot_()
```

You must know if a subprogram is declared with a trailing `$` to suppress the underscore.

Naming C Functions from Fortran

The C compiler does not modify the names of C functions. C functions can have uppercase or mixed-case names, and they have terminal underscores only when specified.

In order to call a C function from a Fortran program you must ensure that the Fortran compiler spells the name correctly. When you control the name of the C function, the simplest solution is to give it a name that consists of lowercase letters with a terminal underscore. For example, the following C function:

```
int fromfort_() {...}
```

could be declared in a Fortran program as follows:

```
EXTERNAL FROMFORT
```

When you do not control the name of a C function, you must direct the Fortran compiler to generate the correct name in the object file. Write the C function name using a terminal `$` character to suppress the terminal underscore. The compiler will not generate an external name with uppercase letters in it.

Verifying Spelling Using `nm`

You can verify the spelling of names in an object file using the `nm`(1) command (or with the `elfdump` command with the `-t` or `-Dt` options). To see the subroutine and common names generated by the compiler, use the nm command with the generated .o (object) or executable file.

Correspondence of Fortran and C Data Types

When you exchange data values between Fortran and C, either as parameters, as function results, or as elements of common blocks, you must make sure that the two languages agree on the size, alignment, and subscript of each data value.

Corresponding Scalar Types

The correspondence between Fortran and C scalar data types is shown in Table 3-1. This table assumes the default precisions. Using compiler options such as -i2 or -r8 affects the meaning of the words LOGICAL, INTEGER, and REAL.

Table 3-1 Corresponding Fortran and C Data Types

Fortran Data Type	Corresponding C type
BYTE, INTEGER*1, LOGICAL*1	signed char
CHARACTER*1	unsigned char
INTEGER*2, LOGICAL*2	short
INTEGER[1], INTEGER*4, LOGICAL[1], LOGICAL*4	int or long
INTEGER*8, LOGICAL*8	long long
REAL[1], REAL*4	float
DOUBLE PRECISION, REAL*8	double
REAL*16	long double
COMPLEX[1], COMPLEX*8	typedef struct{float real, imag; } cpx8;
DOUBLE COMPLEX, COMPLEX*16	typedef struct{ double real, imag; } cpx16;
COMPLEX*32	typedef struct{long double real, imag;} cpx32;

Fortran Data Type	Corresponding C type
`CHARACTER*`*n* (*n*>1)	`typedef char fstr_`*n*`[`*n*`];`
[1] assuming default precision	

The rules governing alignment of variables within common blocks are discussed in "Alignment, Size, and Value Ranges", page 23.

Corresponding Character Types

The Fortran `CHARACTER*1` data type corresponds to the C type unsigned `char`. However, the two languages differ in the treatment of strings of characters.

A Fortran `CHARACTER*`*n* (*n*>1) variable contains exactly *n* characters at all times. When a shorter character expression is assigned to it, it is padded on the right with spaces to reach *n* characters.

A C vector of characters is normally sized 1 greater than the longest string assigned to it. It may contain fewer meaningful characters than its size allows, and the end of meaningful data is marked by a null byte. There is no null byte at the end of a Fortran string. You can create a null byte using the Hollerith constant `\0` but it is not usually done.

Because there is no terminal null byte, most of the string library functions familiar to C programmers (`strcpy()`, `strcat()`, `strcmp()`, and so on) cannot be used with Fortran string values. The `strncpy()`, `strncmp()`, `bcopy()`, and `bcmp()` functions can be used because they depend on a count rather than a delimiter.

Corresponding Array Elements

Fortran and C use different arrangements for the elements of an array in memory. Fortran uses column-major order (when iterating sequentially through memory, the leftmost subscript varies fastest), whereas C uses row-major order (the rightmost subscript varies fastest to generate sequential storage locations). In addition, Fortran array indices are normally origin-1, while C indices are origin-0.

To use a Fortran array in C, you must:

- Reverse the order of dimension limits when declaring the array.
- Reverse the sequence of subscript variables in a subscript expression.

- Adjust the subscripts to origin-0 (usually, decrement by 1).

The correspondence between Fortran and C subscript values is depicted in Figure 3-1, page 34. You can derive the C subscripts for a given element by decrementing the Fortran subscripts and using them in reverse order; for example, Fortran `(99,9)` corresponds to C `[8][98]`.

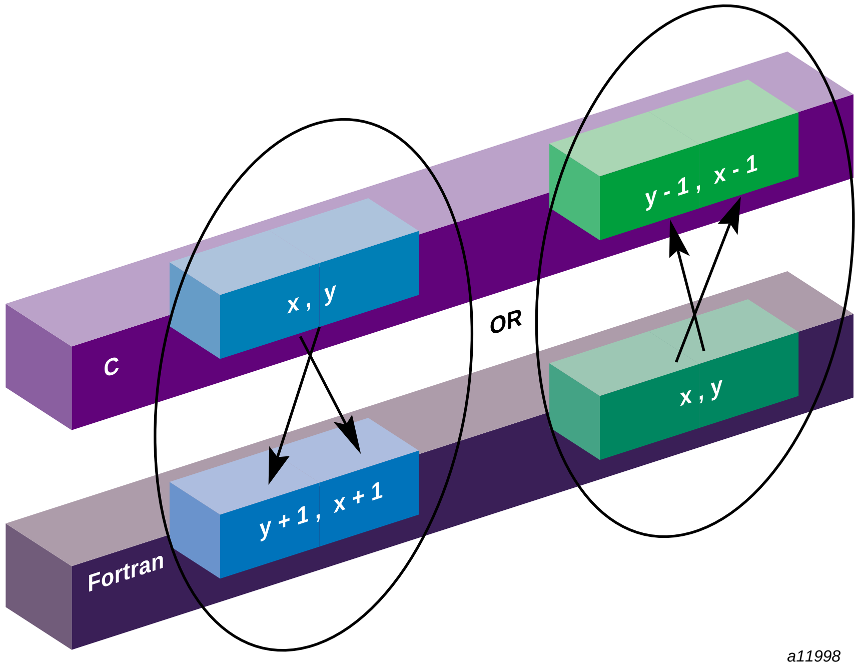

Figure 3-1 Correspondence Between Fortran and C Array Subscripts

For a coding example, see "Using Fortran Arrays in C Code", page 44.

Note: A Fortran array can be declared with some other lower bound than the default of 1. If the Fortran subscript is origin 0, no adjustment is needed. If the Fortran lower bound is greater than 1, the C subscript is adjusted by that amount.

Passing Subprogram Parameters

The Fortran compiler generates code to pass parameters according to simple, uniform rules and it generates subprogram code that expects parameters to be passed according to these rules. When calling non-Fortran functions, you must know how parameters will be passed; and when calling Fortran subprograms from other programming languages you must cause the other language to pass parameters correctly.

Normal Treatment of Parameters

Every parameter passed to a subprogram, regardless of its data type, is passed as the address of the actual parameter value in memory. This rule is extended for two cases:

- The length of each `CHARACTER*`*n* parameter (when *n*>1) is passed as an additional, `INTEGER` value, following the explicit parameters.
- When a function returns type `CHARACTER*`*n* parameter (*n*>1), the address of the space to receive the result is passed as the first parameter to the function and the length of the result space is passed as the second parameter, preceding all explicit parameters.

Example 3-1 Example Subroutine Call

```
COMPLEX*8 cp8
CHARACTER*16 creal, cimag
CALL CPXASC(creal,cimag,cp8)
```

Code generated from the `CALL` in this example prepares these 5 argument values:

1. The address of *creal*
2. The address of *cimag*
3. The address of *cp8*
4. The length of *creal*, an integer value of 16
5. The length of *cimag*, an integer value of 16

Example 3-2 Example Function Call

```
CHARACTER*8 symbl,picksym
CHARACTER*100 sentence
```

```
INTEGER nsym
symbl = picksym(sentence,nsym)
```

Code generated from the function call in this example prepares these 5 argument values:

1. The address of variable *symbl*, the function result space
2. The length of *symbl*, an integer value of 8
3. The address of *sentence*, the first explicit parameter
4. The address of *nsym*, the second explicit parameter
5. The length of *sentence*, an integer value of 100

You can force changes in these conventions using `%VAL` and `%LOC`; see "Calls to C Using `LOC%`, `REF%` and `VAL%`", page 44, for details.

Calling Fortran from C

There are two types of callable Fortran subprograms: subroutines and functions (these units are documented in the *MIPSpro Fortran 77 Language Reference Manual*). In C terminology, both types of subprograms are external functions. The difference is the use of the function return value from each.

Calling Fortran Subroutines from C

From the standpoint of a C module, a Fortran subroutine is an external function returning `int`. The integer return value is normally ignored by a C caller; its meaning is discussed in "Alternate Subroutine Returns", page 38.

The following two examples show a simple Fortran subroutine and a sample of a call to the subroutine.

Example 3-3 Example Fortran Subroutine with `COMPLEX` Parameters

```
SUBROUTINE ADDC32(Z,A,B,N)
COMPLEX*32 Z(1),A(1),B(1)
INTEGER N,I
```

```
    DO 10 I = 1,N
        Z(I) = A(I) + B(I)
10 CONTINUE
    RETURN
    END
```

Example 3-4 C Declaration and Call with COMPLEX Parameters

```
typedef struct{long double real, imag;} cpx32;
extern int
  addc32_(cpx32*pz,cpx32*pa,cpx32*pb,int*pn);
cpx32 z[MAXARRAY], a[MAXARRAY], b[MAXARRAY];
...
    int n = MAXARRAY;
   (void)addc32_(&z, &a, &b, &n);
```

The Fortran subroutine in Example 3-3, page 36, is named in Example 3-4 using lowercase letters and a terminal underscore. It is declared as returning an integer. For clarity, the actual call is cast to `(void)` to show that the return value is intentionally ignored.

The subroutine in the following example takes adjustable-length character parameters.

Example 3-5 Example Fortran Subroutine with String Parameters

```
SUBROUTINE PRT(BEF,VAL,AFT)
CHARACTER*(*)BEF,AFT
REAL VAL
PRINT *,BEF,VAL,AFT
RETURN
END
```

Example 3-6 C Program that Passes String Parameters

```
typedef char fstr_16[16];
extern int
   prt_(fstr_16*pbef, float*pval, fstr_16*paft,
                int lbef, int laft);
main()
{
    float val = 2.1828e0;
    fstr_16 bef,aft;
    strncpy(bef,''Before..........'',sizeof(bef));
    strncpy(aft,''..........After'',sizeof(aft));
    (void)prt_(bef,&val,aft,sizeof(bef),sizeof(aft));
}
```

The C program in Example 3-6 prepares `CHARACTER*16` values and passes them to the subroutine in Example 3-5. Note that the subroutine call requires 5 parameters, including the lengths of the two string parameters. In Example 3-6 the string length parameters are generated using `sizeof()`, derived from the typedef `fstr_16`.

Example 3-7 C Program that Passes Different String Lengths

```
extern int
prt_(char*pbef, float*pval, char*paft, int lbef, int laft);
main()
{
    float val = 2.1828e0;
    char *bef = "Start:";
    char *aft = ":End";
    (void)prt_(bef,&val,aft,strlen(bef),strlen(aft));
}
```

When the Fortran code does not require a specific length of string, the C code that calls it can pass an ordinary C character vector, as shown in Example 3-7. In Example 3-7, page 38, the string length parameter length values are calculated dynamically using `strlen()`.

Alternate Subroutine Returns

In Fortran, a subroutine can be defined with one or more asterisks (`*`) in the position of dummy parameters. When such a subroutine is called, the places of these parameters in the `CALL` statement are supposed to be filled with statement numbers or statement labels. The subroutine returns an integer which selects among the statement numbers, so that the subroutine call acts as both a call and a computed `GOTO`. For more details, see the discussions of the `CALL` and `RETURN` statements in the *MIPSpro Fortran 77 Language Reference Manual*.

Fortran does not generate code to pass statement numbers or labels to a subroutine. No actual parameters are passed to correspond to dummy parameters given as asterisks. When you code a C prototype for such a subroutine, ignore these parameter positions. A `CALL` statement such as the following:

```
CALL NRET (*1,*2,*3)
```

is treated exactly as if it were the computed `GOTO` written as

```
GOTO (1,2,3), NRET()
```

The value returned by a Fortran subroutine is the value specified on the `RETURN` statement, and will vary between 0 and the number of asterisk dummy parameters in the subroutine definition.

Calling Fortran Functions from C

A Fortran function returns a scalar value as its explicit result. This corresponds to the C concept of a function with an explicit return value. When the Fortran function returns any type shown in Table 3-1, page 32, other than `CHARACTER*`*n* (*n*>1), you can call the function from C and use its return value exactly as if it were a C function returning that data type.

Example 3-8 Fortran Function Returning COMPLEX*16

```
COMPLEX*16 FUNCTION FSUB16(INP)
COMPLEX*16 INP
FSUB16 = INP
END
```

This function accepts and returns `COMPLEX*16` values. Although a `COMPLEX` value is declared as a structure in C, it can be used as the return type of a function.

Example 3-9 C Program that Receives `COMPLEX` Return Value

```
typedef  struct{ double real, imag; } cpx16;
extern cpx16 fsub16_( cpx16 * inp );
main()
{
    cpx16 inp = { -3.333, -5.555 };
    cpx16 oup = { 0.0, 0.0 };
    printf("testing fsub16...");
    oup = fsub16_( &inp );
    if ( inp.real == oup.real && inp.imag == oup.imag )
        printf("Ok\n");
    else
        printf("Nope\n");
}
```

The C program in this example shows how the function in Example 3-8 is declared and called. Note that the parameters to a function, like the parameters to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Note: In IRIX 5.3 and earlier, you **cannot** call a Fortran function that returns `COMPLEX` (although you can call one that returns any other arithmetic type). The register conventions used by compilers prior to IRIX 6.0 do not permit returning a structure value from a Fortran function to a C caller.

Example 3-10 Fortran Function Returning `CHARACTER*16`

```
CHARACTER*16 FUNCTION FS16(J,K,S)
CHARACTER*16 S
INTEGER J,K
FS16 = S(J:K)
RETURN
END
```

The function in this example has a `CHARACTER*16` return value. When the Fortran function returns a `CHARACTER*`*n* (*n*>1) value, the returned value is not the explicit result of the function. Instead, you must pass the address and length of the result area as the first two parameters of the function.

Example 3-11 C Program that Receives `CHARACTER*16` Return

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *pz,int lz,int *pj,int *pk,fstr_16*ps,int ls);
main()
{
    char work[64];
    fstr_16 inp,oup;
    int j=7;
    int k=11;
    strncpy(inp,"0123456789abcdef",sizeof(inp));
    fs16_ ( oup, sizeof(oup), &j, &k, inp, sizeof(inp) );
    strncpy(work,oup,sizeof(oup));
    work[sizeof(oup)] = ’\0’;
    printf("FS16 returns <%s>\n",work);
}
```

This C program calls the function in Example 3-10, page 40. The address and length of the function result are the first two parameters of the function. Because type *fstr_16* is an array, its name, *oup*, evaluates to the address of its first element. The next three parameters are the addresses of the three named parameters; and the final parameter is the length of the string parameter.

Calling C from Fortran

In general, you can call units of C code from Fortran as if they were written in Fortran, provided the C modules follow the Fortran conventions for passing parameters (see "Passing Subprogram Parameters", page 35). When the C program expects parameters passed using other conventions, you can either write special forms of `CALL`, or you can build a “wrapper” for the C functions using the `mkf2c` command.

Normal Calls to C Functions

The C function in this section is written to use the Fortran conventions for its name (lowercase with final underscore) and for parameter passing.

Example 3-12 C Function Written to be Called from Fortran

```
/*
|| C functions to export the facilities of strtoll()
|| to Fortran 77 programs.  Effective Fortran declaration:
||
|| INTEGER*8 FUNCTION ISCAN(S,J)
|| CHARACTER*(*) S
|| INTEGER J
||
|| String S(J:) is scanned for the next signed long value
|| as specified by strtoll(3c) for a "base" argument of 0
|| (meaning that octal and hex literals are accepted).
||
|| The converted long long is the function value, and J is
|| updated to the nonspace character following the last
|| converted character, or to 1+LEN(S).
||
|| Note: if this routine is called when S(J:J) is neither
|| whitespace nor the initial of a valid numeric literal,
|| it returns 0 and does not advance J.
```

```
*/
#include <ctype.h> /* for isspace() */
long long iscan_(char *ps, int *pj, int ls)
{
   int  scanPos, scanLen;
   long long ret = 0;
   char wrk[1024];
   char *endpt;
   /* when J>LEN(S), do nothing, return 0 */
     if (ls >= *pj)
     {
        /* convert J to origin-0, permit J=0 */
        scanPos = (0 < *pj)? *pj-1 : 0 ;

        /* calculate effective length of S(J:) */
        scanLen = ls - scanPos;

        /* copy S(J:) and append a null for strtoll() */
        strncpy(wrk,(ps+scanPos),scanLen);
        wrk[scanLen] = '\0';

        /* scan for the integer */
        ret = strtoll(wrk, &endpt, 0);

        /*
        || Advance over any whitespace following the number.
        || Trailing spaces are common at the end of Fortran
        || fixed-length char vars.
        */
        while(isspace(*endpt)) { ++endpt; }
        *pj = (endpt - wrk)+scanPos+1;
     }
     return ret;
}
```

The following program demonstrates a call to the function in Example 3-12, page 41.

```
    EXTERNAL ISCAN
    INTEGER*8 ISCAN
    INTEGER*8 RET
    INTEGER J,K
    CHARACTER*50 INP
    INP = ’1  -99   3141592  0xfff  033 ’
    J = 0
    DO 10 WHILE (J .LT. LEN(INP))
        K = J
        RET = ISCAN(INP,J)
        PRINT *, K,’: ’,RET,’ -->’,J
10  CONTINUE
    END
```

Using Fortran COMMON in C Code

A C function can refer to the contents of a COMMON block defined in a Fortran program. The name of the block as given in the COMMON statement is altered as described in "Subprogram Names", page 29, (that is, forced to lowercase and extended with an underscore). The name of the “blank common” is _BLNK_ _ (one leading underscore and two final underscores).

Follow these steps to refer to the contents of a COMMON block:

- Declare a structure whose fields have the appropriate data types to match the successive elements of the Fortran common block. (See Table 3-1, page 32, for corresponding data types.)
- Declare the common block name as an external structure of that type.

An example is shown below.

Example 3-13 Common Block Usage in Fortran and C

```
      INTEGER STKTOP,STKLEN,STACK(100)
      COMMON /WITHC/STKTOP,STKLEN,STACK

struct fstack {
    int stktop, stklen;
    int stack[100];<_newline>}
extern fstack withc_;
int peektop_()
{
    if (withc_.stktop) /* stack not empty */
        return withc_.stack[withc_.stktop-1];
    else...
}
```

Using Fortran Arrays in C Code

As described in "Corresponding Array Elements", page 33, a C program must take special steps to access arrays created in Fortran.

Example 3-14 Fortran Program Sharing an Array in Common with C

```
INTEGER IMAT(10,100),R,C
COMMON /WITHC/IMAT
R = 74
C = 6
CALL CSUB(C,R,746)
PRINT *,IMAT(6,74)
END
```

This Fortran fragment prepares a matrix in a common block, then calls a C subroutine to modify the array.

Example 3-15 C Subroutine to Modify a Common Array

```
extern struct { int imat[100][10]; } withc_;
int csub_(int *pc, int *pr, int *pval)
{
    withc_.imat[*pr-1][*pc-1] = *pval;
    return 0; /* all Fortran subrtns return int */
}
```

This C function stores its third argument in the common array using the subscripts passed in the first two arguments. In the C function, the order of the dimensions of the array are reversed. The subscript values are reversed to match, and decremented by 1 to match the C assumption of 0-origin indexing.

Calls to C Using LOC%, REF% and VAL%

Using the special intrinsic functions `%VAL`, `%REF`, and `%LOC` you can pass parameters in ways other than the standard Fortran conventions described under "Passing Subprogram Parameters", page 35. These intrinsic functions are documented in the *MIPSpro Fortran 77 Language Reference Manual*.

Using %VAL

%VAL is used in parameter lists to cause parameters to be passed by value rather than by reference. Examine the following function prototype (from the `random` reference page).

```
char *initstate(unsigned int seed, char *state, int n);
```

This function takes an integer value as its first parameter. Fortran would normally pass the address of an integer value, but %VAL can be used to make it pass the integer itself. The following example demonstrates a call to function `initstate()` and the other functions of the `random()` group.

Example 3-16 Fortran Function Calls Using %VAL

```
C declare the external functions in random(3b)
C random() returns i*4, the others return char*
      EXTERNAL RANDOM$, INITSTATE$, SETSTATE$
      INTEGER*4 RANDOM$
      INTEGER*8 INITSTATE$,SETSTATE$
C We use "states" of 128 bytes, see random(3b)
C Note: An undocumented assumption of random() is that
C a "state" is dword-aligned!  Hence, use a common.
      CHARACTER*128 STATE1, STATE2
      COMMON /RANSTATES/STATE1,STATE2
C working storage for state pointers
      INTEGER*8 PSTATE0, PSTATE1, PSTATE2
C initialize two states to the same value
      PSTATE0 = INITSTATE$(%VAL(8191),STATE1)
      PSTATE1 = INITSTATE$(%VAL(8191),STATE2)
      PSTATE2 = SETSTATE$(%VAL(PSTATE1))
C pull 8 numbers from state 1, print
      DO 10 I=1,8
         PRINT *,RANDOM$()
10    CONTINUE
C set the other state, pull 8 numbers & print
      PSTATE1 = SETSTATE$(%VAL(PSTATE2))
      DO 20 I=1,8
         PRINT *,RANDOM$()
20    CONTINUE
      END
```

The use of `%VAL(8191)` or `%VAL(PSTATE1)` causes that value to be passed, rather than an address of that value.

Using `%REF`

`%REF` is used in parameter lists to cause parameters to be passed by reference, that is, to pass the address of a value rather than the value itself.

Parameters passed by reference is the normal behavior of SGI FORTRAN 77 compilers; therefore, no effective difference exists between writing `%REF(`*parm*`)` and writing *parm* alone in a parameter list for non-character parameters. Using `%REF(`*parm*`)` for character parameters causes the character string length not to be added to the end of the parameter list as in the normal case. Thus, using the `%REF(`*parm*`)` guarantees that only the address of the parameter is parsed.

When calling a C function that expects the address of a value rather than the value itself, you can write `%REF(`*parm*`)`, as in the following example:

```
int gmatch (const char *str, const char *pattern);
```

This function `gmatch()` could be declared and called from Fortran.

Example 3-17 Fortran Call to `gmatch()` Using `%REF`

```
LOGICAL GMATCH$
CHARACTER*8 FNAME,FPATTERN
FNAME = 'foo.f\0'
FPATTERN = '*.f\0'
IF ( GMATCH$(%REF(FNAME),%REF(FPATTERN)) )...
```

The use of `%REF()` in this example illustrates the fact that `gmatch()` expects addresses of character strings.

Using %LOC

`%LOC` returns the address of its argument. It can be used in any expression (not only within parameter lists), and is often used to set `POINTER` variables.

Making C Wrappers with `mkf2c`

The `mkf2c` command provides an alternate interface for C routines called by Fortran. See the `mkf2c`(1) reference page for more details.

The `mkf2c` command reads a file of C function prototype declarations and generates an assembly language module. This module contains one callable entry point for each C function. The entry point, or "wrapper," accepts parameters in the Fortran calling convention, and passes the same values to the C function using the C conventions.

The following is a simple case of using a function as input to `mkf2c`:

```
simplefunc (int a, double df)
{ /* function body ignored */ }
```

For this function, the `mkf2c` command (with no options) generates a wrapper function named `simplefunc_` (with an underscore appended). The wrapper function expects two parameters, an integer and a REAL*8, passed according to Fortran conventions; that is, by reference. The code of the wrapper loads the values of the parameters into registers using C conventions for passing parameters by value, and calls `simplefunc()`.

Parameter Assumptions by `mkf2c`

Because `mkf2c` processes only the C source, not the Fortran source, it treats the Fortran parameters based on the data types specified in the C function header. These treatments are summarized in Table 3-2.

Note: Through compiler release 6.0.2, `mkf2c` does not recognize the C data types `long long` and `long double` (`INTEGER*8` and `REAL*16`). It treats arguments of this type as `long` and `double` respectively.

Table 3-2 How `mkf2c` treats function arguments

Data Type in C Prototype	Treatment by Generated Wrapper Code
unsigned `char`	Load `CHARACTER*1` from memory to register, no sign extension
`char`	Load `CHARACTER*1` from memory to register; sign extension only when the `-signed` option is specified
unsigned `short`, unsigned `int`	Load `INTEGER*2` or `INTEGER*4` from memory to register, no sign extension
`short`	Load `INTEGER*2` from memory to register with sign extension

`int`, `long`	Load `INTEGER*4` from memory to register with sign extension
`long long`	(Not supported through 6.0.2)
`float`	Load `REAL*4` from memory to register, extending to `double` unless `-f` is specified
`double`	Load `REAL*8` from memory to register
`long double`	(Not supported through 6.0.2)
`char` *name*`[]`, *name*`[`*n*`]`	Pass address of `CHARACTER*`*n* and pass length as integer parameter as Fortran does
`char *`	Copy `CHARACTER*`*n* value into allocated space, append null byte, pass address of copy

Character String Treatment by `mkf2c`

In Table 3-2 notice the different treatments for an argument declared as a character array and one declared as a character address (even though these two declarations are semantically the same in C).

When the C function expects a character address, `mkf2c` generates the code to dynamically allocate memory and to copy the Fortran character value, for its specified length, to the allocated memory. This creates a null-terminated string. In this case, the following occurs:

- The address passed to C points to the allocated memory
- The length of the value is not passed as an implicit argument
- There is a terminating null byte in the value
- Changes in the string are **not** reflected back to Fortran

A character array specified in the C function is processed by `mkf2c` just like a Fortran `CHARACTER*`*n* value. In this case,

- The address prepared by Fortran is passed to the C function
- The length of the value is passed as an implicit argument (see "Normal Treatment of Parameters", page 35)
- The character array contains no terminating null byte (unless the Fortran programmer supplies one)

- Changes in the array by the C function are visible to Fortran

Because the C function cannot declare the extra string-length parameter (if it declared the parameter, `mkf2c` would process it as an explicit argument) the C programmer has a choice of ways to access the string length. When the Fortran program always passes character values of the same size, the length parameter can be ignored. If its value is needed, the `varargs` macro can be used to retrieve it.

For example, if the C function prototype is specified as follows, `mkf2c` passes a total of six parameters to C:

```
void func1 (char carr1[],int i, char *str, char carr2[]);
```

The fifth parameter is the length of the Fortran value corresponding to *carr1*. The sixth is the length of *carr2*. The C function can use the `varargs` macros to retrieve these hidden parameters. `mkf2c` ignores the *varargs* macro *va_alist* appearing at the end of the parameter name list.

When `func1` is changed to use `varargs`, the C source file is as follows:

Example 3-18 C Function Using `varargs`

```
#include "varargs.h"
void
func1 (char carr1[],int i,char *str,char carr2[],va_alist);
{}
```

The C routine would retrieve the lengths of *carr1* and *carr2*, placing them in the local variables *carr1_len* and *carr2_len* using code like this fragment:

Example 3-19 C Code to Retrieve Hidden Parameters

```
va_list ap;
int carr1_len, carr2_len;
va_start(ap);
carr1_len = va_arg (ap, int)
carr2_len = va_arg (ap, int)
```

Restrictions of `mkf2c`

When it does not recognize the data type specified in the C function, `mkf2c` issues a warning message and generates code to pass the pointer passed by Fortran. It does this in the following cases:

- Any nonstandard data type name, for example a data type that might be declared using typedef or a data type defined as a macro
- Any structure argument
- Any argument with multiple indirection (two or more asterisks, for example *char***)

Because `mkf2c` does not support structure-valued arguments, it does not support passing `COMPLEX*`*n* values.

Using `mkf2c` and `extcentry`

`mkf2c` processes only a limited subset of the C grammar. This subset includes common C syntax for function entry point, C-style comments, and function constructs. However, it does not include constructs such as typedefs, external function declarations, or C preprocessor directives.

To ensure that only the constructs understood by `mkf2c` are included in wrapper input, place special comments around each function for which Fortran-to-C wrappers are to be generated (see the example below).

The special comments `/* CENTRY */` and `/* ENDCENTRY */` surround the section that is to be made Fortran-callable. After these special comments are placed around the code, use the `excentry` command before `mkf2c` to generate the input file for `mkf2c`.

Example 3-20 Source File for Use with `extcentry`

```
typedef unsigned short grunt [4];
struct {
   long 1,11;
   char *str;
} bar;
main ()
{
   int kappa =7;
   foo (kappa,bar.str);
}
/* CENTRY */
foo (integer, cstring)
int integer;
char *cstring;
```

```
{
   if (integer==1) printf("%s",cstring);
} /* ENDCENTRY */
```

Example 3-20 illustrates the use of `extcentry`. It shows the C file `foo.c` containing the function `foo`, which is to be made Fortran-callable.

To generate the assembly language wrapper `foowrp.s` from the above file `foo.c`, use the following set of commands:

```
% extcentry foo.c foowrp.fc
% mkf2c foowrp.fc foowrp.s
```

The programs `mkf2c` and `extcentry` are stored in the directory `/usr/bin`.

Makefile Considerations

The `make` command uses default rules to help automate the control of wrapper generation. The following example of a makefile illustrates the use of these rules. In the example, an executable object file is created from the files `main.f` (a Fortran main program) and `callc.c`:

```
test:  main.o callc.o
    f77 -o test main.o callc.o
callc.o: callc.fc
clean:
   rm -f *.o test *.fc
```

In this program, `main` calls a C routine in `callc.c`. The extension `.fc` has been adopted for Fortran-to-call-C wrapper source files. The wrappers created from `callc.fc` will be assembled and combined with the binary created from `callc.c`. Also, the dependency of `callc.o` on `callc.fc` will cause `callc.fc` to be recreated from `callc.c` whenever the C source file changes. The programmer is responsible for placing the special comments for `extcentry` in the C source as required.

Note: Options to `mkf2c` can be specified when `make` is invoked by setting the `make` variable *FC2FLAGS*. Also, do not create a `.fc` file for the modules that need wrappers created. These files are both created and removed by `make` in response to the `file.o:file.fc` dependency.

The `makefile` above controls the generation of wrappers and Fortran objects. You can add modules to the executable object file in one of the following ways:

- If the file is a native C file whose routines are not to be called from Fortran using a wrapper interface, or if it is a native Fortran file, add the `.o` specification of the final make target and dependencies.
- If the file is a C file containing routines to be called from Fortran using a wrapper interface, the comments for `extcentry` must be placed in the C source, and the `.o` file placed in the target list. In addition, the dependency of the `.o` file on the `.fc` file must be placed in the makefile. This dependency is illustrated in the example makefile above where `callf.o` depends on `callf.fc`.

Chapter 4

System Functions and Subroutines

This chapter describes extensions to FORTRAN 77 that are related to the IRIX compiler and operating system.

- "Library Functions", page 53, summarizes the Fortran run-time library functions.
- "Extended Intrinsic Subroutines", page 60, describes the extensions to the Fortran intrinsic subroutines.
- "Extended Intrinsic Functions", page 63, describes the extensions to the Fortran functions.

Library Functions

The Fortran library functions provide an interface from Fortran programs to the IRIX system functions. System functions are facilities that are provided by the IRIX system kernel directly, as opposed to functions that are supplied by library code linked with your program.

Table 4-1 summarizes the functions in the Fortran run-time library. In general, the name of the interface routine is the same as the name of the system function that would be called from a C program. For details on a system interface routine use the following command:

`man 2` *name_of_function*

Note: You must declare the `time` function as `EXTERNAL`; if you do not, the compiler will assume you mean the VMS-compatible intrinsic `time` function rather than the IRIX system function. It is a usually a good idea to declare any library function in an `EXTERNAL` statement as documentation.

Table 4-1 Summary of System Interface Library Routines

Function	Purpose
`abort`(3f)	abnormal termination
`access`(2)	determine accessibility of a file
`acct`(2)	enable/disable process accounting
`alarm`(3f)	execute a subroutine after a specified time
`barrier`	perform barrier operations
`blockproc`(2)	block processes
`brk`(2)	change data segment space allocation
`close`	close a file descriptor
`creat`	create or rewrite a file
`ctime`(3f)	return system time
`dtime`(3f)	return elapsed execution time
`dup`	duplicate an open file descriptor
`etime`(3f)	return elapsed execution time
`exit`(2)	terminate process with status
`fcntl`(2)	file control
`fdate`(3f)	return date and time in an ASCII string
`fgetc`(3f)	get a character from a logical unit
`flush`(3f)	flush output to a logical unit
`fork`(2)	create a copy of this process
`fputc`(3f)	write a character to a Fortran logical unit
`free_barrier`	free barrier
`fseek`(3f)	reposition a file on a logical unit
`fseek64`(3f)	reposition a file on a logical unit for 64-bit architecture
`fstat`(2)	get file status
`ftell`(3f)	reposition a file on a logical unit

Function	Purpose
`ftell64`(3f)	reposition a file on a logical unit for 64-bit architecture
`gerror`(3f)	get system error messages
`getarg`(3f)	return command line arguments
`getc`(3f)	get a character from a logical unit
`getcwd`	get pathname of current working directory
`getdents`(2)	read directory entries
`getegid`(2)	get effective group ID
`gethostid`(2)	get unique identifier of current host
`getenv`(3f)	get value of environment variables
`geteuid`(2)	get effective user ID
`getgid`(2)	get user or group ID of the caller
`gethostname`(2)	get current host ID
`getlog`(3f)	get user's login name
`getpgrp`	get process group ID
`getpid`	get process ID
`getppid`	get parent process ID
`getsockopt`(2)	get options on sockets
`getuid`(2)	get user or group ID of caller
`gmtime`(3f)	return system time
`iargc`(3f)	return command line arguments
`idate`(3f)	return date or time in numerical form
`ierrno`(3f)	get system error messages
`ioctl`(2)	control device
`isatty`(3f)	determine if unit is associated with tty
`itime`(3f)	return date or time in numerical form
`kill`(2)	send a signal to a process
`link`(2)	make a link to an existing file

Function	Purpose
`loc`(3f)	return the address of an object
`lseek`(2)	move read/write file pointer
`lseek64`(2)	move read/write file pointer for 64-bit architecture
`lstat`(2)	get file status
`ltime`(3f)	return system time
`m_fork`	create parallel processes
`m_get_myid`	get task ID
`m_get_numprocs`	get number of subtasks
`m_kill_procs`	kill process
`m_lock`	set global lock
`m_next`	return value of counter
`m_park_procs`	suspend child processes
`m_rele_procs`	resume child processes
`m_set_procs`	set number of subtasks
`m_sync`	synchronize all threads
`m_unlock`	unset a global lock
`mkdir`(2)	make a directory
`mknod`(2)	make a directory/file
`mount`(2)	mount a filesystem
`new_barrier`	initialize a barrier structure
`nice`	lower priority of a process
`open`(2)	open a file
`oserror`(3f)	get/set system error
`pause`(2)	suspend process until signal
`perror`(3f)	get system error messages
`pipe`(2)	create an interprocess channel
`plock`(2)	lock process, test, or data in memory

Function	Purpose
`prctl`(2)	control processes
`profil`(2)	execution-time profile
`ptrace`	process trace
`putc`(3f)	write a character to a Fortran logical unit
`putenv`(3f)	set environment variable
`qsort`(3f)	quick sort
`read`	read from a file descriptor
`readlink`	read value of symbolic link
`rename`(3f)	change the name of a file
`rmdir`(2)	remove a directory
`sbrk`(2)	change data segment space allocation
`schedctl`(2)	call to scheduler control
`send`(2)	send a message to a socket
`setblockproccnt`(2)	set semaphore count
`setgid`	set group ID
`sethostid`(2)	set current host ID
`setoserror`(3f)	set system error
`setpgrp`(2)	set process group ID
`setsockopt`(2)	set options on sockets
`setuid`	set user ID
`sginap`(2)	put process to sleep
`sginap64`(2)	put process to sleep in 64-bit environment
`shmat`(2)	attach shared memory
`shmdt`(2)	detach shared memory
`sighold`(2)	raise priority and hold signal
`sigignore`(2)	ignore signal
`signal`(2)	change the action for a signal

Function	Purpose
`sigpause`(2)	suspend until receive signal
`sigrelse`(2)	release signal and lower priority
`sigset`(2)	specify system signal handling
`sleep`(3f)	suspend execution for an interval
`socket`(2)	create an endpoint for communication TCP
`sproc`(2)	create a new share group process
`stat`(2)	get file status
`stime`(2)	set time
`symlink`(2)	make symbolic link
`sync`	update superblock
`sysmp`(2)	control multiprocessing
`sysmp64`(2)	control multiprocessing in 64-bit environment
`system`(3f)	issue a shell command
`taskblock`	block tasks
`taskcreate`	create a new task
`taskctl`	control task
`taskdestroy`	kill task
`tasksetblockcnt`	set task semaphore count
`taskunblock`	unblock task
`time`(3f)	return system time (must be declared EXTERNAL)
`ttynam`(3f)	find name of terminal port
`uadmin`	administrative control
`ulimit`(2)	get and set user limits
`ulimit64`(2)	get and set user limits in 64-bit architecture
`umask`	get and set file creation mask
`umount`(2)	dismount a file system
`unblockproc`(2)	unblock processes

Function	Purpose
`unlink`(2)	remove a directory entry
`uscalloc`	shared memory allocator
`uscalloc64`	shared memory allocator in 64-bit environment
`uscas`	compare and swap operator
`usclosepollsema`	detach file descriptor from a pollable semaphore
`usconfig`	semaphore and lock configuration operations
`uscpsema`	acquire a semaphore
`uscsetlock`	unconditionally set lock
`usctlsema`	semaphore control operations
`usdumplock`	dump lock information
`usdumpsema`	dump semaphore information
`usfree`	user shared memory allocation
`usfreelock`	free a lock
`usfreepollsema`	free a pollable semaphore
`usfreesema`	free a semaphore
`usgetinfo`	exchange information through an arena
`usinit`	semaphore and lock initialize routine
`usinitlock`	initialize a lock
`usinitsema`	initialize a semaphore
`usmalloc`	allocate shared memory
`usmalloc64`	allocate shared memory in 64-bit environment
`usmallopt`	control allocation algorithm
`usnewlock`	allocate and initialize a lock
`usnewpollsema`	allocate and initialize a pollable semaphore
`usnewsema`	allocate and initialize a semaphore
`usopenpollsema`	attach a file descriptor to a pollable semaphore
`uspsema`	acquire a semaphore

Function	Purpose
`usputinfo`	exchange information through an arena
`usrealloc`	user share memory allocation
`usrealloc64`	user share memory allocation in 64-bit environment
`ussetlock`	set lock
`ustestlock`	test lock
`ustestsema`	return value of semaphore
`usunsetlock`	unset lock
`usvsema`	free a resource to a semaphore
`uswsetlock`	set lock
`wait`(2)	wait for a process to terminate
`write`	write to a file

You can use the `datapool` statement to cause Fortran interprocess data sharing. However, this is a nonstandard statement. The `datapool` statement is a way that different processes can use to access the same pool of common symbols. Any processes can access the shared datapool by linking with the datapool DSO. For more information see the `datapool`(5) reference page.

Extended Intrinsic Subroutines

This section describes the intrinsic subroutines that are extensions to FORTRAN 77. The intrinsic **functions** that are standard to FORTRAN 77 are documented in Appendix A of the *MIPSpro Fortran 77 Language Reference Manual*. The rules for using the names of intrinsic subroutines are also discussed in that appendix.

DATE

The `DATE` routine returns the current date as set by the system; the format is as follows:

```
CALL DATE (buf)
```

The *buf* argument is a variable, array, array element, or character substring nine bytes long. After the call, *buf* contains an ASCII variable in the format *dd-mmm-yy*, where *dd* is the date in digits, *mmm* is the month in alphabetic characters, and *yy* is the year in digits.

IDATE

The `IDATE` routine returns the current date as three integer values representing the month, date, and year; the format is as follows:

```
CALL IDATE (m, d,y)
```

The *m*, *d*, and *y* arguments are either `INTEGER*4` or `INTEGER*2` values representing the current month, day and year. For example, the values of *m*, *d*, and *y* on August 10, 1989, are the following:

```
m = 8
d = 10
y = 89
```

ERRSNS

The `ERRSNS` routine returns information about the most recent program error; the format is as follows:

```
CALL ERRSNS (arg1, arg2, arg3, arg4, arg5)
```

The arguments (*arg1*, *arg2*, and so on) can be either `INTEGER*4` or `INTEGER*2` variables. On return from `ERRSNS`, the arguments contain the information shown in the following table.

Argument	**Contents**
arg1	IRIX global variable *errno*, which is then reset to zero after the call
arg2	Zero
arg3	Zero

arg4	Logical unit number of the file that was being processed when the error occurred
arg5	Zero

Although only *arg1* and *arg4* return relevant information, *arg2*, *arg3*, and *arg5* are always required.

EXIT

The `EXIT` routine causes normal program termination and optionally returns an exit-status code; the format is as follows:

```
CALL EXIT (status)
```

The *status* argument is an `INTEGER*4` or `INTEGER*2` argument containing a status code.

TIME

The `TIME` routine returns the current time in hours, minutes, and seconds; the format is as follows:

```
CALL TIME (clock)
```

The *clock* argument is a variable, array, array element, or character substring; it must be eight bytes long. After execution, *clock* contains the time in the format *hh:mm:ss*, where *hh*, *mm*, and *ss* are numerical values representing the hour, the minute, and the second.

MVBITS

The `MVBITS` routine transfers a bit field from one storage location to another; the format is as follows:

```
CALL MVBITS (source,sbit,length,destination,dbit)
```

Arguments can be declared as `INTEGER*2`, `INTEGER*4`, or `INTEGER*8`. The following list describes each argument:

Argument	**Description**
source	Integer variable or array element. Source location of bit field to be transferred.
sbit	Integer expression. First bit position in the field to be transferred from source.
length	Integer expression. Length of the field to be transferred from source.
destination	Integer variable or array element. Destination location of the bit field.
dbit	Integer expression. First bit in destination to which the field is transferred.

Extended Intrinsic Functions

The following sections provide an overview of the intrinsic functions added as extensions to FORTRAN 77.

SECNDS

`SECNDS` is an intrinsic routine that returns the number of seconds since midnight, minus the value of the passed argument; the format is as follows:

```
s = SECNDS(n)
```

After execution, *s* contains the number of seconds past midnight less the value specified by *n*. Both *s* and *n* are single-precision, floating point values.

RAN

`RAN` generates a pseudo-random number. The format is as follows:

```
v = RAN(s)
```

The argument *s* is an `INTEGER*4` variable or array element. This variable serves as a seed in determining the next random number. It should initially be set to a large, odd integer value. You can compute multiple random number series by supplying different variables or array elements as the seed argument to different calls of `RAN`.

Caution: Because `RAN` modifies the *s* argument, calling the function with a constant value can cause a core dump.

The algorithm used in `RAN` is the linear congruential method. The code is similar to the following fragment:

```
S = S * 1103515245L + 12345
RAN = FLOAT(IAND(RSHIFT(S,16),32767))/32768.0
```

`RAN` is supplied for compatibility with VMS. For demanding applications, use the functions described on the `random` reference page. These can all be called using techniques described under "Using `%VAL`", page 45.

Chapter 5

OpenMP Fortran API Multiprocessing Directives

This chapter provides an overview of the supported multiprocessing directives. These directives are based on the OpenMP Fortran application program interface (API) standard. Programs that use these directives are portable and can be compiled by other compilers that support the OpenMP standard.

The complete OpenMP standard is available at `http://www.openmp.org/specs`. See that documentation for complete examples, rules of usage, and restrictions. **This chapter provides only an overview of the supported directives and does not give complete details about usage or restrictions.**

To enable recognition of the OpenMP directives, specify `-mp` on the `f77`(1) command line. The `-mp` option must be specified in order for the compiler to honor any `-MP:...` options that may also be specified on the command line. The `-MP:open_mp=ON` option is on by default and must be in effect during compilation.

The following example command line can compile program `ompprg.f`, which contains OpenMP Fortran API directives:

```
f77 -mp ompprg.f
```

In addition to directives, the OpenMP Fortran API describes several library routines and environment variables. See the standard for complete details.

Using Directives

All multiprocessing directives are case-insensitive and are of the following form:

prefix directive [*clause*[`[,]` *clause*]`...`]

Directives cannot be embedded within continued statements, and statements cannot be embedded within directives. Comments cannot appear on the same line as a directive.

Comments are allowed inside directives. Comments can appear on the same line as a directive. The comment extends to the end of the source line and is ignored. If the first nonblank character after the initial prefix (or after a continuation directive line in fixed source form) is an exclamation point, the line is ignored.

Conditional Compilation

Fortran statements can be compiled conditionally as long as they are preceded by one of the following conditional compilation prefixes: `C$`, or `*$`. The prefix must be followed by a Fortran statement on the same line. During compilation, the prefix is replaced by two spaces, and the rest of the line is treated as a normal Fortran statement.

The prefixes must start in column one and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the line. Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six.

Your program must be compiled with the `-mp` option in order for the compiler to honor statements preceded by conditional compilation prefixes; without the `mp` command line option, statements preceded by conditional compilation prefixes are treated as comments.

You must define the `_OPENMP` symbol to be used for conditional compilation. This symbol is defined during OpenMP compilation to have the decimal value `YYYYMM` where `YYYY` and `MM` are the year and month designators of the version of the OpenMP Fortran API is supported.

Parallel Region Constructs

The `PARALLEL` and `END PARALLEL` directives define a *parallel region*. A parallel region is a block of code that is to be executed by multiple threads in parallel. This is the fundamental OpenMP parallel construct that starts parallel execution.

The `END PARALLEL` directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution past the end of a parallel region.

Work-sharing Constructs

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. A work-sharing construct must be enclosed within a parallel region in order for the directive to execute in parallel. When a work-sharing construct is not enclosed dynamically within a parallel region, it is treated as though the thread that encounters it were a team of size one. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The following restrictions apply to the work-sharing directives:

- Work-sharing constructs and `BARRIER` directives must be encountered by all threads in a team or by none at all.
- Work-sharing constructs and `BARRIER` directives must be encountered in the same order by all threads in a team.

If `NOWAIT` is specified on the `END DO`, `END SECTIONS`, `END SINGLE`, or `END WORKSHARE` directive, an implementation may omit any code to synchronize the threads at the end of the worksharing construct. In this case, threads that finish early may proceed straight to the instructions following the work-sharing construct without waiting for the other members of the team to finish the work-sharing construct.

The following list summarizes the work-sharing constructs:

- The `DO` directive specifies that the iterations of the immediately following `DO` loop must be divided among the threads in the parallel region. If there is no enclosing parallel region, the `DO` loop is executed serially.

  The loop that follows a `DO` directive cannot be a `DO WHILE` or a `DO` loop without loop control. If an `END DO` directive is not specified, it is assumed at the end of the `DO` loop.

- The `SECTIONS` directive specifies that the enclosed sections of code are to be divided among threads in the team. It is a noniterative work-sharing construct. Each section is executed once by a thread in the team.

  Each section must be preceded by a `SECTION` directive, though the `SECTION` directive is optional for the first section. The `SECTION` directives must appear within the lexical extent of the `SECTIONS/END SECTIONS` directive pair. The last section ends at the `END SECTIONS` directive. Threads that complete execution of their sections wait at a barrier at the `END SECTIONS` directive unless a `NOWAIT` is specified.

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team. Threads in the team that are not executing the SINGLE directive wait at the END SINGLE directive unless NOWAIT is specified.
- The WORKSHARE directive divides the work of executing the enclosed code into separate units of work, and causes the threads of the team to share the work of executing the enclosed code such that each unit is executed only once. The units of work may be assigned to threads in any manner as long as each unit is executed exactly once.

Combined Parallel Work-sharing Constructs

The combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a PARALLEL directive followed by a single work-sharing construct.

The following list describes the combined parallel work-sharing directives:

- The PARALLEL DO directive provides a shortcut form for specifying a parallel region that contains a single DO directive.

  If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

  The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.
- The PARALLEL SECTIONS/END PARALLEL directives provide a shortcut form for specifying a parallel region that contains a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.
- The PARALLEL WORKSHARE directive provides a shortcut form for specifying a parallel region that contains a single WORKSHARE directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a WORKSHARE directive.

Synchronization Constructs

The following list describe the synchronization constructs:

- The code enclosed within `MASTER` and `END MASTER` directives is executed by the master thread.
- The `CRITICAL` and `END CRITICAL` directives restrict access to the enclosed code to one thread at a time.

  A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name. All unnamed `CRITICAL` directives map to the same name. Critical section names are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.
- The `BARRIER` directive synchronizes all the threads in a team. When it encounters a barrier, a thread waits until all other threads in that team have reached the same point.
- The `ATOMIC` directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.
- The `FLUSH` directive identifies synchronization points at which thread-visible variables are written back to memory. This directive must appear at the precise point in the code at which the synchronization is required.

  Thread-visible variables include the following data items:

  – Globally visible variables (common blocks and modules)

  – Local variables that do not have the `SAVE` attribute but have had their address taken and saved or have had their address passed to another subprogram

  – Local variables that do not have the `SAVE` attribute that are declared shared in a parallel region within the subprogram

  – Dummy arguments

  – All pointer dereferences
- The code enclosed within `ORDERED` and `END ORDERED` directives is executed in the order in which it would be executed in a sequential execution of an enclosing parallel loop.

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. This DO directive must have the ORDERED clause specified. For information on directive binding, see "Directive Binding", page 71.

Only one thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it is guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel. ORDERED sections that bind to different DO directives are independent of each other.

Data Environment Constructs

The THREADPRIVATE directive makes named common blocks and named variables private to a thread but global within the thread.

In addition to the THREADPRIVATE directive, several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct. Not all of the clauses are allowed on all directives; usually, if no data scope clauses are specified for a directive, the default scope for variables affected by the directive is SHARED.

The following list describes the data scope attribute clauses:

- The PRIVATE clause declares variables to be private to each thread in a team.
- The SHARED clause makes variables shared among all the threads in a team. All threads within a team access the same storage area for SHARED data.
- The DEFAULT clause allows the user to specify a PRIVATE, SHARED, or NONE default scope attribute for all variables in the lexical extent of any parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause.
- The FIRSTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause.
- The LASTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause.

  When the LASTPRIVATE clause appears on a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct. Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

- The REDUCTION clause performs a reduction on the variables specified, with the operator or the intrinsic specified.

  At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the (shared) reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler can freely reassociate the computation of the final value (the partial results of a subtraction reduction are added to form the final value).

  The value of the shared variable becomes undefined when the first thread reaches the containing clause, and it remains so until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct; however, if the REDUCTION clause is used on a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed to ensure that all the threads have completed the REDUCTION clause.

- The COPYIN clause applies only to common blocks that are declared THREADPRIVATE. A COPYIN clause on a parallel region specifies that the data in the master thread of the team be copied to the thread private copies of the common block at the beginning of the parallel region.

- The COPYPRIVATE clause uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members.

There are several rules and restrictions that apply with respect to data scope. See the OpenMP specification at http://www.openmp.org/specs for complete details.

Directive Binding

Some directives are *bound* to other directives. A binding specifies the way in which one directive is related to another. For instance, a directive is bound to a second directive if it can appear in the dynamic extent of that second directive. The following rules apply with respect to the dynamic binding of directives:

- A parallel region is available for binding purposes, whether it is serialized or executed in parallel.

- The `DO`, `SECTIONS`, `SINGLE`, `MASTER`, `BARRIER`, and `WORKSHARE` directives bind to the dynamically enclosing `PARALLEL` directive, if one exists. The dynamically enclosing `PARALLEL` directive is the closest enclosing `PARALLEL` directive regardless of the value of the expression in the `IF` clause, should the clause be present.
- The `ORDERED` directive binds to the dynamically enclosing `DO` directive.
- The `ATOMIC` directive enforces exclusive access with respect to `ATOMIC` directives in all threads, not just the current team.
- The `CRITICAL` directive enforces exclusive access with respect to `CRITICAL` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing `PARALLEL`.

Directive Nesting

The following rules apply to the dynamic nesting of directives:

- A `PARALLEL` directive dynamically inside another `PARALLEL` directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- `DO`, `SECTIONS`, `SINGLE`, and `WORKSHARE` directives that bind to the same `PARALLEL` directive cannot be nested one inside the other.
- `DO`, `SECTIONS`, `SINGLE`, and `WORKSHARE` directives are not permitted in the dynamic extent of `CRITICAL` and `MASTER` directives.
- `BARRIER` directives are not permitted in the dynamic extent of `DO`, `SECTIONS`, `SINGLE`, `WORKSHARE`, `MASTER`, `CRITICAL`, and `ORDERED` directives.
- `MASTER` directives are not permitted in the dynamic extent of `DO`, `SECTIONS`, `SINGLE`, `WORKSHARE`, `MASTER`, `CRITICAL`, and `ORDERED` directives.
- `ORDERED` directives must appear in the dynamic extent of a `DO` or `PARALLEL DO` directive which has an `ORDERED` clause.
- `ORDERED` directives are not allowed in the dynamic extent of `SECTIONS`, `SINGLE`, `WORKSHARE`, `CRITICAL`, and `MASTER` directives.
- `CRITICAL` directives with the same name are not allowed to be nested one inside the other.

- Any directive set that is legal when executed dynamically inside a `PARALLEL` region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Chapter 6

Compiling and Debugging Parallel Fortran

This chapter gives instructions on how to compile and debug a parallel Fortran program. It contains the following sections:

- "Compiling and Running Parallel Fortran ", page 75, explains how to compile and run a parallel Fortran program.
- "Profiling a Parallel Fortran Program", page 77, describes how to use the system profiler, `prof`, to examine execution profiles.
- "Debugging Parallel Fortran", page 77, presents some standard techniques for debugging a parallel Fortran program.

Compiling and Running Parallel Fortran

After you have written a program for parallel processing, you should debug your program in a single-processor environment by using the Fortran compiler with the `f77` command. After your program has executed successfully on a single processor, you can compile it for multiprocessing.

To enable multiprocessing, add `-mp` to the `f77` command line. This option causes the Fortran compiler to generate multiprocessing code for the files being compiled. When linking, you can specify both object files produced with the `-mp` option and object files produced without it. If any or all of the files are compiled with `-mp`, the executable must be linked with `-mp` so that the correct libraries are used.

Using the `-static` Option

Multiprocessing implementation demands some use of the stack to allow multiple threads of execution to execute the same code simultaneously. Therefore, the parallel `DO` loops themselves are compiled with the `-automatic` option, even if the routine enclosing them is compiled with `-static`.

This means that `SHARE` variables in a parallel loop behave correctly according to the `-static` semantics but that `LOCAL` variables in a parallel loop do not (see "Debugging Parallel Fortran", page 77, for a description of `SHARE` and `LOCAL` variables).

Finally, if the parallel loop calls an external routine, that external routine cannot be compiled with `-static`. You can mix static and multiprocessed object files in the same executable; the restriction is that a static routine cannot be called from within a parallel loop.

Examples of Compiling

This section steps you through a few examples of compiling code using `-mp`.

The following command line compiles and links the Fortran program `foo.f` into a multiprocessor executable:

```
% f77 -mp foo.f
```

In the following example, the Fortran routines in the file `snark.f` are compiled with multiprocess code generation enabled:

```
% f77 -c -mp -O2 snark.f
```

The optimizer is also used. A standard `snark.o` binary file is produced, which must be linked:

```
% f77 -mp -o boojum snark.o bellman.o
```

Here, the `-mp` option signals the linker to use the Fortran multiprocessing library. The `bellman.o` file did not have to be compiled with the `-mp` option, although it could be.

After linking, the resulting executable can be run like any standard executable. Creating multiple execution threads, running and synchronizing them, and task termination are all handled automatically.

When an executable has been linked with `-mp`, the Fortran initialization routines determine how many parallel threads of execution to create. This determination occurs each time the task starts; the number of threads is not compiled into the code. The default is to use whichever is less: 4 **or** the number of processors that are on the machine (the value returned by the system call `sysmp(MP_NAPROCS)`; see the `sysmp`(2) reference page). You can override the default by setting the `MP_SET_NUMTHREADS` shell environment variable. If it is set, Fortran tasks use the specified number of execution threads regardless of the number of processors physically present on the machine. `MP_SET_NUMTHREADS` can be a value from 1 to 64.

Profiling a Parallel Fortran Program

After converting a program, you need to examine execution profiles to judge the effectiveness of the transformation. Good execution profiles of the program are crucial to help you focus on the loops that consume the most time.

IRIX provides profiling tools that can be used on Fortran parallel programs. Both `pixie`(1) and pc-sample profiling can be used (pc-sampling can help show the system overhead). On jobs that use multiple threads, both these methods will create multiple profile data files, one for each thread. You can use the standard profile analyzer `prof`(1) to examine this output. Also, `timex`(1) indicates if the parallelized versions performed better overall than the serial version.

The profile of a Fortran parallel job is different from a standard profile. To produce a parallel program, the compiler pulls the parallel `DO` loops out into separate subroutines, one routine for each loop. Each of these loops is shown as a separate procedure in the profile. Comparing the amount of time spent in each loop by the various threads shows how well the workload is balanced.

In addition to the loops, the profile shows the special routines that actually do the multiprocessing. The `__mp_parallel_do` routine is the synchronizer and controller. Slave threads wait for work in the routine `__mp_slave_wait_for_work`. The less time they wait, the more time they work. This gives a rough estimate of a program's parallelization.

Debugging Parallel Fortran

This section presents some standard techniques to assist in debugging a parallel program.

General Debugging Hints

The following list describes some standard debugging tips:

- Debugging a multiprocessed program is much more difficult than debugging a single-processor program. Therefore you should do as much debugging as possible on the single-processor version.
- Try to isolate the problem as much as possible. Ideally, try to reduce the problem to a single `C$DOACROSS` loop.

- Before debugging a multiprocessed program, change the order of the iterations on the parallel `DO` loop on a single-processor version. If the loop can be multiprocessed, then the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order frequently causes the single-processor version to fail, and standard single-process debugging techniques can be used to find the problem.
- When debugging a program using `dbx`, use the `ignore TERM` command. When debugging a program using `cvd`, select **Views/Signal Panel**, then select **disable SIGTERM**. Debugging is possible without these commands, but the program may not terminate gracefully after execution is complete.

Example 6-1 Erroneous `C$DOACROSS`

In this example, the two references to `a` have the indexes in reverse order, causing a bug. If the indexes were in the same order (if both were `a(i,j)` or both were `a(j,i)`), the loop could be multiprocessed. As written, there is a data dependency, so the `C$DOACROSS` is a mistake.

```
c$doacross local(i,j)
      do i = 1, n
         do j = 1, n
            a(i,j) = a(j,i) + x*b(i)
         end do
      end do
```

Because a (correct) multiprocessed loop can execute its iterations in any order, you could rewrite this as:

```
c$doacross local(i,j)
      do i = n, 1, -1
         do j = 1, n
            a(i,j) = a(j,i) + x*b(i)
         end do
      end do
```

This loop no longer gives the same answer as the original even when compiled without the `-mp` option. This reduces the problem to a normal debugging problem. When a multiprocessed loop is giving the wrong answer, perform the following checks:

- Check the `LOCAL` variables when the code runs correctly as a single process but fails when multiprocessed. Carefully check any scalar variables that appear in the left-hand side of an assignment statement in the loop to be sure they are all declared LOCAL. Be sure to include the index of any loop nested inside the parallel loop.

A related problem occurs when you need the final value of a variable but the variable is declared LOCAL rather than LASTLOCAL. If the use of the final value happens several hundred lines farther down in the code, or if the variable is in a COMMON block and the final value is used in a completely separate routine, a variable can look as if it is LOCAL when in fact it should be LASTLOCAL. To fix this problem, declare all the LOCAL variables LASTLOCAL when debugging a loop.

- Check for arrays with complicated subscripts. If the array subscripts are simply the index variables of a DO loop, the analysis is probably correct. If the subscripts are more involved, they are a good choice to examine first.
- Check for EQUIVALENCE problems. Two variables of different names may in fact refer to the same storage location if they are associated through an EQUIVALENCE.
- Check for the use of uninitialized variables. Some programs assume uninitialized variables have a value of 0. This works with the -static option, but without it, uninitialized values assume the value that is left on the stack. When compiling with -mp, the program executes differently and the stack contents are different. You should suspect this type of problem when a program compiled with -mp and run on a single processor gives a different result than when it is compiled without -mp. One way to check a problem of this type is to compile suspected routines with -static. If an uninitialized variable is the problem, it should be fixed by initializing the variable rather than by continuing to compile with -static.
- Try compiling with the -C option for range checking on array references. If arrays are indexed out of bounds, a memory location may be referenced in unexpected ways. This is particularly true of adjacent arrays in a COMMON block.
- If the analysis of the loop was incorrect, one or more arrays that are SHARE may have data dependencies. This sort of error is seen only when running multiprocessed code. When stepping through the code in the debugger, the program executes correctly. This sort of error is usually seen only intermittently; the program works correctly most of the time.
- As a final solution, print out all the values of all the subscripts on each iteration through the loop. Then use the uniq(1) command to look for duplicates. If duplicates are found, there is a data dependency.

EQUIVALENCE Statements and Storage of Local Variables

EQUIVALENCE statements affect storage of local variables and can cause data dependencies when parallelizing code. EQUIVALENCE statements with local variables cause the storage location to be statically allocated (initialized to zero and saved between calls to the subroutine).

In particular, if a loop without equivalenced variables calls a subroutine that appears in the scope of a directive ASSERT CONNCURENT CALL which does have equivalenced local variables, a data dependency occurs. This is because the equivalenced storage locations are statically allocated.

Run-Time Error Messages

The following table lists possible Fortran run-time I/O errors. Other errors given by the operating system may also occur. See the `intro`(2) and `perror`(3f) reference pages for details.

Each error is listed on the screen alone or with one of these phrases appended to it:

- apparent state: unit *num* named `user filename`
- last format: *string*
- lately (reading, writing) (sequential, direct, indexed)
- formatted, unformatted (external, internal) IO

When the Fortran run-time system detects an error, the following actions take place:

- A message describing the error is written to the standard error unit (Unit 0).
- A core file, which can be used with `dbx` (the debugger) to inspect the state of the program at termination, is produced if the `f77_dump_flag` environment variable is defined and set to `y`.

When a run-time error occurs, the program terminates with one of the error messages shown in the following table. The errors are output in the format `user filename` : *message*.

Table A-1 Run-Time Error Messages

Number	Message/Cause
100	`error in format` Illegal characters are encountered in `FORMAT` statement.
101	`out of space for I/O unit table` Out of virtual space that can be allocated for the I/O unit table.
102	`formatted io not allowed` Cannot do formatted I/O on logical units opened for unformatted I/O.
103	`unformatted io not allowed` Cannot do unformatted I/O on logical units opened for formatted I/O.

Number	Message/Cause
104	`direct io not allowed` Cannot do direct I/O on sequential file.
106	`can’t backspace file` Cannot perform BACKSPACE/REWIND on file.
107	`null file name` Filename specification in `OPEN` statement is null.
108	`can’t stat file` The directory information for the file is not accessible.
109	`file already connected` The specified filename has already been opened as a different logical unit.
110	`off end of record` Attempt to do I/O beyond the end of the record.
112	`incomprehensible list input` Input data for list-directed read contains invalid character for its data type.
113	`out of free space` Cannot allocate virtual memory space on the system.
114	`unit not connected` Attempt to do I/O on unit that has not been opened or cannot be opened.
115	`read unexpected character` Unexpected character encountered in formatted or directed read.
116	`blank logical input field` Invalid character encountered for logical value.
117	`bad variable type` Specified type for the namelist element is invalid. This error is most likely caused by incompatible versions of the front end and the run-time I/O library.
118	`bad namelist name` The specified namelist name cannot be found in the input data file.
119	`variable not in namelist` The namelist variable name in the input data file does not belong to the specified namelist.

Number	Message/Cause
120	`no end record` `$END` is not found at the end of the namelist input data file.
121	`namelist subscript out of range` The array subscript of the character substring value in the input data file exceeds the range for that array or character string.
122	`negative repeat count` The repeat count in the input data file is less than or equal to zero.
123	`illegal operation for unit` You cannot set your own buffer on direct unformatted files.
124	`off beginning of record` Format edit descriptor causes positioning to go off the beginning of the record.
125	`no * after repeat count` An asterisk (`*`) is expected after an integer repeat count.
126	`’new’ file exists` The file is opened as new but already exists.
127	`can’t find ’old’ file` The file is opened as old but does not exist.
128	`unknown system error` An unexpected error was returned by IRIX.
129	`requires seek ability` The file is on a device that cannot do direct access.
130	`illegal argument` Invalid value in the I/O control list.
131	`duplicate key value on write` Cannot write a key that already exists.
132	`indexed file not open` Cannot perform indexed I/O on an unopened file.
133	`bad isam argument` The indexed I/O library function receives a bad argument because of a corrupted index file or bad run-time I/O libraries.
134	`bad key description` The key description is invalid.

Number	Message/Cause
135	`too many open indexed files` Cannot have more than 32 open indexed files.
136	`corrupted isam file` The indexed file format is not recognizable. This error is usually caused by a corrupted file.
137	`isam file not opened for exclusive access` Cannot obtain lock on the indexed file.
138	`record locked` The record has already been locked by another process.
138	`key already exists` The key specification in the `OPEN` statement has already been specified.
140	`cannot delete primary key` `DELETE` cannot be executed on a primary key.
141	`beginning or end of file reached` The index for the specified key points beyond the length of the indexed data file. This error is probably because of corrupted ISAM files or a bad indexed I/O run-time library.
142	`cannot find request record` The requested key for indexed `READ` does not exist.
143	`current record not defined` Cannot execute `REWRITE`, `UNLOCK`, or `DELETE` before doing a `READ` to define the current record.
144	`isam file is exclusively locked` The indexed file has been exclusively locked by another process.
145	`filename too long` The indexed filename exceeds 128 characters.
148	`key structure does not match file structure` Mismatch between the key specifications in the `OPEN` statement and the indexed file.
149	`direct access on an indexed file not allowed` Cannot have direct-access I/O on an indexed file.
150	`keyed access on a sequential file not allowed` Cannot specify keyed access together with sequential organization.

Number	Message/Cause
151	`keyed access on a relative file not allowed` Cannot specify keyed access together with relative organization.
152	`append access on an indexed file not allowed` Cannot specify append access together with indexed organization.
153	`must specify record length` A record length specification is required when opening a direct or keyed access file.
154	`key field value type does not match key type` The type of the given key value does not match the type specified in the `OPEN` statement for that key.
155	`character key field value length too long` The length of the character key value exceeds the length specification for that key.
156	`fixed record on sequential file not allowed` `RECORDTYPE=fixed` cannot be used with a sequential file.
157	`variable records allowed only on unformatted sequential file` `RECORDTYPE=variable` can only be used with an unformatted sequential file.
158	`stream records allowed only on formatted sequential file` `RECORDTYPE=stream_lf` can only be used with a formatted sequential file.
159	`maximum number of records in direct access file exceeded` The specified record is bigger than the `MAXREC=` *value* used in the `OPEN` statement.
160	`attempt to create or write to a read-only file` User does not have write permission on the file.
161	`must specify key descriptions` Must specify all the keys when opening an indexed file.
162	`carriage control not allowed for unformatted units` `CARRIAGECONTROL` specifier can be used only on a formatted file.
163	`indexed files only` Indexed I/O can be done only on logical units that have been opened for indexed (keyed) access.

Number	Message/Cause
164	`cannot use on indexed file` Illegal I/O operation on an indexed (keyed) file.
165	`cannot use on indexed or append file` Illegal I/O operation on an indexed (keyed) or append file.
167	`invalid code in format specification` Unknown code is encountered in format specification.
168	`invalid record number in direct access file` The specified record number is less than 1.
169	`cannot have endfile record on non-sequential file` Cannot have an endfile on a direct- or keyed-access file.
170	`cannot position within current file` Cannot perform `fseek()` on a file opened for sequential unformatted I/O.
171	`cannot have sequential records on direct access file` Cannot do sequential formatted I/O on a file opened for direct access.
173	`cannot read from stdout` Attempt to read from `stdout`.
174	`cannot write to stdin` Attempt to write to `stdin`.
175	`stat call failed in f77inode` The directory information for the file is unreadable.
176	`illegal specifier` The I/O control list contains an invalid value for one of the I/O specifiers. For example, `ACCESS=INDEXED`.
180	`attempt to read from a writeonly file` User does not have read permission on the file.
181	`direct unformatted io not allowed` Direct unformatted file cannot be used with this I/O operation.
182	`cannot open a directory` The name specified in `FILE=` must be the name of a file, not a directory.
183	`subscript out of bounds` The exit status returned when a program compiled with the `-C` option has an array subscript that is out of range.

Number	Message/Cause
184	`function not declared as varargs` Variable argument routines called in subroutines that have not been declared in a `$VARARGS` directive.
185	`internal error` Internal run-time library error.
186	`Illegal input character in formatted read.` The numeric input field in a formatted file contains non-blank characters beyond the maximum usable field width of 83 characters.
187	`Position specifier is allowed on sequential files only` Cannot have a position specifier in a format statement for a non-sequential file.
188	`Position specifier has an illegal value` The position specifier has a value that does not make sense.
189	`Out of memory` The system is out of memory for allocation. Try to allocate more swap space.
195	`Cannot keep a file opened as a scratch file.` If a file is opened as a scratch file, it cannot be kept when closed, and is automatically deleted.

Appendix B

Multiprocessing Directives (Outmoded)

The directives which are described in this appendix are outmoded. They are supported for older codes that require this functionality. SGI encourages you to write new codes using the OpenMP directives described in Chapter 5, "OpenMP Fortran API Multiprocessing Directives", page 65.

This chapter contains these sections:

- "Overview", page 90, provides an overview of this chapter.
- "Parallel Loops", page 90, discusses the concept of parallel `DO` loops.
- "Writing Parallel Fortran", page 91, explains how to use compiler directives to generate code that can be run in parallel.
- "Analyzing Data Dependencies for Multiprocessing", page 99, describes how to analyze `DO` loops to determine whether they can be parallelized.
- "Breaking Data Dependencies", page 104, explains how to rewrite `DO` loops that contain data dependencies so that some or all of the loop can be run in parallel.
- "Work Quantum", page 109, describes how to determine whether the work performed in a loop is greater than the overhead associated with multiprocessing the loop.
- "Cache Effects", page 111, explains how to write loops that account for the effect of the cache.
- "Advanced Features", page 115, describes features that override multiprocessing defaults and customize parallelism.
- "`DOACROSS` Implementation", page 123, discusses how multiprocessing is implemented in a `DOACROSS` routine.
- "PCF Directives", page 125, describes how PCF implements a general model of parallelism.
- "Communicating Between Threads Through Thread Local Data", page 138, explains how to use `mp_shmem` to explicitly communicate between threads of a MP Fortran program.
- "Synchronization Intrinsics", page 140, describes synchronization operations.

Overview

The MIPSpro Fortran 77 compiler allows you to apply the capabilities of a SGI multiprocessor workstation to the execution of a single job. By coding a few simple directives, the compiler splits the job into concurrently executing pieces, thereby decreasing the wall-clock run time of the job. This chapter discusses techniques for analyzing your program and converting it to multiprocessing operations. Chapter 6, "Compiling and Debugging Parallel Fortran", page 75, gives compilation and debugging instructions for parallel processing.

Note: You can automatically parallelize Fortran programs by using the optional program `-apo`. For information about this software, contact SGI customer support.

Parallel Loops

The model of parallelism used focuses on the Fortran `DO` loop. The compiler executes different iterations of the `DO` loop in parallel on multiple processors. For example, suppose a `DO` loop consisting of 200 iterations will run on a machine with four processors using the `SIMPLE` scheduling method (described in "`CHUNK`, `MP_SCHEDTYPE`", page 94). The first 50 iterations run on one processor, the next 50 on another, and so on.

The multiprocessing code adjusts itself at run time to the number of processors actually present on the machine. By default, the multiprocessing code does not use more than 8 processors. If you want to use more processors, set the environment variable `MP_SET_NUMTHREADS` (see "Environment Variables for Origin Systems ", page 118, for more information). If the above 200-iteration loop was moved to a machine with only two processors, it would be divided into two blocks of 100 iterations each, without any need to recompile or relink. In fact, multiprocessing code can be run on single-processor machines. So the above loop is divided into one block of 200 iterations. This allows code to be developed on a single-processor SGI workstation, and later run on an IRIS POWER Series multiprocessor.

The processes that participate in the parallel execution of a task are arranged in a master/slave organization. The original process is the master. It creates zero or more slaves to assist. When a parallel `DO` loop is encountered, the master asks the slaves for help. When the loop is complete, the slaves wait on the master, and the master resumes normal execution. The master process and each of the slave processes are called a *thread of execution* or simply a *thread*. By default, the number of threads is set to the number of processors on the machine or 4, whichever is smaller. If you want, you can override the default and explicitly control the number of threads of execution used by a parallel job.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of when any other iteration of the loop is executed. Not all `DO` loops have this property, and loops without it cannot be correctly executed in parallel. However, many of the loops encountered in practice fit this model. Further, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially in parallel.

To provide compatibility for existing parallel programs, SGI has adopted the syntax for parallelism used by Sequent Computer Corporation. This syntax takes the form of compiler directives embedded in comments. These fairly high-level directives provide a convenient method for you to describe a parallel loop, while leaving the details to the Fortran compiler. For advanced users the proposed Parallel Computing Forum (PCF) standard (ANSI-X3H5 91-0023-B Fortran language binding) is available (refer to "PCF Directives", page 125). Additionally, a number of special routines exist that permit more direct control over the parallel execution (refer to "Advanced Features", page 115, for more information.)

Writing Parallel Fortran

The compiler accepts directives that cause it to generate code that can be run in parallel. The compiler directives look like Fortran comments: they begin with a `C` in column one. If multiprocessing is not turned on, these statements are treated as comments. This allows the identical source to be compiled with a single-processing compiler or by Fortran without the multiprocessing option. The directives are distinguished by having a `$` as the second character. There are six directives that are supported: `C$DOACROSS`, `C$&`, `C$`, `C$MP_SCHEDTYPE`, `C$CHUNK`, and `C$COPYIN`. The `C$COPYIN` directive is described in "Local `COMMON` Blocks", page 121. This section describes the others.

C$DOACROSS

The essential compiler directive for multiprocessing is `C$DOACROSS`. This directive directs the compiler to generate special code to run iterations of a `DO` loop in parallel. The `C$DOACROSS` directive applies only to the next statement (which must be a `DO` loop). The `C$DOACROSS` directive has the form

```
C$DOACROSS [clause [ [,] clause ...]
```

where valid values for the optional *clause* are the following:

```
[IF (logical_expression)]
[{LOCAL | PRIVATE} (item[,item ...])]
[{SHARE | SHARED} (item[,item ...])]
[{LASTLOCAL | LAST LOCAL} (item[,item ...])]
[REDUCTION (item[,item ...])]
[MP_SCHEDTYPE=mode ]
[CHUNK=integer_expression]
```

The preferred form of the directive (as generated by WorkShop Pro MPF) uses the optional commas between clauses. This section discusses the meaning of each clause.

IF

The `IF` clause determines whether the loop is actually executed in parallel. If the logical expression is TRUE, the loop is executed in parallel. If the expression is FALSE, the loop is executed serially. Typically, the expression tests the number of times the loop will execute to be sure that there is enough work in the loop to amortize the overhead of parallel execution. Currently, the break-even point is about 4000 CPU clocks of work, which normally translates to about 1000 floating point operations.

LOCAL, SHARE, LASTLOCAL

The `LOCAL`, `SHARE`, and `LASTLOCAL` clauses specify lists of variables used within parallel loops. A variable can appear in only one of these lists. To make the task of writing these lists easier, there are several defaults. The loop-iteration variable is `LASTLOCAL` by default. All other variables are `SHARE` by default.

`LOCAL`	Specifies variables that are local to each process. If a variable is declared as `LOCAL`, each iteration of the loop is given its own uninitialized copy of the variable. You can declare a variable as `LOCAL` if its value does not depend on any other iteration of the loop and if its value is used only within a single iteration. In effect the `LOCAL` variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. The name `LOCAL` is preferred over `PRIVATE`.

SHARE	Specifies variables that are shared across all processes. If a variable is declared as SHARE, all iterations of the loop use the same copy of the variable. You can declare a variable as SHARE if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. The name SHARE is preferred over SHARED.
LASTLOCAL	Specifies variables that are local to each process.Unlike with the LOCAL clause, the compiler saves only the value of the logically last iteration of the loop when it exits. The name LASTLOCAL is preferred over LAST LOCAL.

LOCAL is a little faster than LASTLOCAL, so if you do not need the final value, it is good practice to put the DO loop index variable into the LOCAL list, although this is not required.

Only variables can appear in these lists. In particular, COMMON blocks cannot appear in a LOCAL list (but see the discussion of local COMMON blocks in "Advanced Features", page 115). The SHARE, LOCAL, and LASTLOCAL lists give only the names of the variables. If any member of the list is an array, it is listed without any subscripts.

REDUCTION

The REDUCTION clause specifies variables involved in a reduction operation. In a reduction operation, the compiler keeps local copies of the variables and combines them when it exits the loop. For an example and details see Example B-17, page 107. An element of the REDUCTION list must be an individual variable (also called a scalar variable) and cannot be an array. However, it can be an individual element of an array. In a REDUCTION clause, it would appear in the list with the proper subscripts.

One element of an array can be used in a reduction operation, while other elements of the array are used in other ways. To allow for this, if an element of an array appears in the REDUCTION list, the entire array can also appear in the SHARE list.

The four types of reductions supported are sum(+), product(*), min(), and max(). Note that min(max) reductions must use the min(max) intrinsic functions to be recognized correctly.

The compiler confirms that the reduction expression is legal by making some simple checks. The compiler does not, however, check all statements in the DO loop for illegal reductions. You must ensure that the reduction variable is used correctly in a reduction operation.

CHUNK, MP_SCHEDTYPE

The CHUNK and MP_SCHEDTYPE clauses affect the way the compiler schedules work among the participating tasks in a loop. These clauses do not affect the correctness of the loop. They are useful for tuning the performance of critical loops. See "Load Balancing", page 113, for more details.

For the MP_SCHEDTYPE=*mode* clause, *mode* can be one of the following:

```
[SIMPLE | STATIC]
[DYNAMIC]
[INTERLEAVE|INTERLEAVED]
[GUIDED|GSS]
[RUNTIME]
```

You can use any or all of these modes in a single program. The CHUNK clause is valid only with the DYNAMIC and INTERLEAVE modes. SIMPLE, DYNAMIC, INTERLEAVE, GSS, and RUNTIME are the preferred names for each mode.

The simple method (MP_SCHEDTYPE=SIMPLE) divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process.

In dynamic scheduling (MP_SCHEDTYPE=DYNAMIC) the iterations are broken into pieces the size of which is specified with the CHUNK clause. As each process finishes a piece, it enters a critical section to grab the next available piece. This gives good load balancing at the price of higher overhead.

The interleave method (MP_SCHEDTYPE=INTERLEAVE) breaks the iterations into pieces of the size specified by the CHUNK option, and execution of those pieces is interleaved among the processes. For example, if there are four processes and CHUNK=2, then the first process will execute iterations 1–2, 9–10, 17–18, …; the second process will execute iterations 3–4, 11–12, 19–20,…; and so on. Although this is more complex than the simple method, it is still a fixed schedule with only a single scheduling decision.

The fourth method is a variation of the guided self-scheduling algorithm (MP_SCHEDTYPE=GSS). Here, the piece size is varied depending on the number of iterations remaining. By parceling out relatively large pieces to start with and relatively small pieces toward the end, the system can achieve good load balancing while reducing the number of entries into the critical section.

In addition to these four methods, you can specify the scheduling method at run time (MP_SCHEDTYPE=RUNTIME). Here, the scheduling routine examines values in your run-time environment and uses that information to select one of the other four methods. See "Advanced Features", page 115, for more details.

If both the MP_SCHEDTYPE and CHUNK clauses are omitted, SIMPLE scheduling is assumed. If MP_SCHEDTYPE is set to INTERLEAVE or DYNAMIC and the CHUNK clause are omitted, CHUNK=1 is assumed. If MP_SCHEDTYPE is set to one of the other values, CHUNK is ignored. If the MP_SCHEDTYPE clause is omitted, but CHUNK is set, then MP_SCHEDTYPE=DYNAMIC is assumed.

Example 6-1 Simple DOACROSS

The code fragment

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

could be multiprocessed with the directive:

```
C$DOACROSS LOCAL(I), SHARE(A, B)
     DO 10 I = 1, 100
        A(I) = B(I)
10 CONTINUE
```

Here, the defaults are sufficient, provided A and B are mentioned in a nonparallel region or in another SHARE list. The following then works:

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

Example 6-2 DOACROSS LOCAL

Consider the following code fragment:

```
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

You can be fully explicit, as shown below:

```
C$DOACROSS LOCAL(I, X), share(A, B, C, D, N)
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

You can also use the defaults:

```
C$DOACROSS LOCAL(X)
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

See Example B-8, page 101, for more information on this example.

Example 6-3 `DOACROSS LAST LOCAL`

Consider the following code fragment:

```
     DO 10 I = M, K, N
        X = D(I)**2
        Y = X + X
        DO 20 J = I, MAX
           A(I,J) = A(I,J) + B(I,J) * C(I,J) * X + Y
20 CONTINUE
10 CONTINUE
     PRINT*, I, X
```

Here, the final values of `I` and `X` are needed after the loop completes. A correct directive is shown below:

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X),
C$& SHARE(M,K,N,ITOP,A,B,C,D)
     DO 10 I = M, K, N
         X = D(I)**2
         Y = X + X
         DO 20 J = I, ITOP
             A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20 CONTINUE
10 CONTINUE
     PRINT*, I, X
```

You can also use the defaults:

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(X)
     DO 10 I = M, K, N
         X = D(I)**2
         Y = X + X
         DO 20 J = I, MAX
             A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20 CONTINUE
10 CONTINUE
     PRINT*, I, X
```

I is a loop index variable for the C$DOACROSS loop, so it is LASTLOCAL by default. However, even though J is a loop index variable, it is not the loop index of the loop being multiprocessed and has no special status. If it is not declared, it is assigned the default value of SHARE, which produces an incorrect answer.

C$&

Occasionally, the clauses in the C$DOACROSS directive are longer than one line. Use the C$& directive to continue the directive onto multiple lines. For example:

```
C$DOACROSS share(ALPHA, BETA, GAMMA, DELTA,
C$&  EPSILON, OMEGA), LASTLOCAL(I, J, K, L, M, N),
C$&  LOCAL(XXX1, XXX2, XXX3, XXX4, XXX5, XXX6, XXX7,
C$&  XXX8, XXX9)
```

C$

The C$ directive is considered a comment line except when multiprocessing. A line beginning with C$ is treated as a conditionally compiled Fortran statement. The rest of the line contains a standard Fortran statement. The statement is compiled only if multiprocessing is turned on. In this case, the C and $ are treated as if they are blanks. They can be used to insert debugging statements, or an experienced user can use them to insert arbitrary code into the multiprocessed version.

The following code demonstrates the use of the C$ directive:

```
C$    PRINT 10
C$ 10 FORMAT('BEGIN MULTIPROCESSED LOOP')
C$DOACROSS LOCAL(I), SHARE(A,B)
```

```
DO I = 1, 100
   CALL COMPUTE(A, B, I)
END DO
```

C$MP_SCHEDTYPE and C$CHUNK

The `C$MP_SCHEDTYPE=`*mode* directive acts as an implicit `MP_SCHEDTYPE` clause for all `C$DOACROSS` directives in scope. *mode* is any of the modes listed in the section called "`CHUNK`, `MP_SCHEDTYPE`", page 94. A `C$DOACROSS` directive that does not have an explicit `MP_SCHEDTYPE` clause is given the value specified in the last directive prior to the loop, rather than the normal default. If the `C$DOACROSS` does have an explicit clause, then the explicit value is used.

The `C$CHUNK=`*integer_expression* directive affects the `CHUNK` clause of a `C$DOACROSS` in the same way that the `C$MP_SCHEDTYPE` directive affects the `MP_SCHEDTYPE` clause for all `C$DOACROSS` directives in scope. Both directives are in effect from the place they occur in the source until another corresponding directive is encountered or the end of the procedure is reached.

C$COPYIN

It is occasionally useful to be able to copy values from the master thread's version of the `COMMON` block into the slave thread's version. The special directive `C$COPYIN` allows this. It has the following form:

`C$COPYIN ,`*item*`[,`*item*`]...`

Each *item* must be a member of a local `COMMON` block. It can be a variable, an array, an individual element of an array, or the entire `COMMON` block.

Note: The `C$COPYIN` directive cannot be executed from inside a parallel region.

Nesting C$DOACROSS

The Fortran compiler does not support direct nesting of `C$DOACROSS` loops.

For example, the following is illegal and generates a compilation error:

```
C$DOACROSS LOCAL(I)
        DO I = 1, N
C$DOACROSS LOCAL(J)
           DO J = 1, N
              A(I,J) = B(I,J)
           END DO
        END DO
```

However, to simplify separate compilation, a different form of nesting is allowed. A routine that uses `C$DOACROSS` can be called from within a multiprocessed region. This can be useful if a single routine is called from several different places: sometimes from within a multiprocessed region, sometimes not. Nesting does not increase the parallelism. When the first `C$DOACROSS` loop is encountered, that loop is run in parallel. If while in the parallel loop a call is made to a routine that itself has a `C$DOACROSS`, this subsequent loop is executed serially.

Analyzing Data Dependencies for Multiprocessing

The essential condition required to parallelize a loop correctly is that each iteration of the loop must be independent of all other iterations. If a loop meets this condition, then the order in which the iterations of the loop execute is not important. They can be executed backward or at the same time, and the answer is still the same. This property is captured by the concept of *data independence*. For a loop to be data-independent, no iterations of the loop can write a value into a memory location that is read or written by any other iteration of that loop. It is all right if the same iteration reads and/or writes a memory location repeatedly as long as no others do; it is all right if many iterations read the same location, as long as none of them write to it. In a Fortran program, memory locations are represented by variable names. So, to determine if a particular loop can be run in parallel, examine the way variables are used in the loop. Because data dependence occurs only when memory locations are modified, pay particular attention to variables that appear on the left-hand side of assignment statements. If a variable is not modified or if it is passed to a function or subroutine, there is no data dependence associated with it.

The Fortran compiler supports four kinds of variable usage within a parallel loop: `SHARE`, `LOCAL`, `LASTLOCAL`, and `REDUCTION`. If a variable is declared as `SHARE`, all iterations of the loop use the same copy. If a variable is declared as `LOCAL`, each iteration is given its own uninitialized copy. A variable is declared `SHARE` if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. A variable can be `LOCAL` if its value does not depend on any other iteration and if its value is used only within a single iteration. In effect the LOCAL variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. As a special case, if only the very last value of a variable computed on the very last iteration is used outside the loop (but would otherwise qualify as a LOCAL variable), the loop can be multiprocessed by declaring the variable to be LASTLOCAL. "REDUCTION", page 93, describes the use of REDUCTION variables.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to determine if it fulfills the criteria for LOCAL, LASTLOCAL, SHARE, or REDUCTION. If all of the variables' uses conform, the loop can be parallelized. If not, the loop cannot be parallelized as it stands, but possibly can be rewritten into an equivalent parallel form. (See "Breaking Data Dependencies", page 104, for information on rewriting code in parallel form.)

An alternative to analyzing variable usage by hand is to use the MIPSpro Auto-Parallelizer Option (APO). This optional software package is a Fortran preprocessor that analyzes loops for data dependence. If APO determines that a loop is data-independent, it automatically inserts the required compiler directives (see "Writing Parallel Fortran", page 91). If APO cannot determine whether the loop is independent, it produces a listing file detailing where the problems lie.

The rest of this section is devoted to analyzing sample loops, some parallel and some not parallel.

Example B-4 Simple Independence

```
      DO 10 I = 1,N
10    A(I) = X + B(I)*C(I)
```

In this example, each iteration writes to a different location in A, and none of the variables appearing on the right-hand side is ever written to, only read from. This loop can be correctly run in parallel. All the variables are SHARE except for I, which is either LOCAL or LASTLOCAL, depending on whether the last value of I is used later in the code.

Example B-5 Data Dependence

```
       DO 20 I = 2,N
20     A(I) = B(I) - A(I-1)
```

This fragment contains A(I) on the left-hand side and A(I-1) on the right. This means that one iteration of the loop writes to a location in A and the next iteration reads from that same location. Because different iterations of the loop read and write the same memory location, this loop cannot be run in parallel.

Example B-6 Stride Not 1

```
         DO 20 I = 2,N,2
   20    A(I) = B(I) - A(I-1)
```

This example looks like the previous example. The difference is that the stride of the DO loop is now two rather than one. Now A(I) references every other element of A, and A(I-1) references exactly those elements of A that are not referenced by A(I). None of the data locations on the right-hand side is ever the same as any of the data locations written to on the left-hand side. The data are disjoint, so there is no dependence. The loop can be run in parallel. Arrays A and B can be declared SHARE, while variable I should be declared LOCAL or LASTLOCAL.

Example B-7 Local Variable

```
DO I = 1, N
     X = A(I)*A(I) + B(I)
     B(I) = X + B(I)*X
END DO
```

In this loop, each iteration of the loop reads and writes the variable X. However, no loop iteration ever needs the value of X from any other iteration. X is used as a temporary variable; its value does not survive from one iteration to the next. This loop can be parallelized by declaring X to be a LOCAL variable within the loop. Note that B(I) is both read and written by the loop. This is not a problem because each iteration has a different value for I, so each iteration uses a different B(I). The same B(I) is allowed to be read and written as long as it is done by the same iteration of the loop. The loop can be run in parallel. Arrays A and B can be declared SHARE, while variable I should be declared LOCAL or LASTLOCAL.

Example B-8 Function Call

```
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

The value of X in any iteration of the loop is independent of the value of X in any other iteration, so X can be made a LOCAL variable. The loop can be run in parallel. Arrays A, B, C, and D can be declared SHARE, while variable I should be declared LOCAL or LASTLOCAL.

The interesting feature of this loop is that it invokes an external routine, `SQRT`. It is possible to use functions and/or subroutines (intrinsic or user defined) within a parallel loop. However, make sure that the various parallel invocations of the routine do not interfere with one another. In particular, `SQRT` returns a value that depends only on its input argument, does not modify global data, and does not use static storage. We say that `SQRT` has no *side effects*.

All the Fortran intrinsic functions listed in the *MIPSpro Fortran 77 Language Reference Manual* have no side effects and can safely be part of a parallel loop. For the most part, the Fortran library functions and VMS intrinsic subroutine extensions (listed in Chapter 4, "System Functions and Subroutines", page 53) cannot safely be included in a parallel loop. In particular, `rand` is not safe for multiprocessing. For user-written routines, it is the user's responsibility to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the `-static` option when compiling routines called within a parallel loop.

Example B-9 Rewritable Data Dependence

```
INDX = 0
DO I = 1, N
     INDX = INDX + I
     A(I) = B(I) + C(INDX)
END DO
```

Here, the value of `INDX` survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written. Making `INDX` a `LOCAL` variable does not work; you need the value of `INDX` computed in the previous iteration. It is possible to rewrite this loop to make it parallel (see Example B-14, page 105).

Example B-10 Exit Branch

```
       DO I = 1, N
          IF (A(I) .LT. EPSILON) GOTO 320
          A(I) = A(I) * B(I)
       END DO
   320 CONTINUE
```

This loop contains an exit branch; that is, under certain conditions the flow of control suddenly exits the loop. The Fortran compiler cannot parallelize loops containing exit branches.

Example B-11 Complicated Independence

```
DO I = K+1, 2*K
   W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

At first glance, this loop looks like it cannot be run in parallel because it uses both `W(I)` and `W(I-K)`. Closer inspection reveals that because the value of `I` varies between `K`+1 and `2*K`, then `I-K` goes from 1 to `K`. This means that the `W(I-K)` term varies from `W(1)` up to `W`(`K`), while the `W`(`I`) term varies from `W(K+1)` up to `W(2*K)`. So `W(I-K)` in any iteration of the loop is never the same memory location as `W(I)` in any other iterations. Because there is no data overlap, there are no data dependencies. This loop can be run in parallel. Elements `W`, `B`, and `K` can be declared `SHARE`, while variable `I` should be declared `LOCAL` or `LASTLOCAL`.

This example points out a general rule: the more complex the expression used to index an array, the harder it is to analyze. If the arrays in a loop are indexed only by the loop index variable, the analysis is usually straightforward though tedious. Fortunately, in practice most array indexing expressions are simple.

Example B-12 Inconsequential Data Dependence

```
INDEX = SELECT(N)
DO I = 1, N
   A(I) = A(INDEX)
END DO
```

There is a data dependence in this loop because it is possible that at some point `I` will be the same as `INDEX`, so there will be a data location that is being read and written by different iterations of the loop. In this special case, you can simply ignore it. You know that when `I` and `INDEX` are equal, the value written into `A(I)` is exactly the same as the value that is already there. The fact that some iterations of the loop read the value before it is written and some after it is written is not important because they all get the same value. Therefore, this loop can be parallelized. Array `A` can be declared `SHARE`, while variable `I` should be declared `LOCAL` or `LASTLOCAL`.

Example B-13 Local Array

```
DO I = 1, N
   D(1) = A(I,1) - A(J,1)
   D(2) = A(I,2) - A(J,2)
   D(3) = A(I,3) - A(J,3)
   TOTAL_DISTANCE(I,J) = SQRT(D(1)**2 + D(2)**2 + D(3)**2)
END DO
```

In this fragment, each iteration of the loop uses the same locations in the `D` array. However, closer inspection reveals that the entire `D` array is being used as a temporary. This can be multiprocessed by declaring `D` to be `LOCAL`. The Fortran compiler allows arrays (even multidimensional arrays) to be `LOCAL` variables with one restriction: the size of the array must be known at compile time. The dimension bounds must be constants; the `LOCAL` array cannot have been declared using a variable or the asterisk syntax.

Therefore, this loop can be parallelized. Arrays `TOTAL_DISTANCE` and `A` can be declared `SHARE`, while array `D` and variable `I` should be declared `LOCAL` or `LASTLOCAL`.

Breaking Data Dependencies

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. The essential idea is to locate the statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. If this fails, try to pull the statements out of the loop and into a separate loop, allowing the remainder of the original loop to be run in parallel.

The first step is to analyze the loop to discover the data dependencies (see "Writing Parallel Fortran", page 91). Once you have identified these areas, you can use various techniques to rewrite the code to break the dependence. Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to discover a new parallel method of solving the problem. The rest of this section is devoted to a series of “cookbook” examples on how to deal with commonly occurring situations. These are by no means exhaustive but cover many situations that happen in practice.

Example B-14 Loop Carried Value

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

This code segment is the same as in Example B-9, page 102. `INDX` has its value carried from iteration to iteration. However, you can compute the appropriate value for `INDX` without making reference to any previous value.

For example, consider the following code:

```
C$DOACROSS LOCAL (I, INDX)
    DO I  = 1, N
       INDX = (I*(I+1))/2
       A(I) = B(I) + C(INDX)
    END DO
```

In this loop, the value of `INDX` is computed without using any values computed on any other iteration. `INDX` can correctly be made a `LOCAL` variable, and the loop can now be multiprocessed.

Example B-15 Indirect Indexing

```
     DO 100 I = 1, N
        IX = INDEXX(I)
        IY = INDEXY(I)
        XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
        YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
        IXX = IXOFFSET(IX)
        IYY = IYOFFSET(IY)
        TOTAL(IXX, IYY) = TOTAL(IXX, IYY) + EPSILON
100 CONTINUE
```

It is the final statement that causes problems. The indexes `IXX` and `IYY` are computed in a complex way and depend on the values from the `IXOFFSET` and `IYOFFSET` arrays. We do not know if `TOTAL (IXX,IYY)` in one iteration of the loop will always be different from `TOTAL (IXX,IYY)` in every other iteration of the loop.

We can pull the statement out into its own separate loop by expanding `IXX` and `IYY` into arrays to hold intermediate values:

```
C$DOACROSS LOCAL(IX, IY, I)
      DO I  = 1, N
         IX = INDEXX(I)
         IY = INDEXY(I)
         XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
         YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
         IXX(I) = IXOFFSET(IX)
         IYY(I) = IYOFFSET(IY)
      END DO
      DO 100 I = 1, N
         TOTAL(IXX(I),IYY(I)) = TOTAL(IXX(I), IYY(I)) + EPSILON
100 CONTINUE
```

Here, IXX and IYY have been turned into arrays to hold all the values computed by the first loop. The first loop (containing most of the work) can now be run in parallel. Only the second loop must still be run serially. This will be true if IXOFFSET or IYOFFSET are permutation vectors.

Before we leave this example, note that if we were certain that the value for IXX was always different in every iteration of the loop, then the original loop could be run in parallel. It could also be run in parallel if IYY was always different. If IXX (or IYY) is always different in every iteration, then TOTAL(IXX,IYY) is never the same location in any iteration of the loop, and so there is no data conflict.

This sort of knowledge is, of course, program-specific and should always be used with great care. It may be true for a particular data set, but to run the original code in parallel as it stands, you need to be sure it will always be true for all possible input data sets.

Example B-16 Recurrence

```
DO I = 1,N
   X(I) = X(I-1) + Y(I)
END DO
```

This is an example of recurrence, which exists when a value computed in one iteration is immediately used by another iteration. There is no good way of running this loop in parallel. If this type of construct appears in a critical loop, try pulling the statement(s) out of the loop as in the previous example. Sometimes another loop encloses the recurrence; in that case, try to parallelize the outer loop.

Example B-17 Sum Reduction

```
SUM  = 0.0
DO I = 1,N
   SUM = SUM + A(I)
END DO
```

This operation is known as a *reduction*. Reductions occur when an array of values is combined and reduced into a single value. This example is a sum reduction because the combining operation is addition. Here, the value of `SUM` is carried from one loop iteration to the next, so this loop cannot be multiprocessed. However, because this loop simply sums the elements of `A(I)`, we can rewrite the loop to accumulate multiple, independent subtotals.

Then we can do much of the work in parallel:

```
   NUM_THREADS = MP_NUMTHREADS()
C
C  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
C
   IPIECE_SIZE = (N + (NUM_THREADS -1)) / NUM_THREADS
   DO K = 1, NUM_THREADS
   PARTIAL_SUM(K) = 0.0
C
C  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
C  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
C  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
C  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
C  HENCE THE "MIN" EXPRESSION.
C
   DO I =K*IPIECE_SIZE -IPIECE_SIZE +1, MIN(K*IPIECE_SIZE,N)
      PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
   END DO
   END DO
C
C  NOW ADD UP THE PARTIAL SUMS
   SUM = 0.0
   DO I = 1, NUM_THREADS
      SUM = SUM + PARTIAL_SUM(I)
   END DO
```

The outer `K` loop can be run in parallel. In this method, the array pieces for the partial sums are contiguous, resulting in good cache utilization and performance.

This is an important and common transformation, and so automatic support is provided by the REDUCTION clause:

```
     SUM = 0.0
C$DOACROSS LOCAL (I), REDUCTION (SUM)
   DO 10 I = 1, N
      SUM = SUM + A(I)
10 CONTINUE
```

The previous code has essentially the same meaning as the much longer and more confusing code above. It is an important example to study because the idea of adding an extra dimension to an array to permit parallel computation, and then combining the partial results, is an important technique for trying to break data dependencies. This idea occurs over and over in various contexts and disguises.

Note that reduction transformations such as this do not produce the same results as the original code. Because computer arithmetic has limited precision, when you sum the values together in a different order, as was done here, the round-off errors accumulate slightly differently. It is likely that the final answer will be slightly different from the original loop. Both answers are equally "correct." Most of the time the difference is irrelevant, but sometimes it can be significant, so some caution is in order. If the difference is significant, neither answer is really trustworthy.

This example is a `sum` reduction because the operator is plus (+). The Fortran compiler supports three other types of reduction operations:

1. sum: `p = p+a(i)`
2. product: `p = p*a(i)`
3. min: `m = min(m,a(i))`
4. max: `m = max(m,a(i))`

For example,

```
C$DOACROSS LOCAL(I),REDUCTION(BG_SUM,BG_PROD,BG_MIN,BG_MAX)
          DO I = 1,N
             BG_SUM  = BG_SUM + A(I)
             BG_PROD = BG_PROD * A(I)
             BG_MIN  = MIN(BG_MIN, A(I))
             BG_MAX  = MAX(BG_MAX, A(I)
          END DO
```

One further example of a reduction transformation is noteworthy. Consider this code:

```
DO I = 1, N
   TOTAL = 0.0
   DO J = 1, M
      TOTAL = TOTAL + A(J)
   END DO
   B(I) = C(I) * TOTAL
END DO
```

Initially, it might look as if the inner loop should be parallelized with a `REDUCTION` clause. However, look at the outer `I` loop. Although `TOTAL` cannot be made a `LOCAL` variable in the inner loop, it fulfills the criteria for a `LOCAL` variable in the outer loop: the value of `TOTAL` in each iteration of the outer loop does not depend on the value of `TOTAL` in any other iteration of the outer loop. Thus, you do not have to rewrite the loop; you can parallelize this reduction on the outer `I` loop, making `TOTAL` and `J` local variables.

Work Quantum

A certain amount of overhead is associated with multiprocessing a loop. If the work occurring in the loop is small, the loop can actually run slower by multiprocessing than by single processing. To avoid this, make the amount of work inside the multiprocessed region as large as possible.

Example B-18 Loop Interchange

```
DO K = 1, N
   DO I = 1, N
      DO J = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
      END DO
   END DO
END DO
```

Here you have several choices: parallelize the `J` loop or the `I` loop. You cannot parallelize the `K` loop because different iterations of the `K` loop will all try to read and write the same values of `A(I,J)`. Try to parallelize the outermost `DO` loop possible, because it encloses the most work. In this example, that is the `I` loop. For this example, use the technique called *loop interchange*. Although the parallelizable loops are not the outermost ones, you can reorder the loops to make one of them outermost.

Thus, loop interchange would produce

```
C$DOACROSS LOCAL(I, J, K)
      DO I = 1, N
         DO K = 1, N
            DO J = 1, N
               A(I,J) = A(I,J) + B(I,K) * C(K,J)
            END DO
         END DO
      END DO
```

Now the parallelizable loop encloses more work and shows better performance. In practice, relatively few loops can be reordered in this way. However, it does occasionally happen that several loops in a nest of loops are candidates for parallelization. In such a case, it is usually best to parallelize the outermost one.

Occasionally, the only loop available to be parallelized has a fairly small amount of work. It may be worthwhile to force certain loops to run without parallelism or to select between a parallel version and a serial version, on the basis of the length of the loop.

Example B-19 Conditional Parallelism

```
J = (N/4) * 4
DO I = J+1, N
   A(I) = A(I) + X*B(I)
END DO
DO I = 1, J, 4
   A(I) = A(I) + X*B(I)
   A(I+1) = A(I+1) + X*B(I+1)
   A(I+2) = A(I+2) + X*B(I+2)
   A(I+3) = A(I+3) + X*B(I+3)
END DO
```

Here you are using loop unrolling of order four to improve speed. For the first loop, the number of iterations is always fewer than four, so this loop does not do enough work to justify running it in parallel. The second loop is worthwhile to parallelize if `N` is big enough. To overcome the parallel loop overhead, `N` needs to be around 500.

An optimized version would use the `IF` clause on the `DOACROSS` directive:

```
J = (N/4) * 4
DO I = J+1, N
   A(I) = A(I) + X*B(I)
END DO
```

```
C$DOACROSS IF (J.GE.500), LOCAL(I)
        DO I = 1, J, 4
           A(I) = A(I) + X*B(I)
           A(I+1) = A(I+1) + X*B(I+1)
           A(I+2) = A(I+2) + X*B(I+2)
           A(I+3) = A(I+3) + X*B(I+3)
        END DO
     ENDIF
```

Cache Effects

It is good policy to write loops that take the effect of the cache into account, with or without parallelism. The technique for the best cache performance is also quite simple: make the loop step through the array in the same way that the array is laid out in memory. For Fortran, this means stepping through the array without any gaps and with the leftmost subscript varying the fastest. Note that this does not depend on multiprocessing, nor is it required in order for multiprocessing to work correctly. However, multiprocessing can affect how the cache is used, so it is worthwhile to understand.

Performing a Matrix Multiply

Consider the following code segment:

```
DO I = 1, N
   DO K = 1, N
      DO J = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
      END DO
   END DO
END DO
```

This is the same as Example B-18, page 109. To get the best cache performance, the `I` loop should be innermost. At the same time, to get the best multiprocessing performance, the outermost loop should be parallelized.

For this example, you can interchange the `I` and `J` loops, and get the best of both optimizations:

```
C$DOACROSS LOCAL(I, J, K)
      DO J = 1, N
         DO K = 1, N
            DO I = 1, N
               A(I,J) = A(I,J) + B(I,K) * C(K,J)
            END DO
         END DO
      END DO
```

Understanding Trade-Offs

Sometimes you must choose between the possible optimizations and their costs. Look at the following code segment:

```
DO J = 1, N
   DO I = 1, M
      A(I) = A(I) + B(J)*C(I,J)
   END DO
END DO
```

This loop can be parallelized on `I` but not on `J`. You could interchange the loops to put `I` on the outside, thus getting a bigger work quantum.

```
C$DOACROSS LOCAL(I,J)
   DO I = 1, M
      DO J = 1, N
         A(I) = A(I) + B(J)*C(I,J)
      END DO
   END DO
```

However, putting `J` on the inside means that you will step through the `C` array in the wrong direction; the leftmost subscript should be the one that varies the fastest. It is possible to parallelize the `I` loop where it stands:

```
   DO J = 1, N
C$DOACROSS LOCAL(I)
      DO I = 1, M
         A(I) = A(I) + B(J)*C(I,J)
      END DO
   END DO
```

However, M needs to be large for the work quantum to show any improvement. In this example, A(I) is used to do a sum reduction, and it is possible to use the reduction techniques shown in Example B-17, page 107, of "Breaking Data Dependencies", page 104, to rewrite this in a parallel form. (Recall that there is no support for an entire array as a member of the REDUCTION clause on a DOACROSS.) However, that involves converting array A from a one-dimensional array to a two-dimensional array to hold the partial sums; this is analogous to the way we converted the scalar summation variable into an array of partial sums.

If A is large, however, the conversion can take more memory than you can spare. It can also take extra time to initialize the expanded array and increase the memory bandwidth requirements.

```
   NUM = MP_NUMTHREADS()
   IPIECE = (N + (NUM-1)) / NUM
C$DOACROSS LOCAL(K,J,I)
   DO K = 1, NUM
      DO J = K*IPIECE - IPIECE + 1, MIN(N, K*IPIECE)
         DO I = 1, M
            PARTIAL_A(I,K) = PARTIAL_A(I,K) + B(J)*C(I,J)
         END DO
      END DO
   END DO
C$DOACROSS LOCAL (I,K)
   DO I = 1, M
      DO K = 1, NUM
         A(I) = A(I) + PARTIAL_A(I,K)
      END DO
     END DO
```

You must trade off the various possible optimizations to find the combination that is right for the particular job.

Load Balancing

When the Fortran compiler divides a loop into pieces, by default it uses the simple method of separating the iterations into contiguous blocks of equal size for each process. It can happen that some iterations take significantly longer to complete than other iterations. At the end of a parallel region, the program waits for all processes to complete their tasks. If the work is not divided evenly, time is wasted waiting for the slowest process to finish.

Example 6-20 Load Balancing

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

The previous code segment can be parallelized on the `I` loop. Because the inner loop goes from 1 to `I`, the first block of iterations of the outer loop will end long before the last block of iterations of the outer loop.

In this example, this is easy to see and predictable, so you can change the program:

```
   NUM_THREADS = MP_NUMTHREADS()
C$DOACROSS LOCAL(I, J, K)
   DO K = 1, NUM_THREADS
      DO I = K, N, NUM_THREADS
         DO J = 1, I
            A(J, I) = A(J, I) + B(J)*C(I)
         END DO
      END DO
   END DO
```

In this rewritten version, instead of breaking up the `I` loop into contiguous blocks, break it into interleaved blocks. Thus, each execution thread receives some small values of `I` and some large values of `I`, giving a better balance of work between the threads. Interleaving usually, but not always, cures a load balancing problem.

You can use the `MP_SCHEDTYPE` clause to automatically perform this desirable transformation.

```
C$DOACROSS LOCAL (I,J), MP_SCHEDTYPE=INTERLEAVE
       DO 20 I = 1, N
          DO 10 J = 1, I
          A (J,I) = A(J,I) + B(J)*C(J)
 10    CONTINUE
 20    CONTINUE
```

The previous code has the same meaning as the rewritten form above.

Note that interleaving can cause poor cache performance because the array is no longer stepped through at stride 1. You can improve performance somewhat by adding a `CHUNK`=*integer_expression* clause. Usually 4 or 8 is a good value for *integer_expression*. Each small chunk will have stride 1 to improve cache performance, while the chunks are interleaved to improve load balancing.

The way that iterations are assigned to processes is known as `scheduling`. Interleaving is one possible schedule. Both interleaving and the "simple" scheduling methods are examples of `fixed` schedules; the iterations are assigned to processes by a single decision made when the loop is entered. For more complex loops, it may be desirable to use `DYNAMIC` or `GSS` schedules.

Comparing the output from `pixie`(1) or from pc sampling allows you to see how well the load is being balanced so you can compare the different methods of dividing the load. Refer to the discussion of the `MP_SCHEDTYPE` clause in "C$DOACROSS", page 91, for more information.

Even when the load is perfectly balanced, iterations may still take varying amounts of time to finish because of random factors. One process may take a page fault, another may be interrupted to let a different program run, and so on. Because of these unpredictable events, the time spent waiting for all processes to complete can be several hundred cycles, even with near perfect balance.

Reorganizing Common Blocks To Improve Cache Behavior

You can use the `-OPT:reorg_common` option, which reorganizes common blocks to improve the cache behavior of accesses to members of the common block. This option produces consistent results as long as the code follows the standard and array references are made within the bounds of the array. It produces unexpected results if you violate the standard, for example, if you access an array out of its declared bounds.

The option is enabled by default at `-O3` only if all files referencing the common block are compiled at that optimization level. It is disabled if any file with the common block is compiled at either `-O2` and below, `-OPT:reorg_common=OFF`, or `-Wl,-noivpad`.

Advanced Features

A number of features are provided so that sophisticated users can override the multiprocessing defaults and customize the parallelism to their particular applications. This section provides a brief explanation of these features.

mp_block and mp_unblock

`mp_block` puts the slave threads into a blocked state using the system call `blockproc`. The slave threads stay blocked until a call is made to `mp_unblock`. These routines are useful if the job has bursts of parallelism separated by long stretches of single processing, as with an interactive program. You can block the slave processes so they consume CPU cycles only as needed, thus freeing the machine for other users. The Fortran system automatically unblocks the slaves on entering a parallel region if you neglect to do so.

mp_setup, mp_create, and mp_destroy

The `mp_setup`, `mp_create`, and `mp_destroy` subroutine calls create and destroy threads of execution. This can be useful if the job has only one parallel portion or if the parallel parts are widely scattered. When you destroy the extra execution threads, they cannot consume system resources; they must be re-created when needed. Use of these routines is discouraged because they degrade performance; use the `mp_block` and `mp_unblock` routines in almost all cases.

`mp_setup` takes no arguments. It creates the default number of processes as defined by previous calls to `mp_set_numthreads`, by the `MP_SET_NUMTHREADS` environment variable (described in "Environment Variables for Origin Systems ", page 118), or by the number of CPUs on the current hardware platform. `mp_setup` is called automatically when the first parallel loop is entered to initialize the slave threads.

`mp_create` takes a single integer argument, the total number of execution threads desired. Note that the total number of threads includes the master thread. Thus, `mp_create(`*n*`)` creates one thread less than the value of its argument. `mp_destroy` takes no arguments; it destroys all the slave execution threads, leaving the master untouched.

When the slave threads die, they generate a `SIGCLD` signal. If your program has changed the signal handler to catch `SIGCLD`, it must be prepared to deal with this signal when `mp_destroy` is executed. This signal also occurs when the program exits; `mp_destroy` is called as part of normal cleanup when a parallel Fortran job terminates.

mp_blocktime

The Fortran slave threads spin wait until there is work to do. This makes them immediately available when a parallel region is reached. However, this consumes CPU resources. After enough wait time has passed, the slaves block themselves through `blockproc`. Once the slaves are blocked, it requires a system call to `unblockproc` to activate the slaves again (refer to the `unblockproc`(2) reference page for details). This makes the response time much longer when starting up a parallel region.

This trade-off between response time and CPU usage can be adjusted with the `mp_blocktime` call. `mp_blocktime` takes a single integer argument that specifies the number of times to spin before blocking. By default, it is set to 10,000,000; this takes roughly one second. If called with an argument of 0, the slave threads will not block themselves no matter how much time has passed. Explicit calls to `mp_block`, however, will still block the threads.

This automatic blocking is transparent to the user's program; blocked threads are automatically unblocked when a parallel region is reached.

`mp_numthreads`, `mp_set_numthreads`

Occasionally, you may want to know how many execution threads are available. `mp_numthreads` is a zero-argument integer function that returns the total number of execution threads for this job. The count includes the master thread.

In addition, this routine has the side-effect of freezing (for eternity) the number of threads to the returned value, so use this routine sparingly. To determine the number of threads without this freeze property, see the description of `mp_suggested_numthreads` below.

`mp_set_numthreads` takes a single-integer argument. It changes the default number of threads to the specified value. A subsequent call to `mp_setup` will use the specified value rather than the original defaults. If the slave threads have already been created, this call will not change their number. It only has an effect when `mp_setup` is called.

`mp_suggested_numthreads`

The `mp_suggested_numthreads` (integer*4) uses the supplied value as a hint about how many threads to use in subsequent parallel regions, and returns the previous value of the number of threads to be employed in parallel regions. It does not affect currently executing parallel regions, if any. The implementation may ignore this hint depending on factors such as overall system load. This routine returns the previous value of the number of threads being employed at parallel regions. Therefore, to simply query the number of threads, call it with the value 0.

The `mp_suggested_numthreads` interface is available whether or not dynamic threads is turned on (see "Using Dynamic Threads ", page 120).

`mp_my_threadnum`

`mp_my_threadnum` is a zero-argument function that allows a thread to differentiate itself while in a parallel region. If there are *n* execution threads, the function call returns a value between zero and $n - 1$. The master thread is always thread zero. This function can be useful when parallelizing certain kinds of loops. Most of the time the loop index variable can be used for the same purpose. Occasionally, the loop index may not be accessible, as, for example, when an external routine is called from within the parallel loop. This routine provides a mechanism for those cases.

`mp_setlock`, `mp_unsetlock`, `mp_barrier`

`mp_setlock`, `mp_unsetlock`, and `mp_barrier` are zero-argument subroutines that provide convenient (although limited) access to the locking and barrier functions provided by `ussetlock`, `usunsetlock`, and `barrier`. These subroutines are convenient because you do not need to initialize them; calls such as `usconfig` and `usinit` are done automatically. The limitation is that there is only one lock and one barrier. For most programs, this amount is sufficient. If your program requires more complex or flexible locking facilities, use the `ussetlock` family of subroutines directly.

Environment Variables for Origin Systems

The environment variables are described in these subsections:

- "Using the `MP_SET_NUMTHREADS`, `MP_BLOCKTIME`, `MP_SETUP` environment variables", page 119
- "Using Dynamic Threads ", page 120
- "Controlling the Stacksize of Slave Processes", page 120
- "Specifying Page Sizes for Stack, Data, and Text ", page 121
- "Specifying Run-Time Scheduling ", page 121

Using the `MP_SET_NUMTHREADS`, `MP_BLOCKTIME`, `MP_SETUP` environment variables

The `MP_SET_NUMTHREADS`, `MP_BLOCKTIME`, and `MP_SETUP` environment variables act as an implicit call to the corresponding routine(s) of the same name at program start-up time.

For example, the `csh` command

```
% setenv MP_SET_NUMTHREADS 2
```

causes the program to create two threads regardless of the number of CPUs actually on the machine, just like the source statement

```
CALL MP_SET_NUMTHREADS (2)
```

Similarly, the following `sh` commands

```
% set MP_BLOCKTIME=0
% export MP_BLOCKTIME
```

prevent the slave threads from autoblocking, just like the source statement

```
call mp_blocktime (0)
```

For compatibility with older releases, the environment variable `NUM_THREADS` is supported as a synonym for `MP_SET_NUMTHREADS`.

To help support networks with multiple multiprocessors and multiple CPUs, the environment variable `MP_SET_NUMTHREADS` also accepts an expression involving integers `+`, `–`, `min`, `max`, and the special symbol `all`, which stands for "the number of CPUs on the current machine." For example, the following command selects the number of threads to be two fewer than the total number of CPUs (but always at least one):

```
% setenv MP_SET_NUMTHREADS 'max(1,all-2)'
```

Setting the `_DSM_WAIT` Environment Variable

This variable controls how a thread waits for a synchronization event, such as a lock or a barrier. If this variable is set to YIELD, a waiting thread spins for a while and then invokes `sginap(0)`, surrendering the CPU to another waiting process (if any). If set to SPIN, a waiting thread simply busy-waits in a loop until the synchronization event succeeds. The default value is YIELD.

Using Dynamic Threads

In an environment with long running jobs and varying workloads, you may want to vary the number of threads during execution of some jobs.

Setting MP_SUGNUMTHD causes the run-time 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, it decreases the number of threads, possibly to as few as 1. When MP_SUGNUMTHD has no value, this feature is disabled and multithreading works as before.

Note: The number of threads being used is adjusted only at the **start** of a parallel region (for example, a doacross), and not within a parallel region.

In the past, the number of threads utilized during execution of a multiprocessor job was generally constant, set for example, using MP_SET_NUMTHREADS.

The environment variables MP_SUGNUMTHD_MIN and MP_SUGNUMTHD_MAX are used to limit this feature as desired. When MP_SUGNUMTHD_MIN is set to an integer value between 1 and MP_SET_NUMTHREADS, the process will not decrease the number of threads below that value.

When MP_SUGNUMTHD_MAX is set to an integer value between the minimum number of threads and MP_SET_NUMTHREADS, the process will not increase the number of threads above that value.

If you set any value in the environment variable MP_SUGNUMTHD_VERBOSE, informational messages are written to stderr whenever the process changes the number of threads in use.

Calls to mp_numthreads and mp_set_numthreads are taken as a sign that the application depends on the number of threads in use. The number in use is frozen upon either of these calls; and if MP_SUGNUMTHD_VERBOSE is set, a message to that effect is written to stderr.

Controlling the Stacksize of Slave Processes

Use the environment variable, MP_SLAVE_STACKSIZE, to control the stacksize of slave processes. Set this variable to the desired stacksize in bytes. The default value is 16 MB (4 MB for greater than 64 threads). Note that slave processes only allocate their local data onto their stack; shared data (even if allocated on the master's stack) is not counted.

Specifying Page Sizes for Stack, Data, and Text

Use the environment variables, PAGESIZE_STACK, PAGESIZE_DATA, and PAGESIZE_TEXT, to specify the desired page size (in KB) for each of stack, data, and text segments.

Specifying Run-Time Scheduling

These environment variables specify the type of scheduling to use on DOACROSS loops that have their scheduling type set to RUNTIME. For example, the following csh commands cause loops with the RUNTIME scheduling type to be executed as interleaved loops with a chunk size of 4:

```
% setenv MP_SCHEDTYPE INTERLEAVE
% setenv CHUNK 4
```

The defaults are the same as on the DOACROSS directive; if neither variable is set, SIMPLE scheduling is assumed. If MP_SCHEDTYPE is set, but CHUNK is not set, a CHUNK of 1 is assumed. If CHUNK is set, but MP_SCHEDTYPE is not, DYNAMIC scheduling is assumed.

Specifying Gang Scheduling

Set MPC_GANG to ON specify gang scheduling. Set to OFF to disable gang scheduling.

Local COMMON Blocks

A special ld option allows named COMMON blocks to be local to a process. Each process in the parallel job gets its own private copy of the common block. This can be helpful in converting certain types of Fortran programs into a parallel form.

The common block must be a named COMMON (blank COMMON may not be made local), and it must not be initialized by DATA statements.

To create a local COMMON block, give the special loader directive -Wl,-Xlocal followed by a list of COMMON block names. Note that the external name of a COMMON block known to the loader has a trailing underscore and is not surrounded by slashes. For example, the command

```
% f77 -mp a.o -Wl,-Xlocal,foo_
```

makes the `COMMON` block `/foo/` a local `COMMON` block in the resulting `a.out` file. You can specify multiple `-Wl,-Xlocal` options if necessary.

It is occasionally desirable to be able to copy values from the master thread's version of the `COMMON` block into the slave thread's version. The special directive `C$COPYIN` allows this. It has the form

`C$COPYIN` *item* `[,`*item*`]...`

Each *item* must be a member of a local `COMMON` block. It can be a variable, an array, an individual element of an array, or the entire `COMMON` block.

Note: The `C$COPYIN` directive cannot be executed from inside a parallel region.

For example,

```
C$COPYIN x,y, /foo/, a(i)
```

propagates the values for `x` and `y`, all the values in the `COMMON` block `foo`, and the `i`th element of array `a`. All these items must be members of local `COMMON` blocks. Note that this directive is translated into executable code, so in this example `i` is evaluated at the time this statement is executed.

Compatibility With `sproc`

The parallelism used in Fortran is implemented using the standard system call `sproc`. It is recommended that programs not attempt to use both `C$DOACROSS` loops and `sproc` calls. It is possible, but there are several restrictions:

- Any threads you create may not execute `$DOACROSS` loops; only the original thread is allowed to do this.
- The calls to routines like `mp_block` and `mp_destroy` apply only to the threads created by `mp_create` or to those automatically created when the Fortran job starts; they have no effect on any user-defined threads.
- Calls to routines such as `m_get_numprocs` do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the routine `kill` with the arguments `0` and `sig` (for example, `kill`(0,`sig`))

to signal all members of the process group might kill threads used to execute C$DOACROSS. If you choose to intercept the SIGCLD signal, you must be prepared to receive this signal when the threads used for the C$DOACROSS loops exit; this occurs when mp_destroy is called or at program termination.

- Note in particular that m_fork is implemented using sproc, so it is not legal to m_fork a family of processes that each subsequently executes C$DOACROSS loops. Only the original thread can execute C$DOACROSS loops.

DOACROSS Implementation

This section discusses how multiprocessing is implemented in a DOACROSS routine. This information is useful when you use a debugger or interpret the results of an execution profile.

Loop Transformation

When the Fortran compiler encounters a C$DOACROSS directive, it spools the body of the corresponding DO loop into a separate subroutine and replaces the loop with a call to a special library routine __mp_parallel_do.

The newly created routine is named by appending .pregion to the name of the original routine, followed by the number of the parallel loop in the routine (where 0 is the first loop). For example, the first parallel loop in a routine named foo is named foo.pregion0, the second parallel loop is foo.pregion1, and so on.

If a loop occurs in the main routine and if that routine has not been given a name by the PROGRAM statement, its name is assumed to be main. Any variables declared to be LOCAL in the original C$DOACROSS statement are declared as local variables in the spooled routine. References to SHARE variables are resolved by referring back to the original routine.

Because the spooled routine is now just a DO loop, the routine uses subroutine arguments to specify which part of the loop a particular process is to execute. The spooled routine has three arguments: the starting value for the index, the number of times to execute the loop, and a special flag word. As an example, the following routine that appears on line 1000:

```
   SUBROUTINE EXAMPLE(A, B, C, N)
   REAL A(*), B(*), C(*)
C$DOACROSS LOCAL(I,X)
```

```
   DO I = 1, N
      X = A(I)*B(I)
      C(I) = X + X**2
   END DO
   C(N) = A(1) + B(2)
   RETURN
   END
```

produces this spooled routine to represent the loop:

```
   SUBROUTINE EXAMPLE.pregion
X ( _LOCAL_START, _LOCAL_NTRIP, _THREADINFO)
   INTEGER*4 _LOCAL_START
   INTEGER*4 _LOCAL_NTRIP
   INTEGER*4 _THREADINFO
   INTEGER*4 I
   REAL X
   INTEGER*4 _DUMMY
   I = _LOCAL_START
   DO _DUMMY = 1,_LOCAL_NTRIP
       X = A(I)*B(I)
       C(I) = X + X**2
   I = I + 1
   END DO
   END
```

Executing Spooled Routines

The set of processes that cooperate to execute the parallel Fortran job are members of a process share group created by the system call `sproc`. The process share group is created by special Fortran start-up routines that are used only when the executable is linked with the `-mp` option, which enables multiprocessing.

The first process is the master process. It executes all the nonparallel portions of the code. The other processes are slave processes; they are controlled by the routine `mp_slave_control`. When they are inactive, they wait in the special routine `__mp_slave_wait_for_work`.

The `__mp_parallel_do` routine divides the work and signals the slaves. The master process then calls the spooled routine to do its share of the work. When a slave is signaled, it wakes up from the wait loop, calculates which iterations of the spooled *x*`DO` loop it is to execute, and then calls the spooled routine with the appropriate arguments. When a slave completes its execution of the spooled routine, it reports that it has finished and returns to `__mp_slave_wait_for_work`.

When the master completes its execution of its portion of the spooled routine, it waits in the special routine `mp_wait_for_loop_completion` until all the slaves have completed processing. The master then returns to the main routine and continues execution.

PCF Directives

In addition to the simple loop-level parallelism offered by the `C$DOACROSS` directive (described in "Parallel Loops", page 90), the compiler supports a more general model of parallelism. This model is based on the work done by the Parallel Computing Forum (PCF), which itself formed the basis for the proposed ANSI-X3H5 standard. The compiler supports this model through compiler directives, rather than extensions to the source language.

The main concept in this model is the `parallel region`, which can be any arbitrary section of code (not just a `DO` loop). Within the parallel region, there are special `work-sharing constructs` that can be used to divide the work among separate processes or threads. The parallel region can also contain a `critical section` construct, where exactly one process executes at a time.

The master thread executes the user program until it reaches a parallel region. It then spawns one or more slave threads that begin executing code at the beginning of a parallel region. Each thread executes all the code in the region until a work sharing construct is encountered. Each thread then executes some portion of the work sharing construct, and then resumes executing the parallel region code. At the end of the parallel region, all the threads synchronize, and the master thread continues execution of the user program.

The PCF directives, summarized in Table B-1, page 126, implement the general model of parallelism. They look like Fortran comments, with a `C` in column one. The compiler recognizes these directives when multiprocessing is enabled with either the `-mp` option. (Multiprocessing is also enabled with the `-apo` option if you have purchased the MIPSpro Auto-Parallelizing Option.) If multiprocessing is not enabled, the compiler treats these statements as comments. Therefore, you can compile identical source with a single-processing compiler or by Fortran without the multiprocessing option. The PCF directives start with the characters `C$PAR`.

Table B-1 Summary of PCF Directives

Directive	Description
`C$PAR BARRIER`	Ensures that each process waits until all processes reach the barrier before proceeding.
`C$PAR [END] CRITICAL SECTION`	Ensures that the enclosed block of code is executed by only one process at a time by using a global lock.
`C$PAR [END] PARALLEL`	Encloses a parallel region, which includes work-sharing constructs and critical sections.
`C$PAR PARALLEL DO`	Precedes a single `DO` loop for which separate iterations are executed by different processes. This directive is equivalent to the `C$DOACROSS` directive.
`C$PAR [END] PDO`	Separate iterations of the enclosed loop are executed by different processes. This directive must be inside a parallel region.
`C$PAR [END] PSECTION[S]`	Parcels out each block of code in turn to a process.
`C$PAR SECTION`	Signifies a starting line for an individual section within a parallel section.
`C$PAR [END] SINGLE PROCESS`	Ensures that the enclosed block of code is executed by exactly one process.
`C$PAR &`	Continues a PCF directive onto multiple lines.

Parallel Region

A parallel region encloses any number of PCF constructs (described in "PCF Constructs", page 127). It signifies the boundary within which slave threads execute. A user program can contain any number of parallel regions. The syntax of the parallel region is

```
C$PAR PARALLEL [clause [[,] clause]...]
            code
C$PAR END PARALLEL
```

where valid clauses are

[IF (*logical_expression*)]
[{LOCAL | PRIVATE}(*item* [,*item*])...]
[{SHARE | SHARED}(*item* [,*item*])...]

The `IF`, `LOCAL`, and `SHARED` clauses have the same meaning as in the `C$DOACROSS` directive (refer to "Writing Parallel Fortran", page 91).

The preferred form of the directive has no commas between the clauses. The `SHARED` clause is preferred over `SHARE` and `LOCAL` is preferred over `PRIVATE`.

In the following code, all threads enter the parallel region and call the routine `foo`:

```
        subroutine ex1(index)
        integer i
C$PAR PARALLEL LOCAL(i)
        i = mp_my_threadnum()
        call foo(i)
C$PAR END PARALLEL
        end
```

PCF Constructs

The three types of PCF constructs are work-sharing constructs, critical sections, and barriers. All master and slave threads synchronize at the bottom of a work-sharing construct. None of the threads continue past the end of the construct until they all have completed execution within that construct.

The four work-sharing constructs are

- parallel `DO`
- `PDO`
- parallel sections
- single process

If specified, the `PDO`, parallel section, and single process constructs must appear inside of a parallel region; the parallel `DO` construct cannot. Specifying a parallel `DO` construct inside of a parallel region produces a syntax error.

The critical section construct protects a block of code with a lock so that it is executed by only one thread at a time. Threads do not synchronize at the bottom of a critical section.

The barrier construct ensures that each process that is executing waits until all others reach the barrier before proceeding.

Parallel DO

The parallel DO construct is the same as the C$DOACROSS directive (described in "C$DOACROSS", page 91) and conceptually the same as a parallel region containing exactly one PDO construct and no other code. Each thread inside the enclosing parallel region executes separate iterations of the loop within the parallel DO construct. The syntax of the parallel DO construct is

```
C$PAR PARALLEL DO [clause [[,] clause]...]
```

"C$DOACROSS", page 91, describes valid values for *clause* with the exception of the MP_SCHEDTYPE=*mode* clause. For the C$PAR PARALLEL DO directive, MP_SCHEDTYPE= is optional; you can just specify *mode*.

PDO

Each thread inside the enclosing parallel region executes a separate iteration of the loop within the PDO construct. The syntax of the PDO construct, which can only be specified within a parallel region, is

```
C$PAR PDO [clause [[,] clause]]...]
    code
[C$PAR END PDO [NOWAIT]]
```

The valid values for *clause* are

```
[{LOCAL | PRIVATE} (item[,item])...]
[{LASTLOCAL | LAST LOCAL} (item[,item])...]
[(ORDERED)]
[ sched ]
[ chunk ]
```

LOCAL, LASTLOCAL, s*ched*, and *chunk* have the same meaning as in the C$DOACROSS directive (refer to "Writing Parallel Fortran", page 91). Note in particular that it is legal to declare a data item as LOCAL in a PDO even if it was declared as SHARED in the enclosing parallel region. The (ORDERED) clause is equivalent to a *sched* clause of DYNAMIC and a *chunk* clause of 1. The parenthesis are required.

LASTLOCAL is preferred over LAST LOCAL and LOCAL is preferred over PRIVATE.

The `END PDO` directive is optional. If specified, this directive must appear immediately after the end of the `DO` loop. The optional `NOWAIT` clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify `NOWAIT`, the processes will wait until all have reached the directive before proceeding.

As an example of the PDO construct, consider the following code:

```
        subroutine ex2(a,n)
        real a(n)
C$PAR PARALLEL local(i) shared(a)
C$PAR PDO
        do i = 1, n
          a(i) = a(i) + 1.0
        enddo
C$PAR END PARALLEL
        end
```

This sample code is the same as a `C$DOACROSS` loop. In fact, the compiler recognizes this as a special case and generates the same (more efficient) code as for a `C$DOACROSS` directive.

Parallel Sections

The parallel sections construct is a parallel version of the Fortran 90 `SELECT` statement. Each block of code is parcelled out in turn to a separate thread. The syntax of the parallel sections construct is

`C$PAR PSECTION[S]` [*clause* [[,]*clause*]]...
    code
[`C$PAR SECTION`
    code] ...
`C$PAR END PSECTION[S] [NOWAIT]`

where the only valid value for *clause* is

`[{LOCAL | PRIVATE}` (*item* [,*item*])...]

`LOCAL` is preferred over `PRIVATE` and has the same meaning as for the `C$DOACROSS` directive (refer to "C$DOACROSS", page 91). Note in particular that it is legal to declare a data item as `LOCAL` in a parallel sections construct even if it was declared as `SHARED` in the enclosing parallel region.

The optional NOWAIT clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify NOWAIT, the processes will wait until all have reached the END PSECTION directive before proceeding.

Parallel sections must appear within a parallel region. They can contain critical section constructs (described in "Critical Section ", page 134) but cannot contain any of the following types of constructs:

- PDO
- parallel DO or C$DOACROSS
- single process

Each code block is executed in parallel (depending on the number of processes available). The code blocks are assigned to threads one at a time, in the order specified. Each code block is executed by only one thread. For example, consider the following code:

```
        subroutine ex3(a,n1,b,n2,c,n3)
        real a(n1), b(n2), c(n3)
C$PAR PARALLEL local(i) shared(a,b,c)
C$PAR PSECTIONS
C$PAR SECTION
        do i = 1, n1
          a(i) = 0.0
        enddo
C$PAR SECTION
        do i = 1, n2
          b(i) = 0.5
        enddo
C$PAR SECTION
        call normalize(c,n3)
        do i = 1, n3
          c(i) = c(i) + 1.0
        enddo
C$PAR END PSECTION
C$PAR END PARALLEL
        end
```

The first thread to enter the parallel sections construct executes the first block, the second thread executes the second block, and so on. This example has only three sections, so if more than three threads are in the parallel region, the fourth and higher threads wait at the `C$PAR END PSECTION` directive until all threads are finished. If the parallel region is being executed by only two threads, whichever thread finishes its block first continues and executes the remaining block.

This example uses `DO` loops, but a parallel section can be any arbitrary block of code. Be aware of the significant overhead of a parallel construct. Make sure the amount of work performed is enough to outweigh the extra overhead.

The sections within a parallel sections construct are assigned to threads one at a time, from the top down. There is no other implied ordering to the operations within the sections. In particular, a later section cannot depend on the results of an earlier section, unless some form of explicit synchronization is used. If there is such explicit synchronization, you must be sure that the lexical ordering of the blocks is a legal order of execution.

Single Process

The single process construct, which can only be specified within a parallel region, ensures that a block of code is executed by exactly one process. The syntax of the single process construct is

```
C$PAR SINGLE PROCESS [clause [[,] clause]...]
    code
C$PAR END SINGLE PROCESS [NOWAIT]
```

where the only valid value for clause is

```
[{LOCAL | PRIVATE} (item [,item]...)]
```

`LOCAL` is preferred over `PRIVATE` and has the same meaning as for the `C$DOACROSS` directive (refer to "C$DOACROSS", page 91). Note in particular that it is legal to declare a data item as `LOCAL` in a single process construct even if it was declared as `SHARED` in the enclosing parallel region.

The optional `NOWAIT` clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify `NOWAIT`, the processes will wait until all have reached the directive before proceeding.

This construct is semantically equivalent to a parallel sections construct with only one section. The single process construct provides a more descriptive syntax. For example, consider the following code:

```
        real function ex4(a,n, big_max, bmax_x, bmax_y)
        real a(n,n), big_max
        integer bmax_x, bmax_y
C$ volatile big_max, bmax_x, bmax_y
C$ volatile cur_max, index_x, index_y
        index_x = 0
        index_y = 0
        cur_max = 0.0
C$PAR PARALLEL local(i,j)
C$PAR& shared(a,n,index_x,index_y,cur_max,
C$PAR& big_max,bmax_x,bmax_y)
C$PAR PDO
        do j = 1, n
          do i = 1, n
            if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
              if (a(i,j) .gt. cur_max) then
                index_x = i
                index_y = j
                cur_max = a(i,j)
              endif
C$PAR END CRITICAL SECTION
            endif
          enddo
        enddo
C$PAR SINGLE PROCESS
        if (cur_max .gt. big_max) then
          big_max = (big_max + cur_max) / 2.0
          bmax_x = index_x
          bmax_y = index_y
        endif
C$PAR END SINGLE PROCESS
C$PAR PDO
        do j = 1, n
          do i = 1, n
            a(i,j) = a(i,j)/big_max
          enddo
        enddo
C$PAR END PARALLEL
        ex4 = cur_max
        end
```

The first thread to reach the single process section executes the code in that block. All other threads wait at the end of the block until the code has been executed.

This example contains a number of interesting points to be examined. First, note the use of the `VOLATILE` declaration. Any data item that might be written by one thread and then read by a different thread must be marked as `VOLATILE`. Making a variable `VOLATILE` can reduce opportunities for optimization, so the declarations are prefixed by `C$` to prevent the single-processor version of the code from being penalized. Refer to the *MIPSpro Fortran 77 Language Reference Manual*, for more information about the `VOLATILE` statement. Also see "Synchronization Intrinsics", page 140.

Second, note the use of the odd looking repetition of the `IF` test in the first parallel loop:

```
           if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
             if (a(i,j) .gt. cur_max) then
```

This practice is usually called `test&test&set`. It is a multi-processing optimization. Note that the following straight forward code segment is incorrect:

```
        do i = 1, n
          if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
                index_x = i
                index_y = j
                cur_max = a(i,j)
C$PAR END CRITICAL SECTION
          endif
        enddo
```

Because many threads execute the loop in parallel, there is no guarantee that once inside the critical section, `cur_max` still has the same value it did in the `IF` test outside the critical section (some other thread may have updated it). In particular, `cur_max` may now have a value that is larger than `a(i,j)`. Therefore, the critical section must be locked before testing the value of `cur_max`. Changing the previous code into the equally straightforward

```
            do i = 1, n
C$PAR CRITICAL SECTION
                if (a(i,j) .gt. cur_max) then
                  index_x = i
                  index_y = j
                cur_max = a(i,j)
              endif
C$PAR END CRITICAL SECTION
         enddo
```

works correctly, but suffers from a serious performance penalty: the critical section lock must be acquired and released (an expensive operation) for each element of the array. Because the values are rarely updated, this process involves a lot of wasted effort. It is almost certainly slower than just executing the loop serially.

Combining the two methods, as in the original example, produces code that is both fast and correct. If the `IF` test outside of the critical section fails, you can be certain that the values will not be updated, and can proceed. You can expect that the outside `IF` test will account for the majority of cases. If the outer `IF` test passes, then the values `might` be updated, but you cannot always be certain. To ensure correctness, you must perform the test again after acquiring the critical section lock.

You can prefix one of the two identical `IF` tests with `C$` to reduce overhead in the non-multiprocessed case.

Lastly, note the difference between the single process and critical section constructs. If several processes arrive at a critical section construct, they execute the code one at a time. However, they will `all` execute the code. If several processes arrive at a single process construct, only one process executes the code. The other processes bypass the code and wait at the end of the construct for the chosen process to finish.

Critical Section

The critical section construct restricts execution of a block of code so that only one process can execute it at a time. Another process attempting to gain entry to the critical section must wait until the previous process has exited.

The critical section construct can appear anywhere in a program, including inside and outside a parallel region and within a `C$DOACROSS` loop. The syntax of the critical section construct is

```
C$PAR CRITICAL SECTION [ ( lock_variable ) ]
        code
C$PAR END CRITICAL SECTION
```

The *lock_variable* is an optional integer variable that must be initialized to zero. The parenthesis are required. If you do not specify *lock_variable*, the compiler automatically supplies a global lock. Multiple critical section constructs inside the same parallel region are considered to be independent of each other unless they use the same explicit *lock_variable*.

Consider the following code:

```
        integer function num_exceptions(a,n,biggest_allowed)
        double precision a(n,n,n), biggest_allowed
         integer count
        integer lock_var
        volatile count
        unt = 0
          lock_var = 0
C$PAR PARALLEL  local(i,j,k) shared(count,lock_var)
C$PAR PDO
        do 10 k = 1,n
          do 10 j = 1,n
            do 10 i = 1,n
              if (a(i,j,k) .gt. biggest_allowed) then

C$PAR CRITICAL SECTION (lock_var)
                count = count + 1
C$PAR END CRITICAL SECTION (lock_var)
              else
                call transform(a(i,j,k))
                if (a(i,j,k) .gt. biggest_allowed) then
C$PAR CRITICAL SECTION (lock_var)
                  count = count + 1
C$PAR END CRITICAL SECTION (lock_var)
                endif
             endif
 10    continue
C$PAR END PARALLEL
       num_exceptions = count
       return
       end
```

This example demonstrates the use of the lock variable (`lock_var`). A `C$PAR CRITICAL SECTION` directive ensures that no more than one process executes the enclosed block of code at a time. However, if there are multiple critical sections, different processes can be in different critical sections at the same time. This example does not allow different processes to be in different critical sections at the same time because both critical sections control access to the same variable (`count`). Specifying the same lock variable for both critical sections ensures that no more than one process is executing either of the critical sections that use that lock variable. Note that the `lock_var` must be `SHARED` (so that all processes use the same lock), and that `count` must be `volatile` (because other processes might change its value). Refer to "Synchronization Intrinsics", page 140.

Barrier Constructs

A barrier construct ensures that each process waits until all processes reach the barrier before proceeding. The syntax of the barrier construct is

```
C$PAR BARRIER
```

C$PAR &

Occasionally, the clauses in PCF directives are longer than one line. You can use the `C$PAR &` directive to continue a directive onto multiple lines.

For example,

```
C$PAR PARALLEL local(i,j)
C$PAR& shared(a,n,index_x,index_y,cur_max,
C$PAR& big_max,bmax_x,bmax_y)
```

Restrictions

The three work-sharing constructs, `PDO`, `PSECTION`, and `SINGLE PROCESS`, must be executed by all the threads executing in the parallel region (or none of the threads). The following is illegal:

```
       .
       .
       .
C$PAR PARALLEL
        if (mp_my_threadnum() .gt. 5) then
C$PAR SINGLE PROCESS
            many_processes = .true.
C$PAR END SINGLE PROCESS
        endif
         .
         .
         .
```

This code will hang forever when run with enough processes. One or more process will be stuck at the `C$PAR END SINGLE PROCESS` directive waiting for all the threads to arrive. Because some of the threads never took the appropriate branch, they will never encounter the construct. However, the following kind of simple looping is supported:

```
     code
C$PAR PARALLEL local(i,j) shared(a)
        do i= 1,n
C$PAR PDO
          do j = 2,n
     code
```

The distinction here is that all of the threads encounter the work-sharing construct, they all complete it, and they all loop around and encounter it again.

Note that this restriction does not apply to the critical section construct, which operates on one thread at a time without regard to any other threads.

Parallel regions cannot be lexically nested inside of other parallel regions, nor can work-sharing constructs be nested. However, as an aid to writing library code, you can call an external routine that contains a parallel region even from within a parallel region. In this case, only the first region is actually run in parallel. Therefore, you can create a parallelized routine without accounting for whether it will be called from within an already parallelized routine.

Effects on timing

The more general PCF constructs are typically slower than the special case parallelism offered by the `C$DOACROSS` directive. They are slower because of the extra synchronization required. When a `C$DOACROSS` loop executes, there is a synchronization point at entry and another at exit. When a parallel region executes, there is a synchronization point at entry to the region, another at each entry to a work-sharing construct, another at each exit from a work-sharing construct, and one at exit from the region. Thus, several separate `C$DOACROSS` loops typically execute faster than a single parallel region with several PDO constructs. Limit your use of the parallel region construct to those few cases that actually need it.

Communicating Between Threads Through Thread Local Data

The routines described below allow you to perform explicit communication between threads within their MP Fortran program. These communication mechanisms are similar to message-passing, one-sided-communication, or `shmem`, and may be desirable for reasons of performance and/or style.

The operations allow a thread to fetch from (get) or send to (put) data belonging to other threads. Therefore these operations can be performed only on data that has been declared to be `-Xlocal` (that is, each thread has its own private copy of that data; see the `ld`(1) reference page for details on `Xlocal`), the equivalent of the Cray TASKCOMMON directive. A *get operation* requires that source point to Xlocal data, while a *put operation* requires that target point to Xlocal data.

The routines are similar to the original `shmem` routines (see the `shmem` reference page), but are prefixed by `mp_`.

Routines are listed below.

```
mp_shmem_get32  (integer*4 target,
                 integer*4 source,
                 integer*4 length,
                 integer*4 source_thread)

mp_shmem_put32  (integer*4 target,
                 integer*4 source,
                 integer*4 length,
                 integer*4 target_thread)

mp_shmem_iget32 (integer*4 target,
                 integer*4 source,
                 integer*4 target_inc,
                 integer*4 source_inc,
                 integer*4 length,
                 integer*4 source_thread)

mp_shmem_iput32 (integer*4 target,
                 integer*4 source,
                 integer*4 target_inc,
                 integer*4 source_inc,
                 integer*4 length,
                 integer*4 target_thread)
```

```
mp_shmem_get64  (integer*8 target,
                 integer*8 source,
                 integer*4 length,
                 integer*4 source_thread)

mp_shmem_put64  (integer*8 target,
                 integer*8 source,
                 integer*4 length,
                 integer*4 target_thread)

mp_shmem_iget64 (integer*8 target,
                 integer*8 source,
                 integer*4 target_inc,
                 integer*4 source_inc,
                 integer*4 length,
                 integer*4 source_thread)

mp_shmem_iput64 (integer*8 target,
                 integer*8 source,
                 integer*4 target_inc,
                 integer*4 source_inc,
                 integer*4 length,
                 integer*4 target_thread)
```

For the routines listed above:

- Both source and target are pointers to 32-bit quantities for the 32-bit versions, and to 64-bit quantities for the 64-bit versions of the calls. The actual type of the data is not important, since the routines perform a bit-wise copy.
- For a put operation, the target must be Xlocal. For a get operation, the source must be Xlocal.
- Length specifies the number of elements to be copied, in units of 32/64-bit elements, as appropriate.
- *Source_thread/target_thread* specify the thread-number of the remote PE.
- A "get" copies FROM the remote PE, and "put" copies TO the remote PE.
- *Target_inc/source_inc* are specified for the strided iget/iput operations. They specify the "increment" (in units of 32/64 bit elements) along each of source and target

when performing the data transfer. The number of elements copied during a strided put/get operation is still determined by “length.”

Call these routines only after the threads have been created (typically, the first doacross/parallel region). Performing these operations while the program is still serial leads to a run-time error since each thread’s copy has not yet been created.

In the example below, compiling with `-Wl,-Xlocal,mycommon_` ensures that each thread has a private copy of `x` and `y`.

```
integer x
real*8 y(100)
common /mycommon/ x, y
```

The following example copies the value of `x` on thread 3 into the private copy of `x` for the current thread.

```
call mp_shmem_get32 (x, x, 1, 3)
```

The next example copies the value of `localvar` into the thread-5 copy of `x`.

```
call mp_shmem_put32 (x, localvar, 1, 5)
```

The example below fetches values from the thread-7 copy of array `y` into `localarray`.

```
call mp_shmem_get64 (localarray, y, 100, 7)
```

The next example copies the value of every other element of `localarray` into the thread-9 copy of `y`.

```
call mp_shmem_iput64 (y, localarray, 2, 2, 50, 9)
```

Synchronization Intrinsics

The intrinsics described in this section provide a variety of primitive synchronization operations. Besides performing the particular synchronization operation, each of these intrinsics has two key properties:

- The function performed is guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load-linked and/or store-conditional instructions in a loop).

- Associated with each instrinsic are certain *memory barrier* properties that restrict the movement of memory references to *visible data* across the intrinsic operation (by either the compiler or the processor).

  A *visible memory reference* is a reference to a data object potentially accessible by another thread executing in the same shared address space. A visible data object can be one of the following types:

  – Fortran `COMMON` data

  – data declared extern

  – volatile data

  – static data (either file-scope or function-scope)

  – data accessible via function parameters

  – automatic data (local-scope) that has had its address taken and assigned to some object that is visible (recursively)

  The memory barrier semantics of an intrinsic can be one of the following types:

  – `acquire barrier`, which disallows the movement of memory references to visible data from after the intrinsic (in program order) to before the intrinsic (this behavior is desirable at lock-acquire operations)

  – `release barrier`, which disallows the movement of memory references to visible data from before the intrinsic (in program order) to after the intrinsic (this behavior is desirable at lock-release operations)

  – `full barrier`, which disallows the movement of memory references to visible data past the intrinsic (in either direction), and is thus both an acquire and a release barrier. A barrier only restricts the movement of memory references to visible data across the intrinsic operation: between synchronization operations (or in their absence), memory references to visible data may be freely reordered subject to the usual data-dependence constraints.

Caution: Conditional execution of a synchronization intrinsic (such as within an `if` or a `while` statement) does not prevent the movement of memory references to visible data past the overall `if` or `while` construct.

Synopsis

```
integer*4 i4, j4, k4, jj4
integer*8 i8, j8, k8, jj8
logical*4 l4
logical*8 l8
```

Atomic fetch-and-op Operations

```
i4 = fetch_and_add (j4, k4)
i8 = fetch_and_add (j8, k8)
i4 = fetch_and_sub (j4, k4)
i8 = fetch_and_sub (j8, k8)
i4 = fetch_and_or (j4, k4)
i8 = fetch_and_or (j8, k8)
i4 = fetch_and_and (j4, k4)
i8 = fetch_and_and (j8, k8)
i4 = fetch_and_xor (j4, k4)
i8 = fetch_and_xor (j8, k8)
i4 = fetch_and_nand (j4, k4)
i8 = fetch_and_nand (j8, k8)
```

Behavior:

1. Atomically performs the specified operation with the given value on `j4`, and returns the old value of `j4`.

   ```
   { tmp = j4;
   j4 = j4 <op> k4;
   return tmp;
   }
   ```

2. Full barrier.

Atomic op-and-fetch Operations

```
i4 = add_and_fetch (j4, k4)
i8 = add_and_fetch (j8, k8)
i4 = sub_and_fetch (j4, k4)
i8 = sub_and_fetch (j8, k8)
i4 = or_and_fetch (j4, k4)
```

```
i8 = or_and_fetch (j8, k8)
i4 = and_and_fetch (j4, k4)
i8 = and_and_fetch (j8, k8)
i4 = xor_and_fetch (j4, k4)
i8 = xor_and_fetch (j8, k8)
i4 = nand_and_fetch (j4, k4)
i8 = nand_and_fetch (j8, k8)
```

Behavior:

1. Atomically performs the specified operation with the given value on `j4`, and returns the new value of `j4`.

   ```
   { j4 op = k4;
   return j4;
   }
   ```

2. Full barrier.

Atomic BOOL Operation

```
l4 = compare_and_swap( j4, k4, jj4)
l8 = compare_and_swap( j8, k8, jj8)
```

Behavior:

1. Atomically do the following: compare `j4` to old value. If equal, store the new value and return 1, otherwise return 0.

   ```
   if (j4 .ne. oldvalue) return 0;
   else {
       j4 = newvalue
       return 1;
   }
   ```

2. Full barrier.

Atomic synchronize Operation

```
call synchronize
```

Behavior:

1. Full barrier.

Atomic lock and unlock Operations

```
i4 = lock_test_and_set (j4 , k4)
i8 = lock_test_and_set (j8 , k8)
```

Behavior:

1. Atomically store the supplied value in `j4` and return the old value of `j4`.

   ```
   { tmp = j4;
   j4 = k4;
   return tmp;
   }
   ```

2. Acquire barrier.

   ```
   call lock_release(i4)
   call lock_release(i8)
   ```

Behavior:

1. Set j4 to 0.

   ```
   { j4 = 0 }
   ```

2. Release barrier.

Example of Implementing a Pure Spin-Wait Lock

The following example shows implementation of a spin-wait lock.

```
integer*4 lockvar
lockvar = 0
DO WHILE (lock_test_and_set (lockvar, 1) .ne. 0) /* acquire lock */
end do
    ... read and update shared variables ...
call lock_release (lockvar)                  /* release lock */
```

The memory barrier semantics of the intrinsics guarantee that no memory reference to visible data is moved out of the above critical section, either before of the lock-acquire or after the lock-release.

Note: Pure spin-wait locks can perform poorly under heavy contention.

The Auto-Parallelizing Option (APO)

The Auto-Parallelizing Option is a compiler extension controlled with options in the command line that invokes the MIPSpro auto-parallelizing compilers. It is an optional software product for programs written for the N32 and N64 application binary interfaces (see the `ABI`(5) man page for information on the N32 and N64 ABIs). 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 SGI system that uses a MIPS processor.

Note: APO is licensed and sold separately from the compiler. APO features in your code are ignored unless you are licensed for this product. For sales and licensing information, contact your SGI sales representative.

The MIPSpro APO is an extension integrated into the compiler; it is not a source-to-source preprocessor as was used prior to the MIPSpro 7.2 release. If the Auto-Parallelizing Option is installed, the compiler is considered a *auto-parallelizing compiler* and is referred to as the MIPSpro Auto-Parallelizing Fortran 77 compiler.

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.

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). 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 C-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.

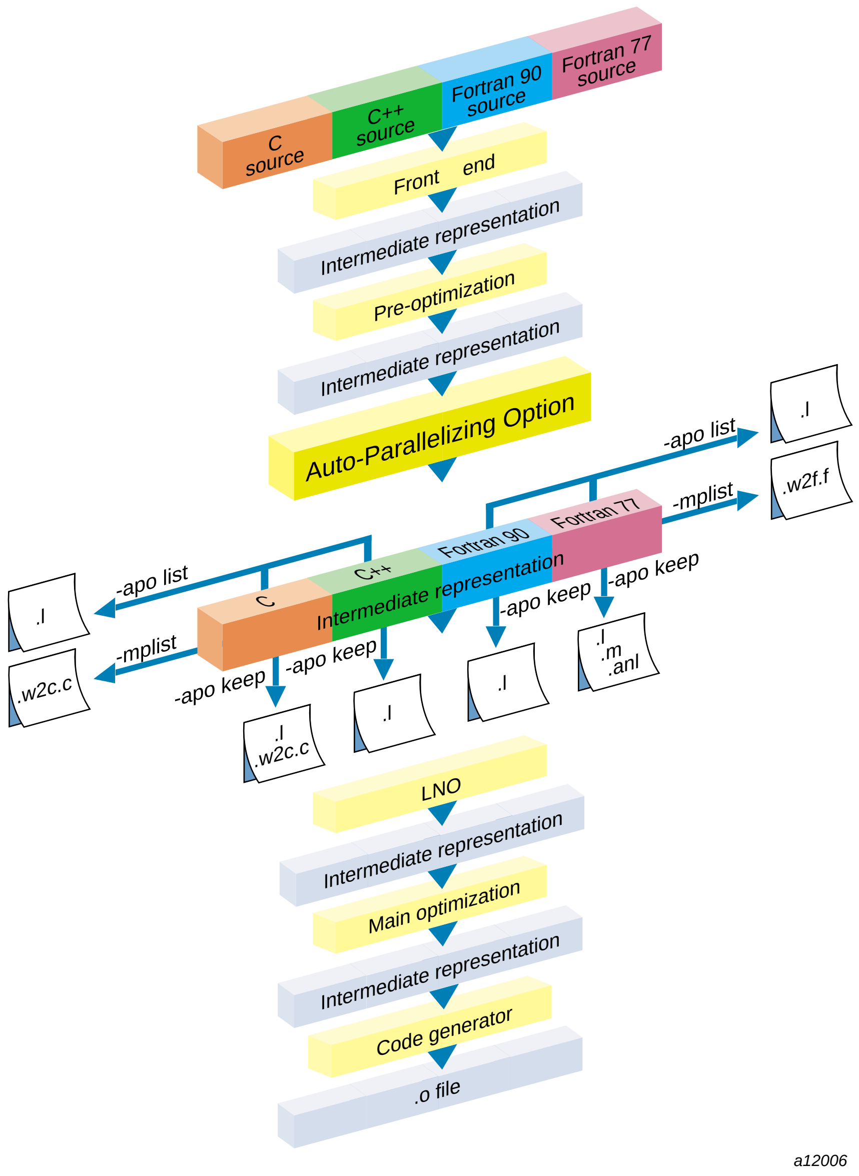

Figure C-1 Files Generated by the MIPSpro Auto-Parallelizing Option

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 APO

You invoke the Auto-Parallelizing Option by including the `-apo` flag with `-n32` or `-64` compiles, on the compiler command line. When using the `-o32` option, the `-apo` option invokes Power Fortran. Additional flags allow you to generate reports to aid in debugging. The syntax for compiling programs with the MIPSpro APO is as follows:

`f77` *options* `-apo` *apo_options* `-mplist` *filename*

The auto-parallelizing compilers may also be invoked using the `-pca` flags (for C) or `-pfa` (for Fortran). **These options are provided for backward compatibility and their use is not recommended.**

The following arguments are used with the compiler command line:

options	The MIPSpro Fortran 77 compiler command-line options. The `-O3` optimization option is recommended for using the APO. See the `f77`(1) man page for details about these options.
`-apo`	Invoke the Auto-Parallelizing Option.
apo_options	*apo_options* can be one of the following values:
	`list`: Invoke the MIPSpro APO and produce a listing of those parts of the program that can run in parallel and those that cannot. The listing is stored in a `.list` file.
	`keep`: Invoke the MIPSpro APO and generate `.list`, `.w2f.f` , `.m`, and `.anl` files. Because of data conflicts, do not use with `-mplist` or the LNO options, `-FLIST`, and `-CLIST`. See "Output files", page 154, for details about all output files.
`-mplist`	Generate the equivalent parallelized program for Fortran 77 in a `.w2f.f` file. These files are discussed in the section "The `.w2f.f` File", page 156. Do not use with `-apo keep`, `-FLIST`, or `-CLIST`.

filename	The name of the file containing the source code.

Starting with the 7.2.1 release of the MIPSpro compilers, the `-apo keep` and `-mplist` options cause Auto-Parallelizing Fortran 77 to generate `.m` and `w2f.f` files based on OpenMP directives.

The following is a typical compiler command line:

```
f77 -apo -O3 -n32 -mips4 -c -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`. The `-n32` option requests an object with an N32 ABI; `-mips4` requests that the code be generated with the MIPS IV instruction set. 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 produce 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` option
  - the `-mp` option
- Because of data set conflicts, you can use only one of the following in a compilation:
  - `-apo keep`
  - `-mplist`
  - `-FLIST` or `-CLIST`

Common 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 options. 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.

The following sections discuss the compiler command-line options most commonly needed with the Auto-Parallelizing Option.

Optimization Options

The `-O3` optimization option 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 requires relaxing of the language rules. The `-O3` option uses transformations that are usually beneficial but can sometimes hurt performance. This optimization may cause noticeable changes in floating-point results due to the relaxation of operation-ordering rules. Floating-point optimization is discussed further in "Other Optimization Options", page 154.

Interprocedural Analysis

Interprocedural analysis (IPA) is invoked by the `-IPA` command-line option. It performs program optimizations that can only be done by examining the whole program, rather than processing each procedure separately. The following are typical IPA optimizations:

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

As of the MIPSpro 7.2.1 release, the Auto-Parallelizing Option with IPA is able to optimize only those loops whose function calls are determined to be “safe” to be parallelized.

If IPA expands subroutines inlined 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. The following are some of the optimizations of the LNO:

- 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, not a preprocessor. There are 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.
- -LNO:ignore_pragmas. This option causes the MIPSpro APO to ignore all of the directives and assertions discussed in "Compiler Directives for Automatic Parallelization", page 167. This includes the C*$* NO CONCURRENTIZE directive.

You can view the transformed code in the original source language after the LNO performs its transformations. Two translators, integrated into the compiler, convert the compiler's internal representation into the original source language. You can invoke the desired translator by using the f77 -FLIST:=on or -flist option (these are equivalent commands). For example, the following command creates an a.out object file and the Fortran file test.w2f.f:

```
f77 -O3 -FLIST:=on test.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 "The `.w2f.f` File", page 156). 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.

Other Optimization Options

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.

With 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 `-OPT;roundoff=1` command-line option. In this example, at the default setting of `-OPT:roundoff=2` for the `-O3` level of optimization, the MIPSpro APO parallelizes this loop.

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

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.

Output files

The MIPSpro APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. With these listings, you may be able to identify small problems that prevent a loop from being made paralle; then you can often remove these data dependences, dramatically improving the program's performance.

When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. To determine where the program spends its execution time, you can use tools such as SpeedShop and the WorkShop Pro MPF Parallel Analyzer View described in "About the `.m` and `.anl` Files", page 158.

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. OpenMP is a collection of directives, library routines, and environment variables and 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 SGI platforms, OpenMP replaces the older Parallel Computing Forum (PCF) and SGI `DOACROSS` directives for Fortran. .

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.

The `.list` File

The `-apo list` and `-apo keep` options generate files that list the original loops in the program along with messages indicating if the loops were parallelized. For loops that were not parallelized, an explanation is given.

Example 6-1 shows a simple Fortran 77 program. The subroutine is contained in a file named `testl.f`.

Example 6-1 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 the following command, the APO produces the file `testl.list`, shown in Example 6-2.

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

Example 6-2 Listing in File `testl.list`

```
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`.

The `.w2f.f` File

The `.w2f.f` file contains code that mimics the behavior of programs after they undergo automatic parallelization. The representation is designed to be readable so that you can see what portions of the original code were not parallelized. You can use this information to change the original program.

The compiler creates the `.w2f.f` file by invoking the appropriate translator to turn the compilers' internal representations into Fortran 77. In most cases, the files contain valid code that can be recompiled, although compiling a `.w2f.f` 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` source file to generate the object file.

The `-flist` 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 more easily understood than the `.w2f.f` file generated using the `-FLIST:=on` option (see "Loop Nest Optimizer Options", page 153). By default, the parallelized program in the `.w2f.f` file uses OpenMP directives.

Consider the subroutine in Example 6-3, contained in a file named `testw2.f`.

Example 6-3 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 the following command, you get an object file, `testw2.o`, and a file, `testw2.w2f.f`, that contains the code shown in Example 6-4.

```
f77 -O3 -n32 -mips4 -c -flist testw2.f
```

Example 6-4 Listing of 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 "The `.list` File", page 155, 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.

About the `.m` and `.anl` Files

The `f77 -apo keep` option generates two files in addition to the `.list` file:

- A `.m` file, which is similar to the `.w2f.f` file. It is based on OpenMP and mimics the behavior of the program after automatic parallelization. It is also annotated with information that is used by Workshop ProMPF.
- A `.anl` file, which is used by Workshop ProMPF.

SGI 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 *ProDev WorkShop: ProMP User's Guide* and the *ProDev WorkShop: Performance Analyzer User's Guide* for details.

SpeedShop, another SGI 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

You invoke a parallelized version of your program using the same command line as that used to run 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 `OMP_NUM_THREADS` environment variable, which tells the system to use a particular number of processors. The following statement causes the program to create two threads regardless of the number of processors available:

```
setenv OMP_NUM_THREADS 2
```

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

Failing to Parallelize Safe Loops

A program's performance may be severely constrained if the APO cannot recognize that a loop is safe to parallelize. A loop is safe if there is no 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 a loop does not appear safe, it does not parallelize that loop. It also often does not parallelize loops containing any of the following constructs:

- function calls in loops, discussed in "Function Calls in Loops", page 159
- `GO TO` statements in loops, discussed in "`GO TO` Statements in Loops", page 160
- problematic array subscripts, discussed in "Problematic Array Subscripts", page 160
- conditionally assigned local variables, discussed in "Local Variables", page 161

However, in many instances such loops can be automatically parallelized after minor changes. Reviewing your program's `.list` file, described in "The `.list` File", page 155, 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. You can, however, use interprocedural analysis (IPA), specified by the `-IPA` command-line option, to 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", page 152, and the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

You can also direct the MIPSpro APO to ignore the dependences of function calls when analyzing the specified loops by using the `CONCURRENT CALL` directive.

`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 cases where array subscripts are too complicated to permit parallelization:

- **The subscripts are indirect array references.** The MIPSpro APO is not able to analyze indirect array references. The following loop cannot be run safely in parallel if the indirect reference *IB(I)* is equal to the same value for different iterations of *I*:

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

  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`", page 172, is appropriate.

- **The subscripts are unanalyzable.** 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`; or 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
```

- **Unknown information.** 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)`", page 169, lets the MIPSpro APO parallelize this loop. You can also use manual parallelization.

Local Variables

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. In the following example, the array *TMP* is used for local scratch space:

```
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
```

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()` because it is not known which processor's private copy of *T* should be used by `S2()`. If *T* were not conditionally assigned, the processor that executed iteration *N* would be used. 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.

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", page 163.
- It has a small trip count, as discussed in "Small Trip Counts", page 163.
- It exhibits poor data locality, as discussed in "Poor Data Locality", page 164.

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.

The `.list` file, described in "The `.list` File", page 155, tells you which loop in a nest was parallelized. Because of the potential for improved performance, it is useful 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. Loops with small trip counts generally run faster when they are not parallelized. Consider how this 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 use a `C*$* ASSERT DO PREFER CONCURRENT` assertion to indicate that it is better to parallelize the *J* loop, or use manual parallelization.

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 the hierarchy. 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, it is probable 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 6-5 and Example 6-6. Assume that the loops are to be parallelized and that there are *p* processors.

Example 6-5 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 6-5, 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 6-6 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 6-6, the Auto-Parallelizing Option may 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.

Unnecessary Parallelization Overhead

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", page 163, loops that have a small number of iterations run faster sequentially than in parallel. The following are two other cases of unnecessary overhead:

- **unknown trip counts**: If the trip count is not known (and sometimes even if it is), the Auto-Parallelizing Option parallelizes the loop conditionally, generating code for both a parallel and a sequential version. By generating two versions, the MIPSpro APO can avoid running a loop in parallel that may have 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.

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

  You can avoid the overhead incurred by having a sequential and parallel version of the loop by using the `C*$* ASSERT DO PREFER` assertions. 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*$* ASSERT DO (SERIAL)`", page 170 and "`C*$* ASSERT DO PREFER (SERIAL)`", page 173.

Strategies for Assisting Parallelization

There are circumstances that interfere with the Auto-Parallelizing Option’s ability to optimize programs. Problems are sometimes caused by coding practices or the MIPSpro APO may not have enough information to make good parallelization decisions. You can pursue three strategies to address 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 "Failing to Parallelize Safe Loops", page 159 and "Parallelizing the Wrong Loop", page 162.

- The second strategy is to assist the MIPSpro APO with the manual parallelization directives. They are described in the *MIPSpro N32/64 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. This 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", page 167. 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 needing to be certain that all the conditions for parallelization are met.
Negative:	The automatic directives are tips 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 (discussed in the documentation with your C compiler).

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. lists the directives, assertions, and pragmas that the MIPSpro APO recognizes.

Table C-1 Auto-Parallelizing Option Directives and Assertions

Compiler Directive	Meaning and Notes
`C*$* NO CONCURRENTIZE`	Varies with placement. Either do not parallelize any loops in a subroutine, or do not parallelize any loops in a file.
`C*$* CONCURRENTIZE`	Override `C*$* NO CONCURRENTIZE`.
`C*$* ASSERT DO (CONCURRENT)`	Do not let perceived dependences between two references to the same array inhibit parallelizing. Does not require `-apo`.
`C*$* ASSERT DO (SERIAL)`	Do not parallelize the following loop.
`C*$* ASSERT CONCURRENT CALL`	Ignore dependences in subroutine calls that would inhibit parallelizing. Does not require `-apo`.
`C*$* ASSERT PERMUTATION` (*array_name*)	Array *array_name* is a permutation array. Does not require `-apo`.
`C*$* ASSERT DO PREFER (CONCURRENT)`	Parallelize the following loop if it is safe.
`C*$* ASSERT DO PREFER (SERIAL)`	Do not parallelize the following loop.

Three compiler directives affect the compiling process even if `-apo` is not specified: `C*$* ASSERT DO (CONCURRENT)` may affect optimizations such as loop interchange; `C*$* ASSERT CONCURRENT CALL` also may affect optimizations such as loop interchange; and `C*$* ASSERT 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 discussed in this section.

C*$* 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

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

C*$* ASSERT DO (CONCURRENT)

C*$* ASSERT DO (CONCURRENT) instructs the MIPSpro APO, when analyzing the loop immediately following this assertion, to ignore all dependences between two references to the same array. If there are real dependences between array references, C*$* 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)
```

Be aware of the following points 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`", page 170.
- The assertion does not affect how the MIPSpro APO analyzes dependences between two potentially 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)

`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

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

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*. 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
```

Note the following points 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)

`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)

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