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.

Copyright © 1994 Thinking Machines Corporation

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 11 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 11 **`-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.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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

Copyright © 1994 Thinking Machines Corporation

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.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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

Copyright © 1994 Thinking Machines Corporation

```
      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

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

```
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**.

Copyright © 1994 Thinking Machines Corporation

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
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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
```


Copyright © 1994 Thinking Machines Corporation

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
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

3.6 Miscellaneous Conversion Hints

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

3.6.1 Limitation on Array Promotion

A limitation of array promotion is that it occurs only when there is a loop in the nest that goes along the array dimension. Thus, array **zz** in this code is not promoted:

```
DO K = 1,30
  DO J = 1,30
    DO I = 1,30
      ZZ(1) = A(I,J,K) * 1.0
      ZZ(2) = A(I,J,K) * 2.0
      ZZ(3) = A(I,J,K) * 3.0
      A(I,J,K) = ZZ(1) + ZZ(2) + ZZ(3)
    END DO
  END DO
END DO
```

There are two workarounds for this limitation.

In some cases, as in the example above, you can rewrite the code by introducing a new loop nest:

```
DO K = 1,30
  DO J = 1,30
    DO I = 1,30
      DO IDX = 1,3
        ZZ(IDX) = A(I,J,K) * IDX
      END DO
      A(I,J,K) = ZZ(1) + ZZ(2) + ZZ(3)
    END DO
  END DO
END DO
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

In other cases, the array can be dispensed with and scalars used instead. The scalars will be promoted in the usual way:

```
DO K = 1,30
  DO J = 1,30
    DO I = 1,30
      ZZ1 = A(I,J,K) * 1.0
      ZZ2 = A(I,J,K) * 2.0
      ZZ3 = A(I,J,K) * 3.0
      A(I,J,K) = ZZ1 + ZZ2 + ZZ3
    END DO
  END DO
END DO
```

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
```

(Version 2.2 of CM Fortran enhances the **INCLUDE** statement to accept relative pathnames and heed the **-I** switch.)

Copyright © 1994 Thinking Machines Corporation

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 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, 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
...
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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
...
```


Copyright © 1994 Thinking Machines Corporation