Chapter 1

System and Compiler Overview

This chapter gives a brief overview of the IRIX™ 6.2 operating system and describes the MIPSpro™ 32-bit, 64-bit and high performance 32-bit (N32) compilers. It contains four sections:

- The first section lists the new features of IRIX 6.2 and describes its compatibility with IRIX 5.*x* and Irix 6.1 (see Figure 1-1).
- The second section describes the various components of both the 64-bit compiler system and the 32-bit compiler.
- The third section describes the Kuck and Associates preprocessor (KAP), a major component of the 64-bit Fortran front end.
- The last section outlines the multiprocessing support provided by the MIPSpro compilers.

IRIX 6.2 Overview

The IRIX 6.2 release is the successor to both IRIX 5.3 and IRIX 6.1 and provides a common operating system across all SGI platforms running on R4000 (rev level 2.2) processors and above. It introduces the high performance 32-bit (N32) programming environment and continues support for both 64-bit virtual addressing and the MIPS R8000 processor. N32 is described in the *N32 ABI Guide*. A brief overview of the R8000 is provided below.

The R8000 processor has the following capabilities:

- Executes MIPS1 user code, compatible with the R3000.
- Executes MIPS2 instruction set extensions introduced in the R4000.
- Executes MIPS3 64-bit addressing and instructions introduced in the R4400.
- Integrates features for floating point performance improvements and integer improvements, which are called MIPS4 instructions.

The MIPS3 and MIPS4 64-bit capabilities provide both 64-bit virtual addressing and instructions which manipulate 64-bit integer data. Processor registers are 64 bits in size.

For more information about the R8000, please consult the *MIPS R8000 Microprocessor User's Manual*.

IRIX 6.2 Objectives

The objectives for the IRIX 6.2 release are:

- Enable intensive floating point programs to compile optimized R8000 code, usually Fortran, and to link to the libraries that they need.
- Supply optimized versions of our own intensive floating point codes, to optimize R8000 performance. Note that a utility program has no need to be ported to a 64-bit version, unless it needs to exploit floating point performance.
- Provide 64-bit user processes with a virtual address space of up to one attribute (2^{40}) in size, provided that the size of physical memory combined with swap space is larger.

- Provide 32-bit user processes with compatibility to IRIX 5.*x*,
- Provide a new extended fleshiest type XFS. Under this filesystem, maximum file and filesystem size on R4x00 systems is 1 terabyte and maximum file and filesystem size on R8000 systems is 2^{63}-1 bytes. Also included are new interfaces such as **lseek64(2)** which allow 32-bit programs access to large files.

What is Not in the IRIX 6.2 Release

- There is no check-point/restart support in IRIX 6.2.
- COFF is no longer supported in IRIX 6.2.

Compatibility and Supported ABIs

With the IRIX 6.2 release on an R8000 system, you can execute programs conforming to any of the following Application Binary Interfaces (ABIs):

- An o32 program built under IRIX 5.x or IRIX 6.2 (32-bit MIPS1 or MIPS2 ABI).
- A 64-bit program built under IRIX 6.1 or IRIX 6.2 (64-bit MIPS3 or MIPS4 ABI).
- An N32 program built under IRIX 6.2 (N32 MIPS3 or MIPS4 ABI).

Figure 1-1 illustrates the ABIs supported by IRIX 6.2.

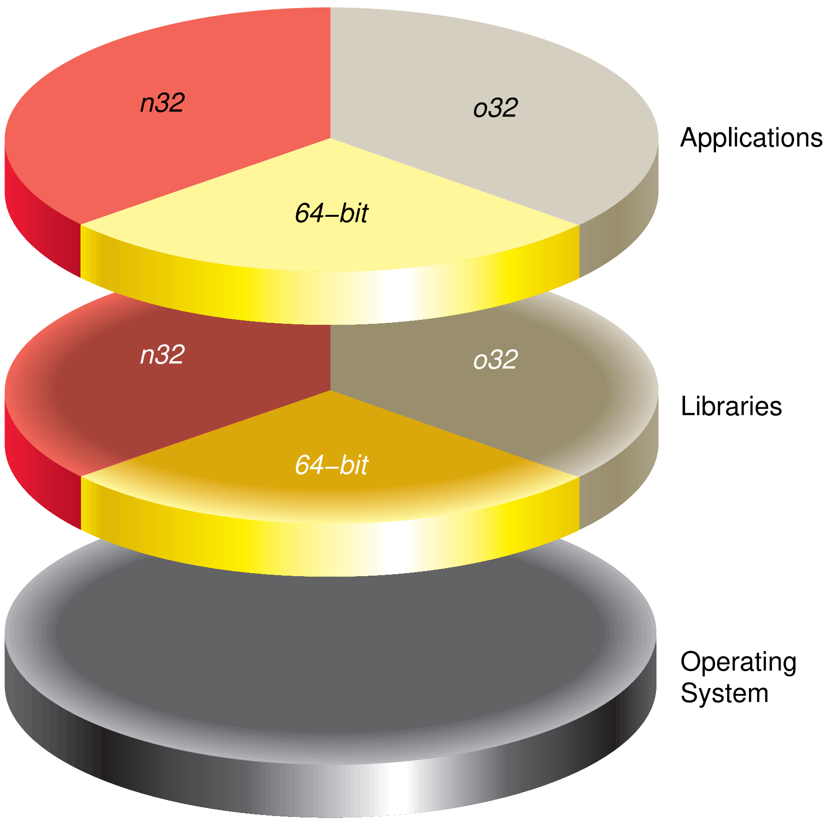

Figure 1-1 ABIs supported by IRIX 6.2

More specifically, the execution and development environments under IRIX 6.2 provide the following functionality:

- 32-bit IRIX 5.*x* binaries and Dynamic Shared Objects (DSOs) execute under IRIX 6.2
- IRIX 6.2 has a set of compilers (32-bit Native Development Environment) that generate 32-bit code. You can mix objects and link with objects produced on IRIX 5.*x*. (We do not, however, guarantee that this code runs on current IRIX 5.*x* systems.)

  For this release, this set of compilers is, in fact, the same (with minor modifications) *ucode* compilers shipped on current IRIX 5.*x* systems.
- IRIX 6.2 also has a set of compilers (64-bit Native Development Environment) that generates either 64-bit or N32 code. This code **cannot** run on current IRIX 5.x systems.
- You can specify which compiler you want to run by using the -**64**, -**n32** or -**32** flags on the compiler command line.

  The compiler driver then executes the appropriate compiler binaries and links with the correct libraries. This also applies to the assembler, linker, and archiver. If these switches are not present, the driver checks an environment variable, *SGI_ABI*, for these values.

  The default under IRIX 6.2 is -**64** for R8000 systems and -**32** for R4x00 systems.
- All of the compiler-related tools (*dbx*, *nm*, *dis*, *pixie*, *prof*) except *pixstats* can work with either 32-bit, N32 or 64-bit binaries. *pixstats* functionality for 64-bit programs is integrated into *prof*.
- You **cannot** mix objects and DSOs produced by the 32-bit compilers with objects and DSOs produced by the 64-bit compilers. In Figure 1-1, this is illustrated by the lines separating the 32-bit, N32 and 64-bit libraries.

  This means:

  – You **cannot** link 32-bit objects with 64-bit or N32 objects and shared libraries

  – You **cannot** link 64-bit objects with 32-bit or N32 objects and shared libraries

  – You **cannot** link N32 objects with 64-bit or 32-bit objects and shared libraries

- The */usr/lib* directory on IRIX 6.2 contains the 32-bit libraries and *.so*s. The 64-bit *.so*s are located in */usr/lib64*. The N32 *.so*s are located in */usr/lib32*. The complete layout looks like this:

  32-bit: This is the IRIX 5.x */usr/lib*, including compiler components:

```
/usr/lib/
          *.so
          mips2/
          *.so
/usr/lib/
          cfe
          fcom
          ugen
          uopt
          as
```

  64-bit: These are the 64-bit-specific components:

```
/usr/lib64/
          *.so
          mips3/
          *.so
          mips4/
          *.so

/usr/lib64/cmplrs
          be
          fec
          fef77
          fef90
          as
```

N32: These are the N32-specific components:

```
/usr/lib32/
          *.so
          mips3/
          *.so
          mips4/
          *.so
```

Known Compatibility Issues

The following issues are known to cause trouble for 32-bit programs running on IRIX 6.2:

- Any access to kernel data structures, for example, through */dev/kmem*. Many of these structures have changed in size. Programs making these kinds of accesses must be ported to 64-bit. 32-bit programs cannot access all of kernel memory, hence probably must also be ported to 64-bit.
- Use of **nlist()** does not work on any 64-bit *.o* or *a.out*. A new **nlist64()** is supplied for 64-bit ELF.
- Any assumption that the page size is 4Kbytes (for example, using **mmap()** and specifying the address). The page size is no longer 4Kbytes. Programs must use **getpagesize()**.
- Ada programs which catch floating point exceptions do not work.
- Any program using */proc* must have some interfaces changed.

It is possible for a program to determine whether it is running on a 64-bit capable kernel in order to work around the issues listed above. Use **sysconf(_SC_KERN_POINTERS)**, which returns 32 or 64.

Source Level System Interface

The IRIX APIs, as defined in SVID and POSIX, are unchanged. Source code requires no changes to be built **-64**. The compiled bit sizes of arguments and fields may indeed be different between the 32-bit and 64-bit definitions, but source code should not have to be changed. For example, *off_t* and *size_t* are now 64 bits when compiled **-64** and 32 bits when compiled **-32**.

The source code interface to IRIX therefore remains the same, with a few exceptions. This interface is defined by the set of header files in */usr/include*. Most of these header files have not changed. The kernel maintains compatibility by converting all system calls from a 32-bit binary into 64-bit internal kernel format. The kernel maintains all states in a superset form (usually 64 bits), converting data structures upon entry to and exit from the kernel. As a result, the kernel continues to keep traditional IRIX form and state information of processes, files, and other structures.

Graphics Environment Support

Table 1-1 summarizes the graphics environments supported under IRIX 6.2. Basically, all of the graphics environments are provided for 32-bit, N32 and 64-bit applications.

Table 1-1 IRIX 6.2 Graphics Environment Support

Package	32-bit Support	64-bit Support	N32 Support
IRIX GL (libgl)	Yes	Yes	Yes
OpenGL	Yes	Yes	Yes
X11R6	Yes	Yes	Yes
Motif1.2.4	Yes	Yes	Yes

Inter-Process Communication

Interfaces across a network connection usually allow 32-bit and 64-bit programs to communicate (for example, NFS works fine). Programs using RPC and XDR should have no problems. Of course, exchanging binary data with size assumptions which do not agree can fail.

Usually, there are no problems with having 32-bit and 64-bit programs interact via system calls. For instance, sending data through a pipe, or sharing file access, or connecting through the network, or shared memory usage, all work. In all cases, however, binary data structure sizes must agree.

Arenas are an exception (see **usinit(3)**). They are not compatible between 32-bit and 64-bit programs. Arenas can be used within a set of 32-bit programs, and within a set of 64-bit programs, but not across program types. See *test_and_set(3)* and *abilock(3)* for locks compatible with 32-bit and 64-bit programs.

Device Driver Interface

The IRIX 6.2 kernel requires MIPS3 or MIPS4 code. MIPS1 drivers cannot be linked. Additionally, the kernel is now linked in ELF format, so the **-coff** option to link 5.*x* drivers is no longer used.

The kernel/driver interface (DDI/DKI) has a few changes and some additions. Usually kernel drivers need a lot of rework due to internal assumptions of pointer sizes, page sizes, and I/O mapping sizes.

Socket **ioctl()** calls to the kernel by 32-bit programs are converted for socket **ioctls** automatically. Hence, whether a 32-bit or 64-bit program is doing the call makes no difference to the kernel.

Block and character drivers have to be responsible for their own **ioctls()**. The approach which IRIX drivers follow is to have a single header file for user code compilation, so the driver must convert fields according to differences between the 32-bit and 64-bit compilation models.

For streams, the driver will not know the user process type, so *ioctls* must be fixed binary data structures. They cannot vary between the 32-bit or 64-bit compilation models. You must fix your header files such that this is true.

Routines have been added to the device kernel interface (DKI) to aid **ioctl** conversions. Please refer to the *IRIX Device Driver Writer's Guide*.

Compiler System Components

As explained earlier, the MIPSpro compiler system on IRIX 6.2 consists of two independent compiler systems. This section describes and compares them.

Fortran

The MIPSpro Fortran 77 compilers support 32-bit, 64-bit and N32 compiler modes. The components of the two systems are described in the section that follows.

Fortran 32-Bit System

The 32-bit (*ucode)* Fortran compiler systems contain the following components:

f77 Fortran driver: Executes the appropriate components below.

cpp C preprocessor: Handles **#include** statements and other *cpp* constructs such as **#define**, **#ifdef**, and so on, in the source file.

fopt Special scalar optimizer: Performs scalar optimization on the Fortran source.

pfa Power Fortran Accelerator™: Automatically parallelizes Fortran source code and performs scalar optimizations. Available with the Power Fortran product.

fcom Fortran front end: Parses the source file into intermediate code (*ucode*).

uopt Optimizer: Performs optimizations on the intermediate file.

ugen Code generator: Generates binary assembly code from the intermediate file.

as1 Binary assembler: Assembles the *binasm* code into an object file.

ld Linker: Links the object file(s) with any libraries.

When you run simple examples through the *ucode* Fortran compilers, the following components are executed by the compiler driver:

```
%f77 -32 foo.f
    cpp --> fcom --> ugen --> as1 --> ld
```

The command

```
%f77 -32 -O foo.f
    cpp --> fcom --> uopt --> ugen --> as1 --> ld
```

also invokes the *ucode* optimizer, *uopt*. The command

```
%f77 -32 -sopt foo.f
    cpp --> fopt --> fcom --> ugen --> as1 --> ld
```

invokes the scalar optimizer but does not invoke the *ucode* optimizer.

The -**mp** option signals *fcom* to recognize inserted parallelization directives:

```
%f77 -32 -mp foo.f
    cpp --> fcom --> ugen --> as1 --> ld
```

The Power Fortran option, -**pfa**:

```
%f77 -32 -pfa foo.f
    cpp --> pfa --> fcom --> ugen --> as1 --> ld
```

invokes the Power Fortran Accelerator to convert programs to parallel code automatically.

Fortran 64-Bit and N32 System

The 64-bit Fortran compiler consists of the following components:

f77	Fortran driver: Executes the appropriate components below.
cpp	K&R C preprocessor: Handles **#include** statements in the source file.
fef77	Fortran 77 front end: Parses the source file into an intermediate representation. It also performs scalar optimization and automatic parallelization. A special version of *fef77*, *fef77p*, is available with the Power Fortran product.
fef77p	Power Fortran front end: Parses the source file into an intermediate representation. It also performs scalar optimization and automatic parallelization.
be	Back end: Generates code and assembles it into an object file. It also performs a variety of optimizations which are described in Chapter Four of this book, "Compilation Issues".
ld64	Linker: Links the object file(s) with any libraries. When you run the examples from this section on the 64-bit compiler, the following components are executed:

```
%f77 -64 foo.f
%f77 -64 -O foo.f
%f77 -64 -sopt foo.f

%f77 -64 -mp foo.f
       cpp --> fef77 --> be --> ld64
%f77 -64 -pfa foo.f
       cpp --> fef77p --> be --> ld64
```

Under the MIPSpro 64-bit compiler, scalar optimizations are performed in *fef77*. Further optimizations are performed in the back end.

Note that when -**O** is used under the 64-bit compiler, as opposed to *ucode*, **both** scalar and back end optimizations are performed.

Note also that -**O3** is available with -**c**. Unlike the *ucode* compilers, -**O3** does not result in interprocedural optimizations being performed.

The -**sopt** switch is supported on the 64-bit compiler and causes *fef77* to perform the desired scalar optimizations.

The -**mp** switch is supported on the 64-bit compiler and causes **fef77** to recognize inserted parallelization directives.

The -**pfa** switch is supported on the 64-bit compiler and causes **fef77p** to perform automatic parallelization.

C

For C, the respective compiler systems are similar to their Fortran counterparts. The front ends, of course, are different in each system. Also, *cpp* functionality was included in the C front end (*cfe*).

C 32-Bit System

The 32-bit (*ucode*) C compiler systems contain the following components:

cc	C driver: Executes the appropriate components below.
acpp	ANSI C preprocessor: Handles **#include** statements and other *cpp* constructs such as **#define**, **#ifdef**, and so on, in the source file.
cfe	C front end: Preprocesses the C file, and then parses the source file into intermediate code (*ucode*).
ccom_mp	C front end for Power C: Interprets parallel directives (K & R version).
accom_mp	C front end for Power C: Interprets parallel directives (ANSI version).
copt	C scalar optimizer: Performs scalar optimization.

pca	Power C analyzer: Automatically parallelizes C source code and performs scalar optimizations. Available with the Power C product.
uopt	Optimizer: Performs optimizations on the intermediate file.
ugen	Code Generator: Generates binary assembly code from the intermediate file.
as1	Binary assembler: Assembles the *binasm* code into an object file.
ld	Linker: Links the object file(s) with any libraries.

When you run simple examples through the *ucode* C compiler, the following components are executed by the compiler driver:

```
%cc -32 foo.c
    cfe --> ugen --> as1 --> ld
```

Note: *cfe* has a built-in C preprocessor.

The command

```
%cc -32 -O foo.c
    cfe --> uopt --> ugen --> as1 --> ld
```

also invokes the *ucode* optimizer, *uopt*.

The command

```
%cc -32 -sopt foo.c
    acpp --> copt --> cfe --> ugen --> as1 --> ld
```

invokes the scalar optimizer but does not invoke the *ucode* optimizer.

The C preprocessor has to be run before *copt* can do its source-to-source translation:

```
%cc -32 -mp foo.c
    acpp --> accom_mp --> ugen --> as1 --> ld
```

-mp signals *accom_mp* to recognize inserted parallelization directives.

The command

```
%cc -32 -pca foo.c
    acpp --> pca --> accom_mp --> ugen --> as1 --> ld
```

invokes the Power C accelerator to convert programs to parallel code automatically.

C 64-Bit and N32 System

The MIPSpro (64-bit) C compiler systems contain the following components:

cc	C driver: Executes the appropriate components below.
acpp	ANSI C preprocessor: Handles **#include** statements and other *cpp* constructs such as **#define**, **#ifdef**, and so on, in the source file.
copt	C scalar optimizer: Performs scalar optimization.
pca	Power C analyzer: Automatically parallelizes C source code and performs scalar optimizations. Available with the Power C product.
mpc	Interpreter: Interprets parallel directives.
fec	C front end: Preprocesses the C file, and then parses the source file into an intermediate representation.
be	Back end: Generates code and assembles it into an object file.
ld	Linker: Links the object file(s) with any libraries.

When you run simple examples through the 64-bit C compilers, the following components are executed by the compiler driver:

```
%cc -64 foo.c
%cc -64 -O foo.c
        fec --> be --> ld64

%cc -64 -sopt foo.c
        acpp --> copt --> fec --> be --> ld64

%cc -64 -mp foo.c
        acpp --> mpc --> fec --> be --> ld64

%cc -64 -pca foo.f
        acpp --> pca --> mpc --> fec --> be --> ld64
```

Fortran and C 64-Bit Components

Table 1-2 summarizes the major components of both the Fortran and C compilers of the new MIPSpro (64-bit and N32) compiler system.

Table 1-2 Major Components of 64-Bit and N32 Fortran and C Compilers

Fortran		C
f77	Driver	cc
cpp	Preprocessor	fec
fef77	Front end	fec
fef77	Scalar optimizer	copt
fef77	MP directive interpreter	mpc
fef77p	Automatic parallelizer	pca
be	Back end	be
ld	Linker	ld

Kuck and Associates Preprocessor (KAP)

KAP (Kuck and Associates Preprocessor) is a Fortran optimizer that analyzes data dependence to guide traditional serial optimizations and automatic parallelization. It is built into *fef77* (rather, *fef77* consists largely of KAP).

This is an example of a transformation that KAP performs. The following routine:

```
subroutine bar(a,b,c,d,j)
real*4 a(1024),b(1024),c(1024)
real*4 d,e,f

sum = 0
do m= 1,j
do i=1,1024
b(i) = b(i) * d
enddo
enddo

call foo(a,b,c)
end
```

is turned into this by the KAP:

```
# 1 "vmuls.f"

subroutine BAR (A, B, C, D, J )
real*4 A(1024), B(1024), C(1024)
real*4 D, E, F
do 2 M=1,J-3,4
do 2 I=1,1024
B(I) = B(I) * D
B(I) = B(I) * D
B(I) = B(I) * D
B(I) = B(I) * D
2        continue
do 3 M=M,J,1
do 3 I=1,1024
B(I) = B(I) * D
3        continue
call FOO (A,B,C)
end
```

As you can see, some outer loop unrolling has occurred.

Passing Options to KAP

You can pass options to KAP using -**WK,option**, for example:

-WK,-cmp=file.cmp

passes -**cmp=file.cmp** to KAP (which puts the equivalent transformed Fortran into *file.cmp*).

KAP's Optimization Flags (-o, -r, -so)

Kap accepts three important optimization flags, -**o**, -**r**, and -**so**. They are described in this section.

The -o Flag

-**o=integer**
Long name: -**optimize=integer**
The -**optimize** option sets the optimization level. Each optimization level is cumulative (that is, each level performs everything up to and including its level). The meaning of each of the different optimization levels is:

0	No optimization is done.
1	Only simple optimizations are done. Induction variable recognition is enabled.
2	Performs lifetime analysis to determine when last-value assignment of scalars is necessary.
3	Triangular loops are recognized and loop interchanging is attempted to improve memory referencing. Special case data dependence tests are used. Special index sets, called wraparound variables, are also recognized.
4	Two versions of a loop are generated, if necessary, to break a data dependence arc.
5	Array expansion and loop fusion are enabled.

The -r Flag

-r=integer
Long name: -**roundoff=integer**
The -**roundoff** option allows you to specify the tolerable change from serial roundoff error. If an arithmetic reduction is accumulated in a different order than in the scalar program, the roundoff error is accumulated differently and the final result may differ from that of the original program's output. Although the difference is usually insignificant, certain restructuring transformations performed by the KAP must be disabled to obtain exactly the same answers as those of the scalar program.

KAP classifies its transformations by the amount of difference in roundoff error that can accumulate, so you can decide what level of roundoff error difference is allowable.

The meanings of the switch values are as follows:

0	Allow no roundoff-changing transformations.
1	Enable expression simplification that might generate overflow or underflow errors differently. Enable simplification of expressions with operands between binary and unary operators. Perform expression simplification that is exposed due to forward substitution. Enable code floating if the -**scalaropt** option is greater than or equal to 1.
2	Allow loop interchanging around arithmetic reductions to be recognized.
3	Recognize real (float) induction variables. Enable sum reductions. Enable memory management optimizations if **scalaropt=3**.

The -so Flag

-so=integer
Long name: -**scalaropt=integer**
The -**scalaropt** option sets the level of dusty-deck and other serial transformations performed.

The following are possible values for **integer**:

0	No scalar optimizations are performed.
1	Only simple scalar optimizations are performed – dead code elimination, global forward substitution and dusty-deck IF transformations.
2	Full range of scalar optimizations are performed – invariant IF's are floated out of loops, loop rerolling/unrolling, array expansion, loop fusion, loop peeling and induction variable recognition.
3	Memory management optimizations are enabled if **roundoff=3**.

Opimization Levels Passed to KAP

Three optimization flags are passed to KAP; their values are dependent on the optimization level specified, as follows:

-**O0** implies -**o=0** -**r=0** -**so=0**
-**O1** implies -**o=1** -**r=0** -**so=2**
-**O2** implies -**o=5** -**r=0** -**so=3**
-**O3** implies -**o=5** -**r=2** -**so=3**
-**sopt** implies -**o=5** -**r=0** -**so=3**
-**mp** implies -**o=0** -**r=0** -**so=0** and recognize parallel directives
-**pfa** implies -**o=5** -**r=0** -**so=3** (plus automatic parallelization)

For combinations of either -**On** -**pfa** or -**On** -**sopt**, the greater of the implied options is taken. For example, with -**O1** -**sopt**, the resulting set of KAP options is -**o=5** -**r=1** -**so=3**.

Multiprocessing Support

IRIX 6.2 and the MIPSpro compilers support multiprocessing primitives for both 32-bit and 64-bit applications. The 64-bit multiprocessor programming environment is a superset of the 32-bit one. It also contains enhancements.

MP Compatibility

This section describes 64-bit and 32-bit Fortran MP compiler compatibility:

- The 64-bit Fortran compiler supports all of the parallelization directives (such as *C$DOACROSS, C$&, C$MP_SCHEDTYPE, C$CHUNK, C$COPYIN*) supported by the 32-bit Fortran compiler. *fef77* also supports PCF style parallel directives, which are documented in the *MIPSpro Fortran 77 Programmer's Guide*.
- The 64-bit Fortran compiler supports the same set of multiprocessing utility subroutine calls (such as **mp_block** and **mp_unblock**) as the 32-bit compiler.

  The 64-bit Fortran compiler supports the same set of environment variables (such as *MP_SET_NUMTHREADS* and *MP_BLOCKTIME*) as the 32-bit compiler.
- The -**mp** and -**pfa** switches are supported on both the 32-bit compilers and the 64-bit compilers.
  - -**mp** allows KAP to recognize hand-inserted parallelization directives.
  - -**pfa** enables automatic parallelization by KAP.

MP Enhancements

The MIPSpro 64-bit Fortran MP I/O library has been enhanced to allow I/O from parallel regions. In other words, multiple threads can read and write to different files as well as read and write to the same file. The latter case, of course, encounters normal overhead due to file locking.

The MIPSpro 64-bit compilers also have been enhanced to allow parallel C and parallel Fortran programs to share a common runtime. This allows you to link parallel C routines with parallel Fortran routines and have a single master. Figure 1-2 illustrates this.

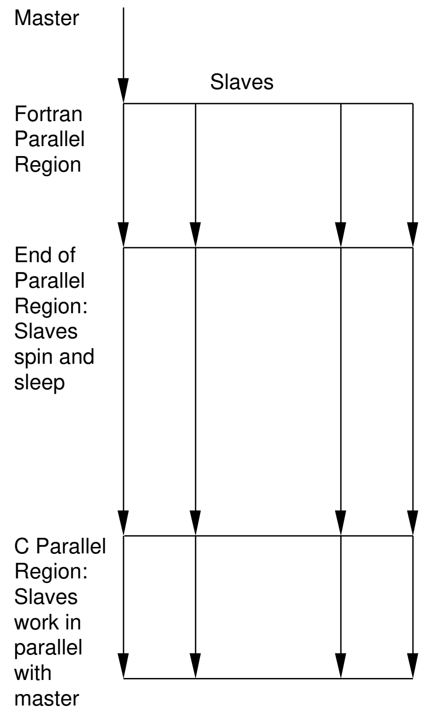

Figure 1-2 Running Parallel C and Parallel Fortran Programs Together

MP Application Testing

To test 64-bit MP applications on the R8000, follow these guidelines:

- First, get the application to run with no parallelization at the highest optimization level.
- When testing the parallel version, first run it with only one thread (either on a single CPU machine or by setting the environment variable *MP_SET_NUMTHREADS* to 1).
- Go down to the **-g** optimization level for the first MP test, and run that version with one thread, then with multiple threads. Then go up the optimization scale, testing both single and multi-threaded.

You can, of course, skip as many steps as you like. In case of failure, however, this method of incremental iterations can most likely help you identify the problem.