The
Connection Machine
System

Using the CMAX Converter

Version 1.0

July 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, July 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000

Contents

Appendixes

Index

Figures

About This Manual

Objectives of This Manual

This manual describes the CMAX Converter for translating Fortran 77 code into CM Fortran. It also provides hints for preparing portable code. The topics covered are:

- The converter's actions and the mechanics of using it
- Fortran 77 constructions and their CMAX-generated equivalents
- The programming conventions of *scalable* Fortran 77, which enhance the portability of code performance
- General portability issues and some variations on the porting process

Intended Audience

The reader of this manual is assumed to have a thorough grasp of Fortran 77 programming and some knowledge of CM Fortran (at the level of the *CM Fortran Programming Guide).* User-level knowledge of UNIX is also required.

Revision Information

This is a new manual.

Related On-Line Documents

- `cmax-1.0.releasenotes`
- `cmax-1.0.bugupdate`

The path is typically `/usr/doc` on Connection Machine CM-5 systems and `/usr/cm/doc` on CM-2/200 systems. See your system administrator for the locations if these files have been moved.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Telephone:	(617) 234–4000

Chapter 1

Overview

CMAX — the CM Automated Translator — converts Fortran 77 programs into CM Fortran. It greatly simplifies the task of porting serial Fortran programs to the massively parallel Connection Machine system.

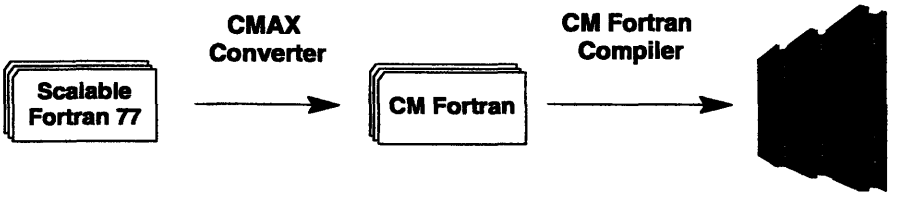
The conversion procedure is the same whether your target is the global data parallel model of CM Fortran or the nodal model that uses the vector units for parallel processing and explicit message passing for communication.

This chapter provides an overview of the converter: its goals and its capabilities. Later chapters address the larger porting process: preparing the program, using the converter, and customizing code for best performance on the CM system.

1.1 Why Use the Converter?

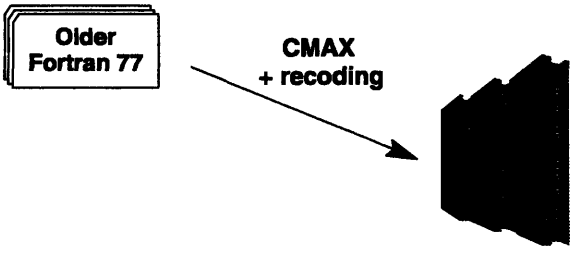
The CMAX Converter has two basic goals:

- It enables developers of new Fortran 77 programs to access the parallel processing resources of the CM system. For programs that follow the few simple conventions of *scalable* Fortran, conversion is largely automatic.

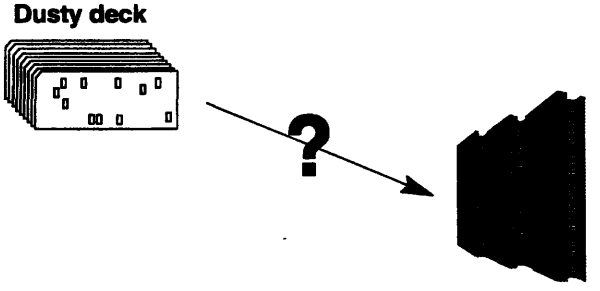


Users in a heterogeneous computer environment and third-party software developers can use the converter as a "preprocessor" for routine CM Fortran compilation. Since the program is maintained in Fortran 77 for portability to multiple platforms, the converter provides a migration path onto and off from the CM system.

- The converter assists users in porting older Fortran 77 programs to CM Fortran. Nonstandard features and non-*scalable* programming idioms may not convert automatically, however, so some manual recoding is usually required.



The amount of recoding depends, naturally, on the size and condition of the input program. It is not a specific goal of the CMAX Converter to convert pre–Fortran 77, "dusty deck" programs, although it can be helpful in the later stages of the porting process.



Dusty decks must first be brought to conformance with the Fortran 77 standard. From there, any of several paths can be followed, depending on whether the program is to run on several architectures or on the CM only.

Chapter 2 of this manual provides more information on the converter's place within an overall porting process.

1.2 A Simple Conversion

The CMAX Converter's basic actions are:

- to *vectorize* `DO` loops on arrays, that is, translate them into CM Fortran array operations
- to indicate the CM home — serial control processor or parallel processing unit — of each array

These actions enable the CM Fortran compiler to generate parallel instructions and to allocate arrays on the appropriate part of the system.

This section illustrates the converter by presenting the straightforward conversion of a simple program. Program BART, Figure 1, numerically integrates the cosine function over a given interval using Simpson's rule. The converter automatically translates BART into the CM Fortran program shown in Figure 2.

1.2.1 Invoking the Converter

The converter is currently available as a batch tool that can be invoked from any shell prompt. The barebones conversion procedure is the following.

1. Create a *package* containing the source program. A package is a set of files that the CMAX Converter treats as a complete program.

   ```
   % cmax bartpack -AddFiles= bart.f [other source files]
   ```

2. Convert the program in the package to CM Fortran. Each converted `.f` file is written to a `.fcm` file.

   ```
   % cmax bartpack
   ```

3. Compile the output file(s) for the CM system of your choice.

   ```
   % cmf [ cmf switches ] -o bart bart.fcm
   ```

4. Execute the program on the appropriate CM system.

   ```
   % bart
   ```

Chapter 2 of this manual presents more information about the `cmax` command and the variations in the conversion procedure.

```
      PROGRAM BART
      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
      PARAMETER (MAXSLICES = 1000)
      REAL LENGTH, EPSILON
      REAL VALUE(MAXSLICES), COEFF(MAXSLICES), 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 1. The BART program, coded in Fortran 77 with DO loops on arrays.

```
      PROGRAM BART
      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
      PARAMETER (MAXSLICES = 1000)
      REAL LENGTH, EPSILON
      REAL VALUE(MAXSLICES), COEFF(MAXSLICES), X, AREA
      INTEGER I
CMF$  LAYOUT COEFF(:NEWS)
CMF$  LAYOUT VALUE(:NEWS)
      REAL  X100(MAXSLICES)
CMF$  LAYOUT X100(:NEWS)

      LENGTH = END - START
      EPSILON = LENGTH / NSLICES

C Evaluate function and compute coefficients:
      FORALL (I = 1:NELTS) X100(I) = START + (I - 1) * EPSILON
      VALUE(:NELTS) = COS(X100(:NELTS))
      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(:NELTS),VALUE(:NELTS))

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

      RETURN
      END
```

Figure 2. The BART program, converted to CM Fortran.

1.2.2 The Converter's Action

Notice what the converter has done to the BART program. It has performed loop conversion to express parallelism, and it has determined where on the CM system each operation will be performed.

This section is a brief introduction only; a more detailed list of the transformations CMAX can perform appears in Appendix B.

Loop Conversion

The converter has translated the two **DO** loops in the Fortran 77 program into array operations. The loop that operates on each array element independently,

```
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
```

turns into these array assignment statements:

```
C Evaluate function and compute coefficients:
      FORALL (I=1:NELTS) X100(I) = START+(I-1)*EPSILON
      VALUE(:NELTS) = COS(X100(:NELTS))
      FORALL (I=1:NELTS) COEFF(I) = 2+2*MOD(I-1,2)
```

The loop that sums the products of the respective elements of **COEFF** and **VALUE** into a scalar,

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

turns into this call to the intrinsic function **DOTPRODUCT**:

```
C Compute total area using Simpson's rule:
      AREA = 0.0
      AREA = AREA +
     .    DOTPRODUCT(COEFF(:NELTS),VALUE(:NELTS))
```

Array Homes

The converter has also inserted directives that control where on the CM system the operations will occur, guaranteeing that the data to be processed in parallel is allocated in the memory of the parallel processors.

The arrays **COEFF** and **VALUE**,

```
      PARAMETER (MAXSLICES = 1000)
      REAL VALUE(MAXSLICES), COEFF(MAXSLICES)
```

become distributed across CM processors in NEWS order:

```
      PARAMETER (MAXSLICES = 1000)
      REAL VALUE(MAXSLICES), COEFF(MAXSLICES)
CMF$  LAYOUT COEFF(:NEWS)
CMF$  LAYOUT VALUE(:NEWS)
```

In addition, the scalar **X**,

```
      REAL X
```

is promoted to an array named **X100** and distributed, since it will be used in array operations:

```
      REAL   X100(MAXSLICES)
CMF$  LAYOUT X100(:NEWS)
```

1.2.3 Is BART Typical?

BART's conversion is painless not only because the program is simple, but also because it is, to a large extent, *scalable*. The basic feature of scalable programs is that they operate on arrays of data, which a compiler can split up among multiple processors for independent or coordinated processing. Contrast BART with program MAGGIE in Figure 3. MAGGIE is also a Simpson integrator, but this implementation does not use arrays. The CMAX Converter will not produce parallel code for MAGGIE.

The following section presents the conventions of scalable Fortran 77 programming. Although these conventions arose from a desire to use data parallel systems effectively, they turn out to be a convenient strategy for achieving good performance on many other architectures as well.

```
      PROGRAM MAGGIE

      REAL START, END
      INTEGER NSLICES

      PRINT *, ' Enter START, END, NSLICES:'
      READ (5, *) START, END, NSLICES

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

      STOP
      END


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

      LENGTH = END - START
      EPSILON = LENGTH/NSLICES

C Evaluate function and accumulate into area:

      AREA = 0
      DO I = 1, NSLICES-1
        X = START + I*EPSILON
        AREA = AREA + (2 + 2*MOD(I,2))*COS(X)
      END DO

      AREA = (COS(START) + AREA +
     .        COS(START + NSLICES*EPSILON))

C Compute total area using Simpson's rule:

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

      RETURN
      END
```

Figure 3. The MAGGIE program, coded in Fortran 77 without arrays.

1.3 Writing Scalable Fortran

Fortran 77 is available on virtually all platforms and thus assures convenient portability. A program's performance, however, is often tuned to the details of a particular architecture, and thus may not port well. With the recent explosion of architectures — vector machines, highly pipelined machines, machines with multiple functional units, massively parallel machines, and machines with combinations of all these features — the portability of a code's performance is not guaranteed, and must be engineered in.

We define *scalability* as the portability of a code's performance. In particular, a scalable program is one designed to execute efficiently on any size data set, large or small, using any number of processors, from one to thousands. Modern data parallel languages like Fortran 90 provide constructs conducive to scalable programming; for older languages like Fortran 77, one must follow certain conventions to ensure scalability.

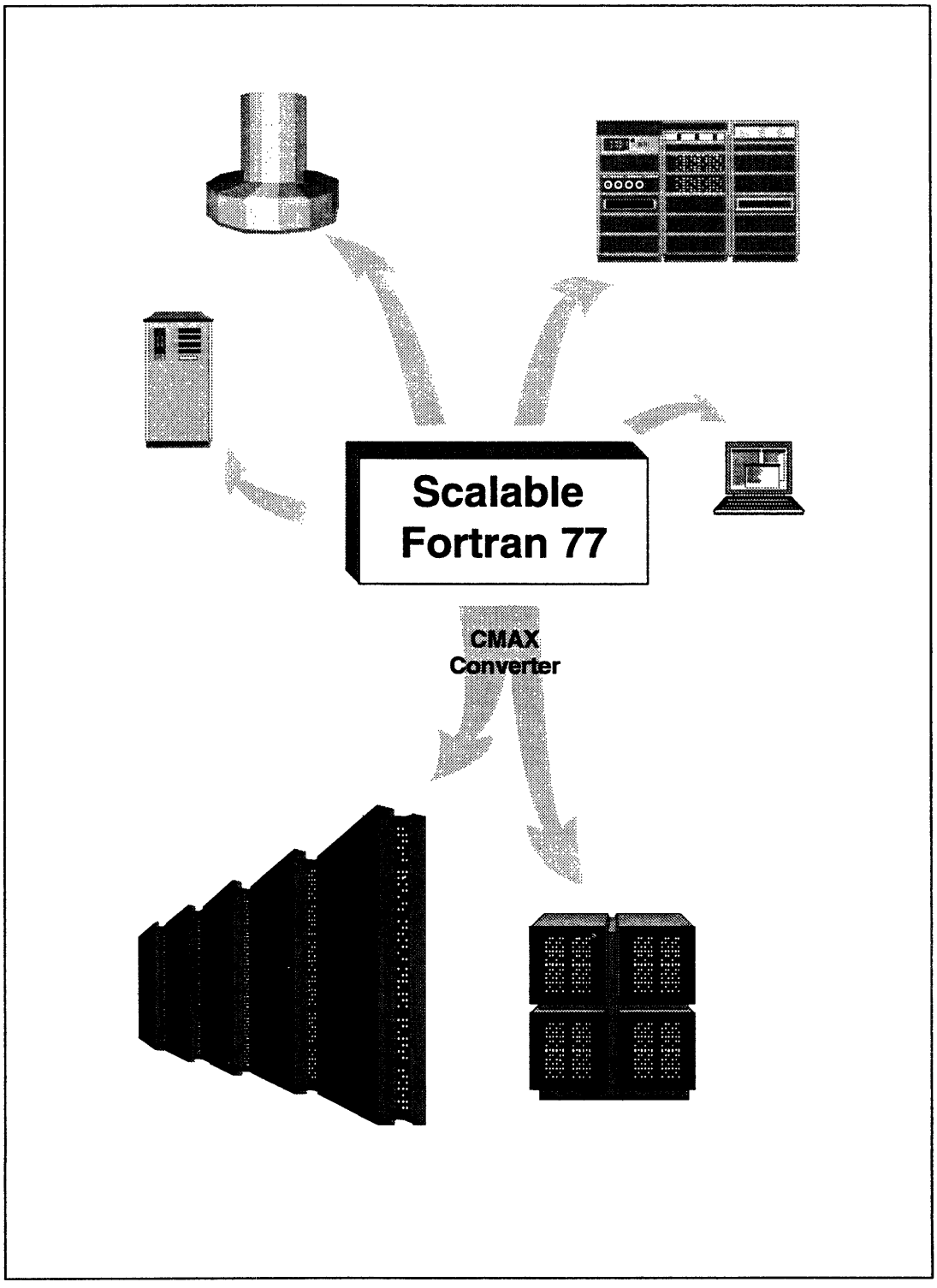
The conventions express three basic objectives:

- Make it easy for a compiler to recognize how data and computations may be split up for independent or coordinated processing.
- Avoid constructions that rely on a particular memory organization.
- Use data layout directives and library procedures (with some conditionalizing convention) to take advantage of the specific performance characteristics of the target platforms.

This section introduces the conventions of scalable programming in Fortran 77, using fragments from a widely used computational aerodynamics program called FLO67™. This program uses a multigrid scheme to simulate the 3-dimensional airflow past a swept wing. It is implemented in about 5000 lines of code.*

NOTE: The conventions are intended as rules of thumb rather than hard-and-fast requirements. In places where different conventions may conflict, as noted in the illustrations below, the goal being sought should guide the programmer's judgment which to follow.

* This section is condensed from "FLO67: A Case Study in Scalable Programming," by Skef Wholey, Clifford Lasser, and Gyan Bhanot, Thinking Machines Technical Report TMC–213, January 1992. The program FLO67 was developed and is owned by Professor Antony Jameson of the Department of Aerospace Engineering at Princeton University.



Rule 1: Scalable programs operate on most or all the data "at once."

In Fortran 77, this convention means looping over as many array axes, and as much of each axis, as possible. The loop below, for example, calculates the air pressure for each cell:

```
      DO 90 K=1,KL
        DO 90 J=1,JL
          DO 90 I=1,IL
            QQ = W(I,J,K,2)**2 + W(I,J,K,3)**2 + W(I,J,K,4)**2
            QQ = .5*QQ/W(I,J,K,1)
            P(I,J,K)=(GAMMA-1.)*DIM(W(I,J,K,5),QQ)
90    CONTINUE
```

The 4-dimensional array **W** implements a 3-dimensional array of 5-element record structures; structure components are selected by indexing along the fourth axis. The first structure component is the air density; the second, third, and fourth components are the momentum densities in the **X**, **Y**, and **Z** directions, respectively; and the fifth component is the total energy density.

NOTE: While data can be distributed across processors along the three spatial dimensions of **W**, the fourth dimension is better left undistributed (in CM Fortran parlance, it is a serial axis) since operations on it are likely to be inherently sequential. This case illustrates the judgmental quality of scalable programming, since a serial axis warrants different treatment from axes in the data domain.

Rule 2: Scalable programs operate on data elements homogeneously, performing identical or similar operations on each.

The programming style encouraged by serial computers (and serial thinking) often violates this convention. The code below computes different values for interior and boundary cells:

```
      DO 10 J=2,M-1
        DO 10 I=2,N-1
          A(I,J) = A(I,J) + B(I,J)   ! Interior
10    CONTINUE

      DO 20 J=1,M
        A(1,J) = C(1,J)              ! Edge
        A(N,J) = C(N,J)
20    CONTINUE
```

```
      DO 30 I=2,N-1
        A(I,1) = C(I,1)                    ! Edge
        A(I,N) = C(I,N)
30    CONTINUE
```

In addition to violating the first rule — operate on all the data at once — these loops perform different operations for different elements. Rather than coding three loops as above, one could set up a logical variable **MASK** which is **.TRUE.** for interior elements and **.FALSE.** for boundary elements. The above computation could then be expressed in this scalable code:

```
      DO 10 J=1,N
        DO 10 I=1,N
          IF (MASK(I,J)) THEN
            A(I,J) = A(I,J) + B(I,J)
          ELSE
            A(I,J) = C(I,J)
          END IF
10    CONTINUE
```

Rule 3: Scalable programs exhibit locality of reference.

In Fortran 77, this convention means that expressions within loops should reference similarly indexed portions of similarly shaped arrays. Elements at the loop indices or nearby are used in computations; distant elements are not. A subroutine of FLO67 computes each element of array **DTL** as a function of nearby elements of array **RAD**:

```
      DO 70 K=1,KL
        DO 70 J=1,JL
          DO 70 I=1,IL
            RADA =  RAD(I+0,J+0,K+0) + RAD(I+1,J+0,K+0) +
     .              RAD(I+0,J+1,K+0) + RAD(I+1,J+1,K+0) +
     .              RAD(I+0,J+0,K+1) + RAD(I+1,J+0,K+1) +
     .              RAD(I+0,J+1,K+1) + RAD(I+1,J+1,K+1)
            DTL(I,J,K)=CFL/RADA
70    CONTINUE
```

In data parallel architectures, this convention allows the compiler to use nearest-neighbor communication instead of the more costly general communication between processors.

Rule 4: Scalable programs exhibit numerical stability.

Some degree of imprecision is unavoidable in floating-point calculations. Since most real numbers cannot be represented exactly in digital form, these numbers are rounded to a representable number. The cumulative round-off error for a series of computations can lead at times to floating-point exceptions or incorrect results.

Scalable programs avoid relying on the correctness of an algorithm or round-off tolerance tuned to a single architecture. This is particularly true of array summations and other reductions, since floating-point reductions are especially sensitive to the order of evaluation. Because they may use a different order of evaluation, vector and parallel machines may produce markedly different results from serial machines, including run-time errors.

For example, if reversing the order of the loop changes the answer in a significant way, then the computation is numerically unstable. Consider this program:

```
      PROGRAM ROUNDABOUT
      REAL A(12), X
      INTEGER I

      A(1)  = +1E+8
      A(2)  = +1E+1
      A(3)  = +2E+8
      A(4)  = +2E+1
      A(5)  = +3E+8
      A(6)  = +3E+1
      A(7)  = -3E+1
      A(8)  = -3E+8
      A(9)  = -2E+1
      A(10) = -2E+8
      A(11) = -1E+1
      A(12) = -1E+8

C     Add up from 1 to 12:

      X = 0.0
      DO I = 1,12
        X = X + A(I)
      END DO
      PRINT *, X
```

```
C     Add up from 12 to 1:

      X = 0.0
      DO I = 12,1,-1
        X = X + A(I)
      END DO
      PRINT *, X

C     Add up odd elements, then even:

      X = 0.0
      DO I = 1,11,2
        X = X + A(I)
      END DO
      DO I = 2,12,2
        X = X + A(I)
      END DO
      PRINT *, X
      END
```

A system using IEEE 32-bit **REAL** values produces this output:

```
 -40.0000
  40.0000
0.
```

Using IEEE 64-bit **DOUBLE PRECISION** instead of 32-bit **REAL**, the output is:

```
0.
0.
0.
```

Scalable programs maintain numerical stability on all target architectures. This example illustrates two options for achieving this:

- Go to higher-precision representation.
- Change the algorithm. In this example, the numerically stable answer is obtained by summing first the odd elements, then the even elements.

Rule 5: Scalable programs use easily recognizable idioms to express common, well-structured dependences.

In the loops shown thus far, each iteration can be executed independently of the others; that is, there are no *loop-carried data dependences*. Most algorithms do have some form of interaction between data items. While code with arbitrary data dependences does not generally execute well on vector or parallel machines, most well-structured data-dependent computations do have efficient low-level algorithms on serial, vector, and parallel machines. Scalable code enables a compiler to recognize such computations by expressing them with a small set of common idioms. (Appendix B of this manual lists many of these idioms.)

Consider this convergence-checking code, from the **EULER** subroutine of FLO67:

```
      NSUP = 0
      HRMS = 0
      DO 80 K=1,KL
        DO 80 J=1,JL
          DO 80 I=1,IL
            V1 = W(I,J,K,2)*W(I,J,K,2) +
     .           W(I,J,K,3)*W(I,J,K,3) +
     .           W(I,J,K,4)*W(I,J,K,4)
            IF (V1.GE.(GAMMA*P(I,J,K)*W(I,J,K,1)))
     .        NSUP = NSUP + 1          ! COUNT
            V1 = W(I,J,K,5) + P(I,J,K)
            V1 = V1/W(I,J,K,1) - H0
            HRMS = HRMS + V1*V1        ! SUM
   80 CONTINUE
      HRMS = SQRT(HRMS/FLOAT((NX-1)*NY*NZ))
```

The variable **NSUP** is incremented once for each cell that exceeds a threshold. An incrementation contained in an **IF** statement is an idiomatic way of expressing the reduction operation named **COUNT** in Fortran 90. The variable **HRMS** is computed using another reduction operation: the **SUM** of the square of a value computed for each cell. This operation is also the idiomatic expression of **DOTPRODUCT**.

Rule 6: Scalable programs do not assume a particular memory model.

Certain features of Fortran 77 — particularly sequence association and storage association — assume a linear model of memory.

- Sequence association is the mapping of a multidimensional object in column-major order to a linear sequence of values.

```
REAL A(10,10), B(10,10)
...
CALL SUBA(A, B(5,5))
...
SUBROUTINE SUBA (C,D)
REAL C(100), D(10)
```

  In this example, the elements of matrices **A** and **B** are sequence associated with elements of the dummy vector arguments **C** and **D**.

- Storage association occurs when two or more variables (or arrays) share the same storage. This feature is exploited by **EQUIVALENCE**,

```
REAL A(2,50)
COMPLEX C(100)
EQUIVALENCE (A(1,1),C(1))
```

  and by some uses of **COMMON**,

```
SUBROUTINE SUBA
COMMON /BOUNDS/ IBX, IBY, IBZ
...
SUBROUTINE SUBB
COMMON /BOUNDS/ IBDS(3)
```

Sequence and storage association are difficult to implement and tend to be inefficient on memory systems that are not linear. These features are included in Fortran 90 for compatibility with Fortran 77, but scalable programs avoid them.

New features of Fortran 90, such as allocatable, automatic, and assumed-shape arrays, reduce the need for some uses of sequence and storage association. Other uses can be replaced by alternative Fortran 77 practices, which are both clearer and more scalable (see Chapter 4).

As general principles of scalable programming, the following are corollaries of Rule 6.

Rule 6a: Multidimensional arrays should be declared as such and be declared consistently throughout a program.

Arrays that change shape, particularly across subroutine boundaries, impose a severe burden on systems with nonlinear memory organization such as the CM system. In addition, performance is greatly affected by the way in which arrays are allocated, and the number and size of dimensions must be made explicit when they are allocated.

The original FLO67 code used storage pools from which it allocated different mesh variables. Pointers into these pools were passed to subroutines, which declared their arguments as multidimensional arrays. To enhance scalability, this original code,

```
DIMENSION W(IDX5),P(IDX), ...
...
K1 = 1
KW = 1
NC = (IE+1) * (JE+1) * (KE+1)
L1 = K1 + NC
LW = KW + 5*NC
...
CALL FLO (W(KW),P(K1), ..., W(LW),P(L1), ...)
```

was recoded as,

```
DIMENSION W_0(0:IEM,0:JEM,0:KEM,5),P_0(0:IEM,0:JEM,0:KEM)
DIMENSION W_1(0:IEM,0:JEM,0:KEM,5),P_1(0:IEM,0:JEM,0:KEM)
...
CALL FLO (W_0,P_0, ..., W_1,P_1, ...)
```

Rule 6b: Data layout directives should be supplied where necessary and helpful.

Most machines' performance is sensitive to data layout, because of its impact on locality of access. Distributed-memory parallel machines are particularly sensitive, since data layout directly influences the frequency of interprocessor communication. Scalable programs reflect an awareness of the layout conven-

tions of the target systems and use machine-specific directives to override the default layout where appropriate.

In the FLO67 program, the 4-dimensional array **W** shown under Rule 1 implements a 3-dimensional array of 5-element record structures; the structure components are selected by indexing along the fourth axis. For the CM system, the fourth dimension is better left undistributed, as specified by the CM Fortran compiler directive **LAYOUT**:

```
      REAL X(0:IE,0:JE,0:KE,5)
CMF$  LAYOUT W(:NEWS, :NEWS, :NEWS, :SERIAL)
```

A program might also benefit from aligning arrays in distributed memory, by means of the directive **ALIGN**. These directives can be inserted into the Fortran 77 source program, since they are ignored by other Fortran compilers.

Rule 6c: Arrays should be of appropriate size and declared such that they are easily changeable.

The optimal size for arrays and array dimensions varies among machines. On Cray machines, certain array sizes can degrade performance because of bank conflicts; the CM-2 system rewards power-of-2 array dimensions; other machines may have constraints stemming from page sizes or cache sizes; and so on. Array sizes are easily changed between compilations if they are declared with parameters instead of literal constants and if the **PARAMETER** statements are easy to locate. (**INCLUDE** files are convenient for this purpose.) This convention is also helpful in scaling a Fortran 77 program to different problem sizes.

Rule 7: Scalable programs use machine-specific libraries where available.

A call to a library routine fulfills the scalability goal of specifying the desired operation without dictating its algorithm. Sorting, matrix multiplication, equation solving, and Fourier transforms are examples of operations where no single algorithm is universally optimal. Using library procedures, along with some conditionalizing convention in the source program, helps make the code machine-independent and also takes advantage of any machine-specific tuning by the library implementor.

1.4 What Does the Converter Do?

CM Fortran includes all of Fortran 77 plus some extensions that support data parallel processing. These extensions include:

- Fortran 90 array operations and other language features that CM Fortran provides to express parallelism
- CM Fortran compiler's conventions for laying out data (in linear versus distributed memory, depending on whether parallel processing is appropriate)
- Fortran 90 method of passing array arguments and the semantics of procedure calls

The converter is designed to deal with all these translation issues. By viewing the whole program, it can provide the `cmf` compiler with code (including directives) that expresses parallelism; that avoids "array home" conflicts by directing that arrays processed in parallel are to be allocated in distributed memory; and that uses correct CM Fortran methods of passing distributed arrays as arguments.

1.4.1 Array Operations

An array operation is a computation that is performed on an array as a single entity, that is, on all the array's elements or a specified subset of them. CM Fortran adopts Fortran 90 array notation to indicate the set of elements. The array reference `A` is short for `A(1:N:1)`, indicating all the elements; the reference `A(1:N/2:2)` indicates every other element in the first half.

This triplet notation (express or implied) is analogous in function to the control variables in a `DO` construct, specifying a range of indices for array elements to be processed. But there is an important difference. A `DO` loop specifies indices in a particular sequential order, and processes one or more entire statements for an index before going on the the next index. An array operation, in contrast, may process indices in any order, but it must perform any given single operation (such as addition, multiplication, or assignment) for *all* indices before performing the next operation. The net effect is that an array operation can process array elements in any order without changing the result. The CM system takes advantage of this feature to process array elements in parallel.

The CMAX Converter analyzes Fortran 77 **DO** constructs to determine whether they are functionally equivalent to any CM Fortran array operation, and if so, converts the array notation and the loop(s).

This section illustrates the converter's loop-conversion capabilities; output may be simplified to clarify the essential transformation. Verbatim output from loop transformations is shown in Appendix B.

Loops without Dependences

Loop operations in which array elements do not interact vectorize straightforwardly into elemental operations:

Fortran 77

```
      DO 10 I=1,N
         A(I)=B(I)+C(I)
         DO 10 J=1,M
            P(I,J)=P(I,J)+R(I,J)
   10 CONTINUE
```

CM Fortran

```
A=B+C
P=P+R
```

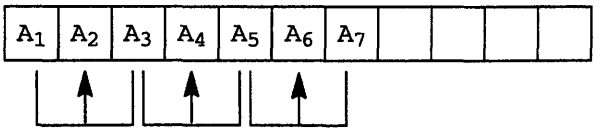
In other loop operations, the data elements may interact but there is no dependence between loop iterations. The converter's dependence analysis detects that a step-by-2 smoothing operation can vectorize to an array assignment. (Notice that the converter and CM Fortran accept the **END DO** statement.)

Fortran 77

```
DO I=2,N-1,2
  A(I)=(A(I-1)+A(I+1))/2
END DO
```

CM Fortran

```
A(2:N-1:2)=(A(:N-2:2)+
.    A(3:N:2))/2
```



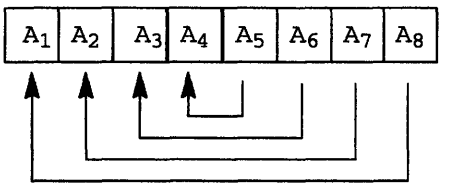
Similarly, the apparent dependence in a "mirror" operation is recognized as not a real dependence:

Fortran 77

```
DO I = 1,N/2
  A(I) = A(N-I+1)
END DO
```

CM Fortran

```
A(1:N/2) =
.    A(N:N/2:-1)
```



Loops That Express Common Idioms

Loops with certain well-structured dependences are functionally equivalent to CM Fortran intrinsic functions, Utility Library procedures, or **FORALL** statements. The converter recognizes the intent of the loop and substitutes the corresponding array operation. For example, consider these idiomatic loop constructions:

Fortran 77

```
X = 0.0
DO I = 1,N
  X = X + A(I)
END DO
```

CM Fortran

```
X = 0.0
X = X + SUM(A)
```

Fortran 77

```
DO J = 1,M
  Y(J) = 1.0
  DO I = 1,N
    Y(J) = Y(J)*B(I,J)
  END DO
END DO
```

CM Fortran

```
Y = 1.0
Y = Y * PRODUCT(B,DIM=1)
```

Loops with Embedded Conditionals

Loops with embedded **IF** statements may convert to a masked array assignment, such as **WHERE** or **FORALL**, or a masked intrinsic. For example:

Fortran 77

```
DO I = 1,N
  IF (A(I) .LT. 0.0) THEN
    B(I) = -B(I)
  ELSE
    C(I) = 0.0
  ENDIF
END DO
```

→

CM Fortran

```
WHERE (A .LT. 0.0)
  B = -B
ELSEWHERE
  C = 0.0
ENDWHERE
```

Other Vectorization Capabilities

The converter analyzes and may restructure code to facilitate vectorization.

For example, it promotes scalar values that are used with arrays into arrays of the appropriate shape, as illustrated above in the conversion of program BART. The array name is derived from the scalar's name, as with **X** and **X100**, and the array is aligned, if necessary, with another CM array to enhance the locality of access of their respective elements.

The converter may also fission loops to isolate vectorizable code from inherently serial operations:

Fortran 77

```
DO I = 1,N
  A(I) = B(I) + C(I)
  PRINT *, D(I)
  X = X + D(I)
END DO
```

→

CM Fortran

```
A = B + C
DO I = 1,N
  PRINT *, D(I)
END DO
X = X + SUM(D)
```

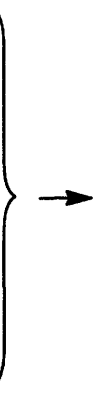
The converter may also transform a loop with an embedded subroutine call into a subroutine that contains the loop, thereby making the loop vectorizable. This *loop-pushing* transformation can be visualized as follows. Notice that the dummy variable `X` becomes an array in the intermediate form, and the loop becomes an array assignment in the CM Fortran version.

Fortran 77

```
SUBROUTINE PUSHME(A,N)
REAL A(N)
DO I = 1,N
   CALL PULLYOU(A(I))
END DO
END

SUBROUTINE PULLYOU(X)
REAL X
X = X + 1
END
```

Intermediate Form

```
SUBROUTINE PUSHME(A,N)
REAL A(N)
CALL PULLYOU_V1(A,N)
END

SUBROUTINE PULLYOU_V1(X,M)
REAL X(M)
DO I = 1,M
   X(I) = X(I) + 1
END DO
END
```



CM Fortran

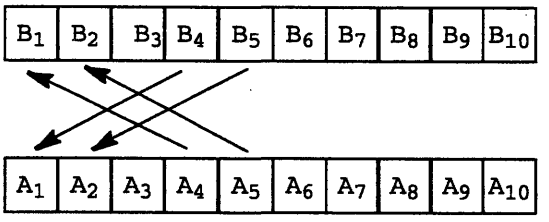
```
SUBROUTINE PUSHME(A,N)
REAL A(N)
CALL PULLYOU_V1(A,N)
END

SUBROUTINE PULLYOU_V1(X,M)
REAL X(M)
X = X + 1
END
```

Limits on Vectorization

If the converter cannot safely vectorize a **DO** loop, it passes the code through unchanged. The CM Fortran compiler then treats it as serial code. Some loop constructions that the converter does not translate are the following.

- Loops that contain a **STOP** or **RETURN** statement, a computed or assigned **GO TO** statement, or a forward or backward branch. In general, nonstructured constructs cannot convert to array operations.
- Loops containing calls to functions that are passed in as arguments.
- Loops that contain certain complicated data dependences. For example, the converter cannot vectorize the following:

```
REAL A(10), B(10)
DO I=1,7
  B(I) = A(I+3) + 10
  A(I) = B(I+3) + 1
END DO
```



- Loops for which the converter cannot resolve a question of data dependence. For example, the following loop can vectorize only if the index offset **M** is zero or positive:

Fortran 77

```
DO I=5,95
  A(I) = A(I+M)**B(I)
END DO
```

→

CM Fortran

```
A(5:95) =
.  A(5+M:95+M)**B(5:95)
```

Where **M** is negative, the loop must execute serially. If **M** is a run-time value, the converter cannot resolve this question.

NOTE: The converter provides directives with which the user can assert information that resolves some dependence issues. If the user asserts, in

this case, **CMAX$NODEPENDENCE** — meaning that **M** is necessarily zero or positive — the converter can vectorize the loop.

- Loops with structured dependences that have no functionally equivalent parallel operation in CM Fortran. For example, a parallel-prefix (or "scan") operation along a dimension is vectorizable. A scan along the diagonal of two dimensions, however, cannot (at present) be computed in parallel, and the converter does not vectorize it.

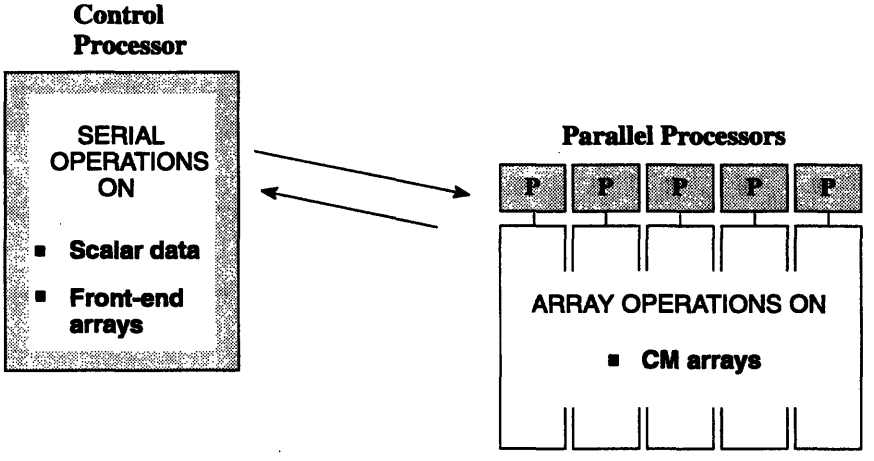
```
DO J=2,N
  DO I=2,N
    A(I,J) = A(I-1,J) + A(I,J-1)
  END DO
END DO
```

1.4.2 Array Homes

Every CM system consists of two processing components with different memory organizations:

- A serial control processor, called the *front end* on a CM-2 or CM-200 and the *partition manager* on the CM-5. This processor has the conventional linear memory organization.
- A parallel processing unit consisting of some number of processing elements, called *processors* (in the abstract) or *nodes* or *vector units* (depending on the CM hardware configuration). The memory of the parallel unit is distributed among the processors.

Data stored on the control processor, including all scalar data and *front-end arrays*, is operated upon serially. Other arrays *(CM arrays)* are laid out across the parallel processors, one or more elements in the local memory of each processor, and are operated upon in parallel. The processing component on which an array is stored is called its *home*. The CMAX Converter guides the compiler's array home decisions.



NOTE: The compiler's decomposition of data and code into "serial" versus "parallel" is the same regardless of whether a CM Fortran program is to run globally or "on a node." In the latter case, each CM-5 node serves as a control processor and the four vector units associated with each node are its dedicated parallel processing unit. See the CMMD documentation set for more information about executing CM Fortran programs "on a node."

The Compiler's Action

The CM Fortran compiler decides the home of each array in a program after attempting to determine whether it is to be processed serially or in parallel. Usually, it goes by whether the array is used in an array operation, although the programmer can control the decision with a compiler directive such as **LAYOUT**.

One of the chores of CM Fortran programming is to keep array homes consistent across procedures. Since the compiler does not at present perform interprocedural analysis, it may give an array different homes in different procedures. However:

- It is an error to perform an array operation on a front-end array.
- There is a severe performance penalty for performing a serial operation on a CM array.
- It is an error to pass an actual array argument from either home to a dummy argument with the other home.

For example:

```
PROGRAM FAILURE
INTEGER A(100,100)

DATA [initialize A]
CALL SUB(A)
PRINT *, A
STOP
END

SUBROUTINE SUB(B)
INTEGER B(100,100)
B = B**2
RETURN
END
```

The compiler allocates array **A** on the control processor, since there is nothing in the main program to indicate that **A** is to be processed in parallel. Array **B**, on the other hand, becomes a CM array by virtue of the array assignment in the subprogram. This program fails at run time when the front-end array is passed to the subroutine.

The Converter's Action

The CMAX Converter spares the programmer the problem of controlling array homes by inserting a **LAYOUT** directive for every array. Since the converter performs interprocedural analysis, it can determine whether an array is used in an array operation anywhere in the program and can thus avoid home mismatches across procedure boundaries. It may also create variants of certain subprograms, so that both front-end arrays and CM arrays can be passed to them as arguments.

The user can of course override the converter's action by means of in-line directives or converter command-line options. These features, and the situations where they might be useful, are described in Chapter 2 of this manual.

The converter makes array home decisions by the following rules:

- A user-supplied **cmf** directive **LAYOUT** or **cmax** option overrides all internal rules. CMAX propagates **LAYOUT** directives across program boundaries and issues a warning if it encounters inconsistency in their use.
- The converter also respects the **cmf** directive **ALIGN** within a program unit, but does not propagate it across program boundaries. (See Chapter 2 for restrictions on **ALIGN**.)
- All arrays that CM Fortran restricts to the control processor are marked as front-end arrays. These include arrays of **CHARACTER** type and arrays subject to an **EQUIVALENCE** statement.
- All arrays whose total size is below a threshhold number of elements are marked as front-end arrays. (The default threshhold size is 8 elements; you can specify a different number with the **-ShortVectorLength=***nn* converter switch. See Chapter 2.)
- All arrays in **COMMON** that change type or shape across program boundaries are marked as front-end arrays.
- All arrays passed as arguments to procedures that define a dummy argument of a different type or shape are marked as front-end arrays, except in those cases where the converter is able to adjust code to meet this CM Fortran restriction.

The last two rules reflect restrictions that CM Fortran imposes on arrays stored in distributed memory. Section 1.4.3 describes the current state of the converter's ability to generate code that meets these restrictions, thereby enabling more arrays to become CM arrays and be subject to array operations.

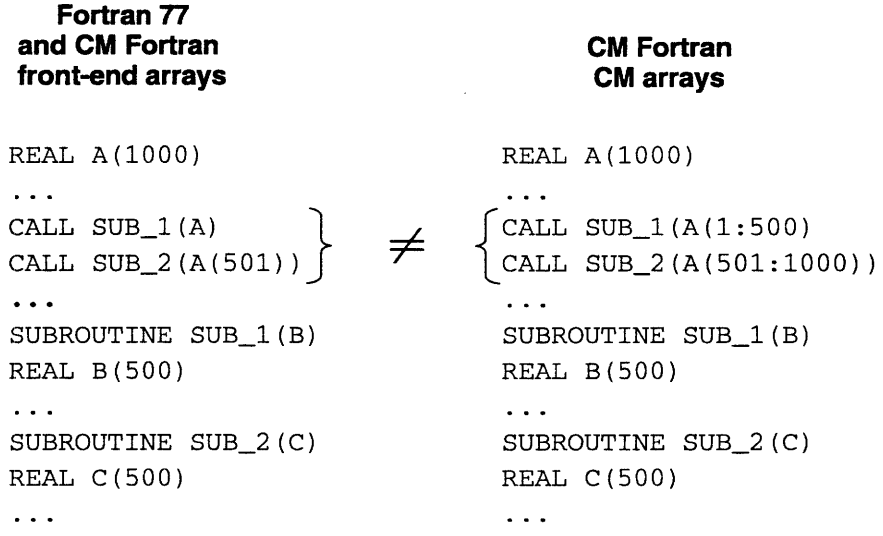
1.4.3 Argument Passing

CM Fortran adopts the Fortran 90 semantics for passing CM array arguments; it retains the Fortran 77 semantics for front-end array arguments. In Fortran 77, where arrays are passed by reference, an array name as an argument indicates the array's first element. In CM Fortran, where CM arrays are passed by descriptor, the reference to the name of a CM array indicates all its elements.

When the argument is a whole array, a procedure call *looks* the same in both cases:

```
REAL A(1000)
....
CALL SUB(A)
```

However, the **A** argument in Fortran 77 is short for **A(1)**, whereas in CM Fortran, the **A** is short for **A(1:1000)**. The semantic difference becomes apparent when the procedure is to operate on only part of the array. For instance, in the following fragment the first subroutine call operates on the first half of the array and second call on the second half:



Both the calls on the left fail when **A** is a CM array because it is an error to change array size or shape across subroutine boundaries. In fact, the argument to **SUB_2** is not an array at all, since the expression **A(501)** indicates a single array element. Where **A** is a CM array, this element is transferred to the control processor and passed as a scalar argument.

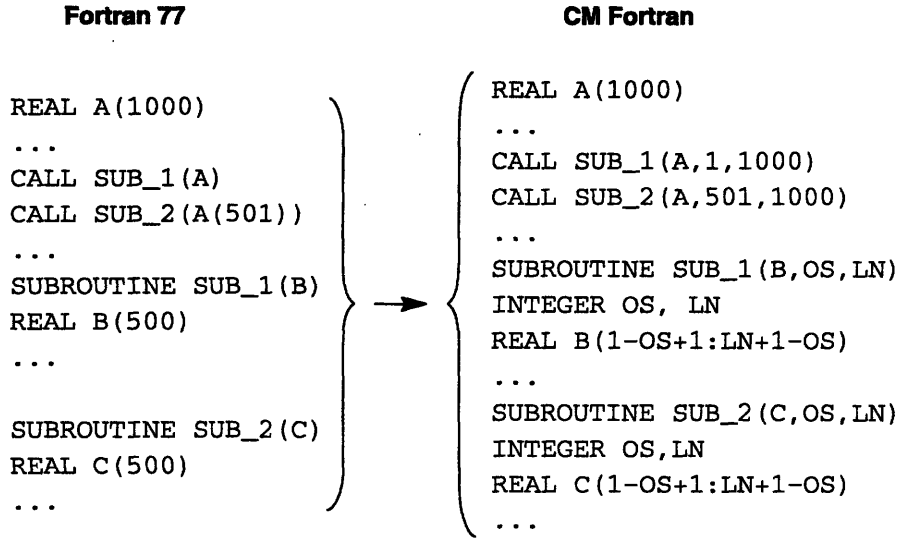
The CMAX Converter's interprocedural analysis enables it to generate code that meets the argument-passing requirements for CM arrays in three particular cases.

- Array arguments with a scalar subscript (one-dimensional)
- Array arguments with elided axes
- Assumed-size dummy array arguments (without change in rank)

Array Arguments with a Scalar Subscript

"Hidden array" arguments, like `A(501)` above, violate CM Fortran argument-passing restrictions by passing a scalar argument to a dummy array.

The converter changes the call to pass the whole array, its length, and an integer index into it, and changes the procedure definition accordingly. At present, this action occurs only if both the actual and the dummy arrays are 1-dimensional.



Array Arguments with Elided Axes

In Fortran 77, one may pass *n*–minus–*k*–dimensional contiguous slices of *n*-dimensional arrays to subprograms in the following way: all indices in the call other than rightmost *k* indices are the lower bound of the corresponding dimension.

```
REAL X(100,200,5)
CALL FOO(X(1,1,3))
     ...
SUBROUTINE FOO(Y)
REAL Y(100,200)
...
```

This type of memory reference violates CM Fortran argument-passing restrictions by reshaping an array across program boundaries. However, CMAX recognizes the intention of axis elision in this situation and transforms the code to pass a 2-dimensional section of the 3-dimensional array. The lower bound indices are turned into colons, indicating the whole dimension. (The user could improve performance of the output by inserting a CM Fortran **LAYOUT** directive to designate the rightmost axis of **X** a serial, or non-distributed, axis.)

```
      REAL X(100,200,5)
      ...
      CALL FOO(X(:,:,3))
      ...

      SUBROUTINE FOO(Y)
      REAL Y(100,200)
CMF$  LAYOUT Y(:NEWS, :NEWS)
      ...
```

Assumed-Size Dummy Array Arguments

CM Fortran does not support assumed-size dummy CM arrays (those with dimension lists ending in *), since Fortran 77 permits collapsing multiple axes into the * axis. Where CMAX has determined that the actual and dummy arguments are in fact of the same rank, and that they meet the other criteria for a CM home and parallel processing, it generates an assumed-shape dummy array. For example, this subroutine:

```
SUBROUTINE SUB(A,MAXA0,NSL)
REAL A(MAXA0, MAXA0, *)
DO IX = 1, MAXA0
  DO IY = 1, MAXA0
    DO ISL = 1, NSL
      A(IX, IY, ISL) = A(IX, IY, ISL) * 2
    END DO
  END DO
```

```
      END DO
      END
```

Is translated to:

```
      SUBROUTINE SUB(A,MAXA0,NSL)
      REAL A(:,:,:)
CMF$  LAYOUT A(:NEWS,:NEWS,:NEWS)
      A(:,:,:NSL) = A(:,:,:NSL) * 2
      END
```

Array Arguments Confined to Front End

The converter confines to the control processor any array arguments that violate CM Fortran argument-passing restrictions in ways that the converter cannot work around. Specifically, it marks as front-end arrays any array arguments that:

- Are passed with a scalar index ("hidden array argument") when either or both the actual and dummy arguments are multidimensional.
- Change type or shape across subprogram boundaries (including array arguments and arrays in COMMON) except for recognized cases of axis elision or of scalar indexing into a 1-dimensional array.
- Are passed to assumed-size arrays when the actual and dummy arguments are of different ranks.

As a result, no loops on these arrays can be vectorized. See Chapter 4 for some ways to work around these restrictions in the Fortran 77 program.

Chapter 2

The Conversion Process

This chapter describes the mechanics of using the CMAX Converter and suggests where the converter might fit into the overall process of porting Fortran 77 programs.

2.1 Invoking the Converter

The CMAX Converter is invoked from a shell prompt with the command **cmax**. This command creates and operates upon *packages*. A package is a set of source files that the converter treats as a complete program. All translation is of packages, not of source files directly, since the converter needs to perform interprocedural analysis to determine arrays' homes, resolve questions of data dependence, and determine the attributes of actual array arguments.

The command operates in one of three modes:

- Information only

  % **cmax** *information-operation*

- Package operation without translation

  % **cmax** *packname package-operation*

- Package translation with or without other package operation

  % **cmax** *packname* [**-T**] [*translation-options*]
  % **cmax** *packname* **-Add=** *sourcefile-list* **-T** [*translation-options*]

The arguments:

- *packname* is a user-supplied name of up to 14 characters, including alphanumerics, underscore, hyphen, or period.
- *sourcefile-list* is a space-delimited list of filenames with the extension `.f` or `.F`. (The extensions `.fcm` and `.FCM` are accepted when the translation option `-CMFortran` is specified.)

The `cmax` command options are summarized in Figure 4 and described below.

2.1.1 Quick Forms

- To create a package and translate it in one step:

  `% cmax mypack -Add= *.f -T` [*translation options*]

- To add file(s) to an existing package and translate it in one step:

  `% cmax mypack -Add=munch.f -T` [*translation options*]

- To translate an existing package:

  `% cmax mypack` [*translation options*]

2.1.2 Specifying Options

The `cmax` options are shown in Figure 4 in mixed case for readibility, but case is ignored and any non-ambiguous abbreviation is accepted. For options that take a list of arguments, a space after = is optional. Binary switches are specified with or without a prepended `no`.

2.1.3 Getting Information

`% cmax` *information-operation*

The *information-operation* may be:

- display a help message: `-Help`
- display a list of the packages in the current directory: `-Pack`

Option	Abbreviation	Default
Information Operations		
-Help	-Help	
-PackagesList	-Pack	
Package Operations		
-AddFiles= *sourcefile-list*	-Add=	
-ContentsList	-Cont	
-DeleteFiles= *sourcefile-list*	-Del=	
-RemovePackage	-Rem	
-TranslatePackage	-T	
Package Translation Options		
Input decisions		
-[no]CMFortran	-[no]CMF	-noCMF
-EntryPoint=*program-unit-name*	-E=	
-LineWidth=*number*	-LineWidth	72
-[no]PermitArraySyntax	-[no]PermitArr	-noPermitArr
-[no]PermitAutomaticArrays	-[no]PermitAuto	-noPermitAuto
-StatementBufferSize=*number*	-StatementBuffer	6700
Output decisions		
-CharForContinuation=*char*	-Char=	&
-[no]LineMapping	-[no]LineMap	-LineMap
-[no]ListingFile[=*filename*]	-[no]List	-noList
	-List[=]	*packname*.lis
-OutputExtension=*string*	-OutputE=	fcm
-OutputFile=*filename*	-OutputF=	
-Verbose=*number*	-Verb=	1
Conversion decisions		
-DefineSymbols= *name-list*	-Define=	
-[no]Dependence	-[no]Dep	-Dep
-[no]PermitKeywordsCMF	-[no]PermitKey	-noPermitKey
-[no]Permutation	-[no]Perm	-noPerm
-[no]Push	-[no]Push	-noPush
-[no]RestructureCode	-[no]Restruct	-Restruct
-[no]UnknownRoutinesSafe	-[no]Unknown	-Unknown
-ShortVectorLength=*number*	-ShortV=	8
-ShortLoopLength=*number*	-ShortL=	8
-[no]Vectorize	-[no]Vec	-Vec
-[no]ZeroArrays	-[no]Zero	-noZero

Figure 4. **cmax** command options, recommended abbreviations, and default values, if any.

2.1.4 Creating and Manipulating Packages (Without Translation)

% **cmax** *packname package-operation*

The *package-operation* may be:

- create a package with specified source files: **-Add=** *sourcefile-list*
- add files to an existing package: **-Add=** *sourcefile-list*
- delete files from a package: **-Del=** *sourcefile-list*
- display a list of the files in a package: **-Cont**
- remove a package: **-Rem**

Packages contain pointers to the source files, not copies of the files. Removing a package or deleting files from it does not affect the source files in any way; only the pointers are removed. These pointers enable the converter to get the latest version of any files on which it performs conversion operations.

2.1.5 Translating packages

% **cmax** *packname* [**-T**] [*translation-options*]
% **cmax** *packname* **-Add=** *sourcefile-list* **-T** [*translation-options*]

The *translation-options* may be any of the options listed under that category in Figure 4. The remainder of this section suggests some uses for various options.

The converter writes converted files and other user-visible output files to the current directory. As shown in Figure 5, it also creates a subdirectory named **CMAX** under the current directory. This subdirectory holds the converter's internal files and directories, which the user does not access directly. These internal items include packages and the package database files generated during conversion.

2.1.6 Removing Package Debris

If you typically use a "scratch" package during development, get into the habit of removing the package (with the **-Remove** option) before each session. Otherwise, you may end up with a package that contains multiple main programs or other anomalies. Any time you are in doubt about the state of a package or of the CMAX subdirectory, don't hesitate to remove them and create them anew.

Also, note that interrupting CMAX can leave the package in an inconsistent state. Remove and recreate the package to fix the problem.

```
% ls
sobel.f          sobel.input        sobel.params

% cmax sobelpack -add=sobel.f
cmax <sobelpack> [ version ]
Creating new package
Adding fortran source file(s) to package <sobelpack>
      sobel.f
CMAX execution complete.

% ls
CMAX             sobel.f            sobel.input
sobel.params

% cmax -pack
Packages in current directory:
      sobelpack
CMAX execution complete.

% cmax sobelpack -cont
cmax <sobelpack> [ version ]
Listing contents of package <sobelpack>
      sobel.f
CMAX execution complete.

% cmax sobelpack -list -verb=1
cmax <sobelpack> [ version ]
Translating package <sobelpack>
CMAX execution complete.

% ls
CMAX             sobel.fcm          sobel.params
sobelpack.lis    sobel.f            sobel.input
sobel.ttab

%
```

Figure 5. Sample package-creation and conversion session.

2.2 Controlling Conversion Output and Input

2.2.1 Output Decisions

Naming Converted Files

By default, the converter creates output files in the present working directory, using the input filename with the extension `.fcm`. To change the output filenames, use the translation option `-OutputFile=` or `-OutputExtension=.` Notice that `-OutputF=` causes the whole program to be written to a single file.

```
% cmax mypack -OutputE=FCM          => filename.FCM
% cmax mypack -OutputE=v1.fcm       => filename.v1.fcm
% cmax mypack -OutputF=whole.fcm    => whole.fcm
```

Suppressing Line-Mapping Files

The output includes a set of files named *filename*`.ttab`. These files are used by the Prism development environment to relate the source line numbers of CMAX input and output programs. If you will not be loading the converted output program into Prism (or using the Emacs utilities suggested in Appendix Section A.4), you can suppress the `.ttab` files with the option `-noLineMapping`.

```
% cmax mypack -noLineMap            => [ no .ttab files]
```

Choosing the Continuation Character

CMAX-generated statements that extend over multiple source lines are continued with the ampersand (`&`) character. Use the option `-CharForContinuation=` to specify another character.

```
% cmax mypack  -Char=>
% cmax mypack  -Char=.
```

Remember to quote characters that would otherwise be interpreted by other software. For example, the UNIX shell interprets `&` as a job-control character, and the `make` utility interprets `$` as a macro reference.

```
% cmax mypack  -Char=/&          & specified at the shell
% cmax mypack  -Char=$$          $ specified in makefile
```

Initializing Arrays to Zero

Some Fortran 77 programs target platforms that automatically initialize user memory to zero. Although this practice is not required by the standard, a program may assume it and not initialize some variables. Such a program could produce unexpected results in CM Fortran, since the Connection Machine system does not implement this practice.

For convenience, CMAX provides the option **-[no]ZeroArrays**, which causes the converter to initialize local CM arrays to zero. Scalar variables, front-end arrays, and arrays in **COMMON** are not affected.

```
% cmax mypack -Zero
```

2.2.2 Getting Conversion Information

Two options provide information about the converter's activities during program analysis and translation.

Progress Messages

The **-Verbose=** option causes CMAX to send a specified level of messages to standard output as it proceeds. The levels are:

0	General startup messages only
1	0 + report of actions
2	1 + messages at start of "passes" over program
3	2 + message at start of transformation of each subprogram
4	3 + message at start of transformation of each **DO** loop

```
% cmax mypack -Verb=1                => [ brief messages ]
```

Efficiency Notes in Listing File

The **-ListingFile** option causes CMAX to generate a summary report of its actions and decisions:

```
% cmax mypack -List                  => mypack.lis

% cmax mypack -List=today.lis        => today.lis
```

The listing file is in three parts:

- The Array Homes section lists every array, by subprogram, stating its home as determined by CMAX. The notes indicate the reason why each front-end array was not designated a CM array.
- The Routine Variants section lists the procedure variants CMAX has created and information concerning array arguments.
- The Statement-Level Efficiency Notes identify loops that did and did not vectorize. For those that did not, the inhibiting cause is shown.

2.2.3 Input Decisions

Accepting Wide Input Lines

The option **`-LineWidth=`** enables CMAX to accept source code lines of any width (in characters) up through the specified argument. The default is 72; the upper limit is 255.

```
% cmax mypack -LineWidth=132
```

Accepting Long Continued Statements

The option **`-StatementBufferSize=`** enables CMAX to accept source code statements of any length (in characters) up through the specified limit. This switch is useful for codes that continue statements over a large number of lines.

```
% cmax mypack -StatementBuffer=10000
```

The argument is the number of characters that will be accepted by CMAX's internal statement buffer. It excludes spaces and the first six characters of continuation lines. The default is 6700; there is no arbitrary upper limit.

2.3 Approaches to a Conversion Process

2.3.1 Performing Conditional Conversion

For programs that are targetted to multiple architectures, it is possible to convert conditionally, selecting only those segments that are appropriate to the CM system. As detailed below in Section 3.5, you can conditionalize a program either in-line or on a file-by-file basis.

Converting In-Line Conditionals

In-line conditionalizing is described below in Section 3.5. The converter recognizes a subset of the syntax of the C preprocessor **cpp**, such as:

```
#ifdef CM
      CALL [ CM library procedure ]
#else
      CALL [ generic routine ]
#endif
```

Symbol names are defined with the **-Define** option on the **cmax** command line:

```
% cmax mypack -Define=CM

% cmax mypack -Define=CM5 CM2 CMSIM
```

During conversion, the converter emulates certain actions of the C preprocessor **cpp**. In its analyses and transformations, it ignores all code that is conditional upon an undefined symbol. When writing the output files, it suppresses such code and also removes the **cpp**-like syntax from around translated code. The output **.fcm** files need no further preprocessing before **cmf** compilation.

NOTE: CMAX does not invoke the C preprocessor **cpp**, but rather emulates a subset of its functionality. If your favorite **cpp** directives are not recognized by CMAX, you might prefer to invoke the preprocessor directly on your program before converting it.

Converting File-Level Conditionals

Another approach to conditional conversion is to isolate machine-specific code in separate files and then process only the appropriate files for each target system. The UNIX **make** utility is useful for this purpose.

For example, suppose a Fortran 77 program consists of the files **main.f**, **munch.f**, **crunch.f**, and **sort-sun.f**. Since the sort algorithm is specially tuned for a Sun, you have an additional file, **sort-cm.fcm**, with a sort routine tuned for the CM and coded in CM Fortran. You would then use a **make** file to select and process the files for the CM system, including **sort-cm.fcm** but excluding **sort-sun.f**.

See Section 2.3.3 below for more information on using **make** with CMAX.

2.3.2 Converting Partial Programs

It is possible to approach a large conversion project by converting one or a few files at a time. You might, for instance, convert the individual subroutines or modules of a program separately.

Selecting the Partial Program

To begin, create a package containing only the subset of files to be converted at each step. If the main program unit is missing, use the **cmax** option **-EntryPoint=** to specify the root node of the subtree.

```
% cmax partprogram -Add=file.f -T -EntryPoint=MYSUB
```

If the main program unit is missing and you fail to specify another entry point, CMAX exits with an error:

```
Unable to locate a main entry point.
Use the -EntryPoint argument to select an
entry point from the following list:
    FOO              BAR              BAZ
```

Once you have specified an entry point, CMAX treats the package as a complete program for the purposes of interprocedural analysis. It looks only at the subprograms that are in the call hierarchy beginning at the specified root node. Any other subprograms in the package are ignored.

This behavior means that you have two options for translating a library of subprograms:

- Write a dummy main program that calls each of the subprograms.
- Create a separate package for each subprogram and specify the subprogram name as the entry point when translating its package.

Managing the Interfaces between Partial Programs

If you convert a program in stages, be aware that you are responsible for consistency among the separate parts. In particular, the converter cannot do the full interprocedural analysis for making valid array home decisions when it sees only part of the program.

The option **-[no]UnknownRoutinesSafe** enables you to control the assumptions the converter makes about unseen parts of the program. The positive form,

```
% cmax partprogram -E=MYSUB -UnknownRoutinesSafe
```

asserts that program units outside the package being translated do not contain code that would constrain any array in the package to a front-end home. The negative form,

```
% cmax partprogram -E=MYSUB -noUnknown
```

causes the converter to make conservative assumptions about unseen parts of the program, with the result that fewer loops may vectorize.

2.3.3 Using make Files

Because it needs to perform interprocedural analysis, CMAX does not support incremental conversion. Since a change to one program unit might have ramifications, such as constraining an array's home, in another unit, the converter needs to build a new program database for each package translation.

Thus, even when invoked via **make**, the converter does not limit itself to the source files that have changed since the last invocation. It does, however, refrain from overwriting previous output files if it has not changed them. The **make** utility then selects only the revised **.fcm** files for compilation and linking.

Provided with CMAX are two sample **make** files that you can adapt as needed. One, reproduced in Figure 6, converts and compiles for the CM Fortran global model; the other converts and compiles for the nodal modal, which uses calls to the CM message-passing library CMMD for communication.

These files, along with some sample **.f** files for conversion, are on line in:

- CM–5 systems: **/usr/examples/cmax/**
- CM–2/200 systems: **/usr/cm/examples/cmax/**

See your system administrator for the locations if these files have been moved.

```
# CMF definitions and defaults. Change CMFFLAGS and LIBS for
# use on non-CM5/VU systems.

CMF    =  cmf
OPT    =  -O
DEBUG  =
CMFFLAGS =  -vu $(OPT) $(DEBUG)
LIBS   =  /usr/lib/libcmax_cm5_vu.a

# CMAX definitions and defaults. Change as desired.

CMAX   =  /usr/bin/cmax
CMAXFLAGS=  -verbose=4 -list

# The package and the files it contains:
TARGET      =  simple
PACKAGE     =  simple
TRANSLATION =  simple-cmaxed
F77SRCS     =  main.f setup.f elwise.f total.f
CMFSRCS     =  $(F77SRCS:%.f=%.fcm)
OBJS   =  $(F77SRCS:%.f=%.o)

# The TRANSLATION (a dummy file) is used to keep track of
# how up-to-date the .fcm files are with respect to the .f
# files. TRANSLATION gets "touched" after CMAX is run. The
# TARGET depends on both the TRANSLATION and the OBJS:

$(TARGET):   $(TRANSLATION) $(OBJS)
        $(CMF) $(CMFFLAGS) $(OBJS) $(LIBS) -o $(TARGET)

# If any .f file is changed, CMAX will be run and .fcm files
# recompiled if needed.

$(TRANSLATION) : $(F77SRCS)
        $(CMAX) $(PACKAGE) -Add=$(F77SRCS) -T $(CMAXFLAGS)
        touch $@

# This rule tells how to compile the .fcm with cmf:

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

clean:
        rm -rf $(CMFSRCS) $(TARGET) $(TRANSLATION) *.o *.lis\
        *.ttab  $(CMAX) $(PACKAGE) -remove
```

Figure 6. Sample `make` file for CMAX, global program execution model

2.4 Controlling Conversion Rules with Options

The CMAX Converter follows certain defaults rules when choosing array homes and deciding whether to transform a loop or clone a procedure. You can control some of its decisions by changing its rules with converter command switches.

You can also control converter behavior on a case-by-case basis by inserting directives into the input program, as shown in Section 2.5.

2.4.1 Controlling Vectorization on Small Arrays or Loops

Data parallel processing is inefficient on a small array, since CM resources are left idle. Data parallel processing is also inefficient for a small number of loop iterations, since the overhead of beginning a parallel operation cannot be amortized effectively. The converter does not vectorize loops when either the iteration count or the array size is below a certain threshhold.

The default threshhold for both interation count and array size is 8. You can change the threshholds by using the **cmax** switches **-ShortVectorLength=** and **-ShortLoopLength=**.

```
% cmax mypack -ShortV=1000
% cmax mypack -ShortL=1000
```

Note that shorter loop and vector lengths may be appropriate for nodal CM Fortran programs compared with global programs.

2.4.2 Controlling Code Restructuring

Two kinds of code transformation can be inhibited or enabled by means of command options. By default, the converter transforms **IF/GOTO** constructions into block **IF/ENDIF** constructs, since only the latter are potentially vectorizable into masked array operations. To inhibit the restructuring of **IF/GOTO** constructions:

```
% cmax mypack -noRestructureCode
```

By default, the converter does not attempt to push loops that contain subroutine calls into a variant of the subroutine (a transformation described above in Section 1.4.1). To enable this behavior on a packagewide basis:

```
% cmax mypack -Push
```

The converter provides a directive that allows you to enable loop pushing for a particular loop or for the loops in a particular program unit (see Section 2.5).

2.4.3 Controlling Vectorization Analysis

Three **cmax** translation options are the global-scope variants of directives that either inhibit vectorization or assert information to permit vectorization. The directives are typically used to override the default command settings for particular cases. The default command settings are:

```
% cmax mypack -Vectorize

% cmax mypack -Dependence

% cmax mypack -noPermutation
```

See Section 2.5 for a discussion of the significance of these options.

2.4.4 Processing Fortran 77 with Array Extensions

Two **cmax** translation options enable the converter to recognize certain Fortran 90 array extensions in Fortran 77 (`.f`) input files. Without these switches, CMAX does not recognize Fortran 90 features in `.f` files and exits with an error when it encounters them.

Array Syntax

The option **-PermitArraySyntax** causes the converter to accept array syntax (references to whole arrays and array sections using Fortran 90 notation).

```
% cmax mypack -PermitArr
```

Automatic Arrays

The option **-PermitAutomaticArrays** causes the converter to accept Fortran 90 automatic arrays. An automatic array is an explicit-shape local array with a bound that of one or more variables.

```
% cmax mypack -PermitAuto
```

2.4.5 Accepting CM Fortran Keywords in .f Files

When CMAX encounters the name of a CM Fortran intrinsic procedure in a `.f` source file, it takes it to be the name of a user variable or procedure. The converter changes such names in the output program to avoid later confusing the `cmf` compiler. For example, if you define and call your own `SUM` function, `cmf` would read it as a call to the `SUM` intrinsic function instead. Renaming the user function avoids the ambiguity.

You can use the CMAX option `-[no]PermitKeywordsCMF` to change this behavior. The positive form, `-PermitKey`, causes the converter to accept the following CM Fortran reserved keywords in `.f` source files and treat them as intrinsic procedure names.

```
ALL
ANY
COUNT
CSHIFT
DIAGONAL
DLBOUND
DOTPRODUCT
DSHAPE
DUBOUND
EOSHIFT
FIRSTLOC
LASTLOC
MATMUL
MAXLOC
MAXVAL
MERGE
MINLOC
MINVAL
MVBITS
PACK
PRODUCT
PROJECT
RANK
REPLICATE
RESHAPE
SPREAD
SUM
TRANSPOSE
UNPACK
```

This option should be used with care. Since it inhibits changing user names, it may suppress some CMAX transformations. For example, if a transformation to the **SUM** intrinsic is appropriate but the program contains a user function of that name, the transformation does not occur. Also, since CMAX passes the user name through unchanged, you may get incorrect results when **cmf** treats **SUM** as a reference to the intrinsic rather than the user function.

2.5 Controlling Conversion with Directives

CMAX converter directives are specialized code comments that control certain translation actions or assert information. The converter also recognizes two CM Fortran compiler directives, and uses the information they assert in its analysis of the program.

2.5.1 CMAX$ Converter Directives

The CMAX Converter supports the following directives:

```
CMAX$ [NO]VECTORIZE      [ L | R ]
CMAX$ [NO]DEPENDENCE     [ L | R ]
CMAX$ [NO]PERMUTATION    [ L | R ]
CMAX$ [NO]PUSH           [ L | R ]
```

The **C** of **CMAX$** must appear in column 1; space after the **$** is optional. A directive can be specified in either its positive or its negative form (although one makes more sense than the other).

The argument specifies the directive's scope: **L** (the default) indicates that the directive applies only to the loop immediately following it (and not to loops nested within that loop); **R** indicates that the directive applies to all subsequent loops in the program unit.

The converter removes **CMAX$** directives when it writes converted files.

Selectively Inhibiting Vectorization

The **NOVECTORIZE** directive instructs the converter not to vectorize the loop or loops within its scope. It can be useful, for instance, when the outer loop of a nested **DO** construct performs an inherently serial operation. You can speed the conversion process by instructing the converter not to bother analyzing the possibility of vectorizing this sort of loop.

```
CMAX$ NOVECTORIZE
      DO J = 1,3
        DO I = 1,N
          PARTICLES(I,J) = PARTICLES(I,J) + DELTAV(I,J)
        END DO
      END DO
```

In some cases, a nonvectorizable outer loop will prevent CMAX from vectorizing inner loops. Putting a **NOVECTORIZE** directive on the outer loop can solve this problem. For example:

```
CMAX$ NOVECTORIZE
      DO K = 1,10
        A(5) = B(2) + B(92)
        B(92) = A(2) + A(5)
        DO I = 1,Npp
          X = X + A(I)
        END DO
      END DO
```

Because loop-level scope is the default, the directive in this example does not inhibit vectorization of the inner loop.

If you do want to suppress vectorization of nested loops, you can either supply a directive for each loop level,

```
CMAX$ NOVECTORIZE
      DO J = 1,M
CMAX$ NOVECTORIZE
        DO I = 1,N
          A(I,J) = B(I,J)**2 + C(I,J)**2
        END DO
      END DO
```

or you can specify a larger scope for the directive:

```
CMAX$ NOVECTORIZE R
      DO J = 1,M
        DO I = 1,N
          A(I,J) = B(I,J)**2 + C(I,J)**2
        END DO
      END DO
```

With routine-wide scope, this directive applies to both the outer and inner loops, and also to any loops that occur later in the program unit.

Asserting Independence

The **NODEPENDENCE** directive asserts that the loop or loops within its scope have no loop-carried data dependences. It is useful for enabling vectorization in cases where the converter cannot determine whether dependence exists. For example, the following loop cannot vectorize if the index offset **M** turns out at run time to be negative:

```
DO I=5,95
   A(I) = A(I+M)**B(I)
END DO
```

If the user asserts non-dependence, however, the loop can vectorize:

Fortran 77

```
CMAX$ NODEPENDENCE
      DO I=5,95
         A(I) = A(I+M)**B(I)
      END DO
```

→

CM Fortran

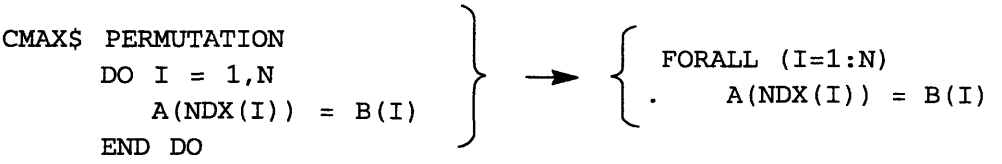
```
A(5:95) =
.   A(5+M:95+M)**B(I)
```

Asserting Uniqueness of Index Values

The **PERMUTATION** directive asserts that an indirection array on the left-hand side of an assignment is a permutation of a sequence, that is, it does not contain duplicate index values. For example, the following loop cannot vectorize if the index array **NDX** contains duplicate values:

```
DO I = 1,N
   A(NDX(I)) = B(I)
END DO
```

If the user asserts that **NDX** is a permutation, however, the loop can vectorize:



Controlling Loop Pushing

The **[NO]PUSH** directive enables or disables loop pushing for the loop or loops within its scope. This transformation is described in Section 1.4.1. The global (packagewide) default is **-noPush**.

2.5.2 CMF$ Compiler Directives

The CMAX Converter recognizes some forms of the CM Fortran compiler directives **LAYOUT** and **ALIGN**, which specify how individual arrays are to be allocated by **cmf**, and accepts their determination of array layout. These directives can appear in the Fortran 77 source program, since they are ignored by compilers other than **cmf**.

The converter parses all lines beginning with **CMAX$** or **CMF$**. Besides its own directives, it recognizes and accepts:

CMF$ LAYOUT	with axis specifiers **:NEWS**, **:SERIAL**, or **:SEND**
CMF$ ALIGN	except with non-zero axis offsets

CMAX does not recognize the following and treats them as syntax errors:

CMF$ COMMON	
CMF$ ALIGN	with non–zero offsets
CMF$ LAYOUT	with axis weights or with assumed-layout or detailed-layout axis ordering

It responds to unrecognized forms with a message like:

```
% cmax mypack -Trans
cmax <mypack> [ version ]
Translating package <mypack>

*******> Ignoring unrecognized CMF$ directive
cmax(_MAIN):CMF$COMMONFEONLY/A/
```

The **LAYOUT** and **ALIGN** directives contain information that needs to be kept consistent across a CM Fortran program. CMAX propagates to other program units the information on array home and layout that it derives from a recognized form of **LAYOUT**. It does not, however, propagate information from **ALIGN**. Because of the potential here for an error in the output program, CMAX issues the following warning whenever it ancounters **ALIGN**:

```
% cmax mypack -Trans
cmax <mypack> [ version ]
Translating package <mypack>

*******> Warning: ALIGN may propagate incorrectly
cmax(_MAIN):CMF$ALIGNB(I)WITHC(1,I)
```

The warning means that CMAX does not check that **ALIGN** is used correctly or consistently, and it does not propagate alignment information between procedures. For example, you might align a 1-dimensional array **N** with a column of a 2-dimensional array **M**, and then pass **N** to a procedure. CMAX does not propagate the noncanonical layout; you have to do it manually by creating 2-dimensional arrays where needed and aligning to them.

It is sometimes possible — and preferable — to use multiple **LAYOUT** directives to avoid an **ALIGN**. CMAX does propagate **LAYOUT** directives of the recognized forms.

Finally, you need to insert CMAX or CM Fortran directives explicitly to preserve the sense of other vendor's directives that already appear in a program. CMAX will derive no useful information from a program fragment such as:

```
C$DIR NO_RECURRENCE
CDIR$ IVDEP
C*$*ASSERT PERMUTATION (JND)
```

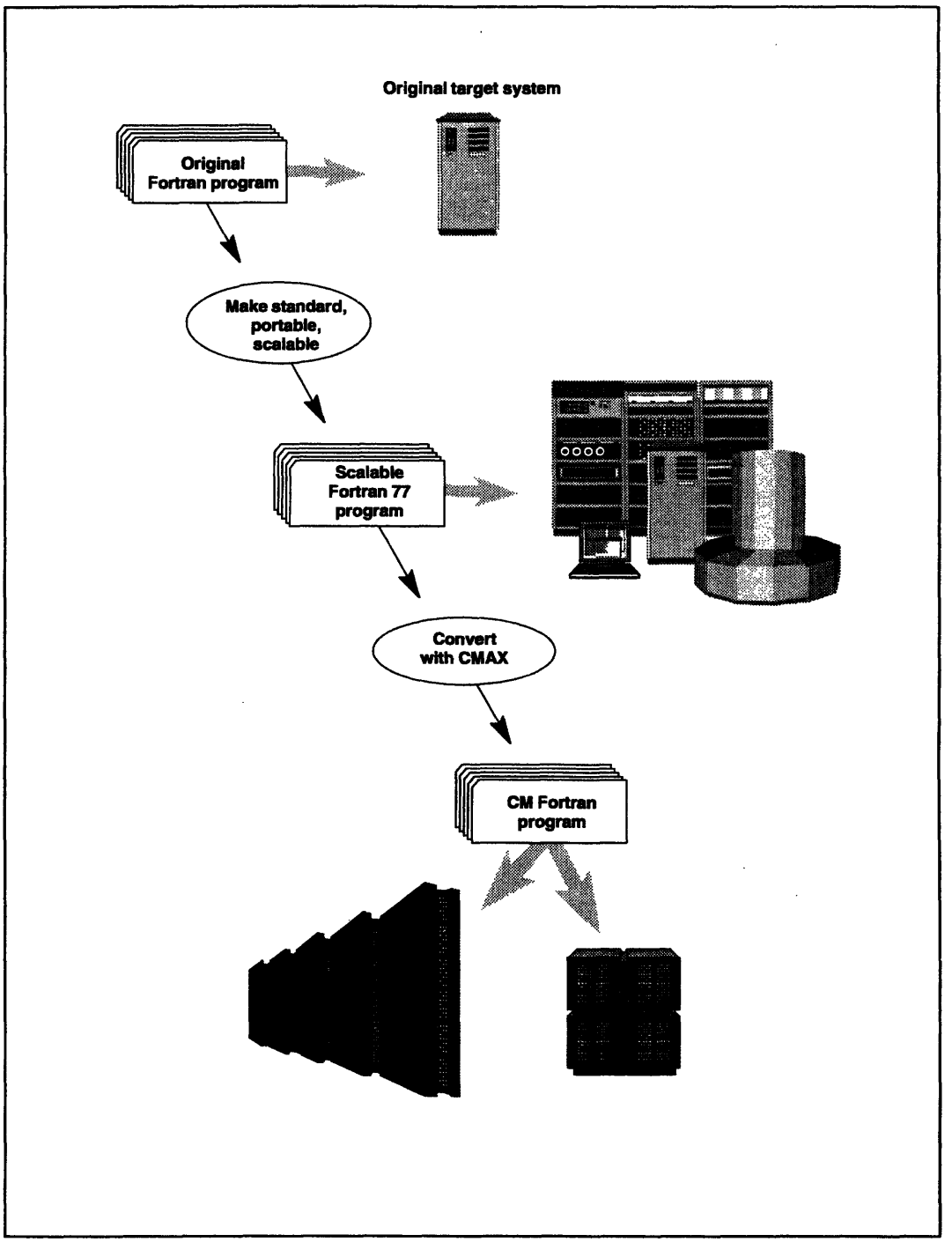


Figure 7. The generic porting process, from older Fortran to the CM and other systems.

2.6 The Porting Process

The CMAX Converter is meant to be used as part of a larger process. Specifically, it automates the translation of standard, scalable Fortran 77 to CM Fortran. For a newly written program that expresses the programmer's intent clearly and does not rely on any architecture-specific features, the translation may be nearly complete. Many older programs, however, are specifically tuned to a given target system or constrained to work around limitations in Fortran 77 by means of rather obscure idioms. For these programs, some amount of manual recoding is needed.

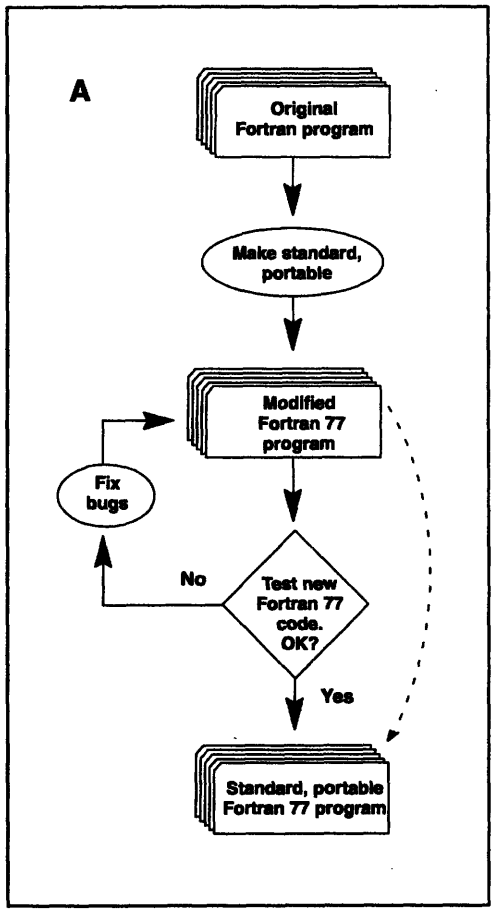
2.6.1 Where Does the Converter Fit In?

The nature of the overall porting process varies according to the state of the original program and whether it is now aimed at one architecture or many. Figure 7 shows a schematic view of the porting process: an older program is first brought to conformance with the Fortran 77 standard and is made scalable for porting to multiple systems, including the CM. What steps are needed to accomplish this, and where the converter can help, depends on the situation.

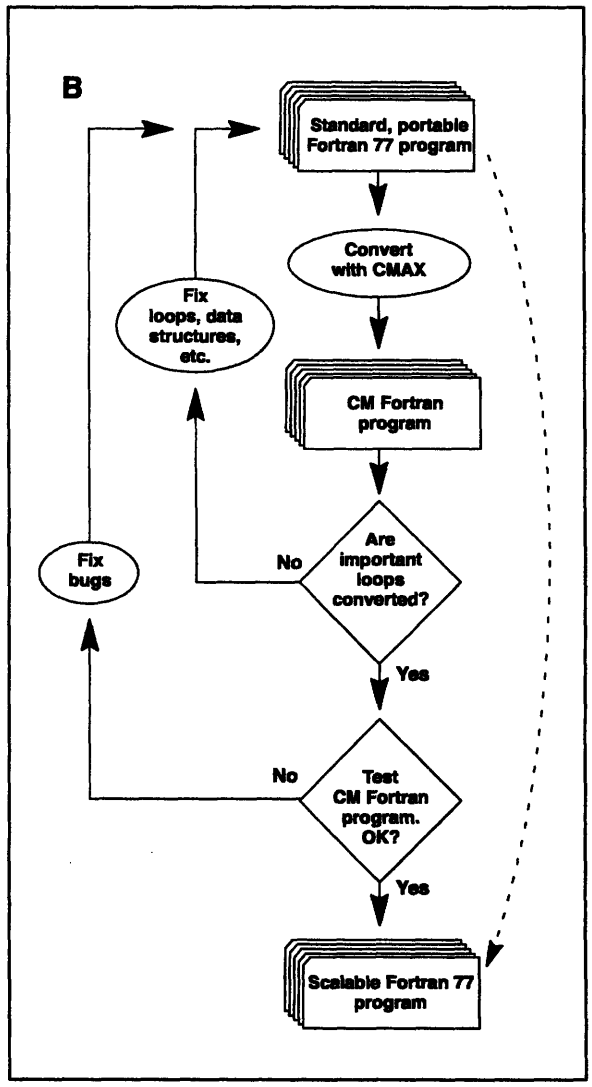
Consider the cross-product of two factors:

		Maintain in	
		Fortran 77	**CM Fortran**
Porting old code		**1** **A, B, C**	**2** **A, B, D**
Writing new code		**3** **B, C**	

Users in groups 1 and 2 are porting existing code to the CM system, with the intention either of maintaining the code in Fortran 77 for portability to other architectures (group 1) or of maintaining it in CM Fortran for the CM only (group 2). All these users need to perform the tasks shown in flowcharts A and B below: make the code standard and make the code scalable. The converter can help with the latter, as shown in chart B.

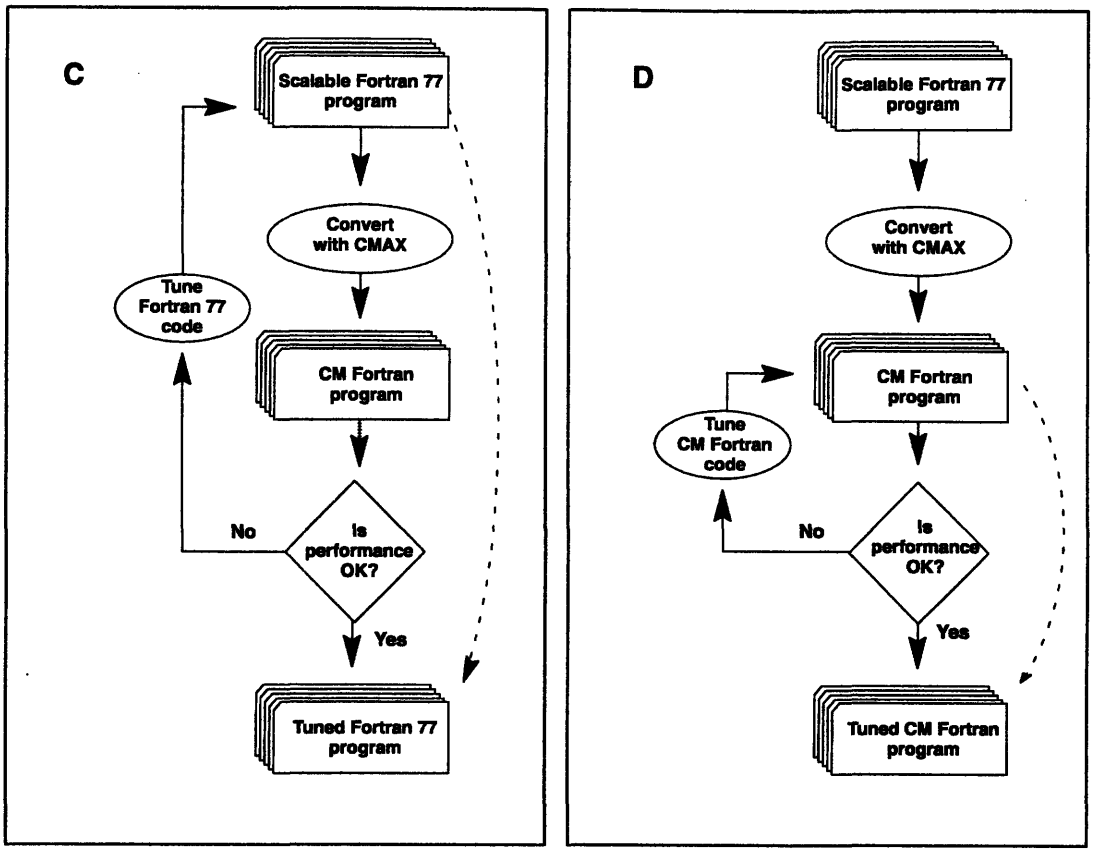


Older Fortran to standard, portable Fortran 77



Standard, portable Fortran 77 to scalable Fortran 77

Contrast the approach of users in group 3, who are writing new code in Fortran 77 for the CM and other systems. Such code could have portability and scalability engineered in, again with some help from the converter during development. The converter's real value, however, to someone maintaining in Fortran 77 is in fairly routine processing for the CM system. Notice in chart B that the converter's output program is being evaluated, but the input program is being developed.



Tuning for the CM among others: tune the input program

Tuning for the CM only: tune the output program

When the program reaches the performance tuning stage, the converter is most helpful to users maintaining in Fortran 77 (chart C). As in the development stage, the converter's output program is evaluated, but the input program is tuned. Users who are maintaining in CM Fortran would naturally tune the output program (chart D).

This manual focuses primarily on the steps in chart B: the one that is relevant to all three categories of users. Chapter 3 does offer some general hints on achieving portability (chart A), although the topic is too large and varied to be treated exhaustively. Chapter 4 focuses on achieving scalability (chart B), with specific programming hints for porting serial Fortran codes to the massively parallel CM system. The remaining step, performance tuning as in charts C and D, is not treated in this manual, since the performance issues in converted codes are the same as those in native CM Fortran codes.

2.6.2 Converting a Program Iteratively

With its default option settings, the converter does not accept CM Fortran (`.fcm`) files as input. However, you may want to feed CMAX its own output files, perhaps after adding directives or other aids to translation.

To cause CMAX to accept `.fcm` files, use the option `-[no]CMFortran`. In addition, you must use either `-OutputExtension` or `-OutputFile` to rename output files, since CMAX exits rather than overwriting an input file with output of the same name.

```
% cmax packname -CMF -OutputE=cmaxed.fcm
```

With this option supplied, CMAX will not attempt to vectorize loops that already contain parallel contructs, but will attempt again to vectorize serial loops.

When `-CMF` is specified, the converter recognizes the Fortran 90 language constructions specified by `-PermitArr`, `-PermitAuto`, and `-PermitKey`, plus other features like `WHERE` and `END WHERE`. It does not recognize nested `WHERE` or array pointers.

When `-CMF` is specified, CMAX parses all files in the package, including `.f` files, as CM Fortran source code. Take note of the caveats mentioned in connection with `-PermitKey` (Section 2.4.5 above) when using this switch: it is important to avoid confusion between user-supplied names and reserved keywords in both the `.fcm` files and `.f` files.

Chapter 3

General Porting Issues

This chapter notes some issues that arise in preparing a serial Fortran program for conversion to CM Fortran. These are general issues that arise in any effort to port an existing program to another system:

- Check for program correctness.
- Bring the program to conformance with the Fortran 77 standard.
- Conditionalize any machine-specific code.
- Make the program portable to the particular target architectures, including their shared language extensions and their respective restrictions and data layout conventions.

In the case of the CM system, the fourth task amounts largely to making the program *scalable*, as defined previously. Chapter 4 gives more information on targetting a program to the CM system, including some optimizations and some specific programming hints for enhancing scalability. This chapter offers some hints that pertain to the first three tasks.

3.1 Check the Input Program for Correctness

The converter cannot, obviously, turn an incorrect program into a correct one. In fact, subtle bugs in the program might become even harder to locate after conversion. It is worthwhile to begin by running the original Fortran program on its target system and verifying its results.

This step also provides baseline results against which to compare the results of later versions of the program. It is advisable to test and debug the program periodically throughout the porting process, checking its results against the baseline results.

3.2 Standardize the Input Program

The converter is designed to recognize and translate standard Fortran 77 code, although it does recognize some common language extensions. Many vendor-specific language extensions and nonstandard coding practices are not translated and may become errors in CM Fortran. Recoding such features is a necessary part of any effort to port to another system.

3.2.1 Some Code-Checking Tools

Many implementations of Fortran 77 provide compiler options that assist the porting process. For example, the Sun `f77` option `-ansi` reports most nonstandard language features; Sun and VAX FORTRAN both provide options that flag out-of-bounds array references; CONVEX provides an option that flags uninitialized variables; and so on. These and other tools may be available on your development platform.

Figure 8 illustrates the output of the Sun `-ansi` option. The sample program shown uses very common language extensions that Sun FORTRAN does in fact support: lowercase letters, names exceeding six characters, and the `IMPLICIT NONE`, `DO WHILE`, and `END DO` statements. Notice that the option inhibits `f77` compilation when nonstandard features are encountered.

CM Fortran and the CMAX Converter also support the features listed in the Figure 8 `-ansi` output, as well as `INCLUDE` lines, the `NAMELIST` statement, and the `DOUBLE COMPLEX` data type.

CMAX provides command options that cause it to accept certain Fortran 90 features in input programs. See Chapter 2 for the options of accepting array syntax, automatic arrays, and/or Fortran 90 intrinsic function names.

```
                program nonstandard
                implicit none
                real x,yisareal
                integer i
                do i=1,5
                  write(*,*) i
                end do
                do while (.false.)
                  x=yisareal*i
                end do
                stop 'This is the end.'
                end


% f77 nonstandard.f
nonstandard.f:
 MAIN nonstandard:

% a.out
   1
   2
   3
   4
   5
STOP: This is the end.
%

% f77 -ansi nonstandard.f
nonstandard.f:
 MAIN nonstandard:
"nonstandard.f", line 1: ANSI extension: input contains
      lower case letters
"nonstandard.f", line 2: ANSI extension: IMPLICIT NONE
      statement
"nonstandard.f", line 3: ANSI extension: symbolic name(s)
      longer than 6 characters
"nonstandard.f", line 5: ANSI extension: nonStandard form
      of DO statement
"nonstandard.f", line 8: ANSI extension: nonStandard form
      of DO statement
%
```

Figure 8. Nonstandard features reported by Sun's `-ansi` compiler option.

3.2.2 Code-Checking on the CM System

If you are converting a program for the CM system only, its general portability is less an issue than is its portability to CM Fortran. In this case, you can compile and execute the program on a CM control processor, using the `cmf` compiler and the Prism debugger to locate unsupported features.

```
% cp my-program.f my-program.fcm
```

If you choose this route, make sure that the compiler allocates common arrays on the control processor:

```
% cmf -fecommon my-program.fcm
```

If you allow the compiler to allocate common blocks on the parallel processing unit (as it does by default), your program will encounter compile-time errors from features that are not supported for CM arrays (such as character type) and will fail at run time from errors such as mismatched array homes. (The CMAX Converter will prevent most of these errors when it converts the program to CM Fortran.)

Once the program compiles and links successfully, you can test it for run-time errors and check the correctness of its results. If necessary, compile the program for debugging with Prism:

```
% cmf -fecommon -g my-program.fcm
```

Since this program executes on the control processor only, it runs much more slowly that it will when its `DO` loops have been converted to array operations.

3.3 Nonstandard Coding Practices

Some coding practices, though nonstandard, are very commonly used because they work on many sequential computers. When porting to CM Fortran, pay particular attention to the following trouble spots. The converter is not guaranteed to flag these practices, and they may show up later as obscure CM Fortran compile-time or run-time errors.

3.3.1 Uninitialized Variables

Some programs neglect to initialize variables, expecting uninitialized memory to contain a known value. It is sometimes assumed that Fortran 77 requires automatic initialization of variables in **COMMON** to zero.

In fact, the Fortran 77 standard does not require automatic initialization of user memory, and CM Fortran does not do so unless run-time safety is enabled. When safety is enabled, with the **cmf** option **-safety=10**, the CM sets user memory in the parallel processors to a NaN. This permits checking that floating-point variables are initialized in the program.

For maximum portability, a program should initialize all variables. However, you can use the **cmax** option **-ZeroArrays** to cause the converter to initialize CM local arrays to zero.

3.3.2 Out-of-Bounds Array References

An out-of-bounds array reference is an error in CM Fortran and should be removed from the input program. Many Fortran 77 compilers provide an option that helps locate instances of this nonstandard practice.

Emulating Assumed-Size Arrays

One place where an out-of-bounds reference might occur is with a dummy array argument declared as length one but used as an assumed-size array. This was a common practice under Fortran 66, and it persists in many widely distributed benchmark programs and numerical algorithms. For example, consider this subroutine from *Numerical Recipes**:

```
SUBROUTINE  ZBRAK(FX,X1,X2,N,XB1,XB2,NB)
DIMENSION XB1(1),XB2(1)
NBB=NB
NB=0
X=X1
DX=(X2-X1)/N
FP=FX(X)
```

(continued next page)

* *Numerical Recipes*: *The Art of Scientific Computing*, by William H. Press, Brian Flannery, Saul Teukolsky, and William Vetterling. Cambridge University Press, 1990

```
      DO 11 I=1,N
        X=X+DX
        FC=FX(X)
        IF(FC*FP.LT.0.) THEN
          NB=NB+1
          XB1(NB)=X-DX              ! Nonstandard reference
          XB2(NB)=X                 ! Nonstandard reference
        ENDIF
        FP=FC
        IF(NBB.EQ.NB)RETURN
   11 CONTINUE
      RETURN
      END
```

This practice usually works on systems with linear memory organization, as long as the value of **NB** does not exceed the storage allocated for the actual arrays passed to **XB1** and **XB2**. In standard Fortran 77, however — and particularly for portability to distributed-memory systems like the CM — you should not declare dummy arrays as length one unless the actual arrays really are.

Although the standard Fortran 77 method of declaring dummy arrays used like those above is the assumed-size array **XB1(*)**, this practice does not give a compiler (or the converter) much more information about the array than does the Fortran 66 practice. Distributed-memory systems are sensitive to array rank and shape, which both these practices obscure. The converter can transform assumed-size dummy arrays into correct CM Fortran code only if the rank of the dummy and actual array arguments match.

Linearizing Multidimensional Arrays

Another source of out-of-bounds references is the practice of linearizing multidimensional arrays instead of writing nested **DO** loops. Some older vectorizing compilers generated more efficient code from single loops, leading programmers to write:

```
REAL A(IM, JM), B(IM, JM), C(IM, JM)
...
DO I = 1,IM*JM
  A(I,1) = B(I,1) * C(I,1)
END DO
```

instead of

```
DO J = 1,JM
   DO I = 1,IM
      A(I,J) = B(I,J) * C(I,J)
   END DO
END DO
```

The first loop, although not standard-conforming, usually works on linear-memory systems because the column-major representation of this array in memory puts element `(1,2)` directly after element `(IM,1)`. This is not the case in distributed memory, however, where the "next" memory location after the end of an array dimension is not a meaningful concept. Linearizing the array obscures the 2-dimensional nature of the computation — an error in CM Fortran which the CMAX Converter may not be able to prevent. Since distributed-memory systems prohibit out-of-bounds array references — and since modern vectorizing compilers can generate efficient code from nested loops — it is best to avoid this practice and use the second form instead.

3.3.3 Aliasing from Above

Aliasing from above refers to overlaps between actual array arguments or between arrays in **COMMON** within a subprogram, such that the same storage is referenced with more than one name. For example, the arguments **X** and **Z** below reference the same memory locations.

```
REAL A(10), B(10)
...
CALL SUB(A,B,A,10)
...
SUBROUTINE SUB(X,Y,Z,N)
INTEGER N
REAL X(N), Y(N), Z(N)
...
```

Avoid this kind of argument-passing in portable code. Neither Fortran 77 nor CM Fortran permits any action within subroutine **SUB** that modifies either array **X** or array **Z**.

3.4 Remove Outmoded "Optimizations"

A number of coding practices, such as the linearizing of arrays noted above, are no longer necessary for their original target systems, and they inhibit the portability of a program to other systems. You can assist the CMAX Converter (and other systems' compilers) by removing outmoded optimizations from the program. This section mentions some examples of these practices.

Strip-Mining

Older vectorizing compilers generated optimal code for arrays of a length suited to their particular vector registers. This led programmers to "strip-mine" arrays for segments of the size needed:

```
REAL S(0:63)
DO I1 = 1,N,64
  DO I2 = 0,63
    S(I2) = A(I1+I2) * B(I1+I2)
    S(I2) = 3.0 * S(I2) + 42.0
    C(I1+I2) = S(I2)
  END DO
END DO
```

Since modern compilers do strip-mining automatically, there is no need to obscure code in this way. The loop above can be written more clearly as:

```
DO I = 1,N
  S = A(I) * B(I)
  S = 3.0 * S + 42.0
  C(I) = S
END DO
```

Unrolling Loops

Many modern compilers can unroll loops when needed for best performance; they no longer require you to perform this optimization by hand. A loop written as,

```
DO I = 1,N,4
  S = S + A(I) + A(I+1) + A(I+2) + A(I+3)
END DO
```

should be rewritten as,

```
DO I = 1,N
  S = S + A(I)
END DO
```

3.5 Conditionalize Machine-Specific Code

It is often necessary to write machine-specific code as part of a portable application. Such code might be:

- Expressions of machine-specific algorithms, such as solvers tuned for each of the target architectures
- Vendor-provided library functions, tuned by the implementors for best performance on their respective systems
- A set of array declarations or **PARAMETER** statements, which might vary between compilations with target system or with problem size
- For the CM system, code written in CM Fortran to express operations that the CMAX Converter does not yet translate from Fortran 77.

Machine-specific code must be encapsulated and conditionalized so that it is visible only to its intended target, and ignored by other systems. This process can be managed however you prefer. Two possible strategies are file-level conditionalizing and in-line conditionalizing.

File-Level Conditionalizing

If machine-specific code segments are placed in separate files, you can use a tool such as the UNIX **make** utility to select the appropriate files for conversion or compilation. See Chapter 2 for information on converting files via **make**.

In-Line Conditionalizing

The converter supports in-line conditionalizing in a manner similar to the C preprocessor **cpp**. As shown in Chapter 2, the **cmax** option **-DefineSymbols=** defines specified symbols during the conversion of a package. The converter ignores any code segments that are made conditional on an undefined symbol.

To conditionalize a CM-specific segment of code:

```
#ifdef CM
      CALL [ CM library procedure ]
#endif
```

The converter also recognizes the following form:

```
#if defined (symbol)
#elif defined (symbol)
#else
#endif
```

For example, when **cmax** is invoked with the option **-DefineSymbols=CM**, the converter processes only the CM-related code in this fragment:

```
#if defined (CM)
      [CM-specific code]
#elif defined (CRAY)
      [Cray-specific code]
#elif defined (CONVEX)
      [CONVEX-specific code]
#else
      [generic code]
#endif
```

The test condition can also be an expression involving symbols:

```
#if expression
#elif  expression
#else
#endif
```

Expressions may contain symbols along with parentheses and the C-style operators logical and (**&&**), or (**||**), and not (**!**). Any other item in an expression causes the expression to be treated as undefined, and the code segment that is conditional upon it is thus ignored. For example:

```
#if ( CM5 || CM200 ) && !CMSIM
...
#elif CMSIM
...
#else
...
#endif
```

If this subset of **cpp**-like syntax does not include features you wish to use, you have the option of invoking **cpp** directly on your input program before running it through CMAX.

3.6 Miscellaneous Conversion Hints

This section notes some programming practices that facilitate conversion via CMAX.

3.6.1 Revealing Reduction Idioms

CMAX recognizes many reductions of calls to intrinsic functions, such as the **MINVAL** of **ABS**. For example, this loop:

```
DO I = 1 , 100
  X = X + SIN(A(I))
  IF (A(I) .GT. 0.0) Y = Y + SQRT(A(I))
  Z = MIN(Z,ABS(A(I)))
ENDDO
```

converts to:

```
X = X + SUM(SIN(A))
Y = Y + SUM(SQRT(A),MASK=A .GT. 0.0)
Z = MIN(Z,MINVAL(ABS(A)))
```

For constructions that are not recognized, you can sometimes manually split the expression to reveal a recognized idiom. For example, reductions of complicated expressions will sometimes not vectorize because the right hand side of the assignment statement cannot be expressed in array syntax. Splitting the expression across two statements will enable vectorization. Consider:

```
DO I = 1 , 100
  X = X + ABS(A(I) * SIN(FLOAT(I)))
ENDDO
```

can be rewritten as:

```
DO I = 1 , 100
  T = ABS(A(I) * SIN(FLOAT(I)))
  X = X + T
ENDDO
```

to be converted into:

```
FORALL (I = 1:100)
&  T100(I) = ABS(A(I) * SIN(FLOAT(I)))
X = X + SUM(T100)
```

3.6.2 Include Files

CMAX converts C-style **#include** directives into Fortran-style **INCLUDE** lines. This code:

```
        SUBROUTINE S002()
        IMPLICIT NONE
        INTEGER I,K,P,M,N

#include "foo.FCM"

        INTEGER SL
        END
```

Is transformed into this code:

```
        SUBROUTINE s002()
        IMPLICIT NONE
        INTEGER I,K,P,M,N

        INCLUDE 'foo.FCM'

        INTEGER SL
        END
```

CMAX expands include files in-line if they are modified during conversion. Also, CMAX-generated **LAYOUT** directives for arrays that are declared in include files appear in the output **.fcm** files rather than in the include files themselves. This behavior can lead to output files that are longer and perhaps less readable than they would otherwise be. You can increase output readability in the course of an iterative porting process by clipping out the generated **LAYOUT** directives (or write your own if you prefer) and putting them in the include files yourself. (Compilers other than **cmf** will ignore them.) Then CMAX will not generate new ones with every run.

CM Fortran Limit on Include Files

CM Fortran versions prior to Version 2.1 Beta 1 place a limit of 20 on the number of include files used in a single source file. If the limit is exceeded, compilation aborts with a `"File stack overflow"` error. There are two workarounds for this limitation:

- Concatenate include files, but be careful to check whether the code uses the same names for variables in different **COMMON** blocks.
- Use C-like **#include** and rename your output file(s) to **.FCM**. The **cmf** command will invoke **cpp** to preprocess the files before invoking the CM Fortran compiler.

In Version 2.1 Beta 1, the **cmf** compiler accepts up to 252 include files per source file, nested to a maximum depth of 19.

3.6.3 SAVE Variables in Procedure Variants

Because CM Fortran requires that actual array arguments match the corresponding dummies in home and shape, CMAX generates procedure variants where needed to accommodate more than one category of array arguments. **SAVE** variables cannot be shared between these independent procedures.

For example, if CMAX needs to clone

```
SUBROUTINE HUMPTY(...)
INTEGER SAT, ONA, WALL
SAVE
...
```

it will warn:

```
Warning: Cloning routine HUMPTY which has 3 local save
variables. SAVE variables will not be shared between
clones:
        SAT
        ONA
        WALL
```

You can preserve the original intent of the subroutine by using a **COMMON** block instead of the **SAVE** attribute:

```
SUBROUTINE HUMPTY(...)
INTEGER SAT, ONA, WALL
COMMON /DUMPTY/ SAT, ONA, WALL
...
```

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

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).

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.

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.

4.2.2 Recoding for Parallel File I/O

Prior to Version 2.1 Beta 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.

In 2.1 Beta 1, the Fortran I/O statements perform parallel I/O on CM arrays, making the utility routines unnecessary.

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 proces-

sor) in versions prior to 2.1 Beta 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 . . .	**. . . like this**

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

4.2.4 Vendor-Specific Difference in READ/WRITE Behavior

A problem you might encounter in porting from Sun or VAX Fortran to CM Fortran is a difference in the behavior of the **READ** and **WRITE** statements. Both the Sun and VAX implementations act on the first character specified, whereas the standard defines it as a carriage control character (which essentially means it is ignored). CM Fortran is standard-conforming in this respect, but its output may be unexpected in a code written for Sun or **VAX**. For example, consider:

```
      PROGRAM OUTPUT
      WRITE(6,100) 'No leading blank'
      WRITE(6,100) ' One leading blank'
100   FORMAT(A)
      END
```

In Sun FORTRAN this gives:

```
% output
No leading blank
One leading blank
```

In CM Fortran this gives:

```
% output
o leading blank
One leading blank
```

A workaround that preserves the expected behavior is to write the **FORMAT** statement as:

```
100  FORMAT (1X,A)
```

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.

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
```

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.

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
```

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
```

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.

```
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 9), 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.

```
      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 9. The HOMER program, coded in Fortran 77 with a common array.

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 10) 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.

```
      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 10. The LISA program, coded in CM Fortran with automatic arrays.

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 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 11 for an example of a Simpson integrator coded with the CMAX dynamic allocation utility.

```
      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 11. The MARGE program, coded in Fortran 77 with the CMAX dynamic allocation utility.

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
```

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

---

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)
...
```

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.

Appendixes

Appendix A

User Interface Reference

This appendix provides reference material for the features of the CMAX Converter's user interface. These are:

- The **cmax** command
- The CMAX converter directives
- The CMAX library
- Some sample EMACS utilities

A.1 The cmax Command

The **cmax** command creates and operates upon *packages*. A package is a specialized subdirectory containing the Fortran source files that the CMAX Converter treats as a complete program. Figure 12 summarizes the command's actions and, when the action is translation of a package, its translation options.

The **cmax** command can be invoked in several modes (as detailed in Section A.1.1). In summary, the modes are:

- Information only

  cmax *information-operation*

- Package operation without translation

  cmax *packname package-operation*

- Package translation with or without other package operation

  cmax *packname* [*translation-options*]
  cmax *packname* **-T** [*translation-options*]
  cmax *packname* **-Add=** *sourcefile-list* **-T** [*translation-options*]

  packname can be up to 14 characters, including any alphanumeric as well as underscore, hyphen, or period. The first character must be alphanumeric; case is ignored.

The following rules apply when specifying any **cmax** command option:

- Case is ignored.
- Any nonambiguous abbreviation is accepted. The abbreviations shown in Figure 12 are recommended, since they are unlikely to conflict with the names of internal options and possible future options.
- Where an = sign appears after an option name, it is required syntax. For options that take a list of arguments (**-Add=**, **-Del=**, and **-Define=**), a space after = is optional. For options that take a single argument, no space after = is permitted.
- Binary switches are specified with or without a prepended **no**. The brackets shown in Figure 12 are a documentation convention only.

Option	Abbreviation	Default
Information Operations		
-Help	-Help	
-PackagesList	-Pack	
Package Operations		
-AddFiles= *sourcefile-list*	-Add=	
-ContentsList	-Cont	
-DeleteFiles= *sourcefile-list*	-Del=	
-RemovePackage	-Rem	
-TranslatePackage	-T	
Package Translation Options		
Input decisions		
-[no]CMFortran	-[no]CMF	-noCMF
-EntryPoint=*program-unit-name*	-E=	
-LineWidth=*number*	-LineWidth	72
-[no]PermitArraySyntax	-[no]PermitArr	-noPermitArr
-[no]PermitAutomaticArrays	-[no]PermitAuto	-noPermitAuto
-StatementBufferSize=*number*	-StatementBuffer	6700
Output decisions		
-CharForContinuation=*char*	-Char=	&
-[no]LineMapping	-[no]LineMap	-LineMap
-[no]ListingFile[=*filename*]	-[no]List	-noList
	-List[=]	*packname*.lis
-OutputExtension=*string*	-OutputE=	fcm
-OutputFile=*filename*	-OutputF=	
-Verbose=*number*	-Verb=	1
Conversion decisions		
-DefineSymbols=*name-list*	-Define=	
-[no]Dependence	-[no]Dep	-Dep
-[no]PermitKeywordsCMF	-[no]PermitKey	-noPermitKey
-[no]Permutation	-[no]Perm	-noPerm
-[no]Push	-[no]Push	-noPush
-[no]RestructureCode	-[no]Restruct	-Restruct
-[no]UnknownRoutinesSafe	-[no]Unknown	-Unknown
-ShortVectorLength=*number*	-ShortV=	8
-ShortLoopLength=*number*	-ShortL=	8
-[no]Vectorize	-[no]Vec	-Vec
-[no]ZeroArrays	-[no]Zero	-noZero

Figure 12. **cmax** command options, recommended abbreviations, and default values, if any.

A.1.1 Invoking the cmax Command

The general rules for invoking the **cmax** command are:

- Options that specify operations (the first two categories in Figure 12) are mutually incompatible, *except* that **-T** (translate) can be used with **-Add=**.
- If a *packname* but no operation is specified on the command line, **-T** is assumed. Any translation option may appear.
- Translation options are incompatible with all operations *except* **-T** (which may be assumed) and the combination of **-Add=** and **-T**. If **-Add=** is specified along with any translation option, **-T** must appear.

From these rules derive the following **cmax** command modes:

- Getting information

  % cmax *information-operation*

  The *information-operation* may be:

  - display a help message: **-Help**
  - display a list of the packages in the current directory: **-Pack**

- Creating and manipulating packages (without translation)

  % cmax *packname package-operation*

  The *package-operation* may be:

  - create a package with specified source files: **-Add=** *sourcefile-list*
  - add files to an existing package: **-Add=** *sourcefile-list*
  - delete files from a package: **-Del=** *sourcefile-list*
  - display a list of the files in a package: **-Cont**
  - remove a package: **-Rem**

  sourcefile-list is a space-delimited list of filenames with extensions **.f**, **.fcm**, **.F**, or **.FCM**.

  Packages contain pointers to source files, not copies. Removing a package or deleting files from it removes pointers only; source files are not affected.

- Translating packages

  % **cmax** *packname* [*translation-options*]
  % **cmax** *packname* **-T** [*translation-options*]
  % **cmax** *packname* **-Add=** *sourcefile-list* **-T** [*translation-options*]

  The *translation-options* may be any of the options listed under that category in Figure 12 and described below.

A.1.2 cmax Translation Options

Input Decisions

The following options control certain decisions the CMAX Converter makes concerning acceptable input.

-[no]CMFortran Default: **-noCMFortran**

When enabled, accept input files with the filename extension **.fcm** or **.FCM**. In this case, CMAX parses all files in the package as CM Fortran files, regardless of their extension. When disabled, CMAX exits with an error when it encounters an **.fcm** source file. The positive form should be used with either **-OutputExtension** or **-OutputFile** to rename output, since CMAX exits rather than overwriting an input file with output of the same name.

-EntryPoint= *program-unit-name*

Use the specified program unit as the entry point (root node) of the program.

-LineWidth=*number* Default: **-LineWidth=72**

Accept source files with line width (in characters) up through the specified *number*. The *number* argument is an integer less than or equal to 255.

-[no]PermitArraySyntax Default: **-noPermitArraySyntax**

When enabled, accept whole arrays and array sections in expressions in **.f** source files.

-[no]PermitAutomaticArrays Default: **-noPermitAutomatic**

When enabled, accept automatic arrays in **.f** source files. An automatic array is an explicit-shape local array with a bound that depends on the values of one or more variables.

-StatementBufferSize=*number* Default: **-StatementBuffer=6700**

Accept source files with line length (in characters and including continuations) up through the specified *number*. There is no aribtrary limit on the value of the integer *number* argument.

Output Decisions

The following options control certain decisions the CMAX Converter makes concerning output from the translation process.

-CharForContinuation=*char* Default: **-CharForContinuation=&**

Specify the continuation character used in generated code.

-[no]LineMapping Default: **-LineMapping**

When enabled, generate files named *sourcefile***.ttab** that contain translation tables. These **.ttab** files are used by the Prism development environment to relate the source code lines in the input and output files.

-[no]ListingFile Default: **-noListingFile**
-ListingFile[=*filename*] Default: *packname***.lis**

When enabled, generate a listing file for the translation operation. This file reports the homes of all arrays, the subprogram variants the converter has generated, and statement-level efficiency notes. The listing file is named *packname***.lis** if no other name is supplied.

-OutputExtension=*string* Default: **-OutputExtension=fcm**

Write converted output files to the present working directory with filenames constructed of the sourcefile names and the extension *string*. The *string* argument may contain periods, such as **cmaxed.fcm**.

-OutputFile=*filename*

Write the converted program to a single file *filename* in the present working directory.

-Verbose=*number* Default: **-Verbose=1**

Set the level of informational messages issued to **stdout** during program analysis and conversion. The levels are:

0	General startup messages only
1	0 + report of actions
2	1 + messages at start of "passes" over program
3	2 + message at start of transformation of each subprogram
4	3 + message at start of transformation of each **DO** loop

Conversion Decisions

The following options control certain decision rules that the CMAX Converter applies in the course of translating a package.

-DefineSymbols= *name-list*

Treat the specified *name* or *names* as defined symbols when preprocessing source files. Code that is conditional upon an undefined symbol is ignored during translation and does not appear in the output program.

-[no]Dependence Default: **-Dependence**

When enabled, asserts that **DO** constructs in the package may exhibit loop-carried dependences. (This option is the global scope variant of the CMAX directive **[NO]DEPENDENCE**.)

-[no]PermitKeywordsCMF Default: **-noPermitKeywordsCMF**

When enabled, accept CM Fortran reserved keywords in **.f** source files and treat them as intrinsic procedure names. If disabled, CMAX changes user-supplied names for variables and procedures when they conflict with CM Fortran keywords.

-[no]Permutation Default: **-noPermutation**

When enabled, asserts that index arrays used in **DO** constructs to perform indirect addressing do not contain duplicate index values. (This option is the global scope variant of the CMAX directive **[NO]PERMUTATION**.)

-[no]Push Default: **-noPush**

Enables or disables "loop pushing" as a technique of making vectorization possible. Loop pushing refers to rewriting a **DO** construct that contains a subroutine call as a call to a subroutine that contains the **DO** construct. Loop pushing is not currently implemented for functions. (This option is the global scope variant of the CMAX directive **[NO]PUSH**.)

-[no]RestructureCode Default: **-RestructureCode**

Enables or disables the conversion of **IF/GOTO** constructions to block **IF/ENDIF** constructs where possible. Block **IF/ENDIF** constructs inside **DO** constructs may be translatable to masked array assignments (**WHERE** or **FORALL**).

-ShortLoopLength=*number* Default: **ShortLoopLength=8**

Do not vectorize **DO** constructs with explicit loop iteration counts (product of extents in loop nest) of less than *number*.

-ShortVectorLength=*number* Default: **ShortVectorLength=8**

Do not vectorize **DO** constructs on arrays whose size (product of dimension extents) is less than *number*.

-[no]UnknownRoutinesSafe Default: **-UnknownRoutinesSafe**

When enabled, asserts that unseen program units (those outside the package being translated) do not contain code that would constrain any array in the package to a front-end home.

-[no]Vectorize Default: **-Vectorize**

Enables or disables the translation of **DO** constructs into array operations. (This option is the global scope variant of the CMAX directive **[NO]VECTORIZE**.)

-[no]ZeroArrays Default: **-noZeroArrays**

When enabled, generate code that initializes local CM arrays to zero. Scalar variables, front-end arrays, and arrays in common are not affected.

A.2 CMAX Converter Directives

CMAX Converter directives are specialized code comments that control certain translation actions or decision rules. A converter directive may:

- Direct the converter not to attempt to vectorize a loop or loops
- Direct the converter not to attempt "loop pushing" on a loop or loops
- Assert that a particular loop or loops do not exhibit loop-carried data dependences
- Assert that the index array(s) used in a loop or loops to perform indirect addressing do not contain duplicate index values

The scope of a converter directive can be either a particular `DO` construct or a program unit. Comparable `cmax` command options apply the directives' actions or assertions globally throughout the package during program translation. Typically, in-line directives are used to override the default command action for particular loops or for particular program units.

A.2.1 Directive Syntax

A directive is a comment line of the form

CMAX$*directive-name* [*scope-spec*]

directive-name is one of:

[NO]VECTORIZE	enable/disable vectorization
[NO]PUSH	enable/disable loop pushing
[NO]DEPENDENCE	assert/deny possibility of data dependence
[NO]PERMUTATION	assert/deny non-repetition of index values

scope-spec is one of:

L [the default]	the subsequent loop (but not loops inside it)
R	all subsequent loops in the program unit

The directive prefix **CMAX$** must begin in column one, and one or more spaces must separate the directive name from the scope specifier (if supplied).

A.2.2 Synopsis of Directives

This section summarizes the forms and the behavior of the CMAX Converter directives and the corresponding command options.

Loop Vectorization

The positive form enables conversion of loop(s) to array operations.

`CMAX$VECTORIZE`	applies to subsequent loop
`CMAX$VECTORIZE L`	applies to subsequent loop
`CMAX$VECTORIZE R`	applies to all subsequent loops in routine
`-Vectorize`	applies to all loops in package

The negative form disables conversion of loop(s) to array operations.

`CMAX$NOVECTORIZE`	applies to subsequent loop
`CMAX$NOVECTORIZE L`	applies to subsequent loop
`CMAX$NOVECTORIZE R`	applies to all subsequent loops in routine
`-noVectorize`	applies to all loops in package

The default global setting is `-Vectorize`.

Loop Pushing

The positive form enables the "pushing" of loop(s) that contain subroutine calls. In this transformation, a variant of the subroutine is created whose body contains the loop that had formerly called it. The loop within the subroutine may then be vectorizable.

`CMAX$PUSH`	applies to subsequent loop
`CMAX$PUSH L`	applies to subsequent loop
`CMAX$PUSH R`	applies to all subsequent loops in routine
`-Push`	applies to all loops in package

The negative form disables the "pushing" of loop(s).

`CMAX$NOPUSH`	applies to subsequent loop
`CMAX$NOPUSH L`	applies to subsequent loop
`CMAX$NOPUSH R`	applies to all subsequent loops in routine
`-noPush`	applies to all loops in package

The default global setting is `-noPush`.

Loop-Carried Data Dependence

The positive form asserts that data dependence may exist in the loop(s). The converter analyzes such loops to determine whether dependence does exist and whether it inhibits vectorization.

CMAX$DEPENDENCE	applies to subsequent loop
CMAX$DEPENDENCE L	applies to subsequent loop
CMAX$DEPENDENCE R	applies to all subsequent loops in routine
-Dependence	applies to all loops in package

The negative form asserts that there is no data dependence in the loop(s). The converter does not perform dependence analysis on such loops.

CMAX$NODEPENDENCE	applies to subsequent loop
CMAX$NODEPENDENCE L	applies to subsequent loop
CMAX$NODEPENDENCE R	applies to all subsequent loops in routine
-noDependence	applies to all loops in package

The default global setting is **-Dependence**

Uniqueness of Values in an Index Array

The positive form asserts that array elements used as indices into another array on the left hand side of an assignment in the specified loop(s) are each unique. Repetition of an index value indicates multiple assignments of an array element and thus a particular type of loop-carried data dependence.

CMAX$PERMUTATION	applies to subsequent loop
CMAX$PERMUTATION L	applies to subsequent loop
CMAX$PERMUTATION R	applies to all subsequent loops in routine
-Permutation	applies to all loops in package

The negative form asserts that array index values may not be unique.

CMAX$NOPERMUTATION	applies to subsequent loop
CMAX$NOPERMUTATION L	applies to subsequent loop
CMAX$NOPERMUTATION R	applies to all subsequent loops in routine
-noPermutation	applies to all loops in package

The default global setting is **-noPermutation**

A.3 The CMAX Library

The CMAX library provides canonical ways to express actions that have no straightforward expression in Fortran 77. These procedures are fully portable to any Fortran 77 platform. The CMAX Converter recognizes them and translates them into the corresponding CM Fortran feature.

The library presently provides utilities for expressing the intent of dynamic array allocation and circular shifting of array elements.

A.3.1 Using the CMAX Library

Header File

Place the library header file in the Fortran 77 program with an **INCLUDE** line:

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

Linking Fortran 77 Programs

To compile and link a Fortran 77 program, the serial link library must be available on the system and specified on the link line as:

```
% f77 myprogram.f libdirectory/libcmax.a
```

A version of this library compiled for a Sun-4 computer is available in the standard CM *libdirectory:*

CM-5:	**/usr/lib/libcmax.a**
CM-2/200:	**/usr/local/lib/libcmax.a**

To compile the serial library for other systems, obtain the library source code from the CMAX on-line examples directory:

CM-5:	**/usr/examples/cmax/libcmax/libcmax.f** **libcmax.c**
CM-2/200:	**/usr/cm/examples/cmax/libcmax/libcmax.f** **libcmax.c**

See your site system adminstrator for locations if any of these files have been moved from their default locations.

Linking CM Fortran Programs

You must link CMAX-generated **.fcm** files with a CMAX library if they contain:

- Converter output from the dynamic allocation utility, since CMAX generates its own dynamic allocation library routines rather than CM Fortran features
- Calls to the CM Fortran Utility Library procedure **CMF_SEND_***combiner*, since CMAX uses its own library routine **CMAX_GRID_TO_SEND_ADDR** to construct send addresses

NOTE: No explicit linking is needed for converted programs that use *only* the circular shift utility from the CMAX library.

Where explicit linking is needed, specify on the **cmf** or linker command line the appropriate link library for the target CM platform and execution model:

```
% cmf myprogram.fcm library
```

where *library* is one of:

libdirectory/**libcmax_cm2.a**
libdirectory/**libcmax_cm200.a**
libdirectory/**libcmax_cm5_sparc.a**
libdirectory/**libcmax_cm5_vu.a**
libdirectory/**libcmax_cm5_cmsim.a**

and *libdirectory* is either:

CM-5 systems:	**/usr/lib/**
CM-2/200 systems:	**/usr/local/lib/**

See your site system adminstrator for locations if any of these files have been moved from their default locations.

A.3.2 The Dynamic Allocation Utility

This utility uses a common array **CMAX_MEMORY**. A call to the subroutine **CMAX_ALLOCATE_***rank* associates an index variable (a scalar or a front-end array element) with a pointer into that memory. Passing the argument **CMAX_MEMORY(INDEX_VAR)** to a subroutine or function makes the memory available as an array. The corresponding dummy argument in the procedure must match the array indicated by **CMAX_MEMORY(INDEX_VAR)** in shape and type.

The converter transforms these subroutines into CM Fortran dynamic array allocation and deallocation. The dynamic arrays may be CM arrays or front-end arrays, depending on how they are treated in the subprogram scope.

```
CMAX_ALLOCATE_1(INDEX_VAR, ELT_TYPE, N1)
CMAX_ALLOCATE_2(INDEX_VAR, ELT_TYPE, N1, N2)
CMAX_ALLOCATE_3(INDEX_VAR, ELT_TYPE, N1, ..., N3)
CMAX_ALLOCATE_4(INDEX_VAR, ELT_TYPE, N1, ..., N4)
CMAX_ALLOCATE_5(INDEX_VAR, ELT_TYPE, N1, ..., N5)
CMAX_ALLOCATE_6(INDEX_VAR, ELT_TYPE, N1, ..., N6)
CMAX_ALLOCATE_7(INDEX_VAR, ELT_TYPE, N1, ..., N7)

CMAX_DEALLOCATE(INDEX_VAR)
```

where,

INDEX_VAR An integer variable *or* an element of an integer front-end array

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

NOTE: An index variable can be used for multiple arrays with different dimension extents, but they must all be of the same rank and layout. Also, all the arrays referenced by the elements of an array of index variables must have the same rank and layout.

Link the output program with the appropriate CMAX library for the target CM system and execution model, as shown in Section A.3.1. The converter translates the subroutines listed above into the following:

```
CMAX_I_ALLOCATE_1
CMAX_I_ALLOCATE_2
CMAX_I_ALLOCATE_3
CMAX_I_ALLOCATE_4
CMAX_I_ALLOCATE_5
CMAX_I_ALLOCATE_6
CMAX_I_ALLOCATE_7
CMAX_I_DEALLOCATE
```

These routines are internal versions of the CMAX allocation/deallocation routines. In addition to the user-supplied arguments, CMAX inserts a bitmask argument to the allocation routines, which specifies the layout of the dynamic array.

The user can opt to change these CMAX routines into CM Fortran dynamic allocation features. No special linking is then required.

A.3.3 The Circular Shift Utility

These subroutines shift the elements on dimension **DIM** of the source array by **SHIFT** element positions and store the result in the destination array. Elements that shift off the end of a dimension wrap around to the other end of that dimension.

```
CMAX_CSHIFT_1(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1)
CMAX_CSHIFT_2(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1, N2)
CMAX_CSHIFT_3(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1, ..., N3)
CMAX_CSHIFT_4(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1, ..., N4)
CMAX_CSHIFT_5(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1, ..., N5)
CMAX_CSHIFT_6(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1, ..., N6)
CMAX_CSHIFT_7(DEST, SOURCE, DIM, SHIFT, ELT_TYPE, N1, ..., N7)
```

where,

DEST	The destination array; must not overlap with the source array and must be of the same type and shape
SOURCE	The source array; must be of the specified type and shape
DIM	An integer between 1 and the rank of the array, 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

The converter transforms these subroutines into references to the CM Fortran intrinsic function **CSHIFT**. 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.

No special linking is required for a CMAX-generated program that performs circular shifts, since the CM Fortran intrinsic function is generated.

A.4 Gnu EMACS Utilities for CMAX (Unsupported)

EMACS editors can be customized in a number of ways to assist with the conversion process. CMAX provides a sample implementation for Gnu Emacs of the utilities described in this section:

- **cmax-viewer** command
- **cmax-subprogram** command
- **cmax-buffer** command

The implementation is in the file **cmax.el**. Customize this file to suit your needs and your local variant of EMACS. (It is not officially supported by Thinking Machines.)

The file is on-line in:

- CM-5 systems: **/usr/examples/cmax/cmax.el**
- CM-2/200 systems: **/usr/cm/examples/cmax/cmax.el**

See your system administrator for the location if this file has been moved.

A.4.1 The CMAX Viewer

The CMAX viewer is an EMACS mode lets you view and move through CMAX input and output files at the same time. Unlike Prism, which requires executable files, this utility loads ASCII source files. By allowing you to compare input and output source, along with efficiency notes, the utility helps you determine what CMAX has done and why.

To activate the CMAX viewer:

1. Translate a **.f** file using the **cmax** option **-LineMap** (the default).
2. Place the cursor within a buffer containing either the **.f** or **.fcm** file.
3. Invoke the EMACS command **M-x cmax-viewer**.

The utility finds the line-mapping (**.ttab**) file and both Fortran files associated with it. It displays the input and output file in two adjoining windows, allowing you to scroll or otherwise move around in both files in synchronized fashion. For programs translated with **-list**, invoke the EMACS command bound to **C-c v** to display efficiency notes for the variable or subprogram name at the cursor.

Type **C-h m** for mode help, and you will see the following message.

CMAX Viewer Mode:

This mode can be entered by calling **cmax-viewer** interactively.

Turn on **cmax-viewer-mode**, a mode for looking at **.f** files and cmax output simultaneously, based on a **.ttab** file.

Moving the cursor in either the input source or output source window will move the cursor in both windows.

If the cursor is in the input source window, any efficiency notes for the pointed to line are displayed at the bottom of the screen. You must run CMAX with **–list** to generate a **.ttab** with efficiency notes.

You can use **C-C v** (below) to display efficiency notes for variables or subprograms when pointing to them in either window.

These special commands operate in both source windows in synchronized fashion:

C-n	Move point to next line.
C-p	Move point to previous line.
C-v	Scroll source windows up.
M-v	Scroll source windows down.
M-<	Go to beginning of buffer.
M->	Go to end of buffer.
C-L	Recenter.
C-c v	Display efficiency notes for the variable or subprogram name at the point.
g	Go to line number of buffer.
q	Quit cmax–viewer mode.

NOTE: If you prefer side-by-side source windows to top-and-bottom source windows, add this line to the EMACS code:

```
(setq cmax-viewer-split-function
      'split-window-horizontally)
```

A.4.2 Command cmax–subprogram

The command **M-x cmax-subprogram** sends the subprogram surrounding the point to a CMAX process and displays the translated output in the buffer named CMAX.

If invoked with a prefix argument (**C-u**), the command prompts for additional switch arguments and passes them to the **cmax** command. Alternatively, set the variable **cmax-default-switches** (it is set to **-verb=4** by default).

A.4.3 Command cmax–buffer

The command **M-x cmax-buffer** sends the contents of the current buffer to a CMAX process and displays the translated output in the buffer named CMAX.

If invoked with a prefix argument (**C-u**), the command prompts for additional switch arguments and passes them to the **cmax** command. Alternatively, set the variable **cmax-default-switches** (it is set to **-verb=4** by default).

Appendix B

Idioms and Transformations

This appendix shows the CM Fortran output code that CMAX generates from particular Fortran 77 loop idioms and other constructions.

B.1 Fortran 90 and CM Fortran Language Constructs

For descriptions of these constructs, see the *CM Fortran Reference Manual.* These examples use one- and two-dimensional arrays; in general, multidimensional forms are recognized.

B.1.1 Array Syntax

This subroutine performs an elementwise addition of two arrays:

```
SUBROUTINE EXAMPLE_ARRAY_ADD(A, B, C, N)
REAL A(N,N), B(N,N), C(N,N)
DO J = 1,N
   DO I = 1,N
      C(I,J) = A(I,J) + B(I,J)
   END DO
END DO
RETURN
END
```

It translates to code which uses array syntax:

```
      SUBROUTINE EXAMPLE_ARRAY_ADD(A,B,C,N)
      REAL A(N,N), B(N,N), C(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT C(:NEWS,:NEWS)
      C = A + B
      RETURN
      END
```

This subroutine computes a more complicated function of several arrays, using scalars which need to be promoted:

```
      SUBROUTINE EXAMPLE_ARRAY_CRUNCH(X1, X2, Y1, Y2, D, N)
      REAL X1(N), X2(N), Y1(N), Y2(N), D(N)
      DO I = 1,N
         D1 = X1(I)-X2(I)
         D2 = Y1(I)-Y2(I)
         D(I) = SQRT(D1*D1 + D2*D2)
      END DO
      RETURN
      END
```

It translates to code which uses array syntax and promoted scalar arrays:

```
      SUBROUTINE EXAMPLE_ARRAY_CRUNCH(X1,X2,Y1,Y2,D,N)
      REAL X1(N), X2(N), Y1(N), Y2(N), D(N)
CMF$  LAYOUT D(:NEWS)
CMF$  LAYOUT X1(:NEWS)
CMF$  LAYOUT X2(:NEWS)
CMF$  LAYOUT Y1(:NEWS)
CMF$  LAYOUT Y2(:NEWS)
      REAL  D2000(N)
CMF$  LAYOUT D2000(:NEWS)
      REAL  D1000(N)
CMF$  LAYOUT D1000(:NEWS)
      D1000 = X1 - X2
      D2000 = Y1 - Y2
      D = SQRT(D1000 * D1000 + D2000 * D2000)
      RETURN
      END
```

This subroutine performs a smoothing operation using nearest–neighbor values:

```
      SUBROUTINE EXAMPLE_ARRAY_SMOOTH(IN, OUT, N)
      REAL IN(N,N), OUT(N,N)
      DO J = 2,N-1
         DO I = 2,N-1
            OUT(I,J) = 0.25*(IN(I-1,J-1) + IN(I-1,J+1) +
     .          IN(I+1,J-1) + IN(I+1,J+1))
         END DO
      END DO
      RETURN
      END
```

It translates to code which uses array sections:

```
      SUBROUTINE EXAMPLE_ARRAY_SMOOTH(IN,OUT,N)
      REAL IN(N,N), OUT(N,N)
CMF$  LAYOUT IN(:NEWS,:NEWS)
CMF$  LAYOUT OUT(:NEWS,:NEWS)
      OUT(2:N - 1,2:N - 1) =
     &    0.25 * (IN(:N - 2,:N - 2) + IN(:N - 2,3:)
     &    + IN(3:,:N - 2) + IN(3:,3:))
      RETURN
      END
```

B.1.2 WHERE

This subroutine performs a simple conditional elementwise operation:

```
      SUBROUTINE EXAMPLE_WHERE_1(TEMP, N, DTEMP)
      REAL TEMP(N)
      DO I=1,N
         IF (TEMP(I) .GT. 98.6) TEMP(I) = TEMP(I) - DTEMP
      END DO
      RETURN
      END
```

It translates to code which uses a **WHERE** statement:

```
      SUBROUTINE EXAMPLE_WHERE_1(TEMP,N,DTEMP)
      REAL TEMP(N)
CMF$  LAYOUT TEMP(:NEWS)
      WHERE (TEMP .GT. 98.6) TEMP = TEMP - DTEMP
      RETURN
      END
```

This subroutine performs a slightly more complex conditional elementwise operation:

```
      SUBROUTINE EXAMPLE_WHERE_2
     &       (PRESSURE,TEMP,RAINING,N,DPRESSURE)
      REAL PRESSURE(N), TEMP(N)
      LOGICAL RAINING(N)
      DO I=1,N
         IF (PRESSURE(I) .LT. 1.0) THEN
            TEMP(I) = TEMP(I) - 5.0
         ELSE
            RAINING(I) = .TRUE.
         END IF
      END DO
      RETURN
      END
```

It translates to code which uses a **WHERE** construct:

```
      SUBROUTINE EXAMPLE_WHERE_2
     &       (PRESSURE,TEMP,RAINING,N,DPRESSURE)
      REAL PRESSURE(N), TEMP(N)
      LOGICAL RAINING(N)
CMF$  LAYOUT PRESSURE(:NEWS)
CMF$  LAYOUT RAINING(:NEWS)
CMF$  LAYOUT TEMP(:NEWS)
      WHERE (PRESSURE .LT. 1.0)
      TEMP = TEMP - 5.0
      ELSEWHERE
      RAINING = .TRUE.
      ENDWHERE
      RETURN
      END
```

B.1.3 FORALL

This subroutine computes APL's **IOTA** function:

```
      SUBROUTINE EXAMPLE_FORALL_1(A, N)
      INTEGER A(N)
      DO I = 1,N
         A(I) = I
      END DO
      RETURN
      END
```

It translates to code which uses a **FORALL**:

```
      SUBROUTINE EXAMPLE_FORALL_1(A,N)
      INTEGER A(N)
CMF$  LAYOUT A(:NEWS)
      FORALL (I = 1:N) A(I) = I
      RETURN
      END
```

This subroutine computes a two-dimensional mask which is true for the "red" squares of a checkerboard:

```
      SUBROUTINE EXAMPLE_FORALL_2(RED, N)
      LOGICAL RED(N,N)
      DO J = 1,N
         DO I = 1,N
            RED(I,J) = (MOD(I+J,2) .EQ. 0)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_FORALL_2(RED,N)
      LOGICAL RED(N,N)
CMF$  LAYOUT RED(:NEWS,:NEWS)
      FORALL (J = 1:N, I = 1:N)
     &       RED(I,J) = (MOD(I + J,2) .EQ. 0)
      RETURN
      END
```

This subroutine multiplies a two-dimensional array with a one-dimensional array spread across its columns:

```
      SUBROUTINE EXAMPLE_FORALL_3(V, A, N)
      REAL V(N), A(N,N)
      DO J = 1,N
         DO I = 1,N
            A(I,J) = A(I,J) * V(I)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_FORALL_3(V,A,N)
      REAL V(N), A(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT V(:NEWS)
      FORALL (J = 1:N, I = 1:N) A(I,J) = A(I,J) * V(I)
      RETURN
      END
```

This subroutine adds an array with the transpose of another array:

```
      SUBROUTINE EXAMPLE_FORALL_4(A, B, N)
      REAL A(N,N), B(N,N)
      DO J = 1,N
         DO I = 1,N
            A(I,J) = A(I,J) + B(J,I)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_FORALL_4(A,B,N)
      REAL A(N,N), B(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
      FORALL (J = 1:N, I = 1:N) A(I,J) = A(I,J) + B(J,I)
      RETURN
      END
```

This subroutine zeros the upper triangle of an array:

```
      SUBROUTINE EXAMPLE_FORALL_5(A, N)
      REAL A(N,N)
      DO J = 1,N
         DO I = 1,J-1
            A(I,J) = 0.0
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_FORALL_5(A,N)
      REAL A(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
      FORALL (J = 1:N, I = 1:N, I .LT. J) A(I,J) = 0.0
      RETURN
      END
```

This subroutine does complicated indirect addressing:

```
      SUBROUTINE EXAMPLE_FORALL_6(A, B, I1, I2, N)
      REAL A(N,N), B(N,N)
      INTEGER I1(N,N), I2(N)
      DO J = 1,N
         DO I = 1,N
            A(I,J) = A(I,J) + B(I1(I,J),I2(J))
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_FORALL_6(A,B,I1,I2,N)
      REAL A(N,N), B(N,N)
      INTEGER I1(N,N), I2(N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT I1(:NEWS,:NEWS)
CMF$  LAYOUT I2(:NEWS)
      FORALL (J = 1:N, I = 1:N)
     &   A(I,J) = A(I,J) + B(I1(I,J),I2(J))
      RETURN
      END
```

B.2 Fortran 90 and CM Fortran Intrinsic Funtions

For descriptions of these intrinsics, see the *CM Fortran Reference Manual.* These examples use one- and two-dimensional arrays; in general, multidimensional forms are recognized.

B.2.1 ALL

This subroutine computes the logical AND of all the elements in an array:

```
SUBROUTINE EXAMPLE_ALL_1(L, FLAG, N)
LOGICAL L(N,N), FLAG
DO J=1,N
   DO I=1,N
      FLAG = FLAG .AND. L(I,J)
   END DO
END DO
RETURN
END
```

It translates to:

```
      SUBROUTINE EXAMPLE_ALL_2(L,FLAG,N)
      LOGICAL L(N,N), FLAG
CMF$  LAYOUT L(:NEWS,:NEWS)
      FLAG = FLAG .AND. ALL(L)
      RETURN
      END
```

This subroutine computes **ALL** over the second dimension of an array:

```
SUBROUTINE EXAMPLE_ALL_2(L, FLAGS, N)
LOGICAL L(N,N), FLAGS(N)
DO J=1,N
   DO I = 1,N
      FLAGS(I) = FLAGS(I) .AND. L(I,J)
   END DO
END DO
```

```
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_ALL_2(L,FLAGS,N)
      LOGICAL L(N,N), FLAGS(N)
CMF$  LAYOUT FLAGS(:NEWS)
CMF$  LAYOUT L(:NEWS,:NEWS)
      FLAGS = FLAGS .AND. ALL(L,DIM=2)
      RETURN
      END
```

B.2.2 ANY

This subroutine computes the logical OR of all the elements in an array:

```
      SUBROUTINE EXAMPLE_ANY_1(L, FLAG, N)
      LOGICAL L(N,N), FLAG
      DO J=1,N
         DO I=1,N
            FLAG = FLAG .OR. L(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_ANY_1(L,FLAG,N)
      LOGICAL L(N,N), FLAG
CMF$  LAYOUT L(:NEWS,:NEWS)
      FLAG = FLAG .OR. ANY(L)
      RETURN
      END
```

This subroutine computes **ANY** over the first dimension of an array:

```
      SUBROUTINE EXAMPLE_ANY_2(L, FLAGS, N)
      LOGICAL L(N,N), FLAGS(N)
      DO J=1,N
         DO I = 1,N
            FLAGS(J) = FLAGS(J) .OR. L(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_ANY_2(L,FLAGS,N)
      LOGICAL L(N,N), FLAGS(N)
CMF$  LAYOUT FLAGS(:NEWS)
CMF$  LAYOUT L(:NEWS,:NEWS)
      FLAGS = FLAGS .OR. ANY(L,DIM=1)
      RETURN
      END
```

B.2.3 COUNT

This subroutine counts the number of zero elements in an array:

```
      SUBROUTINE EXAMPLE_COUNT_1(A, K, N)
      REAL A(N,N)
      DO J = 1,N
         DO I = 1,N
            IF (A(I,J) .EQ. 0.0) K = K + 1
         END DO
      END DO
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_COUNT_1(A,K,N)
      REAL A(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
      K = K + COUNT(A .EQ. 0.0)
      END
```

This subroutine computes **COUNT** over the second dimension of an array:

```
      SUBROUTINE EXAMPLE_COUNT_2(A, K, N)
      REAL A(N,N)
      INTEGER K(N)
      DO J=1,N
         DO I = 1,N
            IF (A(I,J) .EQ. 0.0) K(I) = K(I) + 1
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_COUNT_2(A,K,N)
      REAL A(N,N)
      INTEGER K(N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT K(:NEWS)
      K = K + COUNT(A .EQ. 0.0,DIM=2)
      RETURN
      END
```

B.2.4 DOTPRODUCT

This subroutine computes **DOTPRODUCT**:

```
      SUBROUTINE EXAMPLE_DOTPRODUCT(X, Y, N, A)
      REAL X(N), Y(N), A
      DO I = 1,N
         A = A + X(I) * Y(I)
      END DO
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_DOTPRODUCT(X,Y,N,A)
      REAL X(N), Y(N), A
CMF$  LAYOUT X(:NEWS)
CMF$  LAYOUT Y(:NEWS)
      A = A + DOTPRODUCT(X,Y)
      END
```

B.2.5 MATMUL

These subroutines perform matrix multiplies:

```
SUBROUTINE EXAMPLE_MATMUL_1(A, B, C, N1, N2, N3)
REAL A(N1, N2), B(N2, N3), C(N1, N3)
DO J=1,N3
   DO I=1,N1
      C(I,J)=0.0
   END DO
END DO
DO K=1,N2
   DO J=1,N3
      DO I=1,N1
         C(I,J)=C(I,J)+A(I,K)*B(K,J)
      END DO
   END DO
END DO
RETURN
END

SUBROUTINE EXAMPLE_MATMUL_2(A, B, C, N1, N2, N3)
REAL A(N1, N2), B(N2, N3), C(N1, N3)
DO I=1,N1
   DO J=1,N3
      C(I,J)=0.0
      DO K=1,N2
         C(I,J)=C(I,J)+A(I,K)*B(K,J)
      END DO
   END DO
END DO
RETURN
END
```

They translate to:

```
      SUBROUTINE EXAMPLE_MATMUL_1(A,B,C,N1,N2,N3)
      REAL A(N1, N2), B(N2, N3), C(N1, N3)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT C(:NEWS,:NEWS)
      C = 0.0
      C = C + MATMUL(A,B)
      RETURN
      END

      SUBROUTINE EXAMPLE_MATMUL_2(A,B,C,N1,N2,N3)
      REAL A(N1, N2), B(N2, N3), C(N1, N3)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT C(:NEWS,:NEWS)
      C = MATMUL(A,B)
      RETURN
      END
```

B.2.6 MAXLOC

This subroutine finds the maximum element of an array and its location:

```
      SUBROUTINE MAXLOC_EXAMPLE_1(A, AMI, AMJ, AMAX, N1, N2)
      REAL A(N1,N2), AMAX
      INTEGER AMI, AMJ
      DO J=1,N2
         DO I=1,N1
            IF (A(I,J) .GT. AMAX) THEN
               AMAX = A(I,J)
               AMI = I
               AMJ = J
            END IF
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE MAXLOC_EXAMPLE_1(A,AMI,AMJ,AMAX,N1,N2)
      REAL A(N1,N2), AMAX
      INTEGER AMI, AMJ
CMF$  LAYOUT A(:NEWS,:NEWS)
      INTEGER IJ(2)
CMF$  LAYOUT IJ(:NEWS)
      IJ = MAXLOC(A)
      IF (IJ(1) .GT. 0 .AND. IJ(2) .GT. 0
     &    .AND. A(IJ(1),IJ(2)) .GT. AMAX) THEN
         AMAX = A(IJ(1),IJ(2))
         AMI = IJ(1)
         AMJ = IJ(2)
      ENDIF
      RETURN
      END
```

This subroutine finds the maximum element of an array and its location subject to a mask:

```
      SUBROUTINE MAXLOC_EXAMPLE_2(A,MASK,AMI,AMJ,AMAX,N1,N2)
      REAL A(N1,N2), AMAX
      LOGICAL MASK(N1,N2)
      INTEGER AMI, AMJ
      DO J=1,N2
         DO I=1,N1
            IF (MASK(I,J).AND.(A(I,J).GT.AMAX)) THEN
               AMAX = A(I,J)
               AMI = I
               AMJ = J
            END IF
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE MAXLOC_EXAMPLE_2(A,MASK,AMI,AMJ,AMAX,N1,N2)
      REAL A(N1,N2), AMAX
      LOGICAL MASK(N1,N2)
      INTEGER AMI, AMJ
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT MASK(:NEWS,:NEWS)
      INTEGER IJ(2)
CMF$  LAYOUT IJ(:NEWS)
      IJ = MAXLOC(A,MASK=MASK)
      IF (IJ(1) .GT. 0 .AND. IJ(2) .GT. 0
     &    .AND. A(IJ(1),IJ(2)) .GT. AMAX) THEN
         AMAX = A(IJ(1),IJ(2))
         AMI = IJ(1)
         AMJ = IJ(2)
      ENDIF
      RETURN
      END
```

B.2.7 MAXVAL

This subroutine finds the maximum value of an array:

```
      SUBROUTINE EXAMPLE_MAXVAL_1(A, X, N)
      REAL A(N,N), X
      DO J=1,N
         DO I=1,N
            X = MAX(X,A(I,J))
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_MAXVAL_1(A,X,N)
      REAL A(N,N), X
CMF$  LAYOUT A(:NEWS,:NEWS)
      X = MAX(X,MAXVAL(A))
      RETURN
      END
```

This subroutine finds the maximum values along the first dimension of an array:

```
      SUBROUTINE EXAMPLE_MAXVAL_2(A, X, N)
      REAL A(N,N), X(N)
      DO J=1,N
         DO I = 1,N
            X(J) = MAX(X(J),A(I,J))
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_MAXVAL_2(A,X,N)
      REAL A(N,N), X(N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT X(:NEWS)
      X = MAX(X,MAXVAL(A,DIM=1))
      RETURN
      END
```

B.2.8 MINLOC

This subroutine finds the minimum element of an array and its location:

```
      SUBROUTINE MINLOC_EXAMPLE_1(A, AMI, AMJ, AMIN, N1, N2)
      REAL A(N1,N2), AMIN
      INTEGER AMI, AMJ
      DO J=1,N2
         DO I=1,N1
            IF (A(I,J) .LT. AMIN) THEN
               AMIN = A(I,J)
               AMI = I
               AMJ = J
            END IF
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE MINLOC_EXAMPLE_1(A,AMI,AMJ,AMIN,N1,N2)
      REAL A(N1,N2), AMIN
      INTEGER AMI, AMJ
CMF$  LAYOUT A(:NEWS,:NEWS)
      INTEGER IJ(2)
CMF$  LAYOUT IJ(:NEWS)
      IJ = MINLOC(A)
      IF (IJ(1) .GT. 0 .AND. IJ(2) .GT. 0
     &    .AND. A(IJ(1),IJ(2)) .LT. AMIN) THEN
         AMIN = A(IJ(1),IJ(2))
         AMI = IJ(1)
         AMJ = IJ(2)
      ENDIF
      RETURN
      END
```

This subroutine finds the minimum element of an array and its location subject to a mask:

```
      SUBROUTINE MINLOC_EXAMPLE_2(A,MASK,AMI,AMJ,AMIN,N1,N2)
      REAL A(N1,N2), AMIN
      LOGICAL MASK(N1,N2)
      INTEGER AMI, AMJ
      DO J=1,N2
         DO I=1,N1
            IF (MASK(I,J).AND.(A(I,J).LT.AMIN)) THEN
               AMIN = A(I,J)
               AMI = I
               AMJ = J
            END IF
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE MINLOC_EXAMPLE_2(A,MASK,AMI,AMJ,AMIN,N1,N2)
      REAL A(N1,N2), AMIN
      LOGICAL MASK(N1,N2)
      INTEGER AMI, AMJ
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT MASK(:NEWS,:NEWS)
      INTEGER IJ(2)
CMF$  LAYOUT IJ(:NEWS)
      IJ = MINLOC(A,MASK=MASK)
      IF (IJ(1) .GT. 0 .AND. IJ(2) .GT. 0
     &    .AND. A(IJ(1),IJ(2)) .LT. AMIN) THEN
         AMIN = A(IJ(1),IJ(2))
         AMI = IJ(1)
         AMJ = IJ(2)
      ENDIF
      RETURN
      END
```

B.2.9 MINVAL

This subroutine finds the minimum value of an array:

```
      SUBROUTINE EXAMPLE_MINVAL_1(A, X, N)
      REAL A(N,N), X
      DO J=1,N
         DO I=1,N
            X = MIN(X,A(I,J))
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_MINVAL_1(A,X,N)
      REAL A(N,N), X
CMF$  LAYOUT A(:NEWS,:NEWS)
      X = MIN(X,MINVAL(A))
      RETURN
      END
```

This subroutine finds the minimum values along the second dimension of an array:

```
      SUBROUTINE EXAMPLE_MINVAL_2(A, X, N)
      REAL A(N,N), X(N)
      DO J=1,N
         DO I = 1,N
            X(I) = MIN(X(I),A(I,J))
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_MINVAL_2(A,X,N)
      REAL A(N,N), X(N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT X(:NEWS)
      X = MIN(X,MINVAL(A,DIM=2))
      RETURN
      END
```

B.2.10 PRODUCT

This subroutine computes the product of all the elements in an array:

```
      SUBROUTINE EXAMPLE_PRODUCT_1(A, X, N)
      REAL A(N,N), X
      DO J=1,N
         DO I=1,N
            X = X * A(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_PRODUCT_1(A,X,N)
      REAL A(N,N), X
CMF$  LAYOUT A(:NEWS,:NEWS)
      X = X * PRODUCT(A)
      RETURN
      END
```

This subroutine computes the product of elements along the first dimension of an array:

```
      SUBROUTINE EXAMPLE_PRODUCT_2(A, X, N)
      REAL A(N,N), X(N)
      DO J=1,N
         DO I = 1,N
            X(J) = X(J) * A(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_PRODUCT_2(A,X,N)
      REAL A(N,N), X(N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT X(:NEWS)
      X = X * PRODUCT(A,DIM=1)
      RETURN
      END
```

This subroutine computes the product of all the positive elements of an array:

```
      SUBROUTINE EXAMPLE_PRODUCT_3(A, X, N)
      REAL A(N,N), X
      DO J=1,N
         DO I=1,N
            IF (A(I,J) .GT. 0.0) X = X * A(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_PRODUCT_3(A,X,N)
      REAL A(N,N), X
CMF$  LAYOUT A(:NEWS,:NEWS)
      X = X * PRODUCT(A,MASK=A .GT. 0.0)
      RETURN
      END
```

B.2.11 SUM

This subroutine computes the sum of all the elements in an array:

```
      SUBROUTINE EXAMPLE_SUM_1(A, X, N)
      REAL A(N,N), X
      DO J=1,N
         DO I=1,N
            X = X + A(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_SUM_1(A,X,N)
      REAL A(N,N), X
CMF$  LAYOUT A(:NEWS,:NEWS)
      X = X + SUM(A)
      RETURN
      END
```

This subroutine computes the sum of elements along the second dimension of an array:

```
      SUBROUTINE EXAMPLE_SUM_2(A, X, N)
      REAL A(N,N), X(N)
      DO J=1,N
         DO I = 1,N
            X(I) = X(I) + A(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_SUM_2(A,X,N)
      REAL A(N,N), X(N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT X(:NEWS)
      X = X + SUM(A,DIM=2)
      RETURN
      END
```

This subroutine computes the sum of all the elements of an array under a mask:

```
      SUBROUTINE EXAMPLE_SUM_3(A, MASK, X, N)
      REAL A(N,N), X
      LOGICAL MASK(N,N)
      DO J=1,N
         DO I=1,N
            IF (MASK(I,J)) X = X + A(I,J)
         END DO
      END DO
      RETURN
      END
```

It translates to:

```
      SUBROUTINE EXAMPLE_SUM_3(A,MASK,X,N)
      REAL A(N,N), X
      LOGICAL MASK(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT MASK(:NEWS,:NEWS)
      X = X + SUM(A,MASK=MASK)
      RETURN
      END
```

B.3 CM Fortran Utility Library Subroutines

For descriptions of these subroutines, see the *CM Fortran Utility Library Reference Manual*. These examples use one- and two-dimensional arrays; in general, multidimensional forms are recognized.

B.3.1 CMF_SCAN_

This upward, inclusive add scan:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_1(A,B,N)
      REAL A(N), B(N)
      LAST = 0
      DO I=1,N
         LAST = LAST + A(I)
         B(I) = LAST
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_1(A,B,N)
      REAL A(N), B(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT B(:NEWS)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      LAST = 0
      CALL CMF_SCAN_ADD(B,A,CMF_NULL,1,CMF_UPWARD,
     &       CMF_INCLUSIVE,CMF_NONE,.TRUE.)
      RETURN
      END
```

This downward, inclusive add scan:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_2(A,B,N)
      REAL A(N), B(N)
      LAST = 0
      DO I=N,1,-1
         LAST = LAST + A(I)
         B(I) = LAST
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_2(A,B,N)
      REAL A(N), B(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT B(:NEWS)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      LAST = 0
      CALL CMF_SCAN_ADD(B,A,CMF_NULL,1,CMF_DOWNWARD,
     &       CMF_INCLUSIVE,CMF_NONE,.TRUE.)
      RETURN
      END
```

This upward, exclusive add scan:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_3(A,B,N)
      REAL A(N), B(N)
      LAST = 0
      DO I=1,N
         B(I) = LAST
         LAST = LAST + A(I)
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_3(A,B,N)
      REAL A(N), B(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT B(:NEWS)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      LAST = 0
      CALL CMF_SCAN_ADD(B,A,CMF_NULL,1,CMF_UPWARD,
     &       CMF_EXCLUSIVE,CMF_NONE,.TRUE.)
      RETURN
      END
```

This upward, inclusive, segmented add scan:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_4(A,B,SEGMENT,N)
      REAL A(N), B(N)
      LOGICAL SEGMENT(N)
      LAST = 0
      DO I=1,N
         IF (SEGMENT(I)) LAST = 0
         LAST = LAST + A(I)
         B(I) = LAST
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_4(A,B,SEGMENT,N)
      REAL A(N), B(N)
      LOGICAL SEGMENT(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT B(:NEWS)
CMF$  LAYOUT SEGMENT(:NEWS)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      LAST = 0
      CALL CMF_SCAN_ADD(B,A,SEGMENT,1,CMF_UPWARD,
     &       CMF_INCLUSIVE,CMF_SEGMENT_BIT,.TRUE.)
      RETURN
      END
```

This masked, upward, inclusive, segmented add scan:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_5(A,B,SEGMENT,MASK,N)
      REAL A(N), B(N)
      LOGICAL SEGMENT(N), MASK(N)
      LAST = 0
      DO I=1,N
         IF (SEGMENT(I)) LAST = 0
         IF (MASK(I)) THEN
            LAST = LAST + A(I)
            B(I) = LAST
         END IF
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SCAN_ADD_5(A,B,SEGMENT,MASK,N)
      REAL A(N), B(N)
      LOGICAL SEGMENT(N), MASK(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT B(:NEWS)
CMF$  LAYOUT MASK(:NEWS)
CMF$  LAYOUT SEGMENT(:NEWS)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      LAST = 0
      CALL CMF_SCAN_ADD(B,A,SEGMENT,1,CMF_UPWARD,
     &       CMF_INCLUSIVE,CMF_SEGMENT_BIT,MASK)
      RETURN
      END
```

This upward, inclusive, segmented copy scan:

```
      SUBROUTINE EXAMPLE_SCAN_COPY_1(A,B,SEGMENT,N)
      REAL A(N), B(N)
      LOGICAL SEGMENT(N)
      LAST = A(1)
      DO I=1,N
         IF (SEGMENT(I)) LAST = A(I)
         B(I) = LAST
      END DO
```

```
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SCAN_COPY_1(A,B,SEGMENT,N)
      REAL A(N), B(N)
      LOGICAL SEGMENT(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT B(:NEWS)
CMF$  LAYOUT SEGMENT(:NEWS)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      LAST = A(1)
      CALL CMF_SCAN_COPY(B,A,SEGMENT,1,CMF_UPWARD,
     &       CMF_INCLUSIVE,CMF_SEGMENT_BIT,.TRUE.)
      RETURN
      END
```

B.3.2 CMF_SEND_

This subroutine, which is in essence a combining send:

```
      SUBROUTINE EXAMPLE_SEND_ADD_1(A, B, V1, V2, N1, N2)
      REAL A(N1,N2), B(N1,N2)
      INTEGER V1(N1,N2), V2(N1,N2)
      DO I=1,N1
         DO J=1,N2
            IF (B(I,J).GT.0.0) THEN
               A(V1(I,J),V2(I,J)) = A(V1(I,J),V2(I,J)) + B(I,J)
            END IF
         END DO
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_SEND_ADD_1(A,B,V1,V2,N1,N2)
      REAL A(N1,N2), B(N1,N2)
      INTEGER V1(N1,N2), V2(N1,N2)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT V1(:NEWS,:NEWS)
CMF$  LAYOUT V2(:NEWS,:NEWS)
      INTEGER GRIDTEMP(N1,N2)
CMF$  LAYOUT GRIDTEMP(:NEWS,:NEWS)
      CALL CMAX_GRID_TO_SEND_ADDR(GRIDTEMP,A,V1,V2)
      CALL CMF_SEND_ADD(A,GRIDTEMP,B,B .GT. 0.0)
      RETURN
      END
```

NOTE: This output must be linked with the CMAX library, since it constructs send addresses using **CMAX_GRID_TO_SEND_ADDR** (a CMAX library routine) rather than the CM Fortran Utility Library routines.

B.3.3 CMF_FE_ARRAY_TO_CM and CMF_FE_ARRAY_FROM_CM

This subroutine, which transfers one array from the serial control processor to the parallel processors and one array in the other direction:

```
      SUBROUTINE EXAMPLE_TRANSFERS(CM_A, FE_A, CM_B, FE_B, N)
      REAL CM_A(N,N,N), FE_A(N,N,N)
      REAL CM_B(N,N,N), FE_B(N,N,N)
CMF$  LAYOUT CM_A(:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT FE_A(:SERIAL,:SERIAL,:SERIAL)
CMF$  LAYOUT CM_B(:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT FE_B(:SERIAL,:SERIAL,:SERIAL)
      DO K = 1,N
         DO J = 1,N
            DO I = 1,N
               CM_A(I,J,K) = FE_A(I,J,K)
               FE_B(I,J,K) = CM_B(I,J,K)
            END DO
         END DO
      END DO
      RETURN
      END
```

translates to:

```
      SUBROUTINE EXAMPLE_TRANSFERS(CM_A,FE_A,CM_B,FE_B,N)
      REAL CM_A(N,N,N), FE_A(N,N,N)
      REAL CM_B(N,N,N), FE_B(N,N,N)
CMF$  LAYOUT CM_A(:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT FE_A(:SERIAL,:SERIAL,:SERIAL)
CMF$  LAYOUT CM_B(:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT FE_B(:SERIAL,:SERIAL,:SERIAL)
      CALL CMF_FE_ARRAY_TO_CM(CM_A,FE_A)
      CALL CMF_FE_ARRAY_FROM_CM(FE_B,CM_B)
      RETURN
      END
```

Index

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z