Chapter 18

Array Homes and Argument Passing

This chapter discusses some aspects of array storage in CM Fortran, including layout considerations for CM array argument passing. This chapter assumes some familiarity with compiler directives as described in Appendix A that influence the layout of arrays in memory.

18.1 Determination of the Processor Home of an Array

The general principle of data allocation in CM Fortran is that data used in array operations resides in distributed memory ("CM memory"), while scalar data and arrays used only in scalar operations are stored in the memory of the control processor ("front-end memory"). Recall that the control processor may be the partition manager in a global execution model, or it may be each of the nodes in the nodal or global/local execution models.

An array is a *front-end array* if storage for it is allocated *only* on the control processor. An array is a *CM array* if storage for it is allocated in the memory of the parallel processors. The system component on which an array primarily resides is called its *home*. The home of an array is determined by its declaration (in a program unit or interface block), or by its use, or its home can be designated explicitly using compiler directives.

The compiler switch `-list` causes a listing to be produced that indicates the home of each array in a program unit. See the *CM Fortran User's Guide* for more information.

NOTE: The presence or absence of an initialization expression in a type declaration for an array does not affect the home of that array. For example, a declaration such as

```
INTEGER, DIMENSION(10), DATA :: INDEX = [1:10]
```

provides an initial value for the array `INDEX`, but does *not* force it to be a CM array. (Section 5.4.3 describes this form of type declaration statement.)

Copyright © 1989–1994 Thinking Machines Corporation

18.1.1 Array Home Rules for Local Arrays

The home of a noncharacter *local* array (an array not declared in a common block) is determined by its declaration or its use, unless compiler directives or switches force it to a particular home. The rules below apply for any local array; the rules for common arrays are given in Section 18.1.2. These rules should be applied in numerical order.

1a. A character array `A` is a front-end array.

1b. A deferred-shape (allocatable or pointer) array `A` is a CM array.

1c. An assumed-shape dummy array `A` is a CM array.

1d. A deferred-shape (pointer) dummy array `A` is a CM array.

1e. An automatic array `A` is a CM array.

1f. The result of an array-valued function is a CM array.

2. A local array may be forced to a CM or a front-end home using directives. A `LAYOUT` directive of the form

   `CMF$ LAYOUT A (` *axis-descriptor(s)* `)`

   forces array `A` to be allocated in front-end memory if the *axis-descriptor(s)* list specifies that all its axes are to have `:SERIAL` ordering; `A` is allocated in CM memory if one or more of its axes is given `:NEWS` ordering.

   A `LAYOUT` directive of the form

   `CMF$ LAYOUT A (` *detailed-axis-descriptor(s)* `)`

   forces array `A` to be allocated in CM memory. The detailed axis descriptors are `:BLOCK=:PROCS=` and `:BLOCK=:PDESC=`.

   An `ALIGN` directive of the form

   `CMF$ ALIGN A (` *index-name(s)* `) WITH` *target-spec*

   forces `A` to be allocated in CM memory if the target array is, and in front-end memory otherwise.

3. The home of local arrays that are not constrained by `LAYOUT` or `ALIGN` directives (rule 2) depends on how they are used. A local array `A` is a front-end array if and only if the following two statements hold:

   - The *un*subscripted references to `A` are limited to its use as an argument to external procedures, statement functions, or the array inquiry intrinsic

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

functions (LBOUND, UBOUND, SHAPE, SIZE, and RANK), or as an I/O list item in an input/output statement.

- The *sub*scripted references to A involve only *scalar* subscripts. (Section subscripts, including vector-valued subscripts, are not considered scalar subscripts.)

All other references cause A to be allocated in CM memory.

Rule 3 applies to *all* scalar-subscripted uses of an array name, including its subscripted use in a FORALL statement. For example,

```
FORALL ( I=1:N ) V(I) = I
```

does *not* make V a CM array, since V is referenced only with scalar subscripts, while the section subscript of A

```
FORALL ( I=1:N ) A(I,:) = I
```

causes A to be a CM array. A FORALL index name is considered to be a *scalar* that takes on a range of values in some unspecified order.

18.1.2 Array Home Rules for Arrays in Common

For an array declared in a common block, the following rules determine whether the array is allocated in CM memory or front-end memory. The rules should be applied in numerical order; rules 1a and 1b are mutually exclusive, as are rules 3a and 3b.

0. All (arrays of all) common blocks of a program unit default to a CM home with no front-end initialization memory allocated, unless one of rules 1a through 3b apply.

1a. A character array A in a common block is a front-end array.

1b. The LAYOUT and ALIGN directives may be used to constrain a (noncharacter) array in common to a particular home. See rule 2 of the previous section for details.

2. The COMMON directive may be used to cause all arrays of a specified common block to default to a particular home. The directive affects the home of only those arrays of the common block that are not already constrained by LAYOUT or ALIGN directives (rule 1b).

Copyright © 1989–1994 Thinking Machines Corporation

A directive of the form

```
CMF$  COMMON FEONLY /  block-name  /
```

makes allocation in front-end memory only the default for unconstrained arrays in the specified common block, thereby allowing those arrays to be initialized in a `BLOCK DATA` program unit. All arrays of the specified common block that are subsequently constrained to a CM home by an `ALIGN` or `LAYOUT` directive will retain the front-end initialization memory allocated by this directive.

The forms

```
CMF$  COMMON  CMONLY / block-name /
CMF$  COMMON         / block-name /
```

both make allocation in CM memory only the default, with no provision for static initialization. CM-only allocation is the usual default, and so this form of the directive is unnecessary except to override the effect of the `-common_initialized` switch (see rule 3b below).

The last form of `COMMON` directive

```
CMF$  COMMON INITIALIZED  / block-name /
```

makes CM memory allocation *with* front-end initialization memory the default for all unconstrained arrays in the specified common block, thereby allowing those CM arrays to be initialized in a `BLOCK DATA` program unit.

3a. The `-fecommon` switch may be used to cause all unconstrained arrays of all common blocks declared in a program unit to reside in front-end memory only.

NOTE: This switch affects only those arrays of the common block that are not already constrained by `LAYOUT`, `ALIGN`, or `COMMON` directives (see rules 1 and 2).

3b. The `-common_initialized` switch may be used to cause all unconstrained arrays of all common blocks declared in a program unit to be CM arrays by default *with* front-end initialization memory allocated for all arrays in the common block. The switch affects only those arrays of a common block that are not already constrained by `LAYOUT`, `ALIGN`, or `COMMON` directives (see rules 1 and 2).

This switch should be used when compiling any program unit that declares a common block that is *not* named in a `COMMON INITIALIZED` directive but which contains one or more CM arrays that must be statically initialized.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

NOTE: The home of a common array initialized in a BLOCK DATA program unit is *not* affected by the syntactic form of the initialization. In the type specification statement

```
INTEGER, DIMENSION(3), DATA :: X = [10,20,30]
```

the initialization of array X has the form of an array assignment, but its effect is no different from that achieved by a DATA statement.

The following BLOCK DATA program unit illustrates the use of all combinations of directives on common blocks and arrays.

```
      BLOCK DATA
      COMMON /COM1/ CM1(5), FE1(5)
CMF$  COMMON CMONLY /COM1/
CMF$  LAYOUT CM1(:NEWS)
CMF$  LAYOUT FE1(:SERIAL)  ! overrides COMMON directive
      DATA CM1 / 10, 11, 12, 13, 14 / ! WARNING: has no effect
      DATA FE1 / 20, 21, 22, 23, 24 /

      COMMON /COM2/ CM2(5), FE2(5)
CMF$  COMMON FEONLY /COM2/
CMF$  LAYOUT CM2(:NEWS)    ! overrides COMMON directive
CMF$  LAYOUT FE2(:SERIAL)
      DATA CM2 / 30, 31, 32, 33, 34 /
      DATA FE2 / 40, 41, 42, 43, 44 /

      COMMON /COM3/ CM3(5), FE3(5)
CMF$  COMMON INITIALIZED /COM3/
CMF$  LAYOUT CM3(:NEWS)
CMF$  LAYOUT FE3(:SERIAL)  ! overrides COMMON directive
      DATA CM3 / 50, 51, 52, 53, 54 /
      DATA FE3 / 60, 61, 62, 63, 64 /
      END
```

The compiler does not issue a warning message if an attempt is made to initialize a CM array with no associated front-end memory, as is the case above for the array CM1. The use of -fecommon or -common_initialized switches when compiling the program unit above has no effect, since all arrays and common blocks are named in directives that override the effect of the switches.

The array CM2 is a CM array with front-end initialization memory: the COMMON directive causes it to have initialization memory allocated (and a front-end home by default), but the LAYOUT directive forces it to be a CM array.

Copyright © 1989–1994 Thinking Machines Corporation

18.1.3 Examples Illustrating Array Homes

The following examples illustrate the rules of the previous section.

- A character array is always a front-end array.

```
CHARACTER*20 NAMES(100) ! NAMES is a front-end array
```

- An assumed-shape dummy array is always a CM array.

```
SUBROUTINE FOO(A)
INTEGER A(:)      ! A is a CM array
```

- A deferred-shape array is always a CM array.

```
REAL, ALLOCATABLE, DIMENSION(:) :: A   ! A is a CM array
REAL, POINTER, DIMENSION(:) :: P       ! P is a CM array
```

- A common array is a CM array by default.

```
COMMON /BLOCK/ A
REAL A(1000)      ! A is a CM array
```

- An array is a CM array if its unsubscripted name is used on the left side of an assignment statement or a `WHERE` assignment statement, or as a mask expression of a `WHERE` statement or `WHERE` construct.

```
A = 3                ! A is a CM array
WHERE ( MASK ) A = 3 ! A is a CM array, MASK a CM array
```

- An array is a CM array if its unsubscripted name is used as a numeric primary in a numeric operation, as a logical primary in a logical operation, or as a numeric expression in a relational operation.

```
A + 3          ! array A is a CM array
B .GE. 3       ! array B is a CM array
.NOT. MASK     ! array MASK is a CM array
M1 .AND. M2    ! arrays M1 and M2 are CM arrays
```

- An array is a CM array if its name is used unsubscripted as the vector integer expression in a section subscript of an array section.

```
A(X)  ! If X is a rank-one array, then
      !   X is a CM array (as is A)
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- The home of an array is not affected if its unsubscripted name is used as an actual argument to an external procedure, even if the interface of the procedure is explicit.

```
      EXTERNAL FOO           ! implicit interface
      ...
      INTERFACE
        SUBROUTINE BAR(A,B)  ! explicit interface
        INTEGER A(100)       ! A is a CM array
        INTEGER B(200)       ! B is a FE array
CMF$    LAYOUT A(:NEWS)
      END INTERFACE
      ...
      INTEGER X(100), Y(100)
      ...
      CALL FOO(X)    ! home of X is not affected
      CALL BAR(Y,Z)  ! home of Y and Z are not affected
```

- An array is a CM array if its unsubscripted name is used as an actual argument in a function reference to an elemental or transformational intrinsic function.

```
      M = MAX(A,B)   ! A and B are CM arrays
      M = SUM(X)     ! X is a CM array
```

- The home of an array is not affected if its unsubscripted name is passed as an argument to an inquiry intrinsic function.

```
      L = SIZE(A) ! home of A is unaffected
```

- An array is a CM array if its name is used as the parent array in an array section.

```
      A(1:10:2)   ! A is a CM array
      B(1:1)      ! B is a CM array
      C(X)        ! C is a CM array if X is a rank-one array
```


Copyright © 1989–1994 Thinking Machines Corporation

18.2 Array Argument Passing

Information about a CM array passed as an argument or a CM array in common is maintained in a *descriptor* for that array that is accessed at run time when performing operations on the array. A descriptor is a run-time data structure indicating the shape, current home, memory address, layout, and other characteristics of an array argument. Much of the information in a descriptor is supplied at compile time, but some of it may not be determined until run time.

By default, only CM arrays in common and arrays passed as actual arguments have a descriptor generated for them. A local array that is used in an array operation but not passed as an argument to a procedure is allocated in CM stack memory on entry to the defining procedure and deallocated on exit, and does not require a descriptor.

When a CM array is passed as an argument to a procedure, the address of its descriptor is passed, and the array is said to be passed *by descriptor.* All other arguments are passed *by reference*, meaning that the address of each actual argument is passed (for front-end array arguments, the address of the first element is passed).

The use of compiler directives and interface blocks can greatly affect the efficiency with which array arguments are passed; in some cases, they may be required to achieve the correct results. The use of interface blocks allows the compiler to check the type, shape, and layout of each actual argument against those declared for the corresponding dummy argument. When directives are used to change the layout of dummy argument arrays, it may be necessary for the compiler to generate code that first copies the actual array to a temporary variable with a matching layout, then passes a descriptor of the temporary. The calling procedure copies the temporary back to the actual array if the descriptor indicates that it was modified by the called procedure.

Procedures compiled by CM Fortran use the following argument-passing conventions (in this section, the symbol ⇒ means “is passed to”):

(1) Scalar actual ⇒ scalar dummy

The actual is passed by reference (the address of the scalar is passed).

(2) Front-end array actual ⇒ front-end array dummy

The actual is passed by reference (the address of the first array element is passed).

(3) CM array actual ⇒ CM array dummy (*implicit* interface)

The CM array actual argument is passed *in place, by descriptor*, regardless of its layout, meaning that a descriptor is generated referencing the actual argument and the address of the descriptor is passed. All dummy arguments are assumed to have the layout of the actual.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

(4) CM array actual ⇒ CM array dummy (*explicit* interface)

The array actual is passed by descriptor, but it may be copied to a temporary first. All dummy arguments are assumed to have the layout specified in the interface block.

- A CM array actual argument with a layout matching that of its associated dummy argument is passed in place, by descriptor.
- A CM array actual argument with a layout different from its associated dummy argument is passed through a temporary array with the appropriate layout. The calling procedure first copies the actual argument to a temporary with the correct layout, and the temporary is passed by descriptor. Note that if any of the dummy arguments of a procedure have noncanonical layout, then that procedure may be referenced only from program units that correctly specify the layout of its dummy arguments in an interface block.

(5) Front-end array actual ⇒ CM array dummy
CM array actual ⇒ front-end array dummy

These cases are *not* handled by the run-time system, and will result in a run-time error. The `FORALL` statement or the CM Fortran utility library procedures `CMF_FE_ARRAY_TO/FROM_CM` can be used to transfer arrays between CM memory and front-end memory.

Exceptions to this constraint are the CM Fortran utility library procedures that perform dynamic array allocation or create array aliases. These procedures permit the user to manipulate the contents of an array descriptor *as a front-end* array and then pass it to a subprogram that expects a CM array. These procedures do not cause a run-time error as long as run-time safety checking (which compares the homes of array arguments) is not enabled. See the *CM Fortran Libraries Reference Manual* for more information.

Copyright © 1989–1994 Thinking Machines Corporation

The following three subroutine definitions are assumed to be available in the following examples that illustrate argument passing rules (3) and (4) above for CM arrays. The examples are grouped into three groups: those cases that require no interprocessor communication, those that require general communication, and those that will result in an error.

```
      SUBROUTINE SUBC(D)
      REAL D(100,100,100)  ! D has canonical layout
      ...
      END

      SUBROUTINE SUBX(D)
      REAL D(100,100,100)  ! D has noncanonical layout X
CMF$  LAYOUT D(:SERIAL,:NEWS,:NEWS)
      ...
      END

      SUBROUTINE SUBY(D)
      REAL D(100,100,100)  ! D has noncanonical layout Y
CMF$  LAYOUT D(:SERIAL,5:NEWS,2:NEWS)
      ...
      END
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Passed in place (no interprocessor communication required)

Implicit interface: canonical ⇒ canonical

```
      REAL A(100,100,100)  ! A has canonical layout
      ...
      CALL SUBC(A)         ! passed in place
```

Implicit interface: layout X ⇒ layout X

```
      REAL B(100,100,100)  ! B has layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      CALL SUBX(B)         ! passed in place
```

No assumptions are made about the layout of the dummy array.

Explicit interface: canonical ⇒ canonical

```
      REAL A(100,100,100)    ! A has canonical layout
      ...
      INTERFACE
        SUBROUTINE SUBC(D)
        REAL D(100,100,100)  ! D has canonical layout
      END INTERFACE
      ...
      CALL SUBC(A)           ! passed in place
```

Explicit interface: layout X ⇒ layout X

```
      REAL B(100,100,100)    ! B has noncanonical layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      INTERFACE
        SUBROUTINE SUBX(D)
        REAL D(100,100,100)  ! D has noncanonical layout X
CMF$    LAYOUT D(:SERIAL,:NEWS,:NEWS)
      END INTERFACE
      ...
      CALL SUBX(B)           ! passed in place
```


Copyright © 1989–1994 Thinking Machines Corporation

Passed through a temporary created by the caller (communication required)

Explicit interface: canonical ⇒ layout X

```
      REAL A(100,100,100)     ! A has canonical layout
      ...
      INTERFACE
        SUBROUTINE SUBX(D)
        REAL D(100,100,100)   ! D has noncanonical layout X
CMF$    LAYOUT D(:SERIAL,:NEWS,:NEWS)
      END INTERFACE
      ...
      CALL SUBX(A)            ! copy in/out
```

Explicit interface: layout X ⇒ canonical

```
      REAL B(100,100,100)     ! B has noncanonical layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      INTERFACE
        SUBROUTINE SUBC(D)
        REAL D(100,100,100)    ! D has canonical layout
      END INTERFACE
      ...
      CALL SUBC(B)            ! copy in/out
```

Explicit interface: layout X ⇒ layout Y

```
      REAL B(100,100,100)     ! B has noncanonical layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      INTERFACE
        SUBROUTINE SUBY(D)
        REAL D(100,100,100)   ! D has noncanonical layout Y
CMF$    LAYOUT D(:SERIAL,5:NEWS,2:NEWS)
      END INTERFACE
      ...
      CALL SUBY(B)            ! copy in/out
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Run-time error (possibly detected by lower-level software)

Implicit interface: canonical ⇒ layout X

```
      REAL A(100,100,100)  ! A has canonical layout
      ...
      CALL SUBX(A)   ! error in layout !!
```

Implicit interface: layout X ⇒ canonical

```
      REAL B(100,100,100)  ! B has noncanonical layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      CALL SUBC(B)   ! error in layout !!
```

Implicit interface: layout X ⇒ layout Y

```
      REAL B(100,100,100)  ! B has noncanonical layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      CALL SUBY(B)   ! error in layout !!
```

Explicit interface: layout X ⇒ layout X in interface (layout Y in procedure)

```
      REAL B(100,100,100)  ! B has layout X
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      ...
      INTERFACE
        SUBROUTINE SUBY(D)   ! D appears to have layout X
        REAL D(100,100,100)  ! (it actually has layout Y:
CMF$    LAYOUT D(:SERIAL,:NEWS,:NEWS)   ! weights missing)
      END INTERFACE
      ...
      CALL SUBY(B)   ! error in layout !!
```

This case illustrates the importance of making sure that the interface block is consistent with the subprogram definition. An inconsistency between the two may result in errors signalled from lower-level software that are very difficult to find.

Copyright © 1989–1994 Thinking Machines Corporation

18.3 Arrays in Common

The following rules should be observed in CM Fortran programs that use common blocks:

- All CM arrays in common must have the same layout in all program units in which they appear. Common arrays cannot be assumed-layout arrays.
- If a common array is forced to a particular processor home through the use of directives or switches, it must have been forced to have the same home in all program units in which it is used.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation