Fortran 77 Programmer's Guide

Introduction

This manual provides information on implementing Fortran 77 programs using IRIX™ and the IRIS®–4D ™ series workstation. This implementation of Fortran 77 contains full American National Standard (ANSI) Programming Language Institute Fortran (X3.9-1978). 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 and Fortran 66.

Fortran 77 is referred to as "Fortran" throughout this manual except where distinctions between Fortran 77 and Fortran 66 are being specifically discussed.

Corequisite Publications

Refer to the *Fortran 77 Language Reference Manual* for a description of the Fortran language as implemented by the Silicon Graphics® IRIS–4D series workstation.

Refer to the *IRIS–4D Series Compiler Guide* for information on the following topics:

- an overview of the compiler system
- improving program performance by using the profiling and optimization facilities of the compiler system
- the dump utilities, archiver, and other tools used to maintain Fortran programs

Refer to the *dbx Reference Manual* for a detailed description of the debugger. For information on interfaces to programs written in assembly language, refer to the *Assembly Language Programmer's Guide.*

Organization of Information

This manual contains the following chapters and appendix:

- Chapter 1, "Compiling, Linking, and Running Programs," gives an overview of components of the compiler system, and describes how to compile, link edit, and execute a Fortran program. It also describes special considerations for programs running on IRIX systems, such as file format and error handling.
- Chapter 2, "Storage Mapping," describes how the Fortran compiler implements size and value ranges for various data types and how they are mapped to storage. It also describes how to access misaligned data.
- Chapter 3, "Fortran Program Interfaces," provides reference and guide information on writing programs in Fortran, C, and Pascal that can communicate with each other. It also describes the process of generating wrappers for C routines called by Fortran.
- Chapter 4, "System Functions and Subroutines," describes functions and subroutines that can be used with a program to communicate with the IRIX operating system.

- Chapter 5, "Fortran Enhancements for Multiprocessors," describes programming directives for running Fortran programs in a multiprocessor mode.
- Chapter 6, "Compiling and Debugging Parallel Fortran," describes and illustrates compilation and debugging techniques for running Fortran programs in a multiprocessor mode.
- Appendix A, "Run−Time Error Messages,"lists the error messages that can be generated during program execution.

Typographical Conventions

The following conventions and symbols are used in the text to describe the form of Fortran statements:

Bold	Indicates literal command line options, filenames, keywords, function/subroutine names, pathnames, and directory names.
Italics	Represents user−defined values. Replace the item in italics with a legal value. Italics are also used for command names, manual page names, and manual titles.
`Courier`	Indicates command syntax, program listings, computer output, and error messages.
`Courier bold`	Indicates user input.
[ ]	Enclose optional command arguments.
()	Surround arguments or are empty if the function has no arguments following function/subroutine names. Surround manual page section in which the command is described following IRIX commands.
|	Sseparates two or more optional items.
...	Indicates that the preceding optional items can appear more than once in succession.
#	IRIX shell prompt for the superuser.
%	IRIX shell prompt for users other than superuser.

Here are two examples illustrating the syntax conventions.

`DIMENSION` *a*(*d*) [,*a*(*d*)] …

indicates that the Fortran keyword **DIMENSION** must be written as shown, that the user−defined entity*a*(*d*) is required, and that one or more of *a*(*d*) can be optionally specified. Note that the pair of parentheses ( ) enclosing *d* is required.

`{STATIC | AUTOMATIC}` *v* [,*v*] …

indicates that either the **STATIC** or **AUTOMATIC** keyword must be written as shown, that the user−defined entity*v* is required, and that one or more of *v* items can be optionally specified.

Chapter 1

Compiling, Linking, and Running Programs

This chapter contains the following major sections:

- "Compiling and Linking" 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, Pascal, or other languages supported by the compiler system and how to link them into an executable object program.
- "Driver Options" gives an overview of debugging, profiling, optimizing, and other options provided with the Fortran *f77* driver.
- "Object File Tools" briefly summarizes the capabilities of the *odump*, *stdump*, *nm*, *file*, and *size* programs that provide listing and other information on object files.
- "Archiver" summarizes the functions of the *ar* program that maintains archive libraries.
- "Run−Time Considerations"describes how to invoke a Fortran program, how the operating system treats files, and how to handle run−time errors.

Also refer to the *Fortran Release Notes* for a list of compiler enhancements, possible compiler errors, and instructions on how to circumvent them.

Compiling and Linking

Drivers

Programs called *drivers* invoke the major components of the compiler system: the Fortran compiler, the intermediate code optimizer, the code generator, the assembler, and the link editor. The *f77* command runs the driver that causes your programs to be compiled, optimized, assembled, and link edited.

The format of the *f77* driver command is as follows:

`f77` [*option*] … *filename*`.f` [*option*]

where

f77	invokes the various processing phases that compile, optimize, assemble, and link edit the program.
option	represents the driver options through which you provide instructions to the processing phases. They can be anywhere in the command line. These options are discussed later in this chapter.
*filename***.f**	is the name of the file that contains the Fortran source statements. The filename must always have the suffix **.f**, for example, **myprog.f**.

Compilation

The driver command *f77* can both compile and link edit a source module. Figure 1−1shows the primary drivers phases. It also shows their principal inputs and outputs for the source modules **more.f**.

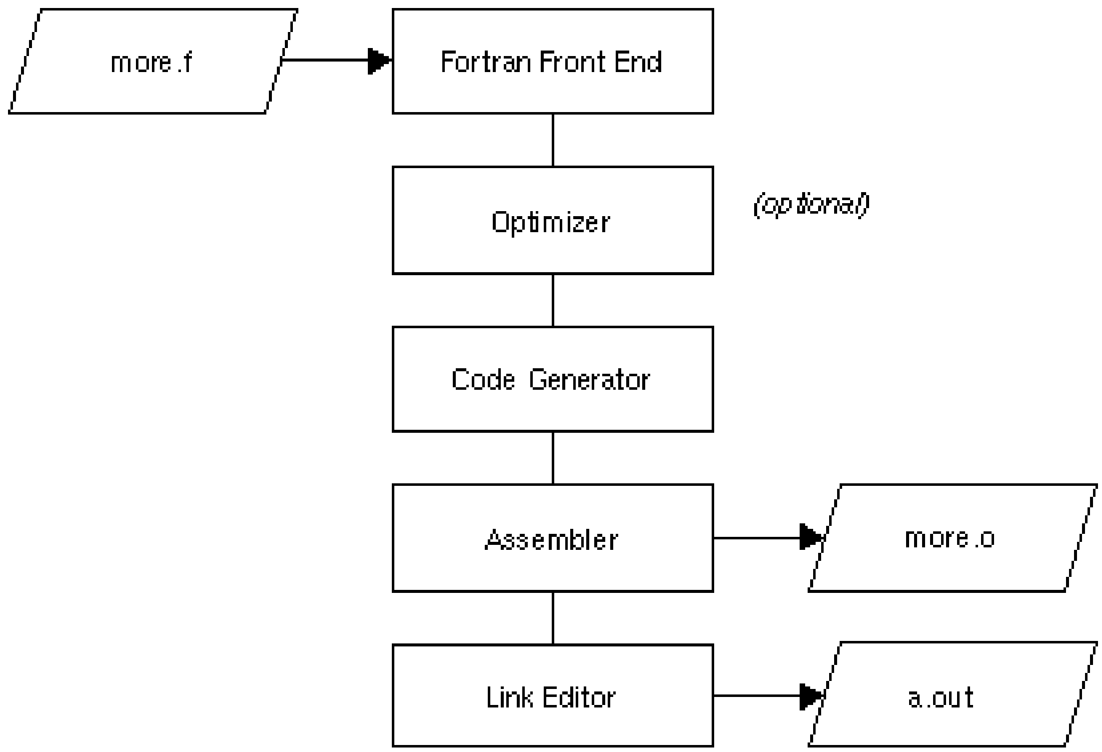

Figure 1–1 Compilation Process

Note the following:

- The source file ends with the required suffixes **.f** or **.F**.
- The source file is passed through the C preprocessor, *cpp*, by default. *cpp* does not accept C–style comments in Hollerith strings. The **-nocpp** option skips the pass through *cpp* and therefore, allows C–style comments in Hollerith strings. (See the **-nocpp** option in "Driver Options" for details.) In the example

  ```
  % f77 myprog.f –nocpp
  ```

  the file **myprog.f** will not be preprocessed by *cpp*.
- The driver produces a linkable object file when you specify the **-c** driver option. This file has the same name as the source file, except with the suffix **.o**. For example, the command line

  ```
  % f77 more.f –c
  ```

  produces the **more.o** file in the above example.
- The default name of the executable object file is **a.out**. For example, the command line

  ```
  % f77 myprog.f
  ```

produces the executable object **a.out**.

- You can specify a name other than **a.out** for the executable object by using the driver option **-o***name*, where *name* is the name of the executable object. For example, the command line

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

  link edits the object module *myprog.o* and produces an executable object named **myprog**.

- The command line

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

  compiles and link edits the source module **myprog.f** and produces an executable object named **myprog**.

Compiling Multilanguage Programs

The compiler system provides drivers for other languages, including C, Pascal, COBOL, and PL/1. 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 *IRIX Series Compiler Guide* for a list of available drivers and the commands that invoke them; refer to Chapter 3 of this manual for conventions you must follow in writing Fortran program interfaces to C and Pascal 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 link editing by specifying the **-c** option, which stops the driver immediately after the assembler phase. For example,

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

The two command lines shown above produce linkable objects named **main.o** and **rest.o**, as illustrated in Figure 1–2

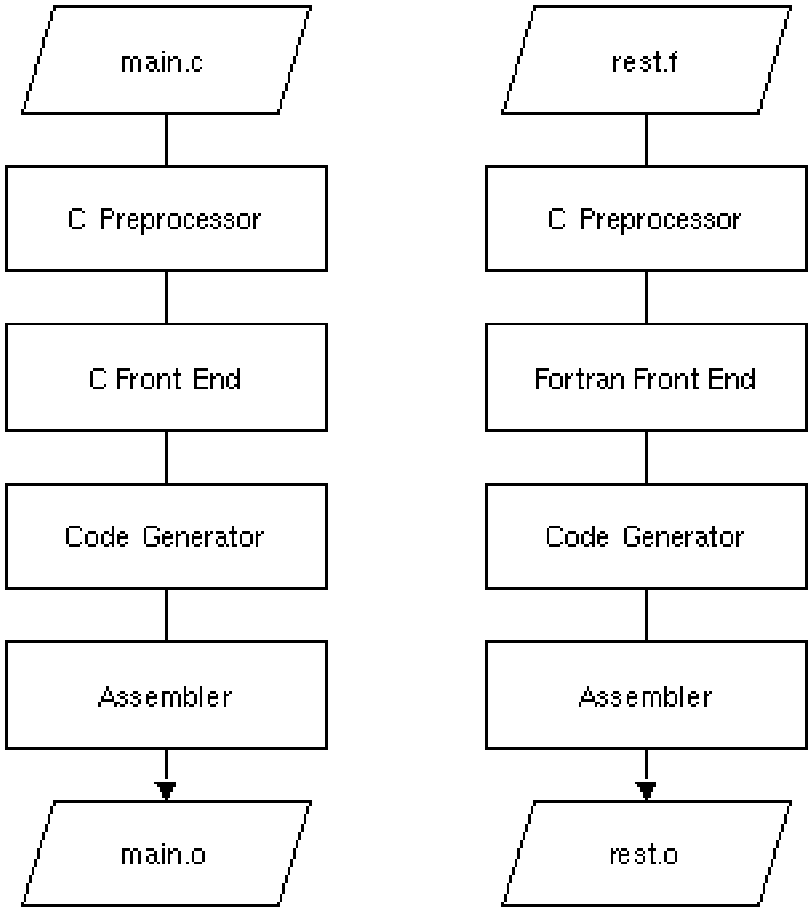

Figure 1–2 Compiling Multilanguage Programs

Linking Objects

You can use the *f77* driver command to link edit separate objects into one executable program when any one of the objects is compiled from a Fortran source. The driver recognizes the **.o** suffix as the name of a file containing object code suitable for link editing and immediately invokes the link editor. The following command link edits the object created in the last example:

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

You can also use the *cc* driver command, as shown below:

```
% cc -o myprog main.o rest.o -lF77 -lU77 -lI77 -lisam -lm
```

Figure 1–3shows the flow of control for this link edit.

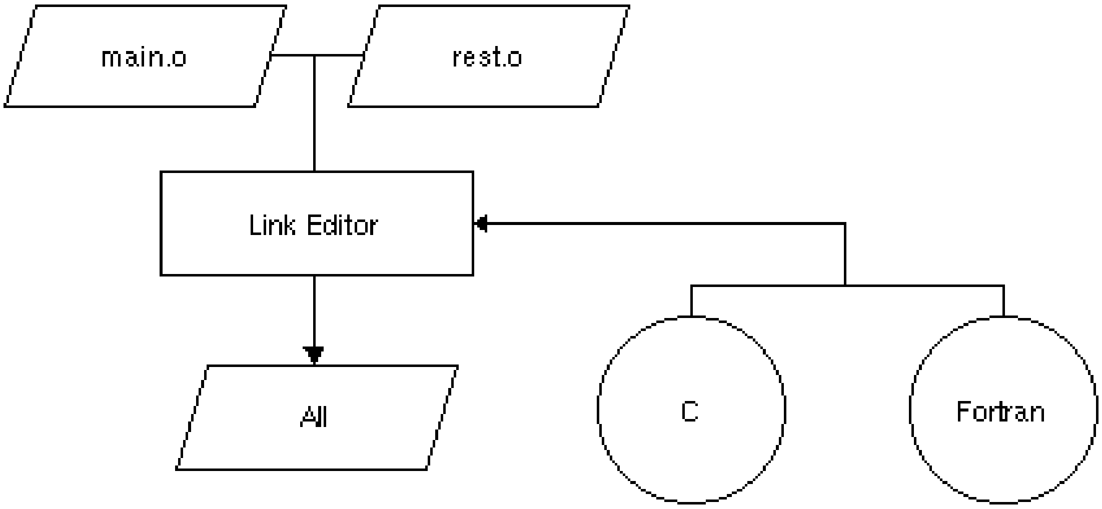

Figure 1–3 Link Editing

Both *f77* and *cc* use the C link library by default. However, the *cc* driver command does not know the names of the link libraries required by the Fortran objects; therefore, you must specify them explicitly to the link editor using the **-l** option as shown in the example. The characters following **-l** are shorthand for link library files as shown in Table 1–1

Table 1–1 Link Libraries

-l	Link Library	Contents
F77	/usr/lib/libF77.a	Fortran intrinsic function library
I77	/usr/lib/libI77.a	Fortran I/O library
I77_mp	/usr/lib/libI77_mp.a	Fortran multiprocessing I/O library
U77	/usr/lib/libU77.a	Fortran IRIX interface library
isam	/usr/lib/libisam.a	Indexed sequential access method library
fgl	/usr/lib/libfgl/a	Fortran graphics library
m	/usr/lib/libm.a	Mathematics library

See the section called "FILES" in the *f77*(1) manual page for a complete list of the files used by the Fortran driver. Also refer to the *ld*(1) manual page for information on specifying the **-l** option.

Specifying Link Libraries

You must explicitly load any required run–time libraries when compiling multilanguage programs. For example, when you link a program written in Fortran and some procedures written in Pascal, you must explicitly load the Pascal library *libp.a* and the math library *libm.a* with the options **-lp** and **-lm** (abbreviations for the libraries *libp.a* and *libm.a*). This procedure is demonstrated in the next example.

```
% f77 main.o more.o rest.o –lp –lm
```

To find the Pascal library, the link editor replaces the **-l** with *lib* and adds an **.a** after **p**. Then, it searches the */lib*, */usr/lib*, and */usr/local/lib* directories for this library. For a list of the libraries that a language uses, see the associated driver manual page, *cc*(1), *f77*(1), or *pc*(1).

You may need to specify libraries when you use IRIX system packages that are not part of a particular language. Most of the manual pages for these packages list the required libraries. For example, the *getwd* (3B) subroutine requires the BSD compatibility library *libbsd.a*. This library is specified 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 link editor searches libraries in the order you specify. Therefore, if you have a library (for example, *libfft.a*) that uses data or procedures from **-lp**, you *must* specify *libfft.a* first.

Driver Options

This section contains a summary of the Fortran-specific driver options. See the *f77*(1) manual page for a complete description of the compiler options; see the *ld*(1) manual page for a description of the link editor options.

-66 Compiles Fortran 66 source programs.

When used at compile time, the following four options generate various degrees of misaligned data in common blocks. They then generate the code to deal with the misalignment.

Note: When specified, these options can degrade program performance; **-align8** causes the greatest degree of degradation, and **-align32** causes the least.

-align8 Aligns objects larger than 8 bits on 8−bit boundaries. Using this option will have the largest impact on performance.

-align16 Aligns objects larger than 16 bits on 16−bit boundaries; 16−bit objects must still be aligned on 16−bit boundaries (MC68000−like alignment rules).

-align32 Aligns objects larger than 32 bits on 32−bit boundaries; 16−bit objects must still be aligned on 16−bit boundaries, and 32−bit objects must still be aligned on 32−bit boundaries.

-align64 Aligns objects larger than 64 bits on 64 bit boundaries. Objects with size 64 bits or smaller must still be aligned on the corresponding boundaries.

The current default alignment is 32 bits. This number may be changed in the future to take advantage of the new 64−bit architecture.

You must specify the appropriate alignment option in the compilation of all modules that reference or define common blocks with misaligned data. Failure to do so could cause core dumps (if the trap handler is not used) or mismatched common blocks.

To load the system libraries capable of handling misaligned data, use the **-L** */usr/lib/align* switch at load time. The trap handler may be needed to handle misaligned data passed to system libraries that are not included in the */usr/lib/align* directory (see *fixade*(3f) and *unalign*(3x)).

-backslash Allows the backslash character to be used as a normal Fortran character instead of the beginning of an escape sequence.

-C Generates code for run−time subscript range checking. The default suppresses range checking. This option will not cause the program to core dump; it will cause the program to exit with a diagnostic message. For details on how to produce a core dump, refer to the information on the **f77_dump_flag** environment variable in Appendix A, "Run−Time Error Messages."

-check_bounds

Causes an error message to be issued at run time when the value of an array subscript expression exceeds the bounds declared for the array. This is equivalent to the **-C** option.

-chunk=*integer*

Has the same effect as putting a **C$CHUNK=***integer* directive at the beginning of the file. See Chapter 5, "Fortran Enhancements for Multiprocessors," and Chapter 6, "Compiling and Debugging Parallel Fortran," for details.

-col72 Sets the source statement format as described in Table 1−2

Table 1−2 Source Statement Settings for −col72 Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

If the source statement contains fewer than 72 characters, no blank padding occurs; the TAB−format facility is disabled.

This option provides the SVS Fortran 72−column option mode.

-col120 Sets the source statement format as described in Table 1−3

Table 1−3 Source Statement Settings for −col120 Option

Column	Contents
1−5	Statement label
6	Continuation indicator
7−120	Statement body
121−end	Ignored

If the source statement contains fewer than 120 characters, no blank padding occurs; the TAB−format facility is disabled.

This option provides the SVS Fortran default mode.

-cpp Runs the C macro preprocessor *cpp* on all source files, including those created by

RATFOR, before compilation. (This option is enabled by default.)

-d_lines Causes any lines with a D in column 1 to be compiled. By default, the compiler treats all lines with a character in column 1 as comment lines.

-expand_include

Expands all include statements in the Fortran source listing file **.L**. This option is only applicable with the **-listing** option.

-extend_source

Sets the source statement format as described in Table 1−4

Table 1−4 Source Statement Settings for −extend_source Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-132	Statement body
133-end	Warning message issued

If the source statement contains fewer than 132 characters, blanks are assumed at the end; the ability of TAB−formatted lines to extend past column 132 is disabled.

This option provides VMS Fortran 132−column mode, except that a warning, instead of a fatal error message, is generated when text extends beyond column 132.

-E Runs only the C macro preprocessor on the files and sends the results to standard output.

-F Calls the RATFOR preprocessor only and puts the output in a **.f** file. Does not produce **.o** files.

-framepointer

Defines the frame pointer register for each subroutine in the source file.

-i2 All small integer constants become **INTEGER*2**. All variables and functions implicitly or explicitly declared type **INTEGER** or **LOGICAL** (without a size designator, that is, **2*, **4*, and so on) will be **INTEGER *2** or **LOGICAL*2**, respectively.

-listing Produces the source listing file with **.L** suffix containing line numbers, error messages, symbol table information, and cross references.

-m If the generic function results do not determine the precision of an integer−valued intrinsic function, the compiler chooses the precisions that return **INTEGER *2**. The default is **INTEGER *4**. Note that **INTEGER *2** and **LOGICAL *2** quantities do not obey the Fortran standard rules for storage location.

−m4 Applies the M4 macro preprocessor to source files to be transformed with RATFOR. The driver puts the result in a **.p** file. Unless you specify the **-K** option, the compiler removes the **.p** file on completion. See the *m4*(1) manual page for details.

-mp Enable the multiprocessing directives. See Chapter 5, "Fortran Enhancements for Multiprocessors," and Chapter 6, "Compiling and Debugging Parallel Fortran," of this book, and the man page on *f77*(1) for further options affecting multiprocessing compilation.

-mp_schedtype=*type*

Has the same effect as putting a **C$MP_SCHEDTYPE=** *type* directive at the beginning of the file. The supported types are **simple**, **interleave**, **dynamic**, **gss**, and **runtime**. See Chapter 5, "Fortran Enhancements for Multiprocessors," and Chapter 6, "Compiling and Debugging Parallel Fortran," of this manual for more details.

-N[qxscnlC]*nnn*

nnn is a decimal number changing the default size of the static tables in the compiler. See the *f77*(1) manual page for details.

-nocpp Does not run the C preprocessor on the source files. Specifying this option allows you to specify C−style comments inside Hollerith strings. Use this option when you want your program to strictly conform to the Fortran 77 standard.

-noexpopt Excludes floating point constant exponent optimization to achieve the same precision as releases prior to 4D1−4.0.

-noextend_source

Sets the source statement format as described in Table 1−5

Table 1−5 Source Statement Settings for −noextend_source Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

If the source statement contains fewer than 72 characters, blanks are assumed at the end; the ability of TAB−formatted lines to extend past Column 72 is disabled.

This option provides VMS Fortran default mode.

-noi4 Same as **-i2** option.

-nof77 Same as **-onetrip** switch except for the following:

- The syntax and behavior of **EXTERNAL** statements are altered.
- The default value for the **BLANK=***clause* in an **OPEN** statement is **ZERO**.
- The default value for the **STATUS=***clause* in an **OPEN** statement is **NEW**.

-noisam Excludes the indexed sequential access library *libisam.a* from being linked to the executable to reduce the size.

-old_rl Interprets the record length specifier for a direct unformatted file as a number of bytes instead of a number of words. This option provides backward compatibility with

4D1−3.1 releases and earlier.

-onetrip Same as **-1** option.

-1 Compiles **DO** loops so that they execute at least once if reached. By default, **DO** loops are not executed if the upper limit is smaller than the lower limit. Similar to the **-nof77** option.

-P Runs only the C macro preprocessor and puts the result of each source file into a corresponding **.i** file. The **.i** file cannot contain # lines.

-pfa Run the *pfa* preprocessor to automatically discover, parallelism in the source code. This also enables the multiprocessing directives. There are two optional arguments:

- **-pfa list** runs *pfa* and produces a listing file with suffix **.l** explaining which loops were parallelized, and if not, why not.
- **-pfa keep** runs *pfa*, produces the listing file, and also keeps the transformed, multiprocessed Fortran intermediate file in a file with suffix **.a**.

-R*flags* *flags* is a valid option for RATFOR; the flags are given in the *ratfor*(1) manual page.

The RATFOR input filename is *filename***.r**. The resulting output is placed in *filename* **.f**. You must specify the **-K** option to retain the output file.

-r8 Uses **REAL*8** and **COMPLEX*16** as the defaults for real and complex variables that are not explicitly declared with a type size.

-static Local variables are saved in a static location, initialized to zeros, and retain values between calls. This option overrides the default **-automatic** option.

-trapeuv Sets unitialized local variables to **0xFFFA5A5A**. This value is treated as a floating point NaN and causes a floating point trap.

-U Causes the compiler to differentiate upper− and lowercase alphabetic characters. For example, the compiler considers a and A as distinct characters. Note that this option causes the compiler to recognize lowercase keywords only. Therefore, lowercase keywords must be used in writing case−sensitive programs (or in writing generic header files).

-u Turns off Fortran default data typing and any data typing explicitly specified in an **IMPLICIT** statement. Forces the explicit declaration of all data types.

-usefpidx Uses the floating point **DO** loop variable as the loop counter instead of a separate integer counter to maintain backward compatibility with releases before 4D1−4.0.

-vms_cc Uses VMS Fortran carriage control interpretation on unit 6.

-vms_endfile

Causes a VMS endfile record to be written when an **ENDFILE** statement is executed, allows records to be written after an endfile record and subsequent reading from an input file after an endfile record is encountered.

-vms_library

	Treats subprograms starting with **LIB$**, **OTS$**, and **SMG$** as VMS run−time routines that accept a variable number of arguments.
-vms_stdin	Allows rereading from *stdin* after **EOF** has been encountered.
-w	Suppresses warning messages.
-w66	Suppresses Fortran 66 compatibility warning messages.

Debugging

The compiler system provides a source−level, interactive debugger called *dbx* that you can use to debug programs as they execute. With *dbx* you can control program execution to set breakpoints, monitor what is happening, modify values, and evaluate results. *dbx* keeps track of variables, subprograms, subroutines, and data types in terms of the symbols used in the source language. You can use this debugger to access the source text of the program, to identify and reference program entities, and to detect errors in the logic of the program.

Reference Information

For a complete list of **-g** driver options, see the *f77*(1) manual page. See the *dbx*(1) manual page for information on the debugger. For a complete description see the *dbx Reference Manual.*

Profiling

The compiler system permits the generation of profiled programs that, when executed, provide operational statistics. This is done through driver option **-p** (which provides pc sampling information) and the *pixie* and *prof* programs.

A variety of options and methods of profiling are available. To learn more about them, read Chapter 2 of the *IRIX Series Compiler Guide*, which describes the advantages and methods of profiling. It also gives examples of the various options and commands to achieve the desired results. See the *prof*(1) manual page for detailed reference information.

Optimizing

The default optimizing option, **-O1**, causes the code generator and assembler phases of compilation to improve the performance of your executable object. You can prevent optimization by specifying **-O0**.

Table 1−6 summarizes the optimizing functions available.

Table 1−6 Optimizer Options

Option	Result
-O3	Performs all optimizations, including global register allocation. With this option, a *ucode* object file is created for each Fortran source file and left in a **.u** file. The newly created *ucode* object files, the *ucode* object files specified on the command lines, the run−time startup routine, and all of the run−time libraries are *ucode* linked. Optimization is done globally on the resulting *ucode* linked file, and then it is linked as normal, producing an *a.out* file. No **.o** file is left from the *ucode* linked result. **-c** cannot be specified with -O3.

-O2	The global optimizer (*uopt*) phase executes. It performs optimization only within the bounds of individual compilation units.
-O1	Default option. The code generator and the assembler perform basic optimizations in a more limited scope.
-O0	No optimization.

The default option, **-O1**, causes the code generator and the assembler to perform basic optimizations such as constant folding, common subexpression elimination within individual statements, and common subexpression elimination between statements.

The global optimizer, invoked with the **-O2** option, is a single program that improves the performance of an object program by transforming existing code into more efficient coding sequences. Although the same optimizer processes all compiler optimizations, it does distinguish between the various languages supported by the compiler system programs to take advantage of the different language semantics involved.

See the *IRIX Series Compiler Guide* for details on the optimization techniques used by the compiler and tips on writing optimal code for optimizer processing.

Performance

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

- The **-feedback** and **-cord** options (see the *f77*(1) manual page) together with the *pixie*(1) and *prof*(1) utilities, can be used to reduce possible machine cache conflicts.
- The link editor **-G num** and **-bestGnum** options control the size of the global data area, which can produce significant performance improvements. See Chapter 2 of the *IRIX Series Compiler Guide* and the *ld*(1) manual page for more information.
- The **-jmpopt** option permits the link editor 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 *ld*(1) for more information.

Object File Tools

The following tools provide information on object files as indicated:

odump	Lists headers, tables, and other selected parts of an object or archive file. Chapters 10 and 11 of the *Assembly Language Programmer's Guide* describe the information provided.
stdump	Lists intermediate–code symbolic information for object files, executables, or symbolic information files.
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 Chapter 10 of the *Assembly Language Programmer's Guide* for a description of the contents and format of section data.

For more information on these tools, see the *odump*(1), *stdump*(1), *nm*(1), *file*(1), or *size*(1) manual pages.

Archiver

An archive library is a file that contains one or more routines in object (*.o*) file format. The term *object* as used in this chapter refers to an *.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 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 *ar*(1) manual page for additional information on the archiver.

Run–Time Considerations

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 *f77*) command line, the executable object module has the name that you specified.

File Formats

Fortran 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 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 made up of records of the given length. When the record length of a direct file is 1, the system treats the file as ordinary system files (as byte strings, in which each byte is addressable). A **READ** or **WRITE** request on such files consumes bytes until satisfied, rather than restricting itself to a single record.

Because of special requirements, sequential unformatted files will probably be 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 program also writes a new line indicator, the I/O system treats it as a separate record.

Preconnected Files

Table 1−7shows the standard preconnected files at program start.

Table 1−7 Preconnected Files

Unit #	Unit
5	Standard input
6	Standard output
0	Standard error

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. 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 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 are positioned by the parent process.

Unknown File Status

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

- If the file does not already 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.

Run−Time Error Handling

When the Fortran run−time system detects an error, the following action takes place:

- A message describing the error is written to the standard error unit (unit 0). See Appendix A, "Run−Time Error Messages,"for a list of the error messages.
- A core file is produced if the **f77_dump_flag** environment variable is set, as described in Appendix A, "Run−Time Error Messages.". You can use *dbx* or *edge* to inspect this file and determine the state of the program at termination. For more information, see the *dbx Reference Manual* and the *edge*(1) manual page.

  To invoke *dbx* using the core file, enter the following:

  `% dbx` *binary−file* **`core`**

  where *binary−file* is the name of the object file output (the default is **a.out**). For more information on *dbx*, see "Debugging".

Trap Handling

The library *libfpe.a* provides two methods for handling floating point exceptions: the subroutine **handle_sigfpes** and the environment variable **TRAP_FPE**. 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. See the *handle_sigfpes*(3F) manual page for more information.

Chapter 2

Storage Mapping

This chapter contains two sections:

- "Alignment, Size, and Value Ranges" describes how the Fortran compiler implements size and value ranges for various data types as well as how data alignment occurs under normal conditions.
- "Access of Misaligned Data" describes two methods of accessing misaligned data.

Alignment, Size, and Value Ranges

Table 2−1contains information about various data types.

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[a]	-32,768…32, 767
INTEGER	INTEGER*4[b]	32 bits	Word[c]	$-2^{31}\ldots2^{31}-1$
LOGICAL*1		8 bits	Byte	0…1
LOGICAL*2		16 bits	Half word[a]	0…1
LOGICAL	LOGICAL*4[d]	32 bits	Word[c]	0…1
REAL	REAL*4	32 bits	Word[c]	See the first note below
DOUBLE PRECISION	REAL*8	64 bits	Double word[e]	See the first note below
COMPLEX	COMPLEX*8	64 bits	Word[c]	
DOUBLE COMPLEX		128 bits	Double word[e]	
CHARACTER		8 bits	Byte	-128…127

[a] Byte boundary divisible by two.

[b] When **-i2** option is used, type **INTEGER** would be equivalent to **INTEGER*2**.

[c] Byte boundary divisible by four.

[d] When **-i2** option is used, type **LOGICAL** would be equivalent to **LOGICAL*2**.

[e] Byte boundary divisible by eight.

f

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

- Table 2−2lists the approximate valid ranges for **REAL** and **DOUBLE**.

Table 2−2 Valid Ranges for REAL and DOUBLE Data Types

Range	REAL	DOUBLE
Maximum	$3.40282356 * 10^{38}$	$1.7976931348623158 * 10^{308}$
Minimum normalized	$1.17549424 * 10^{-38}$	$2.2250738585072012 * 10^{-308}$

Minimum denormalized	$1.40129846 * 10^{-46}$	$2.2250738585072012 * 10^{-308}$

Note: When the compiler encounters a **REAL*16** declaration, it issues a warning message. **REAL*16** items are allocated 16 bytes of storage per element, but only the first 8 bytes of each element are used. Those 8 bytes are interpreted according to the format for **REAL*8** floating numbers.

- When the compiler encounters a **REAL*16** constant in a source program, the compiler issues a warning message. The constant is treated as a double precision (**REAL*8**) constant. **REAL*16** constants have the same form as double precision constants, except the exponent indicator is **Q** instead of **D**.
- Table 2−1states that **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.
- Forcing **INTEGER**, **LOCICAL**, **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. See Chapter 1, "Compiling, Linking, and Running Programs.".
- A **COMPLEX** data item is an ordered pair of real numbers; a double−complex data item is an ordered pair of double−precision numbers. In each case, the first number represents the real part and the second represents the imaginary part.
- **LOGICAL** data items denote only the logical values TRUE and FALSE (written as **.TRUE.** or **.FALSE.**). However, to provide VMS compatibility, **LOGICAL*1** variables can be assigned all values in the range -128 to 127.
- You must explicitly declare an array in a **DIMENSION** declaration or in a data type declaration. To support dimension, the compiler
  - 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 the **COMMON** statements:
  - The compiler assigns data items in the same sequence as they appear in the common statements defining the block. Data items will be padded according to the alignment switches or the default compiler. See "Access of Misaligned Data" 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 size allocated 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.

Access of Misaligned Data

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

As discussed in the previous section, the architecture of the IRIS−4D series assumes a particular alignment of data. ANSI standard Fortran 77 cannot violate the rules governing this alignment. Common extensions to the dialect, particularly small integer types, allowing intermixing of character and non−character data in **COMMON** and **EQUIVALENCE** statements and mismatching the types of formal and actual parameters across a subroutine interface, provide many opportunities for misalignment to occur.

Code using the extensions that compiled and executed 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 compilation system that allow the compilation and execution of programs whose data may be misaligned. Be forewarned that the execution of programs that use these options will be 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 the first method if the number of instances of misaligned data access is small or 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, keep the Fortran front end from padding data to force alignment by compiling your program with one of two options to *f77.*

- Use the **-align8** option if your program expects no restrictions on alignment.
- Use the **-align16** option if your program expects to 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) man page).

Accessing Misaligned Data Without Modifying Source

Use the second method for programs with widespread misalignment or whose source 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. The generation of these more forgiving instructions may be opted for each source file.

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

Chapter 3

Fortran Program Interfaces

This chapter contains the following major sections:

- "Fortran/C Interface" describes the interface between Fortran routines and routines written in C. It contains rules and gives examples for making calls and passing arguments between the two languages.
- "Fortran/C Wrapper Interface" describes the process of generating wrappers for C routines called by Fortran.
- "Fortran/Pascal Interface" describes the interface between Fortran routines and routines written in Pascal. It contains rules and gives examples for making calls and passing arguments between the two languages.

You may need to refer to other sources of information as you read this chapter.

- For information on storage mapping (how the variables of the various languages appear in storage), refer to Chapter 1 for Fortran and to Chapter 2 in the appropriate language programmer's guide for other languages.
- For information on the standard linkage conventions used by the compiler in generating code, see Chapter 7 of the *Assembly Language Programmer's Guide.*

For information on built−in functions that provide access to non−Fortran system functions and library routines, see Chapter 4 of this manual.

Fortran/C Interface

When writing Fortran programs that call C functions, consider procedure and function declaration conventions for both languages. Also, consider the rules for argument passing, array handling, and accessing common blocks of data.

Procedure and Function Declarations

This section discusses items to consider before calling C functions from Fortran.

Names

When calling a Fortran subprogram from C, the C program must append an underscore (_) to the name of the Fortran subprogram. For example, if the name of the subprogram is **matrix**, then call it by the name **matrix_**. When Fortran is calling a C function, the name of the C function must also end with an underscore.

The Fortran compiler changes all its subprogram names to lowercase. Thus, all of the following subprograms refer to the same function **matrix** when interfacing with C:

```
subroutine MATRIX
```

```
subroutine Matrix
subroutine matrix
```

The exception to this rule is when the **-u** option to *f77* is used. This option causes case to be preserved.

Note that only one main routine is allowed per program. The main routine can be written in either C or Fortran. Table 3−1contains an example of a C and a Fortran main routine.

Table 3−1 Main Routines

C	Fortran

main () {
printf("hi!\n");
}
	write (6,10)
	10 format ('hi!')
	end

Invocations

Invoke a Fortran subprogram as if it were an integer−valued function whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the subprogram but cause an indexed branch in the calling subprogram. If the subprogram is *not* a function and has no entry points with alternate return arguments, the returned value is undefined. The Fortran statement

```
call nret (*1,*2Ex,*3)
```

is treated exactly as if it were the computed **goto**

```
goto (1,2,3), nret()
```

A C function that calls a Fortran subprogram can usually ignore the return value of a Fortran subroutine; however, the C function should not ignore the return value of a Fortran function. Table 3−2shows equivalent function and subprogram declarations in C and Fortran programs.

Table 3−2 Equivalent C and Fortran Function Declarations

C Function Declaration	Fortran Function Declaration
double dfort()	double precision function dfort()
double rfort()	real function rfort()
int ifort()	integer function ifort()
int lfort	logical function lfort()

Note the following:

- Avoid calling Fortran functions of type **FLOAT**, **COMPLEX**, and **CHARACTER** from C.
- You cannot write a C function so that it will return a **COMPLEX** value to Fortran.
- A character−valued Fortran subprogram is equivalent to a C language routine with two extra initial arguments: a data address and a length. However, if the length is one, no extra argument is needed and the single character result is returned as in a normal numeric function.

  Thus

  ```
  character*15 function g(...)
  ```

  is equivalent to

```
char result [1];
long int length;
g_(result, length, ...)
...
```

and could be invoked in C by

```
char chars[15]
g_(chars, 15, ...);
```

and

```
character function h(...)
```

could be invoked in C by

```
char c, h();
c=h_(...);
```

Arguments

The following rules apply to arguments passed between Fortran and C:

- All explicit arguments must be passed by reference. All routines must specify an *address* rather than a value. Thus, to pass constants or expressions to Fortran, the C routine must first store their values into variables and then pass the address of the variable. (The only exception occurs when passing the length of a string from C to a Fortran subroutine with a parameter of type **CHARACTER**.)
- When passing the address of a variable, the data representations of the variable in the calling and called routines must correspond, as shown in Table 3–3

Table 3–3 Equivalent Fortran and C Data Types

Fortran	C
integer*2 x	short int x;
integer x	long int x; or just int x;
logical x	long int x; or just int x;
real x	float x;
double precision x	double x;
complex x	struct{float real, imag;) x;
double complex x	struct{double dreal,dimag;} x;
character*6 x	char x[6] [a]

[a] The array length must also be passed, as discussed in the next section.

- Note that in Fortran, **INTEGER** and **LOGICAL** variables occupy 32 bits of memory by default, but this can be changed by using the **-i2** option.
- The Fortran compiler may add items not explicitly specified in the source code to the argument list. The compiler adds the following items under the conditions specified:
  - destination address for character functions, when called

– length of a character variable, when an argument is the address of a character variable

When a C function calls a Fortran routine, the C function must explicitly specify these items in its argument list *in the following order:*

1. If the Fortran routine is a function that returns a character variable of length greater than 1, specify the address and length of the resultant character variable.
2. Specify normal arguments (addresses of arguments or functions).
3. Specify the length of each normal character parameter in the order it appeared in the argument list. The length must be specified as a constant value or **INTEGER** variable (that is, not an address).

The examples on the following pages illustrate these rules.

Example 1

This example shows how a C routine specifies the destination address of a Fortran function (which is only implied in a Fortran program).

Fortran

```
C  Fortran call to SAM, a routine written
C  in Fortran
   EXTERNAL F
   CHARACTER*7 S
   INTEGER B(3)
   …
   CALL SAM (F, B(2), S)
```

C

```
/* C call to SAM, a routine written in Fortran */
/* We pass in the function pointer for the     */
/* Fortran SUBROUTINE F                        */
char s[7];
int  b[3];
extern void sam_(void (*)(), int *, char*);
                        /* Fortran subroutine SAM          */
extern void f_();       /* Fortran subroutine F            */
…
sam_(F, &B[1], S);      /* We pass in pointer to Fortran F */
                        /* for Fortran call-by-reference   */
```

Example 2

This example shows how a C routine must specify the length of a character string (which is only implied in a Fortran call).

Fortran

```
C  Fortran call to F, a function written
C  in Fortran
   EXTERNAL F
   CHARACTER*10 F, G
   G = F()
```

C

```
/* C call to SAM, a routine written in Fortran */
/* which returns a string.                     */
CHAR S[10];
. . .
f_(S, 10);
```

The function F, written in Fortran

```
C  function F, written in Fortran
   CHARACTER*10 FUNCTION F()
   F = '0123456789'
   RETURN
   END
```

Array Handling

Fortran stores arrays in column−major order with the leftmost subscript varying the fastest. C, however, stores arrays in the opposite arrangement (row−major order), with the rightmost subscripts varying the fastest.

Here is how the layout of the Fortran array looks:

```
integer t (2,3)
t(1,1), t(2,1), t(1,2), t(2,2), t(1,3), t(2,3)
```

Here is how the layout of the C array looks:

```
int t [2] [3]
t[0][0], t[0][1], t[0][2], t[1][0], t[1][0], t[1][1],t[1][2]
```

Note that the default for the lower bound of an array in Fortran is 1, where the default in C is 0.

When a C routine uses an array passed by a Fortran subprogram, the dimensions of the array and the use of the subscripts must be interchanged, as shown in Figure 3−1

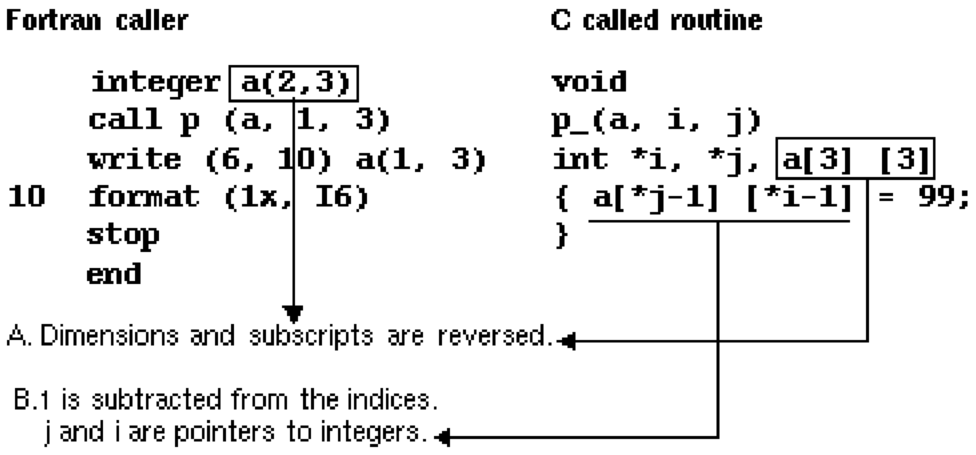

Figure 3–1 Array Subscripts

The Fortran caller prints out the value 99. Note the following:

- Because arrays are stored in column–major order in Fortran and row– major order in C, the dimension and subscript specifications are reversed.
- Because the lower–bound default is 1 for Fortran and 0 for C, 1 must be subtracted from the indexes in the C routine. Also, because Fortran passes parameters by reference, ***j** and ***p** are pointers in the C routine.

Accessing Common Blocks of Data

The following rules apply to accessing common blocks of data:

- Fortran common blocks must be declared by common statements; C can use any global variable. Note that the common block name in C (**sam_**) must end with an underscore.
- Data types in Fortran and C programs must match unless you want equivalencing. If so, you must adhere to the alignment restrictions for the data types described in Chapter 2.
- If the same common block is of unequal length, the largest size is used to allocate space.
- Unnamed common blocks are given the name **_BLNK_**.

The following examples show C and Fortran routines that access common blocks of data.

Fortran

```
subroutine sam()
common /r/ i, r
i = 786
r = 3.2
return
```

```
end
```

C

```
struct S {int i; float j;}r_;
main () {
sam_() ;
printf("%d %f\n",r_.i,r_.j);
}
```

The C routine prints out 786 and 3.2.

Fortran/C Wrapper Interface

This section describes the process of generating wrappers for C routines called by Fortran. If you want to call existing C routines (which use value parameters rather than reference parameters) from Fortran, these *wrappers* convert the parameters during the call.

The program *mkf2c* provides an alternate interface for C routines called by Fortran.

Fortran routines called by C must use the method described in "Fortran/C Wrapper Interface".

The Wrapper Generator mkf2c

The *mkf2c* program uses C data−type declarations for parameters to generate the correct assembly language interface. In generating a Fortran−callable entry point for an existing C−callable function, the C function is passed through *mkf2c*, and *mkf2c* adds additional entry points. Native language entry points are not altered.

Use these rules with *mkf2c*:

- Each function given to *mkf2c* must have the standard C function syntax.
- The function body must exist but can be empty. Function names are transformed as necessary in the output.

A simple case of using a function as input to *mkf2c* is

```
func()
{}
```

Here, the function **func** has no parameters. If *mkf2c* is used to produce a Fortran−to−C wrapper, the Fortran entry is **func_**. The wrapper **func_** simply calls the C routine **func***()*.

Here is another example:

```
simplefunc (a)
int a;
{}
```

In this example, the function **simplefunc** has one argument, **a**. The argument is of type **int**. For this function, **mkf2c** produces three items: a Fortran entry, **simple**, and two pieces of code. The first piece of code dereferences the address of **a**, which was passed by Fortran. The second passes the resulting **int** to C. It then calls the C routine **simplefunc()**.

Using Fortran Character Variables as Parameters

You can specify the length of a character variable passed as a parameter to Fortran either at compilation or at run time. The length is determined by the declaration of the parameter in the Fortran routine. If the declaration contains a length, the passed length must match the declaration. For example, in the following declaration, the length of the string is declared to be 10 characters:

```
character*10 string
```

The passed length must be 10 in order to match the declaration.

When this next declaration is used, the passed length is taken for operations performed on the variable inside the routine:

```
character*(*) string
```

The length can be retrieved by use of the Fortran intrinsic function **LEN**. Substring operations may cause Fortran run−time errors if they do not check this passed length.

Arrays of character variables are treated by Fortran as simple byte arrays, with no alignment of elements. The length of the individual elements is determined by the length passed at run time. For instance, the array **sarray()** can be declared in this manner:

```
character*(*) sarray()
```

This length is necessary to compute the indexes of the array elements. The program *mkf2c* has special constructs for dealing with the lengths of Fortran character variables.

Reduction of Parameters

The program *mkf2c* reduces each parameter to one of seven simple objects. The following list explains each object.

64−bit value — The quantity is loaded indirectly from the passed address, and the result is passed to C. Parameters with the C type double (or long float) are reduced to 64−bit values by converting the 32−bit Fortran **REAL** parameter to double precision (see below).

32−bit value — *mkf2c* uses the passed address to retrieve a 32−bit data value, which is passed to C. Parameters with C types **int** and **long** are reduced to 32−bit values. Any parameter with an unspecified type is assumed to be **int**. If the **-f** option is specified, parameters with the C type **float** are reduced to 32−bit values.

16−bit value — A 16−bit value is loaded using the passed address. The value is either extended (if type is **signed** in the function parameter list) or masked (if type is **unsigned**) and passed to C. Any parameter whose C type is **short** is reduced to a 16−bit value.

8−bit value — The **char** type in C corresponds to the **CHARACTER*1** type in Fortran 77. (There is no mechanism to pass **integer*1** variables to C. A pointer to the value can be passed by declaring the parameter as **int***.) By default the character value is loaded as an unsigned quantity and passed to C. If the **-signed** option has been specified when invoking *mkf2c*, the character value is sign extended before being passed to C.

character string A copy is made of the Fortran character variable, and it is null terminated, and passed as a character pointer to C. Any modifications that C makes to the string will not affect the corresponding character variable in the Fortran routine.

character array When using *mkf2c* to call C from Fortran, the address of the Fortran character variable is passed. This character array can be modified by C. It is not guaranteed to be null terminated. The length of the Fortran character variable is treated differently (as discussed in the next section).

pointer The value found on the stack is treated as a pointer and is passed without alteration. Any array or pointer that is not of type **char**, any parameter with multiple levels of indirection, or any indirect array is assumed to be of type **pointer**. If the type of a parameter is specified but is not one of the standard C types, *mkf2c* will pass it as a pointer.

Below is an example of a C specification for a function:

```
test (i,s,c,ptr1,ar1,u,f,d,d1,str1,str2,str3)
short s;
unsigned char c;
int *ptr1;
char *ptr2[];
short ar1[];
sometype u;
float f;
long float d, *d1;
char *str1;
char str2[],str3[30];
 {
   /* The C function body CAN go here. Nothing
   except the opening and closing braces are
   necessary */
```

If this function were passed to *mkf2c*, the parameters would be transformed as follows:

- **PTR1**, **PTR2**, **AR1**, **D1**, and **U** would be passed as simple pointers.
- *mkf2c* would complain about not understanding the type **SOMETYPE** but, by default, would assume it to be of type **POINTER**.
- **S**, **C**, and **D** would be passed as values of length 16 bits, 64 bits, and 8 bits, respectively. **F** would be converted to a 64−bit **DOUBLE** before being passed, unless the **-f** option had been specified. If the **-f** option had been specified, **F** would be passed as a 32−bit value. Because the type of **I** is not specified, it would be assumed to be **INT** and would also be passed as a 32−bit value. Storing values in any of these parameters would not have any effect on the original Fortran data.

Fortran Character Array Lengths

When the wrapper generator is used, a character variable that is specified as **char*** in the C parameter list is *copied* and null terminated. C may thus determine the length of the string by the use of the standard C function **strlen**.

If a character variable is specified as a character array in the C parameter list, the address of the character variable is passed, making it impossible for C to determine its length, as it is not null terminated. When the call occurs, the wrapper code receives this length from Fortran. For those C functions needing this information, the wrapper passes it by extending the C parameter list.

For example, if the C function header is specified as follows

```
func1 (carr1,i,str,j,carr2)
char carr1[],*str,carr2[];
int i, j;
{}
```

mkf2c will pass a total of seven parameters to C. The sixth parameter will be the length of the Fortran character variable corresponding to **carr1**, and the seventh will be the length of **carr2**. The C function **func1()** must use the **varargs** macros to retrieve these hidden parameters. *mkf2c* will ignore the

varargs macro **va_alist** appearing at the end of the parameter name list and its counterpart **va_alist** appearing at the end of the parameter type list. In the case above, use of these macros would produce the function header

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

The C routine could retrieve the lengths of **carr1** and **carr2**, placing them in the local variables **carr1_len** and **carr2_len** by the following code fragment:

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

Using mkf2c and extcentry

mkf2c understands only a limited subset of the C grammar. This subset includes common C syntax for function entry point, C−style comments, and function bodies. However, it cannot understand constructs such as typedefs, external function declarations, or C preprocessor directives.

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

Once these special comments, **/* CENTRY */** and **/* ENDCENTRY */**, are placed around the code, use the program *excentry*(1) before *mkf2c* to generate the input file for *mkf2c*.

To illustrate the use of *extcentry*, the C file *foo.c* is shown below. It contains the function *foo*, which is to be made Fortran callable.

```
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 */
```

The special comments **/* CENTRY */** and **/* ENDCENTRY */** surround the section that 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 found in the directory */usr/bin* on your workstation.

Makefile Considerations

make(1) contains 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 **F2CFLAGS**. 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 will control 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**.

Fortran/Pascal Interface

This section discusses items you should consider when writing a call between Fortran and Pascal.

Procedure and Function Declarations

This section explains procedure and function declaration considerations.

Names

In calling a Fortran program from Pascal, you must place an underscore (_) as a suffix to routine names and data names.

To call Fortran from Pascal or vice versa, specify an underscore (_) as the suffix of the name of the Fortran or Pascal routine being called. For example, if the routine is called *matrix*, then call it by the name **matrix_**.

In Pascal, always declare the external Fortran subprogram or function with **VAR** parameters.

Note that only one main routine is allowed per program. The main routine can be written either in Pascal or Fortran. Table 3−4contains an example of a Pascal and a Fortran main routine.

Table 3−4 Main Routines

Pascal	Fortran
program p;	write (6,10)
begin	10 format ('hi!')
writeln ('hi!');	stop
end.	end

Invocation

If you have alternate return labels, you can invoke a Fortran subprogram as if it were an integer−valued function whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but cause an indexed branch in the calling subprogram. If the subprogram is *not* a function and has no entry points with alternate return arguments, the returned value is undefined.

The Fortran statement

```
call nret (*1,*2,*3)
```

is treated exactly as if it were the computed *goto*

```
goto (1,2,3), nret()
```

A Pascal function that calls a Fortran subroutine can usually ignore the return value. Table 3−5shows equivalent function declarations in Pascal and Fortran.

Table 3−5 Function Declarations

Pascal	Fortran
function dfort_(): double;	double precision function dfort()
function rfort_(): real;	real function rfort()
function ifort_(): integer;	integer function ifort()

Fortran has a built−in data type**COMPLEX** that does not exist in Pascal. Therefore, there is no compatible way of returning these values from Pascal.

A character−valued Fortran function is equivalent to a Pascal language routine with two initial extra arguments: a data address and a length.

The following Fortran statement

```
character*15 function g (...)
```

is equivalent to the Pascal code

```
type string = array [1..15];
var
   length: integer;
   a: array[1..15] of char;
procedure g_(var a:string;length:integer;...); external;
```

and could be invoked by the Pascal line

```
g_ (a, 15);
```

Arguments

The following rules apply to argument specifications in both Fortran and Pascal programs:

- All arguments must be passed by reference. That is, the argument must specify an *address* rather than a value. Thus, to pass constants or expressions, their values must first be stored into variables and then the addresses of the variables passed.

- When passing the address of a variable, the data representations of the variable in the calling and called routines must correspond, as shown in Table 3–6

Table 3–6 Equivalent Fortran and Pascal Data Types

Pascal	Fortran
integer	integer*4, integer, logical
cardinal, char, boolean, enumeration	character
real	real
double	double precision
procedure	subroutine
record r:real; i:real; end;	complex
record r:double; i:double; end;	double complex

- Note that Fortran requires that each **INTEGER**, **LOGICAL**, and **REAL** variable occupy 32 bits of memory.
- Functions of type **INTEGER**, **REAL**, or **DOUBLE PRECISION** are interchangeable between Fortran and Pascal and require no special considerations.
- The Fortran compiler may add items not explicitly specified in the source code to the argument list. The compiler adds the following items under the conditions specified:
  - destination address for character functions, when called
  - length of character strings, when an argument is the address of a character string

When a Pascal program calls a Fortran subprogram, the Pascal program must explicitly specify these items in its argument list *in the following order*:

1. Destination address of character function.
2. Normal arguments (addresses of arguments or functions).
3. Length of character strings. The length must be specified as an absolute value or **INTEGER** variable. The next two examples illustrate these rules.

Example

The following example shows how a Pascal routine must specify the length of a character string (which is only implied in a Fortran call).

Fortran call to SAM

```
C  SAM IS A ROUTINE WRITTEN IN FORTRAN
```

```
   EXTERNAL F
   CHARACTER*7 S
   INTEGER B(3)
   ...
   CALL SAM (F, B(1), S) <- Length of S is implicit.
```

Pascal call to SAM

```
PROCEDURE F_; EXTERNAL;
S: ARRAY[1..7] OF CHAR;
B: ARRAY[1..3] OF INTEGER;
   ...
SAM_ (F, B[1], S, 7);  <- Length of S is explicit.
```

Execution−Time Considerations

Pascal checks certain variables for errors at execution time, whereas Fortran does not. For example, in a Pascal program, when a reference to an array exceeds its bounds, the error is flagged (if run−time checks are not suppressed). Use the **f77-c** option if you want a Fortran program to detect similar errors when you pass data to it from a Pascal program.

Array Handling

Fortran stores arrays in column−major order, where the leftmost subscripts vary the fastest. Pascal, however, stores arrays in row−major order, with the rightmost subscript varying the fastest. Also, the default lower bound for arrays in Fortran is 1. Pascal has no default; the lower bound must be explicitly specified. Here is an example of the various layouts:

Fortran

```
integer t (2,3)
t(1,1), t(2,1), t(1,2), t(2,2), t(1,3), t(2,3)
```

Pascal

```
var t: array[1..2,1..3] of integer;
t[1,1], t[1,2], t[1,3], t[2,1], t[2,2], t[2,3]
```

When a Pascal routine uses an array passed by a Fortran program, the dimensions of the array and the use of the subscripts must be interchanged. The example below shows the Pascal code that interchanges the subscripts.

In the following example, the Fortran routine calls the Pascal procedure **p**, receives the value 99, and prints it out.

Fortran

```
   INTEGER A(2,3)
   CALL P (A, 1, 3)
   WRITE (6,10) A(1,3)
10 FORMAT (1X, I9)
```

```
   STOP
   END
```

Pascal

```
TYPE ARRY = ARRAY [1..3,1..2];
PROCEDURE P_(VAR A:ARRY; VAR I,J:INTEGER);
BEGIN
 A[I,J] := 99;
END;
```

In the next example, the Pascal routine passes the character string "0123456789" to the Fortran subroutine **S_**, which prints it out and then returns to the calling program.

Pascal

```
TYPE STRING = ARRAY[1..10] OF CHAR;
PROCEDURE S_( VAR A: STRING; I: INTEGER); EXTERNAL;
/* Note the underbar */
PROGRAM TEST;
VAR
 R: STRING;
BEGIN
 R:= "0123456789";
 S_(R,10);
END.
```

Fortran

```
   SUBROUTING S(C)
   CHARACTER*10 C
   WRITE (6,10) C
10 FORMAT (6,10) C
   RETURN
   END
```

Accessing Common Blocks of Data

The following rules apply to accessing common blocks of data:

- Fortran common blocks must be declared by *common* statements; Pascal can use any global variable. Note that the common block name in Pascal (**sam_**) must end with an underscore.
- Data types in the Fortran and Pascal programs must match unless you want implicit equivalencing. If so, adhere to the alignment restrictions for the data types described in Chapter 2, "Storage Mapping."
- If the same common block is of unequal length, the largest size is used to allocate space.
- Unnamed common blocks are given the name **_BLNK_**, where **_** is the underscore character.

Example

The following examples show Fortran and Pascal routines that access common blocks of data.

Pascal

```
VAR
A_: RECORD
    I : INTEGER;
    R : REAL;
END;
PROCEDURE SAM_;
 EXTERNAL;
PROGRAM S;
BEGIN
A_.I := 4;
A_.R := 5.3;
SAM_;
END.
```

Fortran

```
   SUBROUTINE SAM()
   COMMON /A/I,R
   WRITE (6,10) i,r
10 FORMAT (1x,I5,F5.2)
   RETURN
   END
```

The Fortran routine prints out 4 and 5.30.

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" summarizes the Fortran run−time library functions.
- "Intrinsic Subroutine Extensions" describes the extensions to the Fortran intrinsic subroutines.
- "Function Extensions" describes the extensions to the Fortran functions.

Library Functions

The Fortran library functions provide an interface from Fortran programs to the system in the same way that the C library provides for C programs. The compiler automatically loads an interface routine when it processes the associated call.

Table 4−1summarizes the functions in the Fortran run−time library.

Table 4−1 Summary of System Interface Library Routines

Function	Purpose
abort	abnormal termination
access	determine accessibility of a file
acct	enable/disable process accounting
alarm	execute a subroutine after a specified time
barrier	perform barrier operations
blockproc	block processes
brk	change data segment space allocation
chdir	change default directory
chmod	change mode of a file
chown	change owner
chroot	change root directory for a command
close	close a file descriptor
creat	create or rewrite a file
ctime	return system time
dtime	return elapsed execution time
dup	duplicate an open file descriptor
etime	return elapsed execution time
exit	terminate process with status
fcntl	file control
fdate	return date and time in an ASCII string
fgetc	get a character from a logical unit
fork	create a copy of this process
fputc	write a character to a Fortran logical unit
free_barrier	free barrier

fseek	reposition a file on a logical unit
fstat	get file status
ftell	reposition a file on a logical unit
gerror	get system error messages
getarg	return command line arguments
getc	get a character from a logical unit
getcwd	get pathname of current working directory
getdents	read directory entries
getegid	get effective group ID
gethostid	get unique identifier of current host
getenv	get value of environment variables
geteuid	get effective user ID
getgid	get user or group ID of the caller
gethostname	get current host ID
getlog	get user's login name
getpgrp	get process group ID
getpid	get process ID
getppid	get parent process ID
getsockopt	get options on sockets
getuid	get user or group ID of caller
gmtime	return system time
iargc	return command line arguments
idate	return date or time in numerical form
ierrno	get system error messages
ioctl	control device
isatty	determine if unit is associated with tty
itime	return date or time in numerical form
kill	send a signal to a process
link	make a link to an existing file
loc	return the address of an object
lseek	move read/write file pointer
lstat	get file status
ltime	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_rcle_procs	resume child processes
m_set_procs	set number of subtasks
m_sync	synchronize all threads

m_unlock	unset a global lock
mkdir	make a directory
mknod	make a directory/file
mount	mount a filesystem
new_barrier	initialize a barrier structure
nice	lower priority of a process
open	open a file
oserror	get/set system error
pause	suspend process until signal
perror	get system error messages
pipe	create an interprocess channel
plock	lock process, test, or data in memory
prctl	control processes
profil	execution−time profile
ptrace	process trace
putc	write a character to a Fortran logical unit
putenv	set environment variable
qsort	quick sort
read	read from a file descriptor
readlink	read value of symbolic link
rename	change the name of a file
rmdir	remove a directory
sbrk	change data segment space allocation
schedctl	call to scheduler control
send	send a message to a socket
setblockproccnt	set semaphore count
setgid	set group ID
sethostid	set current host ID
setoserror	set system error
setpgrp	set process group ID
setsockopt	set options on sockets
setuid	set user ID
sginap	put process to sleep
shmat	attach shared memory
shmdt	detach shared memory
sighold	raise priority and hold signal
sigignore	ignore signal
signal	change the action for a signal
sigpause	suspend until receive signal
sigrelse	release signal and lower priority
sigset	specify system signal handling
sleep	suspend execution for an interval
socket	create an endpoint for communication TCP

sproc	create a new share group process
stat	get file status
stime	set time
symlink	make symbolic link
sync	update superblock
sysmp	control multiprocessing
system	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[a]	return system time
ttynam	find name of terminal port
uadmin	administrative control
ulimit	get and set user limits
umask	get and set file creation mask
umount	dismount a file system
unblockproc	unblock processes
unlink	remove a directory entry
uscalloc	shared memory allocator
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
usmallopt	control allocation algorithm
usnewlock	allocate and initialize a lock
usnewpollsema	allocate and initialize a pollable semaphore
usnewsema	allocate and initialize a semaphore

usopenpollsem	attach a file descriptor to a pollable semaphore
uspsema	acquire a semaphore
usputinfo	exchange information through an arena
usrealloc	user share memory allocation
ussetlock	set lock
ustest lock	test lock
ustestsema	return value of semaphore
ustrace	trace
usunsetlock	unset lock
usvsema	free a resource to a semaphore
uswsetlock	set lock
wait	wait for a process to terminate
write	write to a file

[a] The library function *time* can be invoked only if it is declared in an external statement.
Otherwise, it will be misinterpreted as the VMS−compatible intrinsic subroutine *time*.

You can display information on a *function* with the *man* command:

% man *function*

Intrinsic Subroutine Extensions

This section describes the intrinsic subroutines that are extensions to Fortran 77. The rules for using the intrinsic subroutines are

- The subroutine names are specially recognized by the compiler. A user−written subroutine with the same name as a system subroutine must be declared in an **EXTERNAL** statement in the calling subprogram.
- Using a user−written subroutine with the same name as a system subroutine in one subprogram does not preclude using the actual system subroutine in a different subprogram.
- To pass the name of a system subroutine as an argument to another subprogram, the name of the system subroutine must be declared in an **INTRINSIC** statement in the calling subprogram.
- When a system subroutine name is passed as an argument to another subprogram, the call to the system subroutine via the formal parameter name in the receiving subprogram must use the primary calling sequence for the subprogram (when there is more than one possible calling sequence).

Table 4−2gives an overview of the system subroutines and their function; they are described in detail in the sections following the table.

Table 4−2 Overview of System Subroutines

Subroutine	Information Returned
DATE	Current date as nine−byte string in ASCII representation
IDATE	Current month, day, and year, each represented by a separate integer
ERRSNS	Description of the most recent error

EXIT	Terminates program execution
TIME	Current time in hours, minutes, and seconds as an eight−byte string in ASCII representation
MVBITS	Moves a bit field to a different storage location

DATE

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

```
CALL DATE (buf)
```

where *buf* 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)
```

where *m*, *d*, and *y* 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

```
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 Table 4−3

Table 4−3 Information Returned by ERRSNS

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 *agr4* 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*)

where *status* 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*)

where *clock* 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*)

Table 4–4defines the arguments. Arguments can be declared as **INTEGER*2** or **INTEGER*4**.

Table 4–4 Arguments to MVBITS

Argument	Type	Contents
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

Function Extensions

Table 4–5gives an overview of the functions added as extensions of Fortran 77.

Table 4–5 Function Extensions

Function	Information Returned
SECNDS	Elapsed time as a floating point value in seconds. This is an intrinsic routine.
RAN	The next number from a sequence of pseudo–random numbers. This is not an intrinsic routine.

These functions are described in detail in the following sections.

SECNDS

SECNDS is an intrinsic routine that returns the number of seconds since midnight, minus the value of the passed arguments; 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

The **RAN** routine generates a random number; the format is as follows:

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

The argument *s* is an **INTEGER*4** variable or array element; *s* serves as a seed in determining the next random number and should initially be set to a large, odd integer value. This permits the computation of multiple random number series by supplying different variable names as the seed argument to **RAN**.

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

Chapter 5

Fortran Enhancements for Multiprocessors

This chapter contains these sections:

- "Overview" provides an overview of this chapter.
- "Parallel Loops" discusses the concept of parallel **DO** loops.
- "Writing Parallel Fortran" explains how to use compiler directives to generate code that can be run in parallel.
- "Analyzing Data Dependencies for Multiprocessing" describes how to analyze **DO** loops to determine whether they can be parallelized.
- "Breaking Data Dependencies" 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" describes how to determine whether the work performed in a loop is greater than the overhead associated with multiprocessing the loop.
- "Cache Effects" explains how to write loops that account for the effect of the cache.
- "Advanced Features" describes features that override multiprocessing defaults and customize parallelism.
- "DOACROSS Implementation" discusses how multiprocessing is implemented in a **DOACROSS** routine.

Overview

The Silicon Graphics Fortran compiler allows you to apply the capabilities of a Silicon Graphics 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 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," gives compilation and debugging instructions for parallel processing.

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, using the **SIMPLE** scheduling method, a **DO** loop consisting of 200 iterations will run on a machine with four processors. 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. Thus, 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 even be run on single–processor machines. The above loop would be divided into one block of 200 iterations. This allows code to be developed on a single–processor Silicon Graphics IRIS–4D Series workstation or Personal IRIS ™, 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 equal to the number of processors on the particular machine. If you want, you can override the default and explicitly control the number of threads of execution used by a Fortran 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 whether 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, any 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, Silicon Graphics has chosen to adopt 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, there are a number of special routines that permit more direct control over the parallel execution. (Refer to "Advanced Features" for more information.)

Writing Parallel Fortran

The Fortran 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". This section describes the others.

C$DOACROSS

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

The **C$DOACROSS** directive has the form

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

where a *clause* is one of the following:

`SHARE (`*variable list*`)`
`LOCAL (`*variable list*`)`
`LASTLOCAL (`*variable list*`)`

```
REDUCTION (scalar variable list)
IF (logical expression)
CHUNK=integer expression
MP_SCHEDTYPE=schedule type
```

The meaning of each clause is discussed below. All of these clauses are optional.

SHARE, LOCAL, LASTLOCAL

These are lists of variables as discussed in the "Analyzing Data Dependencies for Multiprocessing". A variable may 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 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"). 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.

Note: There is a minor flaw in the way unlisted variables default to **SHARE**. There must be at least one reference to the variable in a nonparallel region or at least one appearance of that variable in the **SHARE** list of some loop. If not, the compiler will complain that the variable in the multiprocessed loop has not been previously referenced.

REDUCTION

The **REDUCTION** clause lists those variables involved in a reduction operation. The meaning and use of reductions are discussed in Example 4 of "Breaking Data Dependencies". An element of the **REDUCTION** list must be an individual variable (also called a scalar variable) and may not be an array. However, it may be an individual element of an array. In this case, it would appear in the list with the proper subscripts.

It is possible for one element of an array to 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, it is legal for that array also to appear in the **SHARE** list.

There are four types of reduction supported: *sum*(+), *product*(*), *min*(), and *max*(). Note that *min*(*max*) reductions must use the *min*(*max*) functions in order to be recognized correctly.

The compiler makes some simple checks to confirm that the reduction expression is legal. The compiler does not, however, check all statements in the **DO** loop for illegal reductions. It is up to the programmer to assure legal use of the reduction variable.

IF

The **IF** clause gives a logical expression that is evaluated just before the loop is executed. If the 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 400 CPU clocks of work, which normally translates to about 100 floating point operations.

MP_SCHEDTYPE, CHUNK

These options affect the way the work in the loop is scheduled among the participating tasks. They do not affect the correctness of the loop. They are useful for tuning the performance of critical loops. See"Load Balancing" for more details.

Four methods of scheduling the iterations are supported. A single program may use any or all of them as it finds appropriate.

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

The interleave scheduling method (**MP_SCHEDTYPE=INTERLEAVE**) breaks the iterations up 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.

In dynamic scheduling (**MP_SCHEDTYPE=DYNAMIC**) the iterations are broken into **CHUNK**−sized pieces. 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 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 hope is to achieve good load balancing while reducing the number of entries into the critical section.

In addition to these four methods, the user may specify the scheduling method at run time (**MP_SCHEDTYPE=RUNTIME**). Here, the scheduling routine examines values in the user’s run−time environment and uses that information to select one of the four methods. See "Advanced Features" 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 1

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 2

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

You can be fully explicit:

```
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
```

or you can 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 5 in "Analyzing Data Dependencies for Multiprocessing" for more information on this example.

Example 3

```
   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

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X), 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
```

or you could 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 given the normal default of **SHARE**, which would be wrong.

C$&

Occasionally, the clauses in the **C$DOACROSS** directive are longer than one line. The **C$&** directive is used to continue the directive onto multiple lines.

```
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.

```
C$    PRINT 10
C$ 10 FORMAT('BEGIN MULTIPROCESSED LOOP')
C$DOACROSS LOCAL(Ii), SHARE(A,B)
      DO I = 1, 100
         CALL COMPUTE(A, B, I)
      END DO
```

C$MP_SCHEDTYPE, C$CHUNK

The **C$MP_SCHEDTYPE=***schedule_type* directive acts as an implicit **MP_SCHEDTYPE** clause. A **DOACROSS** directive that does not have an explicit **MP_SCHEDTYPE** clause is given the value specified in the directive, rather than the normal default. If the **DOACROSS** does have an explicit clause, then the explicit value is used.

The **C$CHUNK=***integer_expression* directive affects the **CHUNK** clause of a **DOACROSS** in the same way that the **C$MP_SCHEDTYPE** directive affects the **MP_SCHEDTYPE** clause. 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.

These directives are mostly intended for users of Silicon Graphics POWER Fortran Accelerator™ (PFA). The **DOACROSS** directives supplied by PFA do not have **MP_SCHEDTYPE** or **CHUNK** clauses. These directives provide a method of specifying what kind of scheduling option is desired and allowing PFA to supply the **DOACROSS** directive. These directives are not PFA–specific, however, and can be used by any multiprocessing Fortran programmer.

It is also possible to invoke this functionality from the command line during a compile. The **-mp_schedtype=***schedule_type* and **-chunk=** *integer* command line options have the effect of implicitly putting the corresponding directive(s) as the first lines in the file.

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.

Parallel Blocks

The Silicon Graphics Fortran compiler supports parallel execution of **DO** loops only. However, another kind of parallelism frequently occurs: different blocks of code independent of one another can be executed simultaneously. As a simple example,

```
CALL MAKE1(A, B, C, D)

CALL MAKE2(E, F, G, H)
```

If you know that these two routines do not interfere with each other, you can call them simultaneously. The following example shows how to use **DO** loops to execute parallel blocks of code.

```
C$DOACROSS LOCAL(I), MP_SCHEDTYPE=SIMPLE
      DO I = 1, 2
         IF (I .EQ. 1) THEN
             CALL MAKE1(A, B, C, D)
         ELSEIF (I .EQ. 2) THEN
             CALL MAKE2(E, F, G, H)
         END IF
      END DO
```

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 even at the same time, and the answer is still the same. This property is captured by the notion 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 also 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, 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**. The use of **REDUCTION** variables is discussed later.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to see if it fulfills the criteria for **LOCAL**, **LASTLOCAL**, **SHARE**, or **REDUCTION**. If all the variables 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" for information on rewriting code in parallel form.)

An alternative to analyzing variable usage by hand is to use PFA. This optional software package is a Fortran preprocessor that analyzes loops for data dependence. If it can determine that a loop is data–independent, it automatically inserts the required compiler directives (see"Writing Parallel Fortran"). If PFA cannot determine the loop to be 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 1: 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 2: 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 that 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 3: 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 4: 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 5: 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, that does not modify global data, and that does not use static storage. We say that **sqrt** has no *side effects*.

All the Fortran intrinsic functions listed in Appendix A of the *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,") cannot safely be included in a parallel loop. In particular, **rand** is not safe for multiprocessing. For user−written routines, it is the responsibility of the user to ensure that the routines can be correctly multiprocessed.

Caution: Routines called within a parallel loop cannot be compiled with the **-static** flag.

Example 6: 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 1 in "Breaking Data Dependencies").

Example 7: 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 8: 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 9: 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 particular 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 will read the value before it is written and some after it is written is not important because they will 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 10: 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"). Once the problem areas are identified, various techniques can be used 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 1: Loop Carried Value

```
INDX = 0
DO I = 1, N
INDX = INDX + I
```

```
A(I) = B(I) + C(INDX)
END DO
```

This is the same as Example 6 in "Writing Parallel Fortran". Here, **INDX** has its value carried from iteration to iteration. However, it is possible to compute the appropriate value for **INDX** without making reference to any previous value:

```
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 2: 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.

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 3: 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 4: Sum Reduction

```
        sum  = 0.0
        amax = a(1)
        amin = a(1)
c$doacross local(1), REDUCTION(asum, AMAX, AMIN)
        do i = 1,N
           asum = asum + a(i)
           if (a(i) .gt. amax) then
               imin = a(i)
           else if (a(i) .lt. amin) then
                    imin = a(i)
           end if
      end do
```

This operation is known as a *reduction.* Reductions occur when an array of values are 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
```

This 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 are not strictly correct. 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. Most of the time the difference is irrelevant, but it can be significant, so some caution is in order.

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

1. product: p = p*a(*i*)
2. mm: m = mm(m,a(i))
3. max: m = max(m,a(i))

For example,

```
c$doacross local(1), REDUCTION(asum, AMAX, AMIN)
        do i = 1,N
           big_sum  = big_sum + a(i)
           big_prod = big_prod * a(i)
           big_min  = min(big_min, a(i))
           big_max  = max(big_max, a(i)
        end do
```

One further reduction is noteworthy.

```
      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 may look as if the reduction in the inner loop needs to be rewritten in a parallel form. 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 1: 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 will show 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 2: 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 50.

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.50), 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 optimization 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.

Example 1: Matrix Multiply

```
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 1 in "Work Quantum". 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
```

Example 2: 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
```

but **M** needs to be large for the work quantum to show any improvement. In this particular example, **A(I)** is used to do a sum reduction, and it is possible to use the reduction techniques shown in Example 4 of "Breaking Data Dependencies" 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, that may take more memory than you can spare.

```
   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:

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

This 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, helps cure a load balancing problem.

This desirable transformation is provided to do this automatically by using the **MP_SCHEDTYPE** clause.

```
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
```

This has the same meaning as the rewritten form above.

Note that this can cause poor cache performance because you are no longer stepping through the array at stride 1. This can be somewhat improved by adding a **CHUNK** clause. **CHUNK=** 4 or 8 is often a good choice of value. 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* or from pc–sample profiling 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" 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 have to read the disk, 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.

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(3f) puts the slave threads into a blocked state using the system call **blockproc**(2). The slave threads stay blocked until a call is made to **mp_unblock**(3f). 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 should you neglect to do so.

mp_setup, mp_create, and mp_destroy

The **mp_setup**(3f), **mp_create**(3f), and **mp_destroy**(3f) 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; the **mp_block** and **mp_unblock** routines can be used 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 environment variable **MP_SET_NUMTHREADS**, or by the number of CPUs on the current hardware platform. **mp_setup** is called automatically when the first parallel loop is entered in order 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) man 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**(3f) 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 3 seconds. 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**(3f) is a zero−argument integer function that returns the total number of execution threads for this job. The count includes the master thread.

mp_set_numthreads(3f) 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_my_threadnum

mp_my_threadnum(3f) 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 rare cases.

Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP

These 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 *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 several multiprocessors and several CPUs, the environment variable **MP_SET_NUMTHREADS** also accepts an expression involving integers **+**, **-**, **mm**, **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)
```

Environment Variables: MP_SCHEDTYPE, CHUNK

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.

Environment Variable: MP_PROFILE

By default, the multiprocessing routines use the fastest possible method of doing their job. This can make it difficult to determine where the time is being spent if the multiprocessing routines themselves seem to be a bottleneck. By setting the environment variable **MP_PROFILE**, the multiprocessing routines use a slightly slower method of synchronization, where each step in the process is done in a separate subroutine with a long descriptive name. Thus *pixie* or pc−sample profiling can get more complete information regarding how much time is spent inside the multiprocessing routines.

Note: Only set/unset is important. The value the variable is set to is irrelevant (and typically is null).

mp_setlock, mp_unsetlock, mp_barrier

These zero−argument functions provide convenient (although limited) access to the locking and barrier functions provided by **ussetlock**(3p), **usunsetlock**(3p), and **barrier**(3p). The convenience is that no user initialization need be done because calls such as **usconfig**(3p) and **usinit**(3p) are done automatically. The limitation is that there is only one lock and one barrier. For a great many programs, this is sufficient. Users needing more complex or flexible locking facilities should use the **ussetlock** family of routines directly.

Local COMMON Blocks

A special *ld*(1) option allows named **COMMON** blocks to be local to a process. This means that 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
-Xlocaldata 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 −Xlocaldata foo_
```

would make the **COMMON** block **/foo/** be a local **COMMON** block in the resulting **a.out** file.

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. For example,

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

will propagate 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 **$DOACROSS** loops; only the original thread is allowed to do this.
- The calls to routines like **mp_block** and **mp_destroy** apply only to the threads created by **mp_create** or to those automatically created when the Fortran job starts; they have no effect on any user–defined threads.
- Calls to routines such as **m_get_numprocs**(3p) do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the routine **kill**(2) with the arguments **0** and **sig** (**kill**(0,**sig**)) to signal all members of the process group might possibly result in the death of the threads used to execute **C$DOACROSS**.
- If you choose to intercept the **IGCLD** signal, you must be prepared to receive this signal when the threads used for the **C$DOACROSS** loops exit; this occurs when **mp_destroy** is called or at program termination.
- Note in particular that **m_fork**(3p) is implemented using **sproc**, so it is not legal to **m_fork** a family of processes that each subsequently executes **C$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 the debugger and interpret the results of an execution profile.

Loop Transformation

When the Fortran compiler encounters a **C$DOACROSS** statement, it spools the corresponding **DO** loop into a separate subroutine and replaces the loop statement with a call to a special library routine. Exactly which routine is called depends on the value of **MP_SCHEDTYPE**. For discussion purposes, assume **SIMPLE** scheduling, so the library routine is **mp_simple_sched**.

The newly created subroutine is named using the following conventions. First, underscores are prepended and appended to the original routine name. For example, for a routine named **foo**, the first part of the name is **_foo_**. The next part of the name is the line number where the loop begins. This is the line number in the file, not the line number in the procedure. The last part of the name is a unique, four–character, alphabetic identifier. The first loop in a procedure uses **aaaa**, the second uses **aaab**, and so on. This "counter" is restarted to **aaaa** at the beginning of each procedure (not each file). So if the first parallel loop is at line 1234 in the routine named **foo**, the loop is named **_foo_1234_aaaa**. The second parallel loop, at line 1299, is named **_foo_1299_aaab**, 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 **mp_simple_sched** routine specifies, through subroutine arguments, which part of the loop a particular process is to execute. The spooled routine has four arguments: the starting value for the index, the number of times to execute the loop, the amount to increment the index, 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_1000_aaaa
X ( _LOCAL_START, _LOCAL_NTRIP, _INCR, _THREADINFO)
   INTEGER*4 _LOCAL_START
   INTEGER*4 _LOCAL_NTRIP
   INTEGER*4 _INCR
   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
```

Note: The compiler does not accept user code with an underscore (_) as the first letter of a variable name.

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**.

When the master process calls **mp_simple_sched**, the master passes the name of the spooled routine, the starting value of the **DO** loop index, the number of times the loop is to be executed, and the loop index increment.

The **mp_simple_sched** routine divides the work and signals the slaves. The master process then calls the spooled routine to do its work. When a slave is signaled, it wakes up from the wait loop, calculates which iterations of the spooled **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 the spooled routine, it returns to **mp_simple_sched**, then waits until all the slaves have completed processing. The master then returns to the main routine and continues execution.

Refer to Chapter 6 for an example of debugger output for the stack trace command *where*, which shows the calling sequence.

Chapter 6

Compiling and Debugging Parallel Fortran

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

- "Compiling and Running" explains how to compile and run a parallel Fortran program.
- "Profiling a Parallel Fortran Program" describes how to use the system profiler, *prof*, to examine execution profiles.
- "Debugging Parallel Fortran" presents some standard techniques for debugging a parallel Fortran program.
- "Parallel Programming Exercise" explains how to apply Fortran loop−level parallelism to an existing application.

This chapter assumes you have read Chapter 5, "Fortran Enhancements for Multiprocessors," and have reviewed the techniques and vocabulary for parallel processing in the IRIX environment.

Compiling and Running

After you have written a program for parallel processing, you should debug your program in a single−processor environment by calling the Fortran compiler with the *f77* command. After your program has executed successfully on a single processor, you can compile it for multiprocessing. Check the *f77*(1) manual page for multiprocessing options.

To turn on multiprocessing, add **-mp** to the *f77* command line. This option causes the Fortran compiler to generate multiprocessing code for the particular files being compiled. When linking, you can specify both object files produced with the **-mp** flag 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 Flag

A few words of caution about the **-static** flag: The 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** flag, 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 will not (see "Debugging Parallel Fortran" 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

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

compiles and links the Fortran program **foo.f** into a multiprocessor executable.

In this example

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

the Fortran routines in the file **snark.f** are compiled with multiprocess code generation enabled. The optimizer is also used. A standard **snark.o** binary is produced, which must be linked:

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

Here, the **-mp** flag signals the linker to use the Fortran multiprocessing library. The file **bellman.o** need not have been compiled with the **-mp** flag (although it could have been).

After linking, the resulting executable can be run like any standard executable. Creating multiple execution threads, running and synchronizing them, and task terminating 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 the number of processors that are on the machine (the value returned by the system call **sysmp(MP_NAPROCS)**; see the *sysmp*(2) man page). The default can be overridden by setting the shell environment variable **MP_SET_NUMTHREADS**. If it is set, Fortran tasks will use the specified number of execution threads regardless of the number of processors physically present on the machine. **MP_SET_NUMTHREADS** can be an integer from 1 to 16.

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 consuming 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. On jobs that use multiple threads, both these methods will create multiple profile data files, one for each thread. The standard profile analyzer *prof*(1) can be used to examine this output.

The profile of a Fortran parallel job is different from a standard profile. As mentioned in "Analyzing Data Dependencies for Multiprocessing", 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_simple_sched** 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 how parallel the program is.

"Parallel Programming Exercise" contains several examples of profiling output and how to use the information it provides.

Debugging Parallel Fortran

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

General Debugging Hints

- Debugging a multiprocessed program is much harder than debugging a single−processor program. For this reason, 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.
- Once you have narrowed the bug to a single file, use **-g-mp_keep** to save debugging information and to save the file containing the multiprocessed **DO** loop Fortran code that has been moved to a subroutine. **-mp_keep** will store the compiler−generated subroutines in the following file name:

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

  If you do not set **$TMPDIR**, **/tmp** is used.

Example: Erroneous C$DOACROSS

In this example, the bug is that the two references to **a** have the indexes in reverse order. 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 consiting of 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, 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 combat this problem, simply declare all the **LOCAL** variables **LASTLOCAL** when debugging a loop.

- 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 the value 0. This works with the **-static** flag, but without it, uninitialized values assume the value 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 when it is compiled without **-mp**. One way to track down 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 **-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. In fact, this sort of error often is seen only intermittently, with the program working correctly most of the time.

- The most likely candidates for this error are 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.

- If you suspect this type of error, as a final resort print out all the values of all the subscripts on each iteration through the loop. Then use **uniq**(1) to look for duplicates. If duplicates are found, then there is a data dependency.

Multiprocess Debugging Session

This section takes you through the process of debugging the following incorrectly multiprocessed code.

```
SUBROUTINE TOTAL(N, M, IOLD, INEW)
IMPLICIT NONE
INTEGER N, M
INTEGER IOLD(N,M), INEW(N,M)
DOUBLE PRECISION  AGGREGATE(100, 100)
COMMON /WORK/ AGGREGATE
INTEGER I, J, NUM, II, JJ
DOUBLE PRECISION TMP
C$DOACROSS LOCAL(I,II,J,JJ,NUM)
DO J = 2, M-1
   DO I = 2, N-1
      NUM = 1
      IF (IOLD(I,J) .EQ. 0) THEN
      INEW(I,J) = 1
   ELSE
      NUM = IOLD(I-1,J) + IOLD(I,J-1) + IOLD(I-1,J-1) +
   &        IOLD(I+1,J) + IOLD(I,J+1) + IOLD(I+1,J+1)
      IF (NUM .GE. 2) THEN
         INEW(I,J) = IOLD(I,J) + 1
      ELSE
         INEW(I,J) = MAX(IOLD(I,J)-1, 0)
      END IF
   END IF
   II = I/10 + 1
   JJ = J/10 + 1
   AGGREGATE(II,JJ) = AGGREGATE(II,JJ) + INEW(I,J)
   END DO
END DO
RETURN
END
```

In the program, the **LOCAL** variables are properly declared. **INEW** always appears with **J** as its second index, so it can be a **SHARE** variable when multiprocessing the **J** loop. The **IOLD**, **M**, and **N** are only read (not written), so they are safe. The problem is with **AGGREGATE**. The person analyzing this code reasoned that because **J** is different in each iteration, **J/10** will also be different. Unfortunately, because **J/10** uses integer division, it often gives the same results for different values of **J**.

Although this is a fairly simple error, it is not easy to see. When run on a single processor, the program always gets the right answer. Some of the time it gets the right answer when multiprocessing. The error occurs only when different processes attempt to load from and/or store into the same location in the **AGGREGATE** array at exactly the same time.

After reviewing the debugging hints from the previous section, try reversing the order of the iterations. Replace

```
DO J = 2, M-1
```

with

```
DO J = M-1, 2, -1
```

This still gives the right answer when running with one process and the wrong answer when running with multiple processes. The **LOCAL** variables look right, there are no **EQUIVALENCE** statements, and **INEW** uses only very simple indexing. The likely item to check is **AGGREGATE**. The next step is to use the debugger.

First compile the program with the **-g-mp_keep** options.

```
% f77 -g -mp -mp_keep driver.f total.f -o total.ex
driver.f:
total.f:
```

This debug session is being run on a single–processor machine, which forces the creation of multiple threads.

```
% setenv MP_SET_NUMTHREADS 2
```

Start the debugger.

```
% dbx total.ex
```

```
dbx version 1.31Copyright 1987 Silicon Graphics Inc.Copyright 1987
 MIPS Computer Systems Inc.Type 'help' for help.Reading symbolic
 information of `total.ex' . . .MAIN:14   14  do i = 1, isize
```

Tell *dbx* to pause when **sproc** is called.

```
(dbx) set $promptonfork=1
```

Start the job:

```
(dbx) run
```

```
Warning: MP_SET_NUMTHREADS greater than available cpus
                                                     (MP_SET_NUMTHREADS =
 2; cpus = 1)
Process 19324(total.ex) started
Process 19324(total.ex) has executed the "sproc" system call
Add child to process pool (n if no)?  y
Reading symbolic information of Process 19325 . . .
Process 19325(total.ex) added to pool
Process 19324(total.ex) after sproc [sproc.sproc:38,0x41e130]
Source (of sproc.s) not available for process 19324
```

Make each process stop at the first multiprocessed loop in the routine **total**, which is on line 99. Its name will be **_total_99_aaaa** (see "Loop Transformation"), so enter

```
(dbx) stop in _total_99_aaaa pgrp
```

```
[2] stop in _total_99_aaaa
```

```
[3] stop in _total_99_aaaa
```

Start them all off and wait for one of them to hit a break point.

```
(dbx) resume pgrp
(dbx) waitall
Process 19325(total.ex) breakpoint/trace
 trap[_total_99_aaaa:16,0x4006d0]
  16  j = _local_start
(dbx) showproc
Process 19324(total.ex) breakpoint/trace
 trap[_total_99_aaaa:16,0x4006d0]
Process 19325(total.ex) breakpoint/trace
 trap[_total_99_aaaa:16,0x4006d0]
```

Look at the complete listing of the multiprocessed loop routine.

```
(dbx) list 1,50
     1
     2
     3         subroutine _total_99_aaaa
     4        x ( _local_start, _local_ntrip, _incr, _my_threadno)
     5         integer*4 _local_start
     6         integer*4 _local_ntrip
     7         integer*4 _incr
     8         integer*4 _my_threadno
     9         integer*4 i
    10         integer*4 ii
    11         integer*4 j
    12         integer*4 jj
    13         integer*4 num
    14         integer*4 _dummy
    15
>*  16         j = _local_start
    17         do _dummy = 1,_local_ntrip
    18             do i = 2, n-1
    19
    20            num = 1
    21            if (iold(i,j) .eq. 0) then
    22                inew(i,j) = 1
More (n if no)?y
    23            else
    24               num = iold(i-1,j) + iold(i,j-1) + iold(i-1,j-1) +
    25        $                                   iold(i+1,j) +
 iold(i,j+1) + iold(i+1,j+1)
```

```
26               if (num .ge. 2) then
27              inew(i,j) = iold(i,j) + 1
28               else
29              inew(i,j) = max(iold(i,j)-1, 0)
30               end if
31           end if
32
33           ii = i/10 + 1
34           jj = j/10 + 1
35
36           aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
37
38            end do
39        j = j + 1
40        end do
41
42        end
```

To look at **AGGREGATE**, stop at that line with

```
(dbx) stop at 36 pgrp
```

```
[4] stop at "/tmp/Ptotalkea_11561_":36[5] stop at
 "/tmp/Ptotalkea_11561":36
```

Continue the current process (the master process). Note that *cont* continues only the current process; other members of the process group (**pgrp**) are unaffected.

```
(dbx) cont
```

```
[4] Process 19324(total.ex) stopped at [_total_99_aaaa:36,0x400974]
  36  aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)(dbx) \f8showproc
Process 19324(total.ex) breakpoint/trace
 trap[_total_99_aaaa:36,0x400974]Process 19325(total.ex)
 breakpoint/trace trap[_total_99_aaaa:16,0x4006d0]
```

Check the Slave

Look at the slave process with the following command:

```
(dbx) active 19325
```

```
Process 19325(total.ex) breakpoint/trace
 trap[_total_99_aaaa:16,0x4006d0](dbx) cont
[5] Process 19325(total.ex) stopped at [_total_99_aaaa:36,0x400974]
  36  aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
(dbx) where
>  0 _total_99_aaaa(_local_start = 6, _local_ntrip = 4, _incr = 1,
            my_threadno = 1) ["/tmp/Ptotalkea_11561":36, 0x400974]
```

```
   1 mp_slave_sync(0x0,0x0,0x1,0x1,0x0,0x0)["mp_slave.s":119,
0x402964]
```

The slave process has entered the multiprocessed routine from the slave synchronization routine **mp_slave_sync**. Both processes are now at the **AGGREGATE** assignment statement. Look at the values of the indexes in both processes.

```
(dbx) print ii
1
(dbx) print jj
1
(dbx) print ii pid 19324
1
(dbx) print jj pid 19324
1
```

The indexes are the same in both processes. Now examine the arguments to the multiprocessed routine; note that this information can also be seen in the *where* command above.

```
(dbx) print _local_ntrip
4
(dbx) print _local_start
6
(dbx) print j
6
(dbx) print _local_ntrip pid 19324
4
(dbx) print _local_start pid 19324
2
(dbx) print j pid 19324
2
```

The analysis for this loop assumed that **J/10** would be different for each loop iteration. This is the problem; confirm it by looking further into the loop

```
(dbx) active 19324
Process 19324(total.ex) breakpoint/trace
 trap[_total_99_aaaa:36,0x400974]
(dbx) where
>  0 _total_99_aaaa(_local_start = 2, _local_ntrip = 4, _incr = 1,
                                                  _my_threadno = 0)
  ["/tmp/Ptotalkea_11561":36, 0x400974]
   1 mp_simple_sched_(0x0, 0x0, 0x0, 0x0, 0x0, 0x40034c) [0x400e38]
   2 total.total(n = 100, m = 10, iold = (...), inew = (...))
                                                  ["total.f":15,
 0x4005f4]
   3 MAIN() ["driver.f":25, 0x400348]
```

```
   4 main.main(0x0, 0x7fffc7a4, 0x7fffc7ac, 0x0, 0x0, 0x0)
                                                        ["main.c":35,
 0x400afc]
(dbx) func total
[using total.total]
total:15   15  do j = 2, m-1
(dbx) print m
10
(dbx) quit
Process 19324(total.ex) terminated
Process 19325(total.ex) terminated
%
```

There are several possible ways to correct this problem; they are left as an exercise for the reader.

Parallel Programming Exercise

This section explains the techniques for applying Fortran loop−level parallelism to an existing application. Each program is unique; these techniques must be adapted for your particular needs.

In summary, the steps to follow are these:

1. Make the original code work on one processor.
2. Profile the code to find the time−critical part(s).
3. Perform data dependence analysis on the part(s) found in the previous step.
4. If necessary, rewrite the code to make it parallelizable. Add **C$DOACROSS** statements as appropriate.
5. Debug the rewritten code on a single processor.
6. Run the parallel version on a multiprocessor. Verify that the answers are correct.
7. If the answers are wrong, debug the parallel code. Always return to step 5 (single−process debugging) whenever any change is made to the code.
8. Profile the parallel version to gauge the effects of the parallelism.
9. Iterate these steps until satisfied.

First Pass

The next several pages take you through the process outlined above. The exercise is based on a model of a molecular dynamics program; the routine shown below will not work except as a test bed for the debug exercise.

Step 1: Make the Original Work

Make sure the original code runs on a Silicon Graphics workstation before attempting to multiprocess it.

Multiprocess debugging is much harder than single−process debugging, so fix as much as possible in the single−process version.

Step 2: Profile

Profiling the code enables you to focus your efforts on the important parts. For example, initialization code is frequently full of loops that will parallelize; usually these set arrays to zero. This code typically uses only 1 percent of the CPU cycles; thus working to parallelize it is pointless.

In the example, you get the following output when you run the program with *pixie*. For brevity, we omit listing the procedures that took less than 1 percent of the total time.

```
prof −pixie −quit 1% orig orig.Addrs orig.Counts
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
* −p[rocedures] using basic−block counts; sorted in   *
* descending order by the number of cycles executed in*
* each procedure; unexecuted procedures are excluded  *
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
10864760 cycles
    cycles %cycles  cum %   cycles  bytes procedure (file)
                             /call  /line
  10176621   93.67  93.67   484601    24 calc_ (/tmp/ctmpa00845)
    282980    2.60  96.27    14149    58 move_ (/tmp/ctmpa00837)
    115743    1.07  97.34      137    70 t_putc (lio.c)
```

The majority of time is spent in the **CALC** routine, which looks like this:

```
   SUBROUTINE CALC(NUM_ATOMS,ATOMS,FORCE,THRESHOLD,WEIGHT)
   IMPLICIT NONE
   INTEGER MAX_ATOMS
   PARAMETER(MAX_ATOMS = 1000)
   INTEGER  NUM_ATOMS
   DOUBLE PRECISION  ATOMS(MAX_ATOMS,3), FORCE(MAX_ATOMS,3)
   DOUBLE PRECISION  THRESHOLD
   DOUBLE PRECISION  WEIGHT(MAX_ATOMS)
   DOUBLE PRECISION  DIST_SQ(3), TOTAL_DIST_SQ
   DOUBLE PRECISION  THRESHOLD_SQ
   INTEGER I, J
   THRESHOLD_SQ = THRESHOLD ** 2
   DO I = 1, NUM_ATOMS
      DO J = 1, I-1
      DIST_SQ(1) = (ATOMS(I,1) - ATOMS(J,1)) ** 2
      DIST_SQ(2) = (ATOMS(I,2) - ATOMS(J,2)) ** 2
      DIST_SQ(3) = (ATOMS(I,3) - ATOMS(J,3)) ** 2
      TOTAL_DIST_SQ = DIST_SQ(1) + DIST_SQ(2) + DIST_SQ(3)
      IF (TOTAL_DIST_SQ .LE. THRESHOLD_SQ) THEN
```

```
C
C       ADD THE FORCE OF THE NEARBY ATOM ACTING ON THIS
C       ATOM ...
C
          FORCE(I,1) = FORCE(I,1) + WEIGHT(I)
          FORCE(I,2) = FORCE(I,2) + WEIGHT(I)
          FORCE(I,3) = FORCE(I,3) + WEIGHT(I)
C
C       ... AND THE FORCE OF THIS ATOM ACTING ON THE
C       NEARBY ATOM
C
          FORCE(J,1) = FORCE(J,1) + WEIGHT(J)
          FORCE(J,2) = FORCE(J,2) + WEIGHT(J)
          FORCE(J,3) = FORCE(J,3) + WEIGHT(J)
        END IF
        END DO
     END DO
     RETURN
     END
```

Step 3: Analyze

It is better to parallelize the outer loop, if possible, to enclose the most work. To do this, analyze the variable usage. The simplest and best way is to use the Silicon Graphics POWER Fortran Accelerator™ (PFA). If you do not have access to this tool, you must examine each variable by hand.

Data dependence occurs when the same location is written to and read. Therefore, any variables not modified inside the loop can be dismissed. Because they are read only, they can be made **SHARE** variables and do not prevent parallelization. In the example, **NUM_ATOMS**, **ATOMS**, **THRESHOLD_SQ**, and **WEIGHT** are only read, so they can be declared **SHARE**.

Next, **I** and **J** can be **LOCAL** variables. Perhaps not so easily seen is that **DIST_SQ** can also be a **LOCAL** variable. Even though it is an array, the values stored in it do not carry from one iteration to the next; it is simply a vector of temporaries.

The variable **FORCE** is the crux of the problem. The iterations of **FORCE(I,*)** are all right. Because each iteration of the outer loop gets a different value of **I**, each iteration uses a different **FORCE(I,*)**. If this was the only use of **FORCE**, we could make **FORCE** a **SHARE** variable. However, **FORCE(J,*)** prevents this. In each iteration of the inner loop, something may be added to both **FORCE(I,1)** and **FORCE(J,1)**. There is no certainty that **I** and **J** will ever be the same, so you cannot directly parallelize the outer loop. The uses of **FORCE** look similar to sum reductions but are not quite the same. A likely fix is to use a technique similar to sum reduction.

In analyzing this, notice that the inner loop runs from 1 up to **I**-1. Therefore, **J** is always less than **I**, and so the various references to **FORCE** do not overlap with iterations of the inner loop. Thus the various **FORCE(J,*)** references would not cause a problem if you were parallelizing the inner loop.

Further, the **FORCE(I,*)** references are simply sum reductions with respect to the inner loop (see "Debugging Parallel Fortran" Example 4, for information on modifying this loop with a reduction transformation). It appears you can parallelize the inner loop. This is a valuable fallback position should you be unable to parallelize the outer loop.

But the idea is still to parallelize the outer loop. Perhaps sum reductions might do the trick. However, remember round–off error: accumulating partial sums gives different answers from the original because the precision nature computer arithmetic is limited. Depending on your requirements, sum reduction may not be the answer. The problem seems to center around **FORCE**, so try pulling those statements entirely out of the loop.

Step 4: Rewrite

Rewrite the loop as follows; changes are noted in bold.

```
SUBROUTINE CALC(NUM_ATOMS,ATOMS,FORCE,THRESHOLD, WEIGHT)
IMPLICIT NONE
INTEGER MAX_ATOMS
   PARAMETER(MAX_ATOMS = 1000)
   INTEGER  NUM_ATOMS
   DOUBLE PRECISION  ATOMS(MAX_ATOMS,3), FORCE(MAX_ATOMS,3)
   DOUBLE PRECISION  THRESHOLD, WEIGHT(MAX_ATOMS)
   LOGICAL FLAGS(MAX_ATOMS,MAX_ATOMS)
   DOUBLE PRECISION  DIST_SQ(3), TOTAL_DIST_SQ
   DOUBLE PRECISION  THRESHOLD_SQ
   INTEGER I, J
   THRESHOLD_SQ = THRESHOLD ** 2
C$DOACROSS LOCAL(I,J,DIST_SQ,TOTAL_DIST_SQ)
   DO I = 1, NUM_ATOMS
      DO J = 1, I-1
      DIST_SQ(1) = (ATOMS(I,1) - ATOMS(J,1)) ** 2
      DIST_SQ(2) = (ATOMS(I,2) - ATOMS(J,2)) ** 2
      DIST_SQ(3) = (ATOMS(I,3) - ATOMS(J,3)) ** 2
      TOTAL_DIST_SQ=DIST_SQ(1)+DIST_SQ(2)+ DIST_SQ(3)
C
C               SET A FLAG IF THE DISTANCE IS WITHIN THE
C               THRESHOLD
C
      IF (TOTAL_DIST_SQ .LE. THRESHOLD_SQ) THEN
          FLAGS(I,J) = .TRUE.
      ELSE
          FLAGS(I,J) = .FALSE.
      END IF
      END DO
   END DO
```

```
   DO I = 1, NUM_ATOMS
      DO J = 1, I-1
      IF (FLAGS(I,J)) THEN
C
C        ADD THE FORCE OF THE NEARBY ATOM ACTING ON THIS
C        ATOM ...
C
          FORCE(I,1) = FORCE(I,1) + WEIGHT(I)
          FORCE(I,2) = FORCE(I,2) + WEIGHT(I)
          FORCE(I,3) = FORCE(I,3) + WEIGHT(I)
C
C        ... AND THE FORCE OF THIS ATOM ACTING ON THE
C        NEARBY ATOM
C
         FORCE(J,1) = FORCE(J,1) + WEIGHT(J)
         FORCE(J,2) = FORCE(J,2) + WEIGHT(J)
         FORCE(J,3) = FORCE(J,3) + WEIGHT(J)
      END IF
      END DO
   END DO
   RETURN
   END
```

You have parallelized the distance calculations, leaving the summations to be done serially. Because you did not alter the order of the summations, this should produce exactly the same answer as the original version.

Step 5: Debug on a Single Processor

The temptation might be strong to rush the rewritten code directly to the multiprocessor at this point. Remember, single–process debugging is easier than multiprocess debugging. Spend time now to compile and correct the code without the **-mp** flag to save time later.

A few iterations should get it right.

Step 6: Run the Parallel Version

Compile the code with the **-mp** flag. As a further check, do the first run with the environment variable **MP_SET_NUMTHREADS** set to 1. When this works, set **MP_SET_NUMTHREADS** to 2, and run the job multiprocessed.

Step 7: Debug the Parallel Version

If you get the correct output from the version with one thread but not from the version with multiple threads, you need to debug the program while running multiprocessed. Refer to "General Debugging Hints" for help.

Step 8: Profile the Parallel Version

After the parallel job executes correctly, check whether the run time has improved. First, compare an execution profile of the modified code compiled without **-mp** with the original profile. This is important because, in rewriting the code for parallelism, you may have introduced new work. In this example, writing and reading the **FLAGS** array, plus the overhead of the two new **DO** loops, are significant.

The *pixie* output on the modified code shows the difference:

```
% prof -pixie -quit 1% try1 try1.Addrs try1.Counts
----------------------------------------------------------
*  -p[rocedures] using basic-block counts; sorted in     *
*  descending order by the number of cycles executed in  *
*  each procedure; unexecuted procedures are excluded    *
----------------------------------------------------------
13302554 cycles
    cycles %cycles  cum %   cycles  bytes procedure (file)
                             /call  /line
  12479754   93.81  93.81   594274    25 calc_ (/tmp/ctmpa00857)
    282980    2.13  95.94    14149    58 move_ (/tmp/ctmpa00837)
    155721    1.17  97.11       43    29 _flsbuf (flsbuf.c)
```

The single−processor execution time has increased by about 30 percent.

Look at an execution profile of the master thread in a parallel run and compare it with these single−process profiles:

```
% prof -pixie -quit 1% try1.mp try1.mp.Addrs try1.mp.Counts00421
----------------------------------------------------------
*  -p[rocedures] using basic-block counts; sorted in     *
*  descending order by the number of cycles executed in  *
*  each procedure; unexecuted procedures are excluded    *
----------------------------------------------------------
12735722 cycles
    cycles %cycles  cum %   cycles  bytes procedure (file)
                             /call  /line
   6903896   54.21  54.21   328767    37 calc_ (/tmp/ctmpa00869)
   3034166   23.82  78.03   137917    16 mp_waitmaster
 (mp_simple_sched.s)
   1812468   14.23  92.26    86308    19 _calc_88_aaaa (/tmp/fMPcalc_)
    294820    2.31  94.57   294820    13 mp_create (mp_utils.c)
    282980    2.22  96.79    14149    58 move_ (/tmp/ctmpa00837)
```

Multiprocessing has helped very little compared with the single−process run of the modified code: the program is running slower than the original. What happened? The cycle counts tell the story. The routine **calc_** is what remains of the original routine after the **C$DOACROSS** loop **_calc_88_aaaa** is extracted (refer to "Loop Transformation" for details about loop naming conventions). **calc_** still takes nearly 70 percent of the time of the original. When you pulled the code for **FORCE** into a separate loop, you had to remove too much from the loop. The serial part is still too large.

Additionally, there seems to be a load−balancing problem. The master is spending a large fraction of its time waiting for the slave to complete. But even if the load were perfectly balanced, there would still be the 30 percent additional work of the multiprocessed version. Trying to fix the load balancing right now will not solve the general problem.

Regroup and Attack Again

Now is the time to try a different approach. If the first attempt does not give precisely the desired result, regroup and attack from a new direction.

Repeat Step 3: Analyze

At this point, round−off errors might not be so terrible. Perhaps you can try to adapt the sum reduction technique to the original code.

Although the calculations on **FORCE** are not quite the same as a sum reduction, you can use the same technique: give the reduction variable one extra dimension so that each thread gets its own separate memory location.

Repeat Step 4: Rewrite

As before, changes are noted in bold.

```
      SUBROUTINE CALC(NUM_ATOMS,ATOMS,FORCE,THRESHOLD,WEIGHT)
      IMPLICIT NONE
      INTEGER MAX_ATOMS
      PARAMETER(MAX_ATOMS = 1000)
      INTEGER  NUM_ATOMS
      DOUBLE PRECISION  ATOMS(MAX_ATOMS,3), FORCE(MAX_ATOMS,3)
      DOUBLE PRECISION  THRESHOLD
      DOUBLE PRECISION  WEIGHT(MAX_ATOMS)
      DOUBLE PRECISION  DIST_SQ(3)
      DOUBLE PRECISION  THRESHOLD_SQ
      INTEGER       I, J
      INTEGER       MP_SET_NUMTHREADS, MP_NUMTHREADS
      INTEGER       BLOCK_SIZE, THREAD_INDEX
      EXTERNAL      MP_NUMTHREADS
      DOUBLE PRECISION  PARTIAL(MAX_ATOMS, 3, 4)
      THRESHOLD_SQ = THRESHOLD ** 2
      MP_SET_NUMTHREADS = MP_NUMTHREADS()
C
C INITIALIZE THE PARTIAL SUMS
C
```

```
C$DOACROSS LOCAL(THREAD_INDEX,I,J)
   DO THREAD_INDEX = 1, MP_SET_NUMTHREADS
       DO I = 1, NUM_ATOMS
           DO J = 1, 3
          PARTIAL(I,J,THREAD_INDEX) = 0.0D0
          END DO
       END DO
   END DO
   BLOCK_SIZE = (NUM_ATOMS + (MP_SET_NUMTHREADS-1)) /
     &                MP_SET_NUMTHREADS
C$DOACROSS LOCAL(THREAD_INDEX, I, J, DIST_SQ, TOTAL_DIST_SQ)
   DO THREAD_INDEX = 1, MP_SET_NUMTHREADS
   DO I = THREAD_INDEX*BLOCK_SIZE - BLOCK_SIZE + 1,
     $          MIN(THREAD_INDEX*BLOCK_SIZE, NUM_ATOMS)
      DO J = 1, I-1
      DIST_SQ1 = (ATOMS(I,1) - ATOMS(J,1)) ** 2
      DIST_SQ2 = (ATOMS(I,2) - ATOMS(J,2)) ** 2
      DIST_SQ3 = (ATOMS(I,3) - ATOMS(J,3)) ** 2
      TOTAL_DIST_SQ = DIST_SQ1 + DIST_SQ2 + DIST_SQ3
      IF (TOTAL_DIST_SQ .LE. THRESHOLD_SQ) THEN
C
C     ADD THE FORCE OF THE NEARBY ATOM ACTING ON THIS
C     ATOM ...
C
   PARTIAL(I,1,THREAD_INDEX) = PARTIAL(I,1,
     +          THREAD_INDEX) + WEIGHT(I)
   PARTIAL(I,2,THREAD_INDEX) = PARTIAL(I,2,
     +          THREAD_INDEX) + WEIGHT(I)
   PARTIAL(I,3,THREAD_INDEX) = PARTIAL(I,3,
     +          THREAD_INDEX) + WEIGHT(I)
C
C       ... AND THE FORCE OF THIS ATOM ACTING ON THE
C       NEARBY ATOM
C
   PARTIAL(J,1,THREAD_INDEX) = PARTIAL(J,1,THREAD_INDEX)
     +          + WEIGHT(J)
   PARTIAL(J,2,THREAD_INDEX) = PARTIAL(J,2,THREAD_INDEX)
     +          + WEIGHT(J)
   PARTIAL(J,3,THREAD_INDEX) = PARTIAL(J,3,THREAD_INDEX)
     +          + WEIGHT(J)
      END IF
       END DO
   END DO
```

```
   ENDDO
C
C  TOTAL UP THE PARTIAL SUMS
C
   DO I = 1, NUM_ATOMS
      DO THREAD_INDEX = 1, MP_SET_NUMTHREADS
      FORCE(I,1) = FORCE(I,1) + PARTIAL(I,1,THREAD_INDEX)
      FORCE(I,2) = FORCE(I,2) + PARTIAL(I,2,THREAD_INDEX)
      FORCE(I,3) = FORCE(I,3) + PARTIAL(I,3,THREAD_INDEX)
      END DO
   END DO
   RETURN
   END
```

Repeat Step 5: Debug on a Single Processor

Because you are doing sum reductions in parallel, the answers may not exactly match the original. Be careful to distinguish between real errors and variations introduced by round−off. In this example, the answers agreed with the original for 10 digits.

Repeat Step 6: Run the Parallel Version

Again, because of round−off, the answers produced vary slightly depending on the number of processors used to execute the program. This variation must be distinguished from any actual error.

Repeat Step 7: Profile the Parallel Version

The output from the *pixie* run for this routine looks like this:

```
% prof −pixie −quit 1% try2.mp try2.mp.Addrs try2.mp.Counts00423
----------------------------------------------------------
*  −p[rocedures] using basic−block counts; sorted in     *
*  descending order by the number of cycles executed in  *
*  each procedure; unexecuted procedures are excluded    *
----------------------------------------------------------
10036679 cycles
    cycles %cycles  cum %   cycles  bytes procedure (file)
                             /call  /line
   6016033   59.94  59.94   139908  16 mp_waitmaster
 (mp_simple_sched.s)
   3028682   30.18  90.12   144223  31 _calc_88_aaab (/tmp/fMPcalc_)
    282980    2.82  92.94    14149  58 move_ (/tmp/ctmpa00837)
    194040    1.93  94.87     9240  41 calc_ (/tmp/ctmpa00881)
    115743    1.15  96.02      137  70 t_putc (lio.c)
```

With this rewrite, **calc_** now accounts for only a small part of the total. You have pushed most of the work into the parallel region. Because you added a multiprocessed initialization loop before the main loop, that new loop is now named **_calc_88_aaaa** and the main loop is now **_calc_88_aaab**. The initialization took less than 1 percent of the total time and so does not even appear on the listing.

The large number for the routine **mp_waitmaster** indicates a problem. Look at the *pixie* run for the slave process

```
% prof -pixie -quit 1% try2.mp try2.mp.Addrs try2.mp.Counts00424
----------------------------------------------------------

*  -p[rocedures] using basic-block counts; sorted in      *
*  descending order by the number of cycles executed in   *
*  each procedure; unexecuted procedures are excluded     *
----------------------------------------------------------

10704474 cycles
    cycles %cycles  cum %   cycles  bytes procedure (file)

                             /call  /line

   7701642   71.95  71.95   366745    31 _calc_2_ (/tmp/fMPcalc_)

   2909559   27.18  99.13    67665    32 mp_slave_wait_for_work
 (mp_slave.s)
```

The slave is spending more than twice as many cycles in the main multiprocessed loop as the master. This is a severe load balancing problem.

Repeat Step 3 Again: Analyze

Examine the loop again. Because the inner loop goes from 1 to **I**–1, the first few iterations of the outer loop have far less work in them than the last iterations. Try breaking the loop into interleaved pieces rather than contiguous pieces. Also, because the **PARTIAL** array should have the leftmost index vary the fastest, flip the order of the dimensions. For fun, we will put some loop unrolling in the initialization loop. This is a marginal optimization because the initialization loop is less than 1 percent of the total execution time.

Repeat Step 4 Again: Rewrite

The new version looks like this, with changes in bold:

```
   SUBROUTINE CALC(NUM_ATOMS,ATOMS,FORCE,THRESHOLD,WEIGHT)
   IMPLICIT NONE
   INTEGER MAX_ATOMS
   PARAMETER(MAX_ATOMS = 1000)
   INTEGER  NUM_ATOMS
   DOUBLE PRECISION  ATOMS(MAX_ATOMS,3), FORCE(MAX_ATOMS,3)
   DOUBLE PRECISION  THRESHOLD
   DOUBLE PRECISION  WEIGHT(MAX_ATOMS)
   DOUBLE PRECISION  DIST_SQ(3), TOTAL_DIST_SQ
```

```
      DOUBLE PRECISION  THRESHOLD_SQ
      INTEGER I, J
      INTEGER MP_SET_NUMTHREADS, MP_NUMTHREADS, THREAD_INDEX
      EXTERNAL MP_NUMTHREADS
      DOUBLE PRECISION  PARTIAL(3, MAX_ATOMS, 4)
      THRESHOLD_SQ = THRESHOLD ** 2
      MP_SET_NUMTHREADS = MP_NUMTHREADS()
C
C  INITIALIZE THE PARTIAL SUMS
C
C$DOACROSS LOCAL(THREAD_INDEX,I,J)
      DO THREAD_INDEX = 1, MP_SET_NUMTHREADS
         DO I = 1, NUM_ATOMS
         PARTIAL(1,I,THREAD_INDEX) = 0.0D0
         PARTIAL(2,I,THREAD_INDEX) = 0.0D0
         PARTIAL(3,I,THREAD_INDEX) = 0.0D0
         END DO
      END DO
C$DOACROSS LOCAL(THREAD_INDEX, I, J, DIST_SQ, TOTAL_DIST_SQ)
      DO THREAD_INDEX = 1, MP_SET_NUMTHREADS
      DO I = THREAD_INDEX, NUM_ATOMS, MP_SET_NUMTHREADS
         DO J = 1, I-1
         DIST_SQ1 = (ATOMS(I,1) - ATOMS(J,1)) ** 2
         DIST_SQ2 = (ATOMS(I,2) - ATOMS(J,2)) ** 2
         DIST_SQ3 = (ATOMS(I,3) - ATOMS(J,3)) ** 2
         TOTAL_DIST_SQ = DIST_SQ1 + DIST_SQ2 + DIST_SQ3
         IF (TOTAL_DIST_SQ .LE. THRESHOLD_SQ) THEN
C
C       ADD THE FORCE OF THE NEARBY ATOM ACTING ON THIS
C       ATOM ...
C
      PARTIAL(1,I,THREAD_INDEX) = PARTIAL(1,I, THREAD_INDEX)
     +      + WEIGHT(I)
      PARTIAL(2,I, THREAD_INDEX) = PARTIAL(2,I, THREAD_INDEX)
     +      + WEIGHT(I)
      PARTIAL(3,I,THREAD_INDEX) = PARTIAL(3,I, THREAD_INDEX)
     +      + WEIGHT(I)
C
C
C
      PARTIAL(1,J,THREAD_INDEX) = PARTIAL(1,J, THREAD_INDEX)
     +      + WEIGHT(J)
      PARTIAL(2,J,THREAD_INDEX) = PARTIAL(2,J, THREAD_INDEX)
     +      + WEIGHT(J)
```

```
   PARTIAL(3,J,THREAD_INDEX) = PARTIAL(3,J, THREAD_INDEX)
     +      + WEIGHT(J)
      END IF
      END DO
   END DO
   ENDDO
C
C  TOTAL UP THE PARTIAL SUMS
C
   DO THREAD_INDEX = 1, MP_SET_NUMTHREADS
      DO I = 1, NUM_ATOMS
      FORCE(I,1) = FORCE(I,1) + PARTIAL(1,I,THREAD_INDEX)
      FORCE(I,2) = FORCE(I,2) + PARTIAL(2,I,THREAD_INDEX)
      FORCE(I,3) = FORCE(I,3) + PARTIAL(3,I,THREAD_INDEX)
      END DO
   END DO
   RETURN
   END
```

With these final fixes in place, repeat the same steps to verify the changes:

1. Debug on a single processor.
2. Run the parallel version.
3. Debug the parallel version.
4. Profile the parallel version.

Repeat Step 7 Again: Profile

The *pixie* output for the latest version of the code looks like this:

```
% prof -pixie -quit 1% try3.mp try3.mp.Addrs try3.mp.Counts00425
------------------------------------------------------
*  -p[rocedures] using basic-block counts; sorted in     *
*  descending order by the number of cycles executed in  *
*  each procedure; unexecuted procedures are excluded    *
----------------------------------------------------------
7045818 cycles
    cycles %cycles  cum %   cycles  bytes procedure (file)
                             /call  /line
   5960816   84.60  84.60   283849  31 _calc_2_ (/tmp/fMPcalc_)
    282980    4.02  88.62    14149  58 move_ (/tmp/ctmpa00837)
    179893    2.75  91.37     4184  16 mp_waitmaster
 (mp_simple_sched.s)
    159978    2.55  93.92     7618  41 calc_ (/tmp/ctmpa00941)
```

```
    115743    1.64  95.56      137  70 t_putc (lio.c)
```

This looks good. To be sure you have solved the load−balancing problem, check that the slave output shows roughly equal amounts of time spent in **_calc_2_**. Once this is verified, you are finished.

Epilogue

After considerable effort, you reduced execution time by about 30 percent by using two processors. Because the routine you multiprocessed still accounts for the majority of work, even with two processors, you would expect considerable improvement by moving this code to a four−processor machine. Because the code is parallelized, no further conversion is needed for the more powerful machine; you can just transport the executable image and run it.

Note that you have added a noticeable amount of work to get the multiprocessing correct; the run time for a single processor has degraded nearly 30 percent. This is a big number, and it may be worthwhile to keep two versions of the code around: the version optimized for multiple processors and the version optimized for single processors. Frequently the performance degradation on a single processor is not nearly so large and is not worth the bother of keeping multiple versions around. You can simply run the multiprocessed version on a single processor. The only way to know what to keep is to run the code and time it.

Appendix A

Run−Time Error Messages

Table A−1lists possible Fortran run−time I/O errors. Other errors given by the operating system may also occur.

Each error is listed on the screen alone or with one of the following 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 Table A−1 All of the errors in the table 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.
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.
109	`unit not 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 and 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 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.

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.

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.

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.

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 specifiy 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 only be used on a formatted file.

163 `indexed files only`
Indexed I/O can only be done on logical units that have been opened for indexed (keyed) access.

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.

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= mut 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.

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.