## 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?

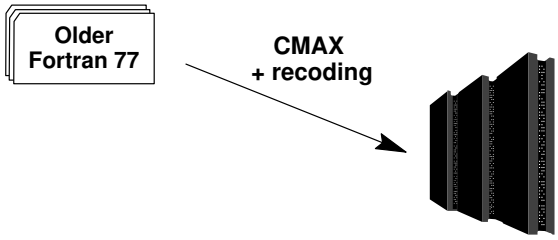
The CMAX Converter has two basic goals:

- 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.

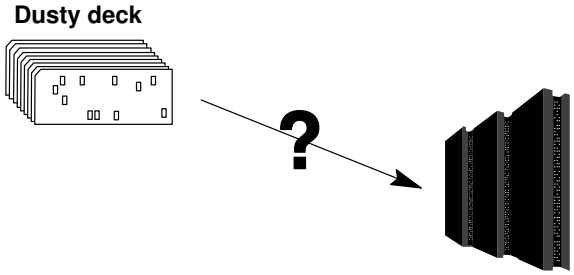
*Copyright © 1994 Thinking Machines Corporation*

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.


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

## 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.


Copyright © 1994 Thinking Machines Corporation

```
      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.**


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

```
      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.**


*Copyright © 1994 Thinking Machines Corporation*

### 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))
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

### 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.


Copyright © 1994 Thinking Machines Corporation

```
      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.**


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

## 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:

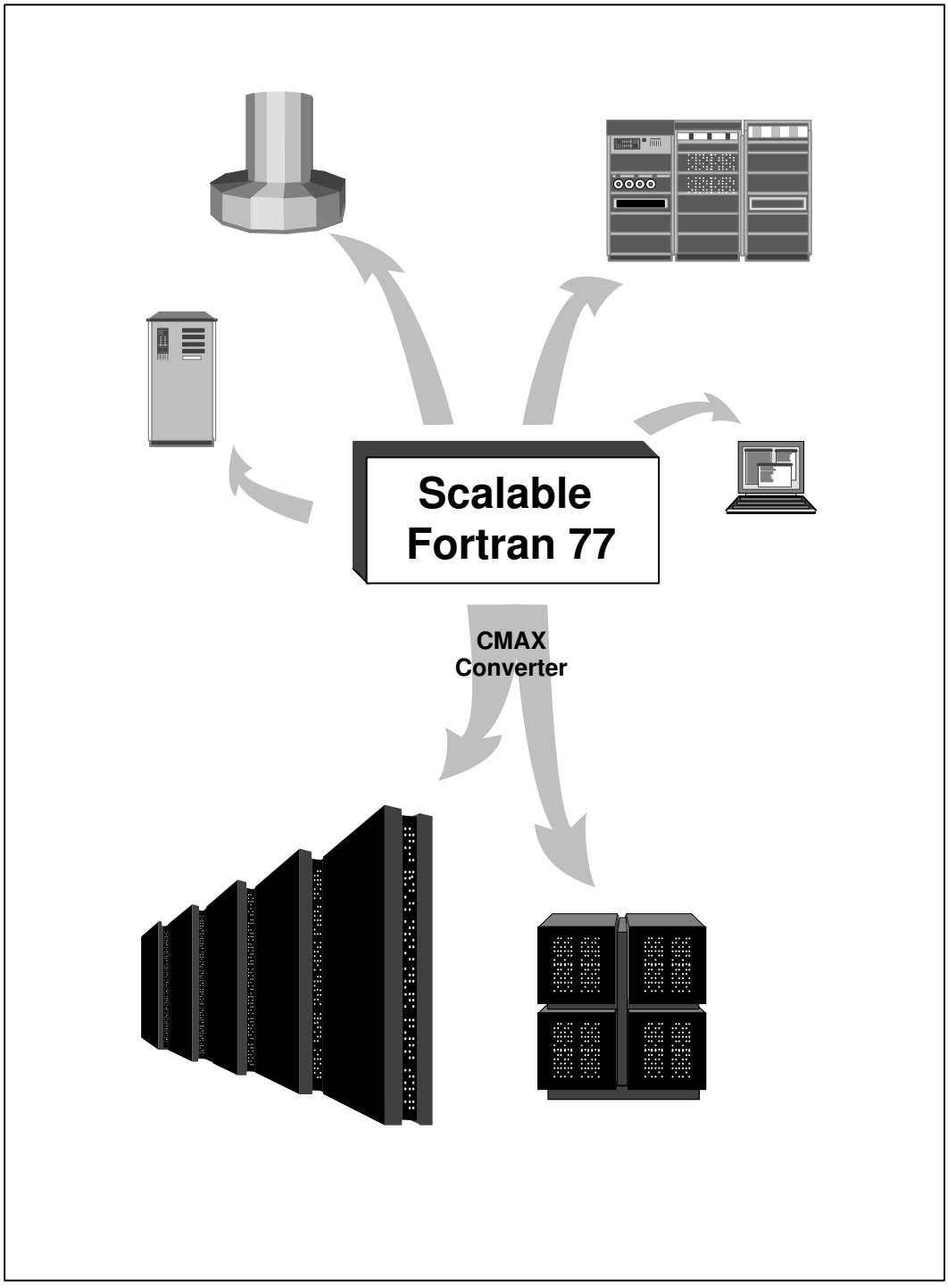
- Make it easy for a compiler to recognize how data and computations may be split up for independent or coordinated processing.
- Avoid constructions that rely on a particular memory organization.
- Use data layout directives and library procedures (with some conditionalizing convention) to take advantage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


Copyright © 1994 Thinking Machines Corporation





*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

### 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.


Copyright © 1994 Thinking Machines Corporation

**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
```



*Copyright © 1994 Thinking Machines Corporation*

### 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:


Copyright © 1994 Thinking Machines Corporation

```
      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
```



*Copyright © 1994 Thinking Machines Corporation*

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.)


Copyright © 1994 Thinking Machines Corporation

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**.


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

### 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.


Copyright © 1994 Thinking Machines Corporation

### 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 conventions 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 sys-



*Copyright © 1994 Thinking Machines Corporation*

tem, 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.


Copyright © 1994 Thinking Machines Corporation

## 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.


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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).

## Loop Conversion

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
```

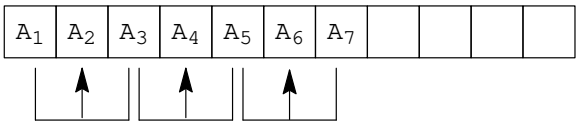
In other loop operations, the data elements may interact but there is no dependence between loop iterations. The converter's dependence analysis detects that a step-by-2 smoothing operation can vectorize to an array assignment. (Notice that the converter and CM Fortran accept the **END DO** statement.)

**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
```





Copyright © 1994 Thinking Machines Corporation

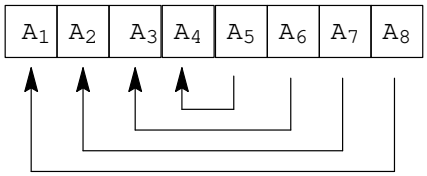
Similarly, the apparent dependence in a "mirror" operation is recognized as not a real dependence:

**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
```

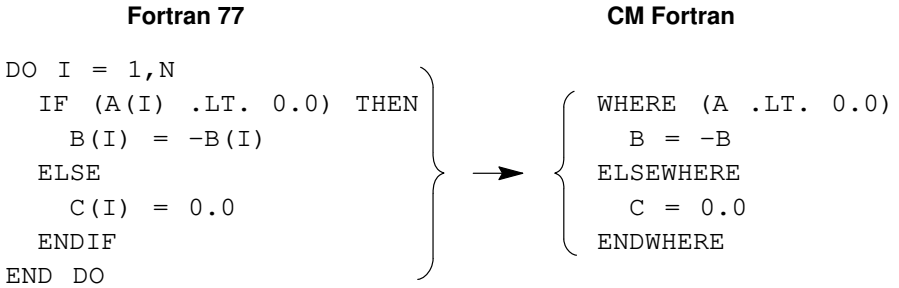
**CM Fortran**

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



*Copyright © 1994 Thinking Machines Corporation*

**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:




## Conversions that Permit Vectorization

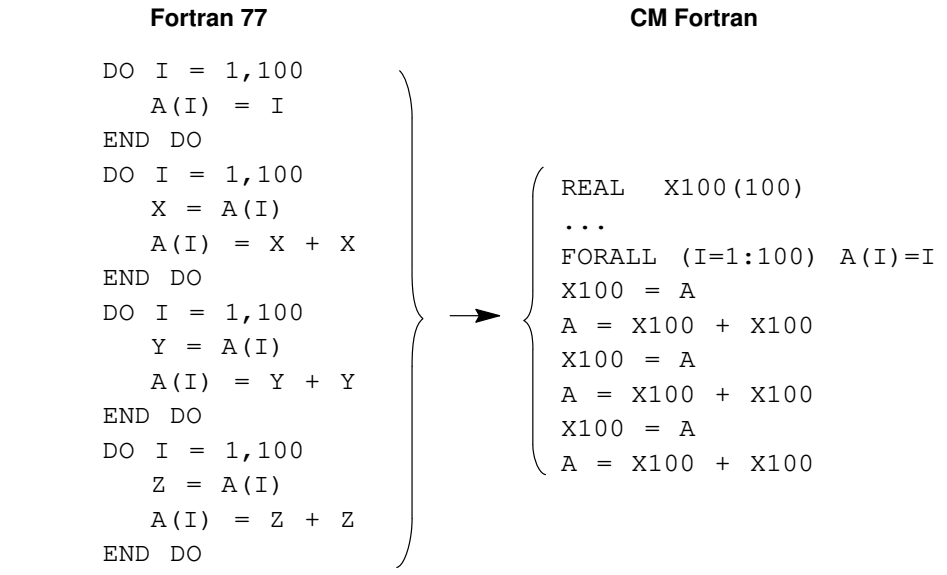
The converter analyzes and may restructure code to facilitate vectorization.

**Scalar Promotion.** The converter 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.

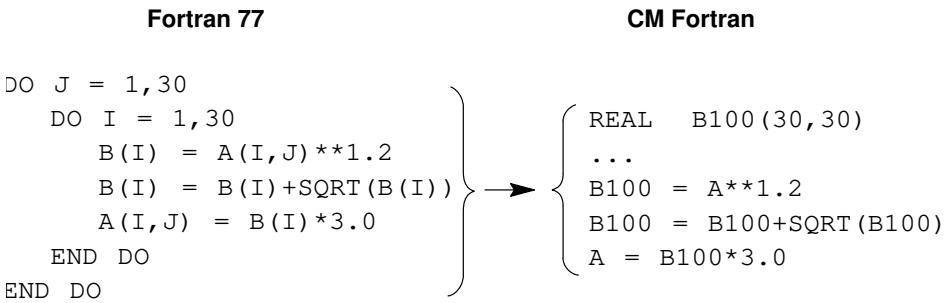
When the converter needs to introduce an array, such as by promoting a scalar or creating a temporary for some other purpose, it tries to reuse a previously introduced array where possible to produce more space-efficient code.

For example, in the following code, all three variables **X**, **Y**, and **Z**, might have been promoted to arrays. However, the converter in fact creates only one array, **X100**, and reuses it.


Copyright © 1994 Thinking Machines Corporation




**Array Promotion.** CMAX also promotes arrays to arrays of higher rank where parallelism will be increased. For example, the following loop uses 2-dimensional array **A** together with 1-dimensional array **B**. By promoting **B** to an array of 2 dimensions, **B100**, the converter can generate code that allows parallel execution along both dimensions of **A**.




A limitation on array promotion is that it occurs only when there is a loop in the next that goes along the array dimension. See Chapter 3 for some workarounds.


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

**Loop Fissioning.** 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)
```

**Statement Splitting.** When the right-hand side of an assignment in a loop is a reduction of a complicated expression, the intention often cannot be represented in simple array syntax. However, the converter may be able to split assignment statements that, inside **DO** loops, comprise such reductions, generating a separate **FORALL** statement and reduction assignment.

For example, the right-hand side of the following **DO** loop cannot be expressed in array syntax, but CMAX vectorizes it by translating it into two statements. (It is generally not necessary to recode complicated reduction expressions by hand to reveal reduction idioms, as advised for CMAX 1.0.)

**Fortran 77**

```
REAL A
REAL X(N),X2(N)
...
DO I = 1,N-3
   A = A + ABS(I + X(I+1) *
.      SQRT(X2(I+2)))
ENDDO
```

→

**CM Fortran**

```
REAL A
REAL X(N),X2(N)
REAL TEMP(N) ! temp CM array
...
FORALL (I = 1:N - 3)
.   TEMP(I+1) = ABS(I + X(I+1) *
.   SQRT(X2(I + 2)))
A = A + SUM(TEMP(2:N - 2))
```


Copyright © 1994 Thinking Machines Corporation

**Loop Pushing.** 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
```



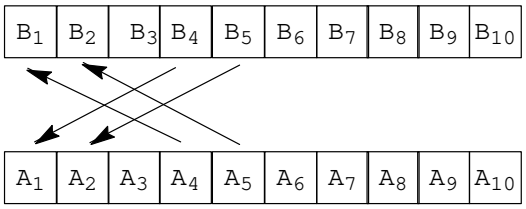
*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

### 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


Copyright © 1994 Thinking Machines Corporation

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
```

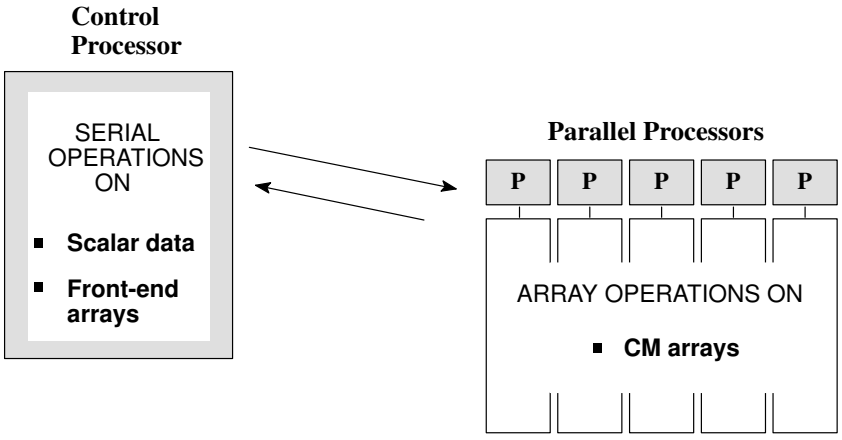

*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

### 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."


Copyright © 1994 Thinking Machines Corporation

### 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.


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

### 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 warns 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 converter switch **-ShortVectorLength**=*nn*. 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


Copyright © 1994 Thinking Machines Corporation

enabling arrays to become CM arrays and be subject to array operations. See also the instructions for converting in "firewalls" mode in Chapter 2, which avoids many of the problems of mismatched array homes.

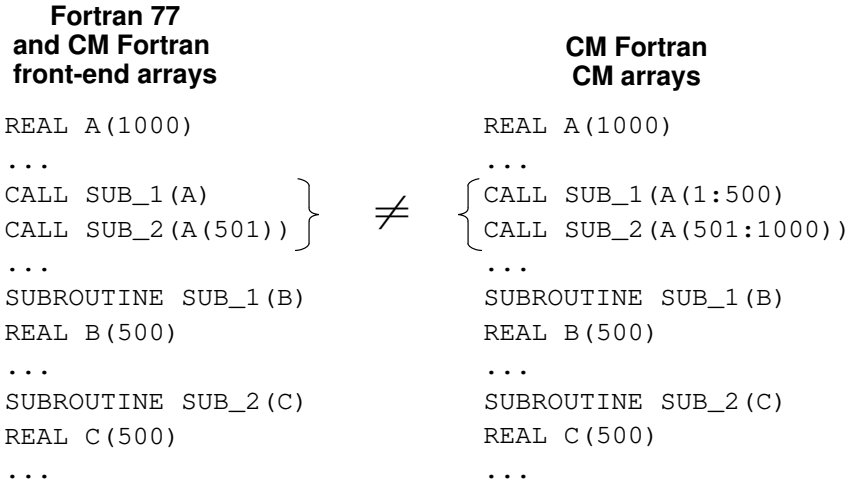
### 1.4.3 Argument Passing

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:





Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

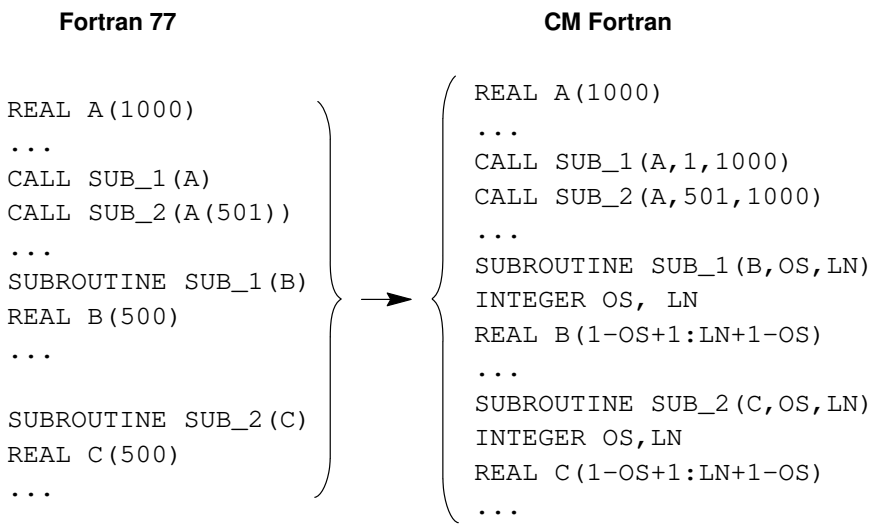
Both the calls on the left fail when **A** is a CM array because it is an error to change array size or shape across subroutine boundaries. In fact, the argument to **SUB_2** is not an array at all, since the expression **A(501)** indicates a single array element. Where **A** is a CM array, this element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





Copyright © 1994 Thinking Machines Corporation

### 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)
      ...
```


*Version 2.0, May 1994*
*Copyright © 1994 Thinking Machines Corporation*

### 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
```


Copyright © 1994 Thinking Machines Corporation

### 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.


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation