Chapter 4

Porting to CM Fortran

This chapter provides a series of hints for programming in Fortran for the CM system. Many of these items are particular ways of engineering scalability into the program; as such, they often benefit the program on other systems as well. Other items pertain to CM-specific procedures and restrictions; these should be included conditionally if the program is targetted to multiple architectures.

The topics described include:

- Declaring CM arrays
- Customizing CM I/O operations
- Avoiding memory model assumptions
- Simulating dynamic array allocation
- Expressing circular element shifts
- Converting to nodal CM Fortran with message passing

Copyright © 1994 Thinking Machines Corporation

4.1 CM Array Declarations

Optimal array sizes and layouts for CM systems are described in the CM Fortran documentation set. This section lists some general hints for declaring and laying out local and common arrays.

4.1.1 Array Layouts

The CMAX Converter generates a **LAYOUT** directive for every array in the program. By default, arrays used in vectorized operations are laid out in NEWS order, and arrays confined to the control processor are described with all dimensions serial. The converter also generates appropriate **ALIGN** directives for the arrays that it creates (for instance, by promoting scalar values).

The converter never overrides user-specified **LAYOUT** and **ALIGN** directives. Insert directives as needed for best performance. For example, an array dimension intended for serial operations only (such as the time dimension in the evolution of a system) should be laid out within processors:

```
      REAL A(10,1000,1000)
CMF$  LAYOUT A(:SERIAL, :NEWS, :NEWS)
```

NEWS ordering is appropriate for most data parallel operations. SEND ordering is used on CM-2 systems to optimize certain procedures of the CM Scientific Software Library; it is redundant with NEWS ordering on the CM-5.

Operations between arrays are most efficient if the arrays are aligned in distributed memory. Replace this,

```
      REAL A(1000,1000), B(1000,1000)
CMF$  LAYOUT A(:SERIAL, :NEWS), B(:NEWS, :NEWS)
```

with this,

```
      REAL A(1000,1000), B(1000,1000)
CMF$  LAYOUT A(:SERIAL, :NEWS), B(:SERIAL, :NEWS)
```

Use the **ALIGN** directive to align specified dimensions of arrays that are of different sizes or ranks. However, take note of the restrictions on CMAX's ability to manage user-supplied **ALIGN** directives (Chapter 2).

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

CM Fortran Version 2.0 gives best performance if all serial dimensions are declared to the left of any parallel dimension (NEWS or SEND). This restriction is removed in Version 2.1.

4.1.2 Arrays in COMMON

The CM Fortran compiler allocates arrays in **COMMON** either on the control processor or on the parallel processors, never on both. In the Fortran 77 input program, it is useful to segregate the common arrays that will be processed in parallel from those that will be processed serially, placing them in separate **COMMON** blocks. If you fail to do so, the compiler will not maintain the storage and sequence association of the front-end arrays in the common block.

CM Fortran Version 2.0 requires that all CM arrays in **COMMON** be declared in the main program unit, regardless of where they are declared and used in the program. You need to add the declarations manually to the output program to avoid a later CM Fortran error. This restriction is removed in Version 2.1.

Copyright © 1994 Thinking Machines Corporation

4.2 Customizing I/O Operations

CM Fortran supports all Fortran 77 I/O statements, as well as certain common extensions such as **NAMELIST**. The CMAX Converter makes no changes to I/O operations.

This section lists some I/O revisions you might wish to consider.

4.2.1 I/O into a Single Vector

Some programs perform I/O by reading and writing a single large vector, often in **COMMON**. The computational kernel of the program then indexes into the vector to get data as needed.

As noted earlier, the converter can deal with such code only in the 1-dimensional case. For greater scalability, programs should declare arrays in the shapes actually used and perform I/O operations separately on each of the arrays.

Or, you could consider converting in "firewalls" mode (described in Chapter 2), leaving the I/O part of the program outside the kernel to be converted. This approach helps avoid some of the problems that arise from reshaping arrays across procedure boundaries.

4.2.2 Recoding for Parallel File I/O

Prior to Version 2.1, CM Fortran implements **READ** and **WRITE** as serial operations only. Data is transferred in a single stream between the control processor and a peripheral device. If a CM array is to be written, it is first moved to the control processor and then transferred.

Avoiding this "front-end bottleneck" can significantly increase the speed of an I/O operation. The CM Fortran Utility Library procedures **CMF_CM_ARRAY_TO/FROM_FILE** perform parallel I/O, reading or writing in multiple streams between the parallel processors and the device.

Beginning in Version 2.1, the Fortran I/O statements perform parallel I/O on CM arrays, making the utility routines unnecessary.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

4.2.3 Block Data Transfers for Serial I/O

If you choose not to recode I/O operations with CM Fortran utility routines, all your **READ** and **WRITE** statements will execute serially (from the control processor) in versions prior to 2.1. The **cmf** compiler usually performs a block transfer of a CM array to facilitate this operation.

The exception is when the I/O operation contains an implied **DO** loop. In this case, the compiler transfers a CM array to or from the control processor one element at a time. To avoid this time-consuming operation, either:

- Using some conditionalizing convention to target the code to the CM system, manually insert the CM Fortran block-transfer utility procedures (**CMF_FE_ARRAY_TO/FROM_CM**) into the program.
- Rewrite the I/O operation in a way that the converter will translate into a block-transfer utility. For example, if you expect array **A** to become a CM array, read instead into a front-end temporary array and then use a vectorizable **DO** loop to copy the values from **TEMP** to **A**.

Rewrite this . . .

```
READ *, (A(I), I=1,N)
```

. . . like this

```
READ *, (TEMP(I), I=1,N)
DO I=1,N
    A(I) = TEMP(I)
END DO
```


Copyright © 1994 Thinking Machines Corporation

4.3 Avoiding Memory Model Assumptions

Although CM Fortran includes all of Fortran 77, some standard features are restricted to the control processor and thus to sequential execution. These features — **EQUIVALENCE**, arrays that change shape across program boundaries, and assumed-size arrays — are ones that rely on a linear model of memory. This section provides some hints for making Fortran 77 programs more scalable by working around these features.

4.3.1 Coding around EQUIVALENCE

The CMAX Converter does not vectorize loops on equivalenced arrays, since CM Fortran supports equivalencing for front-end arrays only.

The **EQUIVALENCE** statement is used for a number of different purposes, many of which can be expressed in another way. In preparing a Fortran 77 program, you should consider these alternatives for arrays that you want to be processed in parallel.

Saving Memory

One use of **EQUIVALENCE** is to save memory while retaining the clarity of code. Two or more arrays that are used in non-overlapping stages of the program may be folded into one with **EQUIVALENCE**, thus reducing the amount of storage used.

```
REAL TOADS(1000000), FROGS(1000000)
EQUIVALENCE(TOADS,FROGS)
...
```

If **TOADS** and **FROGS** are processed entirely separately, it makes sense to reuse the storage and also to clarify the code by giving the storage different names for each of its uses.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

When targetting the CM system, however, it is best to remove the **EQUIVALENCE** statement (at the cost of extra memory use),

```
REAL TOADS(1000000), FROGS(1000000)
...
```

Or to use a single array for both purposes (possibly at the cost of some clarity),

```
REAL FROADS(1000000)
...
```

Either of these approaches permits the CM system to perform parallel operations on the two data sets.

Alternatively, you can use the converter's Fortran 77 dynamic allocation utility, described below in Section 4.4.

Naming a Subarray

In the interest of conciseness, a programmer might use **EQUIVALENCE** to define an abbreviation for a piece of an array:

```
REAL A(100,100), ARIGHT(100)
EQUIVALENCE (ARIGHT, A(1,100))
...
ARIGHT(K) = X
```

Here, **ARIGHT** is aliased to the rightmost (100th) column of **A**. This kind of abbreviation saves keystrokes, although it can make code somewhat less transparent — it is not immediately obvious from looking at a later line that **ARIGHT** and **A** share storage.

When targetting the CM system, it is best to remove the **EQUIVALENCE** statement and write out the real reference in full. A clearer expression of the intent of the above fragment is:

```
REAL A(100,100)
PARAMETER (ARIGHT = 100)
...
A(K,ARIGHT) = X
```


Copyright © 1994 Thinking Machines Corporation

Subverting the Type System

Programmers who are thoroughly familiar with their target memory organization can use **EQUIVALENCE** to avoid the constraints of the type system, perhaps for convenience,

```
REAL A(2000)
COMPLEX B(1000)
EQUIVALENCE (A,B)
...
DO I = 1,2000
  A(I) = 0.0
END DO
```

or perhaps to perform operations on types for which they are not supported:

```
REAL A(1000)
INTEGER B(1000)
EQUIVALENCE (A,B)
...
DO I = 1,1000
  B(I) = [bit-level operation]
END DO
```

These uses of **EQUIVALENCE** are not generally portable, and are unlikely to work on the CM system even if **EQUIVALENCE** were supported for CM arrays. If you need to do something like this to optimize for another architecture, it is best to isolate and conditionalize that section of the program.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

4.3.2 Coding for CM Array Argument Passing

A general rule of scalable programming is to declare multidimensional arrays as such and to avoid reshaping them across program boundaries. Recall that a CM array as an argument is passed by descriptor, which indicates all the array's elements as well as its shape (see Section 1.4.3).

The CMAX Converter confines to the control processor any arrays that are subject to the following kinds of operations:

- Reshaping or retyping arrays across subroutine boundaries, with certain recognized exceptions
- Passing array arguments with scalar subscripts when either the actual or the dummy argument is multidimensional

As a result, loops on these arrays cannot be vectorized anywhere in the program. Some manual recoding is required to permit vectorization in these cases.

Reshaping across Subroutine Boundaries

To permit vectorization, rewrite this fragment as shown.

Rewrite this . . .

```
REAL A(N1,N2,N3)
CALL ZEROFILL(A, N1*N2*N3)
...

SUBROUTINE ZEROFILL(A,N)
REAL A(N)
DO I = 1,N
  A(I) = 0.0
END DO
RETURN
```

. . . like this

```
REAL A(N1,N2,N3)
CALL ZEROFILL(A,N1,N2,N3)
...

SUBROUTINE
.   ZEROFILL(A,N1,N2,N3)
REAL A(N1,N2,N3)
DO K = 1,N3
  DO J = 1,N2
    DO I = 1,N1
      A(I,J,K) = 0.0
    END DO
  END DO
END DO
RETURN
```


Copyright © 1994 Thinking Machines Corporation

Array Arguments with Scalar Subscripts

The converter cannot convert "hidden array arguments" (arrays referenced with a scalar subscript) when either the actual or the dummy argument is multidimensional. To permit vectorization in the 1-dimensional case, write subroutine calls and definitions as shown on the left. Such code converts as shown on the right:

Fortran 77

```
SUBROUTINE HIDDEN1(A,N)
REAL A(N)
DO I=1,N
  A(I)=SQRT(A(I))+2.0
END DO
RETURN
END

SUBROUTINE HIDDEN(AA, NN)
REAL AA(NN)
CALL HIDDEN1(AA(5), 57)
RETURN
END
```

→

CM Fortran

```
      SUBROUTINE
     .    HIDDEN1(A,OS,LN,N)
      INTEGER LN, OS
      REAL A(1-OS+1:LN+1-OS)
      A(1:N)=SQRT(A(1:N))+2.0
      RETURN
      END

      SUBROUTINE HIDDEN(AA,NN)
      REAL AA(NN)
      ...
      CALL HIDDEN1(AA,5,NN,57)
      RETURN
      END
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

4.4 Dynamic Array Allocation

Scalable software allows some array sizes to be determined at run time, so that the program scales easily for different size data sets and also makes best use of memory in the target architectures. For this purpose, CM Fortran provides Fortran 90 automatic and allocatable arrays.

The CMAX Converter provides library procedures that simulate dynamic array allocation on any Fortran 77 platform. When you convert the program to CM Fortran, CMAX transforms these procedures into dynamic allocation on the CM system.

4.4.1 Simulating Dynamic Allocation in Fortran 77

Fortran 77 does not provide any straightforward way to express the allocation of arrays whose size is determined at run-time. This forces programmers to devise indirect ways of expressing this intent.

Constant Array Sizes

Some Fortran 77 programmers opt to "hard-wire" array sizes. For example, the BART program shown in Chapter 1 is fully portable and converts cleanly into CM Fortran. However, the computation is performed on arrays (**VALUE** and **COEFF**) whose size is specified with the parameter **MAXSLICES**; the run-time value **NSLICES** indicates how much of the storage to use. The specification part of BART's function definition reads:

```
FUNCTION SIMPSON(START, END, NSLICES, NELTS)
REAL START, END
INTEGER NSLICES, NELTS
PARAMETER (MAXSLICES = 1000)
REAL LENGTH, EPSILON
REAL VALUE(MAXSLICES), COEFF(MAXSLICES), X, AREA
INTEGER I
```

On distributed-memory systems, constant array sizes can mean poor utilization of memory and processing resources. If **MAXSLICES** is 1024 and the array **VALUE** is evenly distributed over 128 processors, each processor holds 8 contiguous elements.

Copyright © 1994 Thinking Machines Corporation

```
Processor   0    1    2    3   . . .   127

Element     0    8   16   24
            1    9   17   25
            2   10   18    .   .
            3   11   19    .     .
            4   12   20    .       .
            5   13   21
            6   14   22
            7   15   23                 1023
```

If **NSLICES** is always equal to **MAXSLICES**, processors will be fully utilized. However, when **NSLICES** is 256, only 32 processors participate in the computation of the integral, leaving 96 processors idle.

Constant array sizes pose other problems for programmers, even on serial and shared-memory computers. To run BART with large **NSLICES**, it may be necessary to recompile the program, setting **MAXSLICES** higher.

If the array size were instead determined dynamically, then the **VALUE** and **COEFF** arrays — and all the related computation — could be evenly distributed regardless of array size and the number of processors in the machine, all without recompilation.

Arrays in COMMON

Another frequent approach to "dynamic" array allocation in Fortran 77 is used in the HOMER program (Figure 12), another Simpson integrator. A large chunk of memory is allocated in **COMMON** at compile time, and then used as needed at run time. Although this approach is more scalable than constant array sizes, it suffers from the same problems of uneven distribution as BART, and it may also obscure the actual type and shape of the data set. Also, the common array (**MEMORY**) may be only partially used at run time, wasting memory on both serial and parallel machines.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

```
      PROGRAM HOMER
      REAL START, END
      INTEGER NSLICES, NELTS
      COMMON /MEMBLK/ MEM(1000000)
      REAL MEM

      PRINT *, ' Enter START, END, NSLICES:'
      READ (5, *) START, END, NSLICES
      NELTS = NSLICES + 1

      PRINT *, ' The integral = ',
     .   SIMPSON(START,END,NSLICES,NELTS,MEM(1),MEM(1+NELTS))
      STOP
      END

      FUNCTION SIMPSON(START,END,NSLICES,NELTS,VALUE,COEFF)
      REAL START, END
      INTEGER NSLICES, NELTS
      REAL VALUE(NELTS), COEFF(NELTS)
      REAL LENGTH, EPSILON
      REAL X, AREA
      INTEGER I

      LENGTH = END - START
      EPSILON = LENGTH/NSLICES

C Evaluate function and compute coefficients:
      DO I = 1,NELTS
         X = START + (I-1)*EPSILON
         VALUE(I) = COS(X)
         COEFF(I) = 2 + 2*MOD(I-1,2)
      END DO

C First and last coefficients are 1.0:
      COEFF(1) = 1.0
      COEFF(NELTS) = 1.0

C Compute total area using Simpson's rule:
      AREA = 0.0
      DO I = 1,NELTS
         AREA = AREA + COEFF(I)*VALUE(I)
      END DO

      SIMPSON = (LENGTH/(3*NSLICES))*AREA

      RETURN
      END
```

Figure 12. The HOMER program, coded in Fortran 77 with a common array.

Copyright © 1994 Thinking Machines Corporation

4.4.2 A CM Fortran Solution

Because Fortran 77 lacks the features of automatic and allocatable arrays, programmers cannot express their intent clearly. In the BART and HOMER programs, for instance, the CMAX Converter cannot determine whether the programmer who wrote a **PARAMETER** statement or a **COMMON** statement would actually have preferred dynamic allocation.

One approach to porting these memory management schemes to CM Fortran is to rewrite the array allocation using the appropriate Fortran 90 dynamic allocation feature. Program LISA (Figure 13) illustrates the use of CM Fortran automatic arrays (which follow a stack discipline). In LISA, the run-time value **NSLICES**, not the compile-time value **MAXSLICES**, specifies the size of the arrays. For another algorithm, an allocatable array or array pointer (which follow a heap discipline) might be preferred.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

```
      PROGRAM LISA
      REAL START, END
      INTEGER NSLICES, NELTS

      PRINT *, ' Enter START, END, NSLICES:'
      READ (5, *) START, END, NSLICES
      NELTS = NSLICES + 1

      PRINT *, ' The integral = ',
     .         SIMPSON(START, END, NSLICES, NELTS)
      STOP
      END

      FUNCTION SIMPSON(START, END, NSLICES, NELTS)
      REAL START, END
      INTEGER NSLICES, NELTS

      REAL VALUE(NELTS), COEFF(NELTS), X(NELTS)
CMF$  LAYOUT VALUE(:NEWS), COEFF(:NEWS), X(:NEWS)

      REAL LENGTH, EPSILON
      REAL AREA
      INTEGER I

      LENGTH = END - START
      EPSILON = LENGTH/NSLICES

C Evaluate function and compute coefficients:

      FORALL (I = 1:NELTS) X(I) = START + (I - 1) * EPSILON
      VALUE = COS(X)
      FORALL (I = 1:NELTS) COEFF(I) = 2 + 2 * MOD(I - 1,2)

C First and last coefficients are 1.0:

      COEFF(1) = 1.0
      COEFF(NELTS) = 1.0

C Compute total area using Simpson's rule:

      AREA = 0.0
      AREA = AREA + DOTPRODUCT(COEFF,VALUE)

      SIMPSON = (LENGTH / (3 * NSLICES)) * AREA

      RETURN
      END
```

Figure 13. The LISA program, coded in CM Fortran with automatic arrays.

Copyright © 1994 Thinking Machines Corporation

4.4.3 A Scalable Fortran 77 Solution

The CMAX Converter provides a canonical, portable way of expressing dynamic memory allocation in Fortran 77. Programs that use this utility can be compiled with any Fortran 77 compiler (source code is provided). The converter recognizes this utility and translates it into dynamic array allocation in CM Fortran.

The utility package consists of the header file **cmax.h** and a set of libraries for serial and parallel linking. (See Appendix A for information on linking programs that make use of the dynamic allocation utility.) The utility defines a set of subroutines, one for each possible rank of the array to be allocated.

CMAX_ALLOCATE_*rank* **(INDEX_VAR, ELT_TYPE, N1 ..., N7)**

Returns a value in **INDEX_VAR** that can (only) be used to index into the common array **CMAX_MEMORY** when passed as an argument to a procedure.

INDEX_VAR	An integer variable or front-end array element
ELT_TYPE	A predefined integer constant, one of **CMAX_LOGICAL**, **CMAX_INTEGER**, **CMAX_REAL**, **CMAX_DOUBLE**, **CMAX_COMPLEX**, or **CMAX_DOUBLE_COMPLEX**
N1 ..., N7	Integer extents for array dimensions; the number of extent arguments must correspond to the rank specified in the procedure name

---

The common array **CMAX_MEMORY** is declared by the header file in an **INCLUDE** line. A call to **CMAX_ALLOCATE_***rank* associates an index variable with a pointer into that memory. The memory thus indicated becomes available as an array when you pass a reference to it, **CMAX_MEMORY(INDEX_VAR)**, as an argument to a subroutine or function. The corresponding dummy argument in the procedure must match the shape and type of the array allocated; it can be either a CM array or a front-end array in the subprogram scope.

An index variable can be used for multiple arrays of different dimension extents, although all must be of the same rank and layout. You can also build arrays of dynamically allocated arrays, by using a front-end array element at

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

the index variable. All dynamic arrays pointed to by elements of an index array must be of the same rank and layout.

The library procedure **CMAX_DEALLOCATE** performs dynamic deallocation.

CMAX_DEALLOCATE (INDEX_VAR)

Frees the memory associated with the **INDEX_VAR**.

INDEX_VAR An integer variable previously defined as an index into **CMAX_MEMORY** by a call to **CMAX_ALLOCATE_***rank*.

For example, this fragment allocates two arrays, a real array of rank 2 and an integer array of rank 1.

```
INCLUDE 'cmax.h'
INTEGER I_A, I_B
...
CMAX_ALLOCATE_2(I_A, CMAX_REAL, K1, K2)
CMAX_ALLOCATE_1(I_B, CMAX_INTEGER, K1*K2)
...
CALL SUBMARINE(CMAX_MEMORY(I_A),
.               CMAX_MEMORY(I_B), K1, K2, K1*K2)
...
CALL CMAX_DEALLOCATE(I_A)
CALL CMAX_DEALLOCATE(I_B)
...
SUBROUTINE SUBMARINE(A, B, N1, N2, N3)
REAL A(N1, N2)
REAL B(N3)
...
```

The calls to **CMAX_ALLOCATE** and **CMAX_DEALLOCATE** need not immediately surround the call to the procedure that uses the storage. Any code can come between allocation and use, there can be multiple uses of an allocated array, and the allocated arrays can be deallocated in any order. These features are also true of the CM Fortran dynamic array allocation to which the CMAX Converter translates this utility.

See Figure 14 for an example of a Simpson integrator coded with the CMAX dynamic allocation utility.

Copyright © 1994 Thinking Machines Corporation

```
      PROGRAM MARGE
      REAL START, END
      INTEGER NSLICES, NELTS
      INTEGER I_VALUE, I_COEFF
      INCLUDE 'cmax.h'

      PRINT *, ' Enter START, END, NSLICES:'
      READ (5, *) START, END, NSLICES
      NELTS = NSLICES + 1

      CALL CMAX_ALLOCATE_1(I_VALUE, CMAX_REAL, NELTS)
      CALL CMAX_ALLOCATE_1(I_COEFF, CMAX_REAL, NELTS)
      PRINT *, ' The integral = ', SIMPSON(START,END,
     .   NSLICES,NELTS,CMAX_MEMORY(I_VALUE),CMAX_MEMORY(I_COEFF))
      CALL CMAX_DEALLOCATE(I_VALUE)
      CALL CMAX_DEALLOCATE(I_COEFF)
      STOP
      END

      FUNCTION SIMPSON(START,END,NSLICES,NELTS,VALUE,COEFF)
      REAL START, END
      INTEGER NSLICES, NELTS
      REAL VALUE(NELTS), COEFF(NELTS)
      REAL LENGTH, EPSILON
      REAL X, AREA
      INTEGER I

      LENGTH = END - START
      EPSILON = LENGTH/NSLICES
C Evaluate function and compute coefficients:
      DO I = 1,NELTS
         X = START + (I-1)*EPSILON
         VALUE(I) = COS(X)
         COEFF(I) = 2 + 2*MOD(I-1,2)
      END DO
C First and last coefficients are 1.0:
      COEFF(1) = 1.0
      COEFF(NELTS) = 1.0
C Compute total area using Simpson's rule:
      AREA = 0.0
      DO I = 1,NELTS
         AREA = AREA + COEFF(I)*VALUE(I)
      END DO

      SIMPSON = (LENGTH/(3*NSLICES))*AREA
      RETURN
      END
```

Figure 14. The MARGE program, coded in Fortran 77 with the CMAX dynamic allocation utility.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

It is advisable to control the layout of the index array and the allocated arrays with explicit **LAYOUT** directives.

- If an array, the **INDEX_VAR** argument itself must be placed on the front end (layout **:SERIAL**).
- The **INDEX_VAR** argument (array or scalar) can be placed in **COMMON**. If so, the user is responsible for seeing that all the arrays it points to have the same rank and layout. CMAX cannot enforce this restriction.
- The allocated arrays can be designated either front-end or CM by explicit **LAYOUT** directive.
- If the procedure that allocates an array never uses **CMAX_MEMORY()** — that is, never passes the array to a subprogram where it is declared and used — the layout defaults to all **:NEWS**. If you wish to control layout (or home) explicitly, insert a subprogram that does nothing more than declare and lay out the array.

When the allocated array is given an explicit all **:SERIAL** layout in a called subroutine, **CMAX_ALLOCATE_***rank* allocates a front-end array. For example, in the following program, the dynamic array **ISER** will become a CM Fortran front-end array.

```
      PROGRAM CEREAL
      INCLUDE '/usr/include/cm/cmax.h'

      INTEGER ISER

      CALL CMAX_ALLOCATE_1(ISER, CMAX_INTEGER, 100)
      CALL USIT(CMAX_MEMORY(ISER))

      END

      SUBROUTINE USIT(K)
      INTEGER K(100)
CMF$  LAYOUT K(:SERIAL)

      DO I = 1,100
        K(I) = I
      END DO

      PRINT *, K

      END
```


Copyright © 1994 Thinking Machines Corporation

4.5 Circular Element Shifts

Fortran 77 provides no concise way to express a circular shift of array elements, where the element(s) that shift off the end of a dimension are "wrapped" around to the beginning. This operation is expressed by the Fortran 90 intrinsic function **CSHIFT**.

The CMAX library provides a canonical, portable way to express circular shifts in Fortran 77. This utility can be compiled with any Fortran 77 compiler, or translated by the CMAX Converter into references to the **CSHIFT** function. (A future version of the converter will recognize a circular shift idiom.)

The utility package consists of the header file **cmax.h** and a library for serial linking. (See Appendix A for information on linking programs that use the circular shift utility.) The utility defines a set of subroutines, one for each possible rank of the array to be shifted.

CMAX_CSHIFT_*rank* **(DEST, SOURCE,**
DIM, SHIFT, ELT_TYPE, N1 ..., N7)

Shifts the elements of the specified **DIM** of the source array by **SHIFT** distance and returns the result in the destination array. Both the source and destination arrays must be of the specified type and shape.

DEST	The destination array; must not overlap with the source array
SOURCE	The source array
DIM	An integer between 1 and *rank* indicating the dimension along which to shift elements
SHIFT	An integer indicating the distance (number of element positions) to shift
ELT_TYPE	A predefined integer constant, one of **CMAX_LOGICAL**, **CMAX_INTEGER**, **CMAX_REAL**, **CMAX_DOUBLE**, **CMAX_COMPLEX**, or **CMAX_DOUBLE_COMPLEX**
N1 ..., N7	Integer extents for array dimensions; the number of extent arguments must correspond to the rank specified in the procedure name

---

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

For example, this fragment shifts the elements in real vector **A** by 3 positions in the negative direction (upward, in terms of array element order) and shifts the second dimension of matrix **B** by 1 position in the positive direction (downward). The elements that shift off the end wrap around to the other end of the dimension. The subroutines store the results in arrays **X** and **Y**, respectively, which are the same type and shape as the source arrays.

```
INCLUDE 'cmax.h'
...
REAL A(K1), B(K2,K3)
REAL X(K1), Y(K2,K3)
...
CALL CMAX_CSHIFT_1(X,A,1,-3,CMAX_REAL,K1)
...
CALL CMAX_CSHIFT_2(Y,B,2,1,CMAX_REAL,K2,K3)
...
```

The converter recognizes these subroutines and transforms them into references to the CM Fortran **CSHIFT** intrinsic function. The converter uses argument keywords in the output code so that the order of the **DIM** and **SHIFT** arguments is compatible with any version of CM Fortran.

```
...
X = CSHIFT(A, DIM=1, SHIFT=-3)
...
Y = CSHIFT(B, DIM=2, SHIFT=1)
...
```


Copyright © 1994 Thinking Machines Corporation

4.6 Converting to Nodal CM Fortran with Message Passing

CMAX supports both global and nodal CM Fortran for the CM-5. Since the CM Fortran source is largely the same for both execution models (barring some I/O differences), CMAX users notice little difference in targeting one or the other.

One slight difference is that you might choose different values for the cmax command options **-ShortVectorLength** and **-ShortLoopLength**. On the node, shorter lengths are often appropriate; in global programming, longer lengths are appropriate.

The major difference you might notice is in the need to control array homes explicitly in nodal programs. CMAX flags the CMMD message-passing routines as unknown external routines and makes assumptions about their impact on array homes according to the setting of **-UnknownRoutinesSafe**. However, many CMMD routines take arguments that essentially specify the homes of arrays. To avoid problems, you should always insert explicit **LAYOUT** directives for arrays that will be passed to CMMD routines.

For example, to send a block of data, you might write either:

```
      REAL X(1000)
CMF$  LAYOUT X(:NEWS)
      INTEGER RESULT

      RESULT = CMMD_SEND_BLOCK(DEST_NODE,
     &    CMMD_DEFAULT_TAG, X, CMMD_PARALLEL_ARRAY)
```

or:

```
      REAL X(1000)
CMF$  LAYOUT X(:SERIAL)
      INTEGER RESULT

      RESULT = CMMD_SEND_BLOCK(DEST_NODE,
     &    CMMD_DEFAULT_TAG, X, 1000 * 4)
```

depending on whether **X** is to have a **:NEWS** or **:SERIAL** layout. It is advisable to specify that layout for **X** in the CMAX input program to avoid home mismatches in the output program.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation