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**.

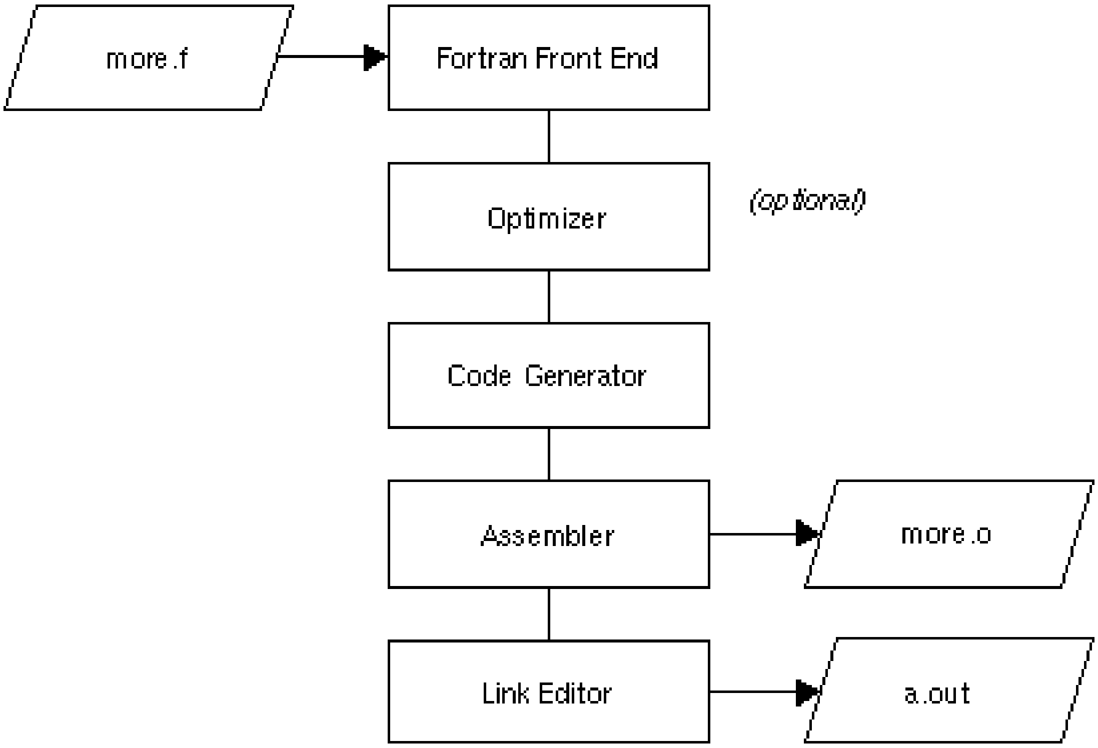

Figure 1–1 Compilation Process

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

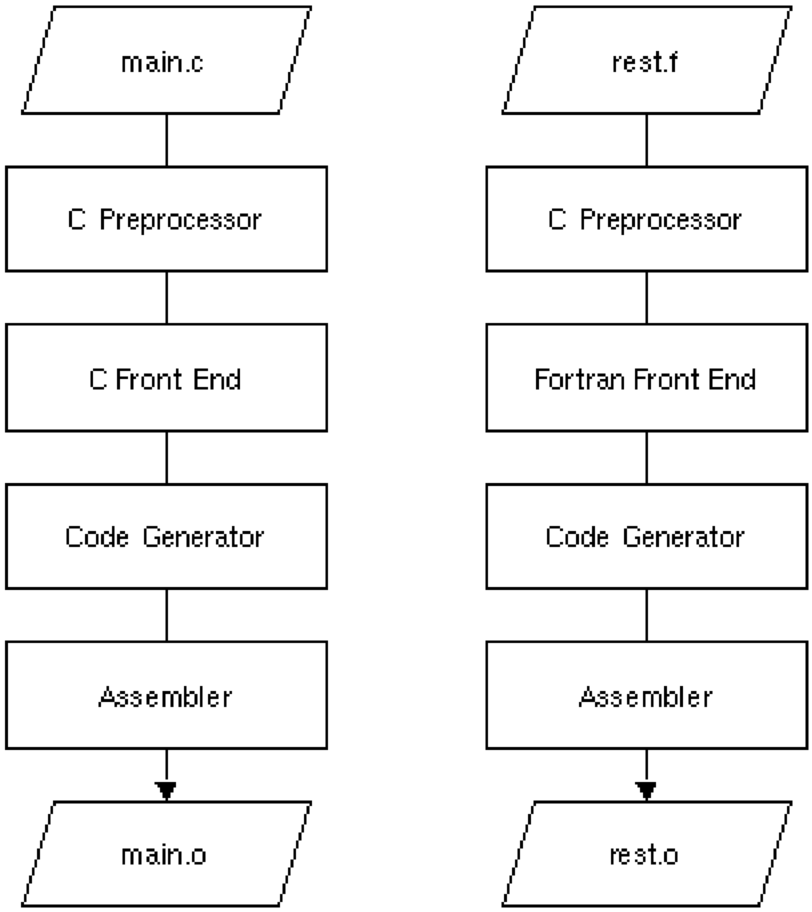

Figure 1–2 Compiling Multilanguage Programs

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.

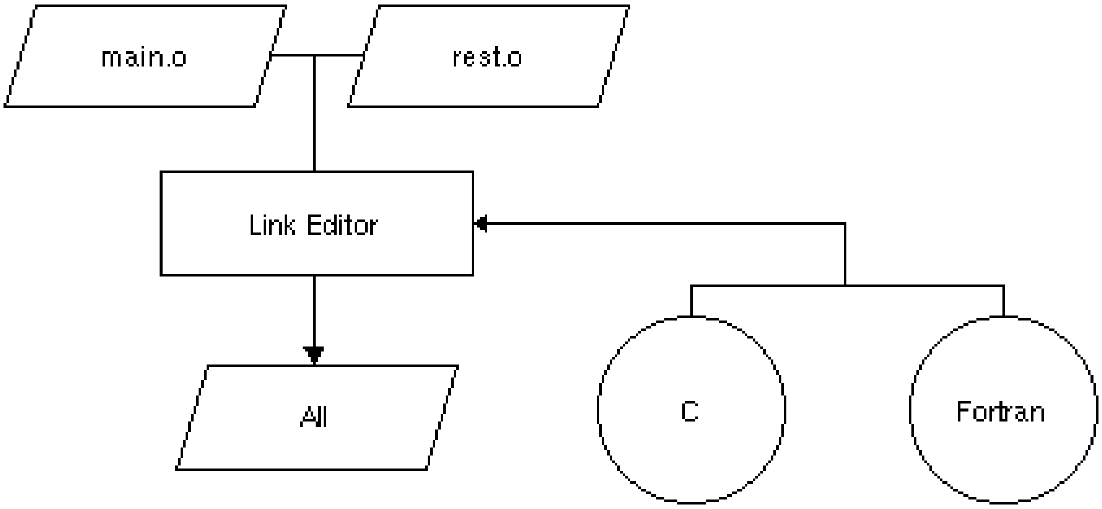

Figure 1−3 Link Editing

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 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.