Chapter 7

Dynamic Array Allocation

CM Fortran provides two mechanisms by which the user can manage parallel memory dynamically:

- Fortran 90 *deferred-shape arrays* (allocatables and pointers) and the statements **ALLOCATE** and **DEALLOCATE**
- utility library procedures **CMF_ALLOCATE/DEALLOCATE_ARRAY** and related procedures

With these facilities, the programmer does not need to resort to such common Fortran 77 practices as allocating all the available memory at compile time and indexing into it as needed, or "hard-wiring" array sizes and then having to recompile to change problem size. The dynamic allocation features allow you to express memory management straightforwardly in a CM Fortran program.

The utility procedures were originally intended as stopgaps before the Fortran 90 features were available. However, there are some differences in behavior between the language features and the library procedures that might still influence the programmer's choice.

Both these mechanisms apply to parallel memory only, and thus only to CM arrays. For dynamic allocation of front-end memory — for front-end arrays and scalars — the cmf77 library provides the **FMALLOC** and **FFREE** subroutines, to be used with the operator **%VAL**.

Copyright © 1989–1994 Thinking Machines Corporation

7.1 Array Pointers

Array pointers are arrays declared with the attribute **POINTER**, along with type and rank, but without extents. The extents are supplied to the **ALLOCATE** statement, which allocates the storage at run time and associates it with the pointer. The pointer can then be used like any other CM array name.

NOTE

This release provides a subset of the Fortran 90 pointer functionality. The subset does not include the **TARGET** attribute or the pointer assignment statement. You cannot declare pointers and targets separately and then associate them at run time.

7.1.1 Declaring Array Pointers

An array can acquire the **POINTER** attribute either in an attributed type declaration or by means of the **POINTER** statement:

```
REAL, POINTER, DIMENSION(:) :: A,B
```

or,

```
...
POINTER A,B
```

The array's shape must be specified as a *deferred-shape*, meaning a comma-separated list of colons, one for each dimension. The deferred-shape specification can be attached to the array name in a type declaration or **POINTER** statement, or it can be supplied with a separate **DIMENSION** or **ARRAY** specification. (It cannot be attached to the **POINTER** attribute in a type declaration.)

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

For example, here are several equivalent ways to declare three array pointers, **P**, **Q**, and **R**:

```
INTEGER, POINTER :: P(:), Q(:), R(:,:)
```

or,

```
INTEGER P,Q,R
POINTER P(:), Q(:), R(:,:)
```

or,

```
INTEGER, POINTER, DIMENSION(:) :: P,Q
INTEGER, POINTER, DIMENSION(:,:) :: R
```

or,

```
INTEGER P,Q,R
DIMENSION P(:), Q(:), R(:,:)
POINTER P,Q,R
```

7.1.2 Allocating Pointer "Target" Arrays

The targets of array pointers are allocated and deallocated by execution of the statements **ALLOCATE** and **DEALLOCATE**. The **ALLOCATE** statement specifies the shape of the array, allocates the storage, and associates it with the pointer:

```
REAL, POINTER, DIMENSION(:) :: A
...
ALLOCATE (A(N))
...
DEALLOCATE (A)
```

The **DEALLOCATE** statement disassociates the pointer and releases the target storage. It is an error to deallocate a pointer array that is not currently allocated. Since pointers are stored in CM heap memory (the static area), they can be deallocated in any order.

The pointer can become associated with other storage by another **ALLOCATE** statement. It is not an error to allocate a pointer array that is already allocated; in this case, the pointer becomes associated with the new storage, and the previous target storage becomes inaccessible.

Copyright © 1989–1994 Thinking Machines Corporation

Use the intrinsic inquiry function **ASSOCIATED** to check the allocation status of an array pointer:

```
ASSOCIATED( POINTER )
```

For example:

```
IF (ASSOCIATED(P)) THEN
   PRINT *, 'P is already allocated.'
ELSE
   ALLOCATE (P(0:M))
END IF
```

NOTE: Those who are familiar with Fortran 90 will notice that **ASSOCIATED** in CM Fortran takes only one argument. The second Fortran 90 argument, **TARGET**, is not provided, since CM Fortran provides no mechanism to associate a target with more than one pointer at a time. Also, CM Fortran does not yet implement the **STAT** argument to **ALLOCATE**.

The system deallocates the array upon exit from the program unit that created it, unless the array has the attribute **SAVE**. With the **SAVE** attribute, the array remains defined and preserves its contents between calls to the program unit that created it.

7.1.3 Using Array Pointers

Once allocated, an array pointer can be used like any other CM array. It can be a primary in an expression, the left-hand side of an array assignment, an argument to a procedure, or an item in an I/O list. An array pointer cannot, however, be a function result.

Pointers are particularly useful for declaring dynamic arrays in common to avoid having to pass them as arguments. For example, a program that operates on numerous dynamic arrays can declare their pointers in common, perhaps in an include file, so that all program units have access to them.

```
REAL, POINTER, DIMENSION(:) :: P1,P2,P3, ... P100
COMMON /C/ P1,P2,P3, ... P100
```


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

If you do use array pointers as arguments, notice several constraints on declaring dummy array arguments as pointers:

- If a dummy array argument is a pointer, the actual argument passed to it must also be a pointer (not necessarily allocated). In addition, the calling procedure must contain an interface block that declares the dummy argument as a pointer.

```
INTEGER, POINTER, DIMENSION(:,:,:) :: ACTUAL_D3
...
INTERFACE
   SUBROUTINE SUB(D3)
   INTEGER, POINTER, DIMENSION(:,:,:) :: D3
END INTERFACE
...
ALLOCATE (ACTUAL_D3 (N,M,N+M))
...
CALL SUB( ACTUAL_D3 )
...
```

- An actual argument that is a pointer may be passed to a nonpointer dummy argument. In this case, no interface block is required.

- A dummy array argument that is a pointer may not have the **INTENT** attribute.

Copyright © 1989–1994 Thinking Machines Corporation

7.2 Allocatable Arrays

An alternative to array pointers is a deferred-shape array declared with the attribute **ALLOCATABLE**. Its type and rank are declared, but its axis extents are not determined until the **ALLOCATE** statement executes. The allocation status of allocatable arrays is reported by the intrinsic function **ALLOCATED**. For example:

```
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE ( A(N), B(100,0:M) )
...
DEALLOCATE (B)
...
DEALLOCATE (A)
```

A allocatable array cannot appear in **COMMON**; nor can it be a dummy array argument or a function result.

For example, the array **T1** in the following fragment is not allocated until after the user has supplied its length at run time:

```
      PROGRAM DYNAMIC
      REAL START, END
      INTEGER N
      REAL, ALLOCATABLE :: T1(:) ! T1 is a CM array

      PRINT *, ' Enter START, END, N:'
      READ (5, *) START, END, N
      ALLOCATE(T1(0:N))
      PRINT *, ' The result = ',
     &           FIGURE_IT(START, END, N, T1)
      DEALLOCATE(T1)
      END

      FUNCTION FIGURE_IT(START, END, N, VALUE)
      REAL START, END
      INTEGER N
      REAL VALUE(0:N)
CMF$  LAYOUT VALUE(:NEWS)     ! VALUE is a CM array

C     [ some operation ]

      END
```


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

7.2.1 Compared with Pointers

For arrays that are local to a program unit, pointers provide the same capabilities as allocatable arrays. The difference between them is that array pointers can appear in common, whereas allocatable arrays cannot. Pointers thus give you the ability to manage global memory dynamically.

In addition, an unallocated pointer can be passed as an argument (as long as the dummy argument is also a pointer), whereas unallocated allocatable arrays cannot be passed as arguments.

In effect, allocatable arrays are a restricted form of pointers that cannot "escape" from the program unit and be deallocated or reallocated elsewhere.

7.3 Utility Library Procedures

Several utility library procedures give you the ability to allocate CM arrays dynamically. The basic form is:

```
CMF_ALLOCATE_ARRAY( ARRAY, EXTENTS, RANK, TYPE )

CMF_DEALLOCATE_ARRAY( ARRAY )
```

Related procedures, such as **CMF_ALLOCATE_LAYOUT_ARRAY**, give you additional control over the array's layout in parallel memory. See the *CM Fortran Libraries Reference Manual* for information on calling these procedures.

The allocation procedures work by creating an array descriptor; the **ARRAY** argument is a front-end array of size **CMF_SIZEOF_DESCRIPTOR** (a predefined constant). The other arguments fill in descriptor slots with additional information about the array to be allocated.

Since the array is a front-end array in the program unit that creates it, it cannot be used as a CM array until you pass it to a subprogram that expects a CM array. For example, the array **IN** in the following program is an assumed-shape dummy array, and thereby constrained to a CM home.

Copyright © 1989–1994 Thinking Machines Corporation

```
 SUBROUTINE ALLOCATE()
 IMPLICIT NONE
 INTEGER NEW_ARRAY(CMF_SIZEOF_DESCRIPTOR)
 INTEGER EXTENTS(7), RANK, I
 PARAMETER (RANK=3)

 INCLUDE 'CMF_defs.h'

 DO I=1,RANK
    EXTENTS(I) = I * 10
 END DO

 CALL CMF_ALLOCATE_ARRAY
&   (NEW_ARRAY, EXTENTS, RANK, CMF_S_INTEGER)

 CALL PRINT_DIMS3D(NEW_ARRAY)    ! Arg is an FE array

 CALL CMF_DEALLOCATE_ARRAY(NEW_ARRAY)
 END SUBROUTINE ALLOCATE
 SUBROUTINE PRINT_DIMS3D(IN)
 IMPLICIT NONE
 INTEGER IN(:,:,:)      ! Dummy is a CM array

 PRINT *,"Shape of DUMMY is (",UBOUND(IN,1),
&        ",",UBOUND(IN,2),
&        ",",UBOUND(IN,3),")"

 END SUBROUTINE PRINT_DIMS3D
```

Because of their very different implementation, the dynamic allocation utilities have a number of advantages and disadvantages in comparison with Fortran 90 deferred-shape arrays (pointers and allocatables created with **ALLOCATE**):

- The dynamic allocation utilities are incompatible with run-time safety checking, since this facility flags home mismatches across program boundaries. Deferred-shape arrays can be used in programs compiled with **-safety**.

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

- Data types and ranks of deferred-shape arrays must be known at compile time. With **CMF_ALLOCATE_ARRAY**, they can be decided at run time (although used only in subroutines where the appropriate type and rank are declared).

- Storage allocated for deferred-shape arrays is deallocated upon exit from the program unit that created them. Storage allocated by a utility subroutine must be deallocated explicitly by a call to **CMF_DEALLOCATE_ARRAY**.

- **ALLOCATABLE** arrays cannot appear in **COMMON**, so their names are not available to all program units. Array pointers and arrays created with **CMF_ALLOCATE_ARRAY** or one of its variants can be globally available.

- If a deferred-shape array is subject to a **LAYOUT** directive, the directive must appear in the specification part of the program unit (before any executable code), and the values of any variables used in the directive are those established upon entry. If you use the utility **CMF_ALLOCATE_LAYOUT_ARRAY** or **CMF_ALLOCATE_DETAILED_ARRAY** instead, you can compute before the call to determine layout-related values. The variable values used are those established at the time of execution. (See Chapter 12 for information on specifying array layouts.)

Copyright © 1989–1994 Thinking Machines Corporation

7.4 Front-End Arrays and Scalars

Neither the CM Fortran language nor its utility library provides for dynamic allocation of front-end arrays and scalars. For this purpose, you can use the subroutines **FMALLOC** and **FFREE** in the cmf77 library (described in the *CM Fortran Libraries Reference Manual).* These subroutines provide an interface to the standard **malloc** and **free** functionality that, together with the **%VAL** operator, enable you to manage front-end storage.

```
FMALLOC(NUM-BYTES, POINTER)

FFREE(POINTER)
```

FMALLOC allocates a block of front-end memory of at least **NUM-BYTES** bytes, suitable for storing any data type. It returns the address of the start of that area in **POINTER**. **FFREE** releases a block of memory allocated by **FMALLOC**.

The value returned in **POINTER** by **FMALLOC** is used with the **%VAL** operator to pass the argument by value to a subprogram. The **%VAL** operator is a predefined argument-list function: it indicates that the argument following it is a pointer that is to be passed by value.

The following example shows the use of **FMALLOC** with **%VAL** to allocate front-end storage and treat it as an array:

```
PROGRAM MALLOC_ME
PARAMETER (M=3,NB=M*4)
INTEGER POINTER

CALL FMALLOC (NB, POINTER)
PRINT *,POINTER

CALL DO_IT(%VAL(POINTER), M)

CALL FFREE (POINTER)
END

SUBROUTINE DO_IT(ARRAY,M)
REAL ARRAY(M)

ARRAY(2) = 14.333
PRINT *,ARRAY(2)
END
```


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