3 *Developing Apply Programs*

This chapter describes how to compile, link, and load an Apply procedure for parallel execution. There are two main steps in compiling an Apply procedure and linking it to a C main program. The first step is to translate the Apply procedure into a parallel representation with the Apply translator. The second step is to include the translated Apply procedure in a C main program for processing by the pre-compiler. The pre-compiler generates two cellprograms; one for the master cell and one for the slave cells. These cellprograms are compiled and linked to form executable cellprograms. Figure 3-1 illustrates these steps.

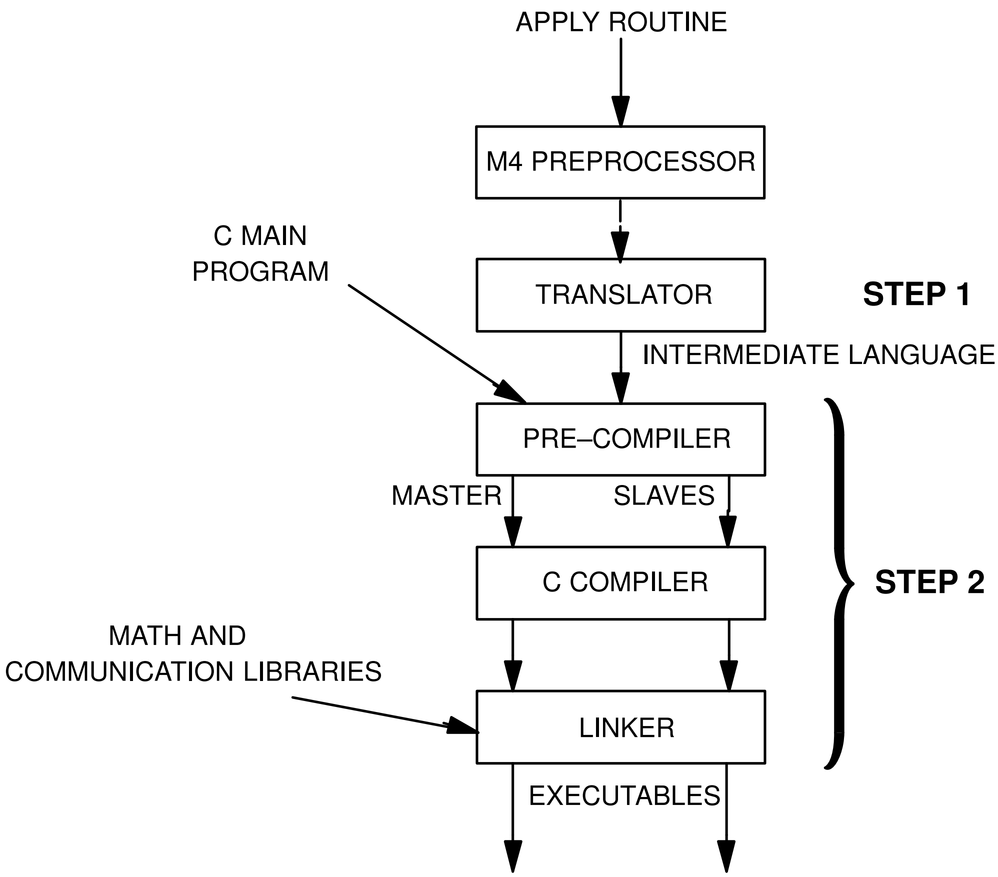

Figure 3-1: Apply Parallel Compilation and Linking Process

NOTE

All Apply procedures pass through the m4 preprocessor, allowing you to use m4 commands when writing Apply procedures.

Compiling and Linking

The apply command invokes the Apply translator. The apply command syntax for translating an Apply procedure to a parallel representation is the following:

```
apply [option ... ] procedure.app
```

procedure.app is the name of the apply procedure. The .app file extension is required and specifies that *procedure* is an apply source file. The Apply translator produces an output file *procedure*.pc.

If you do not select any options with the Apply command, the program will be compiled for a 512 x 512 image with one slave processor.

The apply command syntax to link the translated Apply procedure to a C main program and create an executable program is the following:

```
apply [option ... ] main_program.pc
```

main_program.pc is a C language main program that calls the Apply procedure. The .pc file extension is required. The main program must contain a #include statement that specifies the name of the Apply procedure file, such as #include ‘‘example.pc’’.

option can be any of the following:

-c	Create intermediate files, but do not execute final compilation.
-d	List expanded commands for intermediate files. This option is used with the -x or -y option.
-g	Insert calls to debugging software. The default is no debugging calls.
-h <*value*>	Specify *value* iterations down rows in image. The default is 512. If you are not using SAMPLE (refer to Chapter 2), *value* would be the number of rows in the image. If you are using SAMPLE, *value* would be the number of rows divided by the sample size.
-I*pathname*	Add *pathname* to the list of directories in which to search for #include files with relative pathnames. This option follows Unix conventions.
-k	Retain intermediate files in the current directory. By default, intermediate files are deleted.
-l*x*	This option is an abbreviation for the library name lib*x*.a, where *x* is a string. This option is used with the -L option. This option follows Unix conventions.
-L*dir*	Add *dir* to the list of directories containing object library routines. Directories specified with -L are searched before the standard directory, /iwarp/lib. This option only affects -l*x* options which follow on the command line. This option follows Unix conventions.
-n <*value*>	Specify *value* cells. The default is 2 cells. You must specify the total number of cells in the iWarp array.
-s	Execute sequential compilation. The default is parallel compilation.

-S	Compile with the Sun host compiler rather than the iWarp compiler.
-v	Specify iWarp architecture version. The default is 3.2.
-V	Print version information for each tool and perform compilation as specified.
-VV	Print version information for each tool but do not perform compilation.
-w <*value*>	Specify *value* iterations across columns in image. The default is 512. If you are not using SAMPLE (refer to Chapter 2), *value* would be the number of columns in the image. If you are using SAMPLE, *value* would be the number of columns divided by the sample size.
-W*c*, *arg1* [, *arg2*...]	Pass any specified arguments to pass *c* of the compiler chain, where *c* is one of the following: p preprocessor 0 compiler first pass q PCO pass 1 compiler second pass 2 PCI a assembler l link editor
-x	List the expanded compilation commands to the screen before executing the commands.
-y	List the expanded compilation commands to the screen, but do not execute the commands.
-Y*c dir*	Get Apply facility *c* from directory *dir*, where *c* is one of the following: a .app to .pc compiler c .pc to .c compiler d data movement software p pathlib software t direct files to temporary directory *dir*

You can combine the translation and linking steps into a single command by typing the following:

```
apply [options] procedure.app main_program.pc
```

CAUTION

If the C main program and the Apply procedure have the same name, the main program will be replaced by the translated Apply procedure. In the following example the Apply translator creates the file badexample.pc from the Apply procedure badexample.app. This file will replace the C main program of the same name, and the main program will be lost.

```
apply badexample.app badexample.pc
```

Executing

The arun command executes the program. The syntax for the arun command is the following:

```
arun [option ...] main_program
```

main_program is the name of the C main program, without the .pc file extension.

option can be any of the following:

-sib <*number*>	Specifies <*number*> as the Sun Interface Board. The default is 0.
-x	Lists the expanded execution commands to the screen before executing the commands.
-y	Lists the expanded execution commands to the screen, but does not execute the commands.

An Apply Example

The following is an example of a C main program that calls the Apply procedure addconst to add a constant to each element of a two-dimensional array. The main program is stored in a file called example.pc. Note that the main program contains an include statement for the Apply procedure addconst.pc.

```
#include "addconst.pc"

#define   HEIGHT   512
#define   WIDTH    512

main ()
{
   int i;
   int constant;
   static  unsigned char a[HEIGHT][WIDTH];
   static  unsigned char c[HEIGHT][WIDTH];

   /* create the data */

   for (i=0; i<HEIGHT*WIDTH; i++)
      a[i/WIDTH][i%WIDTH] = i;

   printf("\nInput data\n\n");

   for (i=0; i<HEIGHT*WIDTH; i++)
      printf("%d %s", a[i/WIDTH][i%WIDTH], (i+1)%WIDTH?"\t":"\n");

   constant = 3;

   addconst(a, constant, c);

   printf("\nAdd 3 ...\n\n");

   for (i=0; i<HEIGHT*WIDTH; i++)
      printf("%d%s",  c[i/WIDTH][i%WIDTH], (i+1)%WIDTH?"\t":"\n");

}
```

The following is the Apply procedure addconst:

```
procedure addconst(   image1   : in byte,
                      constant : const integer,
                      image2   : out byte)
```

```
is
begin
   image2 := image1 + constant;

end addconst ;
```

Parameters *image1* and *image2* are two-dimensional arrays of unsigned bytes that are received from and returned to the calling program. The parameter *constant* is a scalar passed by value from the calling program. The procedure adds the value of *constant* to each element of *image1*, storing the results in the corresponding element of *image2*.

The Apply procedure is stored in a file named addconst.app. To translate the Apply procedure to a parallel representation, execute the apply command as follows:

```
apply addconst.app
```

To link the Apply procedure to the main program and compile them into an executable master/slave pair of cellprograms, execute the apply command as follows:

```
apply example.pc
```

This command generates a master cellprogram called example_m.c and a slave cellprogram called example_s.c. These cellprograms are compiled and linked to form executable cellprograms. The command also generates a file called example, which defines the cells on which the master and slave programs execute.

To execute the program, type the following:

```
arun example
```

This command executes the program on the array and prints the results.

NOTE

> Apply places its intermediate files in /tmp. On some systems this area is small and may run out of space, causing a compilation failure. You can redirect intermediate files to an alternate directory with the –Y option of the Apply command.

Sequential Compilation

When developing a program, you may want to first compile it as a sequential program for testing and debugging. To compile a sequential program, use the -s option as follows:

apply –s *procedure*.app *main_program*.pc

You can also use the -S option with the -s option, specifying that compilation is done with the Sun host compiler rather than the iWarp compiler.

The resulting program can be run on the Sun host, giving you access to dbx and dbx tools to debug the program, if you also use the -g option to generate symbols.