Chapter 12

Sample DPEAC Routine

This chapter presents a simple example of a DPEAC subroutine, and shows how to call it from a program written in a higher-level language (in this case, a CM Fortran program).

This is the best way to make use of DPEAC: letting the high-level language compiler manage the details of memory management and data layout, while you concentrate on writing short DPEAC subroutines for the parts of your program that require large amounts of efficient computation.

The program described in this chapter uses a DPEAC routine to calculate a specific formula, applied elementwise to a set of array arguments. The formula is:

$$d = \frac{b^2 + c}{\sqrt{3.69a + 25.0b}}$$

Each of the letters *a*, *b*, *c*, and *d* represents a corresponding array element taken from a CM Fortran array that is passed as an argument to the DPEAC routine.

The CM Fortran program that calls this DPEAC routine handles the allocation of the arrays, and the subsequent processing of the results produced by the DPEAC subroutine.

Important: You do *not* have to structure your programs as shown in this chapter to make use of the CM-5's Vector Units. The CM Fortran compiler itself defines DPEAC routines in the process of compiling standard CM Fortran programs, and thus implicitly uses the Vector Units whenever they are needed. The methods shown in this chapter allow you to duplicate the compiler's work for specific routines that you choose to write yourself.

12.1 Component Files of the Program

To include a DPEAC subroutine in a program, four separate files of code are required (see Figure 11, below):

- The DPEAC subroutine itself, which executes in multiple copies on the nodes. (The DPEAC routine on each node handles that portion of the array arguments that is stored in the node's memory.)
- The *node interface* function, which calls the DPEAC routine. The node function is paired with a similar function on the partition manager (PM).
- The *host interface* function on the PM, which broadcasts a call to the node interface functions on all the nodes.
- The *main program*, written in a high-level language (such as CM Fortran), which calls the host interface function to invoke the DPEAC routine.

This chapter presents examples of each of the above routines, along with a sample "`make`" file that is used to compile and link them into a working program.

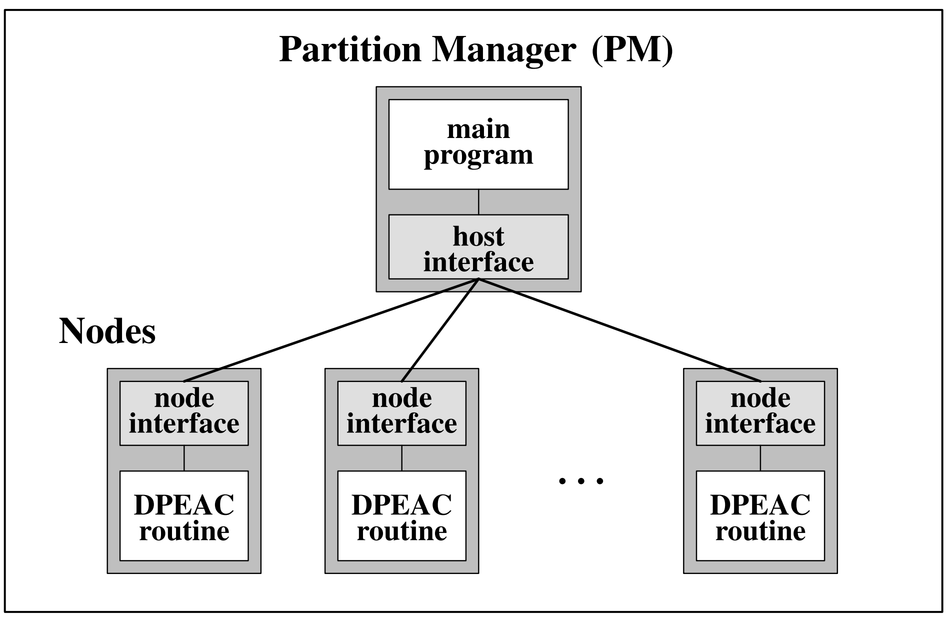

Figure 11. Relationship of code components in a DPEAC program

Note: If you have more than one DPEAC routine in a program, you do not need a separate set of interface files for each routine — you can simply add the additional interface and DPEAC code to the files described above.

For the Curious: There are a number of reasons for the separation of the host and node interface files from the rest of the program:

- The interface files describe the relationship between a specific set of function calls made on the partition manager, and a specific set of functions that are defined on the nodes. The interface code files provide the “glue” that allows these function calls and definitions to compile and link correctly.
- Since the host and node interface functions are required by different components of the CM (the partition manager and the nodes, respectively), they must be in separate files to be compiled properly.
- Finally, the pre-processing utilities applied to the interface files require that the interface files contain C code. Thus, the interface files must by definition be processed and compiled separately from the main program (which is written in CM Fortran) and from the DPEAC routine (which is written in assembly code).

12.2 Host/Node Interface Naming Conventions

The compilation of a CM program that includes a DPEAC subroutine uses many of the same mechanisms that are used in the compilation of a standard high-level CM program. As a result, there are some restrictions that must be observed in the naming of the interface files and functions. (See Figure 12.)

- The host interface function, as called from the main program, can have any name. However, its name as defined in the host interface file must be the name used in the CM Fortran program in *all lower case* and with a trailing underscore character.
- Likewise, the node interface function can have any name in the host interface file. However, its name as defined in the node interface file must be the host interface name with the prefix “**`CMPE_`**” attached.

- The DPEAC routine name must begin with an underscore character. Otherwise, the routine can have any name, although it is typical for the DPEAC routine name to be the same as the node interface name (with the underscore prefix added).

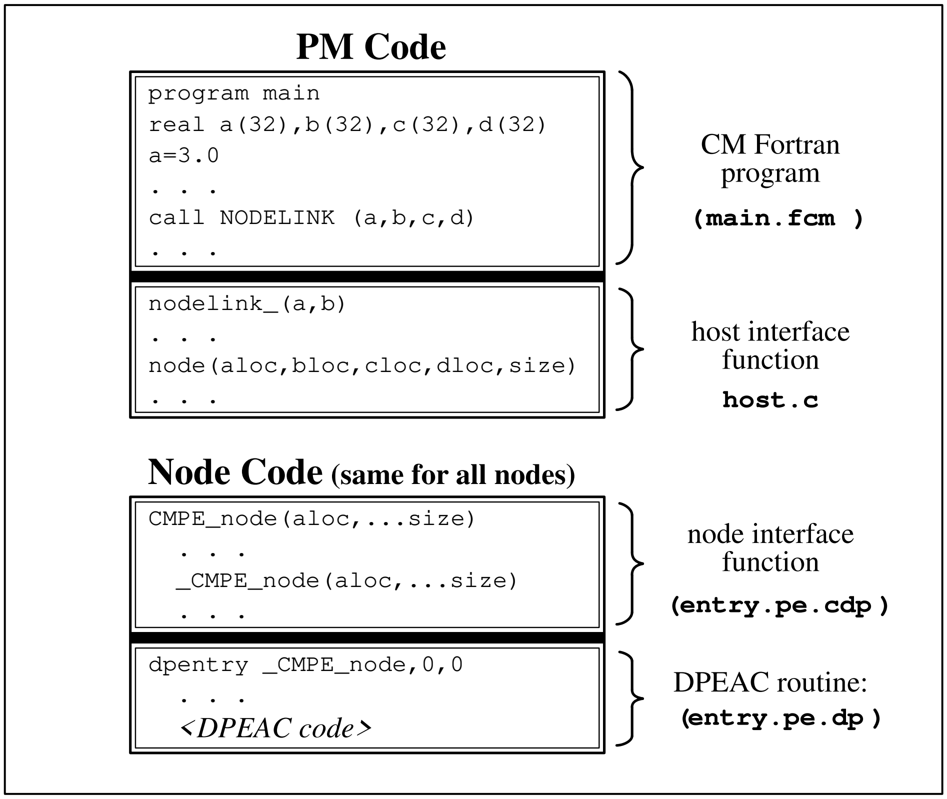

Figure 12. Names and contents of files in the sample program

These restrictions are imposed by the compilation mechanism — special prefix and case requirements allow the compiler and linker to easily determine which functions in the host and node interface files correspond to which routines in the main program and the DPEAC code file.

The "**CMPE_**" prefix for the names of node functions callable from the host has the additional purpose of making it unlikely that a random function call from the main program would happen to match a function defined in the node interface.

12.2.1 Makefile Restrictions

The "**make**" file used to compile this sample program imposes some restrictions on the names of the files. It expects the names of the code files to be as follows:

Main program:	**main.fcm**
Host interface code:	**host.c**
Node interface code:	**entry.pe.cdp**
DPEAC routine:	**entry.pe.dp**

You can assign virtually any filenames you choose for these programs, so long as your compiling and linking scripts (or makefiles) know the correct names of the files. The node interface file and DPEAC code file, however, must have the same name, with extensions of "**.pe.cdp**" and "**.pe.dp**" respectively. This restriction is again imposed by the methods used to compile and link these files.

12.3 Array Access in CM Fortran

In CM Fortran, the name of an array is not the address of the array data itself, but is instead a pointer to a data structure known as an *array descriptor*. This descriptor contains, among other things, the address of the start of the array and the number of elements in the array. The array descriptor is stored on the *partition manager*, and the array location it contains is a memory address in *node* memory. Thus, part of the job of the host interface function will be to extract the appropriate information from the descriptors of any array arguments it receives, and pass this information on to the node interface function.

The contents of an array descriptor can be extracted by calls to special accessor functions that are part of the CM Run-Time System (CMRTS). For example:

```
CMCOM_cm_address_t CMRT_desc_get_cm_location (arr_desc);
int CMRT_desc_get_subgrid_size(arr_desc);
      CMRT_desc_t arr_desc;
```

CMRT_desc_get_cm_location returns the starting address (in node memory) of the array described by **arr_desc**.

CMRT_desc_get_subgrid_size returns the number of elements of the array that are stored in the memory of each VU (the "subgrid size" of the array). This value is required by the DPEAC routine, which must determine how many memory locations to operate on.

12.4 The Main CM Fortran Program (main.fcm)

The CM Fortran program, **main.fcm**, does three things:

- It initializes its arrays by assigning

  a = 3.0
  b = *random numbers between* 0.0 *and* 1.0
  c = 19.0
  d = 0.0

- It evaluates the formula **d = (b*b+c)/sqrt(3.69*a+25.0*b)** twice: first by a ordinary CM Fortran expression (which is internally compiled into DPEAC code by the CM Fortran compiler), and second by a call to the host interface function **NODELINK**.

- For each of the above methods, the program prints out the computed results, to demonstrate that the two methods do return the same values.

```
      program main
      real a(32),b(32),c(32),d(32)
      a=3.0
      b=0.0
      call CMF_RANDOM (b)
      c=19.0
c
      d=0.0
      d=(b*b+c)/sqrt(3.69*a + 25.0*b)
      do 10 i=1,32
      print  20, a(i),b(i),c(i),d(i)
 10   continue
 20   format (f10.2,f10.2,f10.2,f10.2)
c
      print *,'dpeac output follows'
      print *,''
      d=0.0
      call NODELINK (a,b,c,d)
      do 30 i=1,32
      print  40, a(i),b(i),c(i),d(i)
 30   continue
 40   format (f10.2,f10.2,f10.2,f10.2)
      stop
      end
```

12.5 The Host Interface File (host.c)

The host interface file contains a single function, **`nodelink_`**, which does three things:

- It calls **`CMRT_desc_get_cm_location`** once for each of the array arguments to get the actual node memory locations of the arrays.
- Because all the array arguments must have the same size and shape, the host interface function calls **`CMRT_desc_get_subgrid_size`** just once to get the subgrid size of the array arguments.
- Finally, the host interface function makes a call to the corresponding **`node`** function to execute the DPEAC routine.

```
#include <cm/cmrt.h>
#include <cm/cmrt_types.h>
void nodelink_ (a,b,c,d)
  CMRT_desc_t  a,b,c,d ;
  {
   CMCOM_cm_address_t aloc,bloc,cloc,dloc;
   int size;
   aloc=CMRT_desc_get_cm_location (a);
   bloc=CMRT_desc_get_cm_location (b);
   cloc=CMRT_desc_get_cm_location (c);
   dloc=CMRT_desc_get_cm_location (d);
   /* size is same for all arrays*/
   size=CMRT_desc_get_subgrid_size (a);
   node (aloc,bloc,cloc,dloc,size);
  }
```

For the Curious: As part of the compilation process, a *stubs* file, called **`pe-call.c`**, is generated. This file contains a stub function (named **`node`** in this case) that uses RTS calls to send its arguments (**`aloc`**, **`bloc`**, etc.) to the function on each node named **`CMPE_node`**.

12.6 The Node Interface File (entry.pe.cdp)

The node interface file contains one node interface function, **CMPE_node**. This function's role in the program is fairly minor. It provides a convenient entry-point for the function call made by the host interface function.

CMPE_node takes the array addresses and subgrid size provided by the host function and passes them directly to the DPEAC routine.

```
void CMPE_node (aloc,bloc,cloc,dloc, size)
  unsigned aloc,bloc,cloc,dloc, size;
  {
    _CMPE_node(aloc,bloc,cloc,dloc, size);
  }
```

12.7 The DPEAC Routine File (entry.pe.dp)

The DPEAC code file contains the DPEAC routine that drives the vector units. It contains both SUN–4 assembly language statements and DPEAC statements:

```
#include <cmsys/dp.h>
#include <cmsys/dpeac.h>

      dpentry _CMPE_node,0,0  ! entry point for DPEAC routine

! Function arguments are, by convention, found in the SPARC
! "input" registers i0, i1, etc. These constants provide
! symbolic names for these registers:

#define A i0      ! Note that the "%" prefix for SPARC
#define B i1      ! register names is not included here,
#define C i2      ! but is used explicitly in code to make
#define D i3      ! the distinction between SPARC and VU
#define Size i4   ! registers clear in usage.

! By default, CM Fortran sets the vector length to 8,
! and the vector mask mode to the condition "always".
! Calling the following macro assures any changes are reset;
! whenever it is not needed, it is simply redundant:

      set_vector_length_and_vmmode 8, always

#define VECTOR_LENGTH 8
```

```
!The formula being evaluated is
!         d=(b*b+c)/sqrt(3.69*a + 25.0*b)

Label1:
      floadv [%B]:4, V2   ! note memory stride is 4 bytes
      add %B,32,%B        ! mem pointer incr by 4*8 bytes
      floadv [%C]:4, V3; fmadav V2, V2, V3    ! b**2+c in V3
      add %C,32,%C
      floadv [%A]:4, V4; fmulv V4, 0r3.69, V5 ! 3.69a in V5
      add %A,32,%A
      fmadav V2, 0r25.0, V5  ! entire denominator in V5
      fisqtv V5, V5          ! inverse square root
      fmulv V5, V3, V5       ! final result in V5
      fstorev [%D]:4, V5  ! note memory stride is 4 bytes
      addcc %Size,-VECTOR_LENGTH,%Size ! More to strip mine?
      bne Label1           ! If yes, go repeat
                      ! assuming array size
                      ! is multiple of vector
                      ! length.
      add %D,32,%D    ! delay slot used to incr reg
      dpretn
```

A few notes on the structure of this program:

- Note that the **floadv** and **fstorev** instructions specify the memory stride to 4 within the instruction. An option would have been to use the macro **set_mem_stride**.

- CM Fortran sets the following VU Control registers as defaults:

dp_vector_length	8
dp_stride_rs1	0
dp_stride_memory	0
dp_vector_mask_mode	always
dp_vector_mask_direction	right
dp_alu_mode_round	nearest, even for tie
dp_alu_mode_fast	fast, not IEEE

  Nevertheless, it is always a good precaution to set these registers to the values you require within your DPEAC code routines, to avoid unnecessary surprises should these defaults change.

- Note that the array size is assumed to be a multiple of 8. Since the vector length is set to 8, there is no remainder, or "tail" of leftover elements. To handle a more general case, any "tail" of remaining values would need to be processed in a separate section of code, by resetting the vector length to the tail length, and repeating the calculation just once for the tail values.

12.8 Makefile for the Sample Program

Below is a makefile that can be used to compile and link the program described above. It performs a number of different operations to bring the pieces of the program together:

- The main CM Fortran program is compiled by **cmf** to produce two object files, one for the PM (**main.o**) and one for the nodes (**main.pe.o**).
- The host interface program is compiled by **cc**, producing an object code file (**host.o**) for the PM.
- The node interface program is compiled by **dpcc** and then passed through a filtering utility to produce an object code file (**pe-call.o**) for the PM.
- The DPEAC routine is assembled by **dpas**, producing a node object file (**entry.pe.o**).
- Finally, the various object code files are linked together (again by **cmf**) to produce a single executable file, **run**.

The executable file **run** can be called from the partition manager of a CM-5 to execute the program.

```
####################################
## Uses CMF 2.0 compiler          ##
## Executable module called "run" ##
####################################

CP_OBJS       =   main.o host.o pe-call.o
PN_OBJS       =   main.pe.o entry.pe.o

STUB_CP       =   pe-call.c
STUB_PN       =   entry.pe.cdp
TARGET        =   run

HOST          =
OSROOT        =
RTSROOT       =
FROOT         =
CDPROOT       =
```

```
include            $(CDPROOT)/usr/doc/cdpeac_makefile
INC_DIR        =   -I$(CDPROOT)/usr/include     \
                   -I$(RTSROOT)/usr/include\
                   -I$(OSROOT)/usr/include

DEBUG          =
OPT            =   -O

CMF            =   cmf2.0
CMFFLAGS       =   -cm5     \
                   -vu      \
                   $(DEBUG)\
                   $(OPT)       \
                   -Zcmld -Bstatic

CFLAGS         =   $(INC_DIR)   \
                   -DCM5_DASH   \
                   $(DEBUG)\
                   $(OPT)
CC             =   cc

DPFLAGS        =   -t
DPAS           =   $(OSROOT)/usr/bin/dpas

DPCFLAGS       =   $(CFLAGS) $(DPFLAGS) -dpas $(DPAS)
DPCC           =   $(CDPROOT)/usr/bin/dpcc

COMPILE.dp     =   $(DPAS) $(DPFLAGS)
MKSTUB         =   $(RTSROOT)/usr/bin/mkpestubs

SUFFIXES  +=  .fcm

.SUFFIXES:
.Suffixes:$(SUFFIXES)

.fcm.o:
      $(HOST) $(CMF) $(CMFFLAGS) -c $<

SUFFIXES  +=  .dp

.SUFFIXES:
.SUFFIXES:$(SUFFIXES)

COMPILE.dp=   $(DPAS) $(DPFLAGS)
OUTPUT_OPTION = -o $@
```

```
.dp.o:
      $(COMPILE.dp) $(OUTPUT_OPTION) $<

$(TARGET) :   $(CP_OBJS) $(PN_OBJS)
      $(HOST) $(CMF) $(CMFFLAGS) $(CP_OBJS) $(PN_OBJS) \
              $(LIB_DIR) -o $(TARGET)

$(STUB_CP):   $(STUB_PN)
      echo '#include <cm/cmcom_types.h>' > $@
      $(DPCC) -E $(INC_DIR) $? > .make.temp
      $(MKSTUB) .make.temp -n >> $@
      rm .make.temp

clean : rm -f $(TARGET) *.o *.i *.join *.s $(STUB_CP)
```

12.9 Sample Run of the Program

Here is a UNIX session in which the sample program is built and run. (The following assumes that you have logged onto the partition manager of a CM-5, and are currently in a directory containing the four code files and the makefile.)

```
% make
cmf2.0 -cm5 -vu  -O -Zcmld -Bstatic -c main.fcm
cmf [CM5 VecUnit 2.0 Beta 2]
Compiling main.fcm

cc -I/usr/include  -I/usr/include  -I/usr/include
-DCM5_DASH  -O  -target sun4 -c -o host.o host.c

echo '#include <cm/cmcom_types.h>' > pe-call.c
/usr/bin/dpcc -E -I/usr/include  -I/usr/include
-I/usr/include entry.pe.cdp > .make.temp
/usr/bin/mkpestubs .make.temp -n >> pe-call.c
rm .make.temp
cc -I/usr/include  -I/usr/include  -I/usr/include
-DCM5_DASH  -O  -target sun4 -c -o pe-call.o pe-call.c
/usr/bin/dpas -t -o entry.pe.o entry.pe.dp

cmf2.0 -cm5 -vu  -O -Zcmld -Bstatic main.o host.o
pe-call.o main.pe.o entry.pe.o -o run
cmf [CM5 VecUnit 2.0 Beta 2]
Linking...done.
```

```
% run

     3.00      0.72     19.00      3.62
     3.00      0.79     19.00      3.53
     3.00      0.79     19.00      3.54
     3.00      0.18     19.00      4.82
     3.00      0.68     19.00      3.67
     3.00      0.41     19.00      4.15
     3.00      0.69     19.00      3.66
      . . .  < other values omitted >
     3.00      0.91     19.00      3.41
     3.00      0.45     19.00      4.06
     3.00      1.00     19.00      3.33

dpeac output follows

     3.00      0.72     19.00      3.62
     3.00      0.79     19.00      3.53
     3.00      0.79     19.00      3.54
     3.00      0.18     19.00      4.82
     3.00      0.68     19.00      3.67
     3.00      0.41     19.00      4.15
     3.00      0.69     19.00      3.66
      . . .  < other values omitted >
     3.00      0.91     19.00      3.41
     3.00      0.45     19.00      4.06
     3.00      1.00     19.00      3.33

 FORTRAN STOP
```