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*]

Copyright © 1994 Thinking Machines Corporation

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 Table 1 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 Table 1 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**

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

Table 1. `cmax` command options, recommended abbreviations, and default values, if any.

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=	
-Interface Routines= *program-unit-name-list*	-Interface=	
-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

Copyright © 1994 Thinking Machines Corporation

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 Table 1. 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 4, 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

Copyright © 1994 Thinking Machines Corporation

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 4. Sample package-creation and conversion session.

Copyright © 1994 Thinking Machines Corporation

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
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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
```


Copyright © 1994 Thinking Machines Corporation

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.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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**.

2.3.2 Using make Files

Because it needs to perform interprocedural analysis, CMAX does not support incremental conversion (except in the "firewalls" mode described in Section 2.3.3). 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 of the whole package for each translation.

Thus, 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 sample **make** files that you can adapt as needed. One, reproduced in Figure 5, converts and compiles for the CM Fortran CM-5 global model; others convert and compile for the CM-5 nodal modal, which uses calls to the CM message-passing library CMMD for communication, or for the CM-2/200 slicewise model.

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

- CM-5 and standalone 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.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

```
# 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 5. Sample `make` file for CMAX, global program execution model

Copyright © 1994 Thinking Machines Corporation

2.3.3 Converting Partial Programs with Firewalls

CMAX provides "firewalls mode" as a way to convert programs partially or incrementally. In this mode, CMAX manages the interfaces and maintains data compatibility — particularly with regard to array homes — between the converted and unconverted portions of the program. The resulting **`.fcm`** files can be compiled and linked with the remainder of the original program into a single executable.

The CMAX user identifies one or more kernels of computationally intensive subroutines on which to focus the conversion effort. In "firewalls" mode, the kernel is converted into CM Fortran while the part outside the kernel remains in Fortran 77. All arrays, including arrays in **`COMMON`**, that are used in a parallel way within the kernel are copied to CM arrays, and, if modified by the kernel, are copied back to front-end arrays on exit. By confining the conversion effort to the kernel, many problems that might restrict array homes to the front end — which often occur in parts of the program that are not computationally intensive — can be avoided.

In addition, program units that are called both within the kernel and outside can be cloned, if necessary, into variants that accept front-end array arguments outside the kernel and CM array arguments within it.

Setting Up Firewalls

The user specifies the boundary at which "firewalls" are erected with the **`-InterfaceRoutines=`** option, followed by one or more program unit names.

The output generated from an identified interface routine **`FOO`** is two separate routines:

- **`FOO_V1`** is the original routine converted to CM Fortran
- **`FOO`** becomes a new interface routine that does the following:
  - creates CM arrays corresponding to all the arrays used in array operations in any of its callees
  - transfers data from each front-end (Fortran 77) array to the corresponding CM array
  - calls the converted routine **`FOO_V1`**

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

- transfers data from CM arrays back to the corresponding front-end arrays if the data has been modified.

For example, Figure 6 shows a simple routine that has been identified as a firewall interface, along with the two routines CMAX generates.

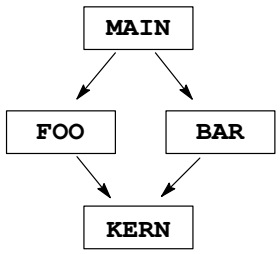
```
    % cmax package-name -T -InterfaceRoutines=FOO


Input in foo.f

      SUBROUTINE FOO(X)
      REAL X(100)
      DO I = 1,100
         X(I) = X(I) + I
      END DO
      END


Output in foo.fcm

      SUBROUTINE FOO(X)     ! generated interface routine
      REAL X(100)
CMF$  LAYOUT X(:SERIAL)     ! its FE array argument
      REAL X0(100)
CMF$  LAYOUT X0(:NEWS)      ! a local CM array
      CALL CMF_FE_ARRAY_TO_CM(X0,X)   ! FE-CM data transfer
      CALL FOO_V1(X)        ! call converted routine below
      CALL CMF_FE_ARRAY_FROM_CM(X,X0) ! CM-FE data transfer
      END

      SUBROUTINE FOO_V1(X)  ! the converted routine
      REAL X(100)
CMF$  LAYOUT X(:NEWS)       ! its CM array argument
      FORALL (I = 1:100) X(I) = X(I) + I
      END
```

Figure 6. Conversion of a specified interface routine.

Copyright © 1994 Thinking Machines Corporation

The package used in "firewalls" conversion should contain the whole program, ideally with each routine in a separate file. The **fsplit** program, available on most Sun systems under **/usr/lang/**, can be used to split files containing multiple subprograms into files containing one subprogram each.

Also, the argument to **-Interface** can be either a program unit name or a list of names. It does not matter whether the subtrees under the interface routines are separate or interdependent. For example, imagine that program **MAIN** calls subroutines **FOO** and **BAR**, and both call the computational kernel **KERN**.



Here, it might make sense to erect the firewall just below **MAIN**:

```
% cmax mypack -Add=main.f foo.f bar.f kern.f
% cmax mypack -T -InterfaceRoutines=foo bar
```

It is also advisable to use the option **-OutputExtension=**, for reasons that will become clear in the following section.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

A More Elaborate Example

This section contains a more elaborate example of conversion in "firewalls" mode. It illustrates the use of common arrays and of subprograms called both inside and outside the kernel. The Fortran 77 input program is shown in Figure 7; it is also on line under **examples/cmax/**.

include.inc

```
      INTEGER N, SQRTN
      PARAMETER (N = 100, SQRTN = 10)
      REAL COEFFICIENTS(N)
      COMMON /SHMOMMON/ COEFFICIENTS
```

main.f

```
      PROGRAM MAIN
      INCLUDE 'include.inc'
      REAL SNOW(N)

      CALL INPUT(SNOW)
      CALL MUNCH(SNOW)
      CALL OUTPUT(SNOW)
      END
```

input.f

```
      SUBROUTINE INPUT(A)
      INCLUDE 'include.inc'
      REAL A(SQRTN,SQRTN)

      DO J = 1,SQRTN
         DO I = 1,SQRTN
            A(I,J) = MOD(11 * I, 7) + MOD(15 * J, 9)
         END DO
      END DO
      DO I = 1,N
         COEFFICIENTS(I) = MOD(33 * I, 6)
      END DO
      CALL CRUNCH(A)
      END
```

program continues

Figure 7. An input program for conversion in "firewalls" mode.

Copyright © 1994 Thinking Machines Corporation

continued:

output.f

```
      SUBROUTINE OUTPUT(A)
      INCLUDE 'include.inc'
      REAL A(N)

      PRINT 10, A
 10   FORMAT(8F9.2)
      END
```

munch.f

```
      SUBROUTINE MUNCH(A)
      INCLUDE 'include.inc'
      REAL A(N)

      CALL GRIND(A)
      CALL CRUNCH(A)
      END
```

grind.f

```
      SUBROUTINE GRIND(A)
      INCLUDE 'include.inc'
      REAL A(N)

      DO I = 1,N
         IF (A(I) .GT. 0.0) THEN
            A(I) = COEFFICIENTS(I) * SQRT(A(I))
         ELSE
            A(I) = -1.0
         END IF
      END DO
      END
```

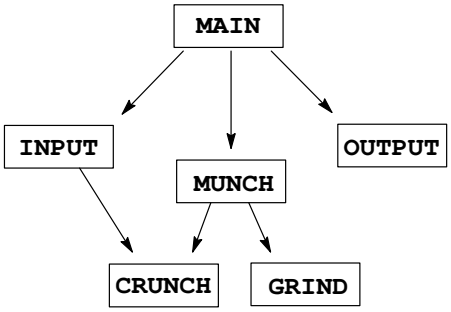
crunch.f

```
      SUBROUTINE CRUNCH(A)
      INCLUDE 'include.inc'
      REAL A(N)

      DO I = 1,N
         IF (A(I) .EQ. 0.0) A(I) = -1.0
      END DO
      END
```

Figure 7. Input program for conversion in "firewalls" mode *(cont'd)*.

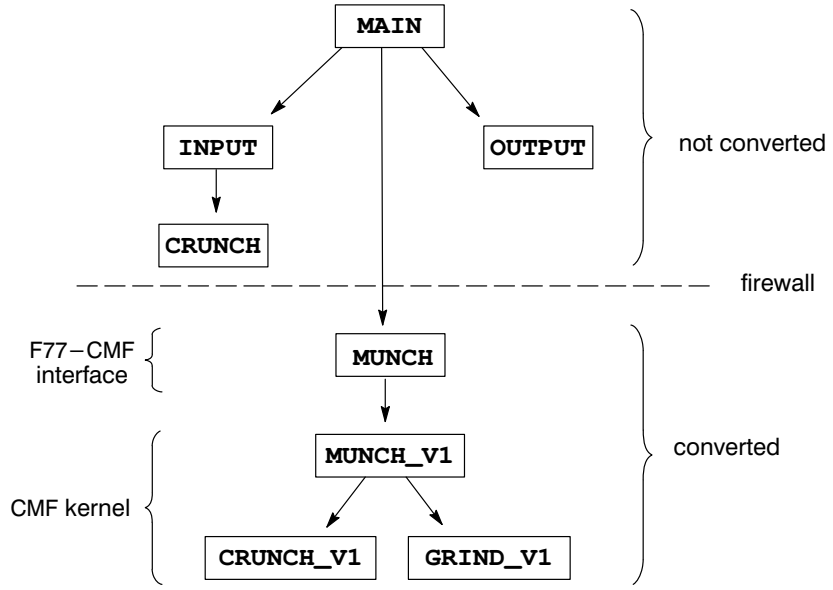
Copyright © 1994 Thinking Machines Corporation

The call tree for the program shown in Figure 7 is:



The computationally intensive kernel of the program is **MUNCH** and its callees **CRUNCH** and **GRIND**. This makes **MUNCH** a good choice as the interface routine. Note that **CRUNCH** is called both from **MUNCH** and from **INPUT**, which is outside the kernel.

After invoking CMAX with **-InterfaceRoutines=MUNCH**, we have a new call structure:



CRUNCH appears twice but now in different variants: in unconverted Fortran 77 form outside the kernel, as well as in CM Fortran form, **CRUNCH_V1**, within the kernel.

Copyright © 1994 Thinking Machines Corporation

MUNCH has been converted to two routines; the new routine **MUNCH** takes front-end arrays, creates CM arrays and transfers the data to them, calls the vectorized **MUNCH_V1**, and then copies the CM array data back to front-end arrays where necessary. From the original code in **munch.f**,

```
      SUBROUTINE MUNCH(A)
      INCLUDE 'include.inc'
      REAL A(N)

      CALL GRIND(A)
      CALL CRUNCH(A)
      END
```

the converted output in **munch.fcm** is an interface routine **MUNCH** and a converted **MUNCH_V1**:

```
      SUBROUTINE MUNCH(A)
      INTEGER N           ! include file expanded
      PARAMETER (N=100)
      REAL A(N)
CMF$  LAYOUT A(:SERIAL)
      REAL A0(N)
CMF$  LAYOUT A0(:NEWS)
      INTEGER LL
      REAL COEFFICIENTS(N)
CMF$  LAYOUT COEFFICIENTS(:SERIAL)
      COMMON /SHMOMMON/COEFFICIENTS
      REAL COEFFICIENTS0(N)
CMF$  LAYOUT COEFFICIENTS0(:NEWS)
      COMMON /CM_SHMOMMON/COEFFICIENTS0 ! CM common block

      CALL CMF_FE_ARRAY_TO_CM(A0,A)
      CALL CMF_FE_ARRAY_TO_CM(COEFFICIENTS0,COEFFICIENTS)

      CALL MUNCH_V1(A0)

      CALL CMF_FE_ARRAY_FROM_CM(A,A0)
      END
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

```
      SUBROUTINE MUNCH_V1(A)
      INTEGER N, SQRTN
      PARAMETER (N = 100, SQRTN = 10)
      REAL COEFFICIENTS(N)
      COMMON /SHMOMMON/COEFFICIENTS0
      REAL A(N)
CMF$  LAYOUT A(:NEWS)
CMF$  LAYOUT COEFFICIENTS(:NEWS)
      REAL COEFFICIENTS0(N)
CMF$  LAYOUT COEFFICIENTS0(:SERIAL)
      COMMON /CM_SHMOMMON/COEFFICIENTS   ! CM common block

      CALL GRIND_V1(A)    ! Body of original MUNCH
      CALL CRUNCH_V1(A)
      END
```

The converted code for **MUNCH** contains declarations and statements to arrange for all the data needed in the parallel code below it (in **CRUNCH** and **GRIND**) to be moved from front-end arrays to CM arrays. The argument array **A** is given a corresponding CM array, **A0**. The front-end common array **COEFFICIENTS**, from the include file, is likewise given a corresponding CM common array, **COEFFICIENTS0**. Two calls to **CMF_FE_ARRAY_TO_CM** before the call to **MUNCH_V1** transfer the data to the CM arrays. Since the kernel modifies **A0** but not **COEFFICIENTS0**, only **A0** is transferred back with a call to **CMF_FE_ARRAY_FROM_CM**.

The converted subroutine **MUNCH_V1** declares its dummy argument and the common block as CM data, since they are used in array operations in its callees. Notice that **MUNCH_V1** uses the original common array name **COEFFICIENTS** to refer to the data in the CM common block **CM_SHMOMMON**. This trick reduces the amount of kernel code that needs to be changed.

Subroutine **INPUT**, which is outside the kernel, also takes array **A** as an argument, but notice what it does with it:

```
      SUBROUTINE INPUT(A)      ! A is 1D in caller
      INCLUDE 'include.inc'
      REAL A(SQRTN,SQRTN)       ! A is 2D in INPUT
      ...
      CALL CRUNCH(A)
      END
```


Copyright © 1994 Thinking Machines Corporation

The subroutine **INPUT** violates CM array argument-passing constraints by redeclaring its 1-dimensional actual argument as a 2-dimensional dummy argument. Since this subroutine does not reside in the kernel **MUNCH**, however, the data need not be constrained to a front-end home inside the firewall. The data in array **A** is transferred to CM array **A0** for use inside the kernel **MUNCH,** and the results later transferred back to **A**. Subroutine **INPUT** also passes array **A** on to **CRUNCH**, the unconverted version of **CRUNCH** that accepts a front-end array argument.

The code in **MUNCH_V1**, **CRUNCH_V1**, and **GRIND_V1** that operates on **A0** is fully vectorized, unconstrained by **A**'s front-end home in **INPUT**.

Using make to Convert and Compile

This section contains a sample Makefile for converting and compiling the program in Figure 7 with a firewall around the kernel **MUNCH**. The Makefile is in Figure 8 and also on line in **examples/cmax**.

One minor problem in CM Fortran 2.1 and earlier versions is that the main program unit of any program that contains CM Fortran code must be compiled with **cmf**. You cannot, therefore, compile the sample program with its original main routine in a **.f** file. You can rewrite the original **MAIN** as a subroutine and provide a dummy **.fcm** main program that calls it (as shown in Figure 9). You may also need a dummy **.f** main program that calls the subroutine **MAIN**, to be used when building a serial version of the program.

The Makefile uses the option setting **-OutputExtension=.cm.fcm** so that both **crunch.o**, created from the Fortran 77 version of **CRUNCH**, and **crunch.cm.o**, created from the converted **CRUNCH_V1**, can be linked into the executable. To avoid link-time errors, you must rename the output files for any routines that will continue to exist in unconverted **.f** files. If a program does not contain subroutines that are called both inside and outside the kernel, the change in output extension is not needed. It is wise precaution, however, to always change the output extension when converting in "firewalls" mode.

Copyright © 1994 Thinking Machines Corporation

```
.SUFFIXES: .f .o .cm.fcm .cm.o
# CMF definitions/defaults. Customize CMFFLAGS and LIBS.
CMF      =  cmf
OPT      =
DEBUG    =
CMFFLAGS =  -cm5 -vu $(OPT) $(DEBUG)
LIBS     =

# CMAX definitions and defaults. Customize as desired.
CMAX        =  cmax
CMAXFLAGS   =  -verbose=4 -list

# The package and the files it contains:
TARGET      =  myprog
PACKAGE     =  myprog
TRANSLATION =  myprog-cmaxed
ALLSRCS  =  main.f input.f output.f munch.f crunch.f grind.f
F77SRCS     =  main.f input.f output.f crunch.f
INTERFACE   =  munch
KERNEL      =  munch crunch grind
KERNELSRCS  =  $(KERNEL:%=%.f)
CMFSRCS  =  dummy-cmf-main.cm.fcm $(KERNELSRCS:%.f=%.cm.fcm)
OBJS     =  $(F77SRCS:%.f=%.o) $(CMFSRCS:%.cm.fcm=%.cm.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. The TARGET depends on both TRANSLATION and 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) : $(ALLSRCS)
        $(CMAX) $(PACKAGE) -Add=$(ALLSRCS) -T \
            -Interface=$(INTERFACE) \
            $(CMAXFLAGS) -OutputExtension=.cm.fcm
        touch $@

# This rule tells how to compile the .fcm with cmf:
%.o: %.cm.fcm
        $(CMF) $(CMFFLAGS) -c $<
clean:
        rm -f $(TARGET) $(TRANSLATION) *.o *.lis *.ttab
        $(CMAX) $(PACKAGE) -remove
```

Figure 8. Sample `make` file for "firewalls" conversion and compilation.

Copyright © 1994 Thinking Machines Corporation

Rewrite `MAIN` as a subroutine:

`main.f`

```
      SUBROUTINE MAIN()
      INCLUDE 'include.inc'
      REAL SNOW(N)

      CALL INPUT(SNOW)
      CALL MUNCH(SNOW)
      CALL OUTPUT(SNOW)
      END
```

Have `make` choose one of these main programs:

`dummy-cmf-main.cm.fcm`

```
      PROGRAM DUMMY_CMF_MAIN
      CALL MAIN()
      END
```

`dummy-f77-main.f`

```
      PROGRAM DUMMY_F77_MAIN
      CALL MAIN()
      END
```

Figure 9. Multiple main routines for Makefile.

2.3.4 Converting Partial Programs without Firewalls

Besides "firewalls," CMAX provides a second way to convert single modules or subtrees of a program. With this older functionality, the user must manage the interfaces between converted and unconverted parts of a program. Specifically, the user must guard against array home mismatches across program boundaries and against vectorizing operations on an array that is constrained to a front-end home in another program unit. With these caveats, the older functionality may be more convenient for some situations, since it avoids the array copying and procedure cloning that CMAX performs in "firewalls" mode.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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. Either write a dummy main program that calls each of the subprograms, or 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 without “firewalls” (Section 2.3.3), 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** controls 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.

Copyright © 1994 Thinking Machines Corporation

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
```


Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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
```


Copyright © 1994 Thinking Machines Corporation

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

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.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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
```


Copyright © 1994 Thinking Machines Corporation

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

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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:

Fortran 77

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

CM Fortran

```
FORALL (I=1:N)
.    A(NDX(I)) = B(I)
```

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**.

Copyright © 1994 Thinking Machines Corporation

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

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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)
```


Copyright © 1994 Thinking Machines Corporation

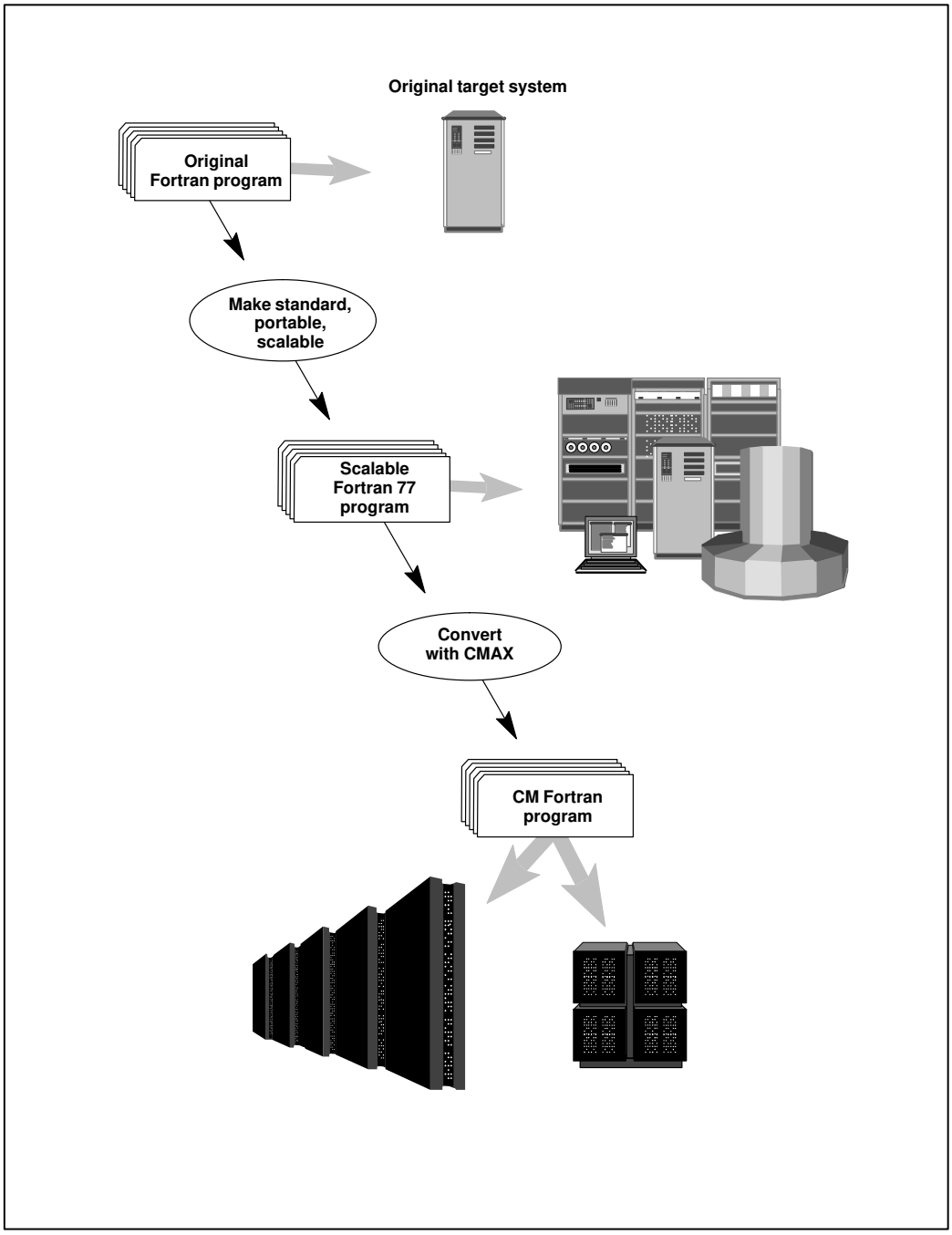


Figure 10. The generic porting process, from older Fortran to the CM and other systems.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation

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 10 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.

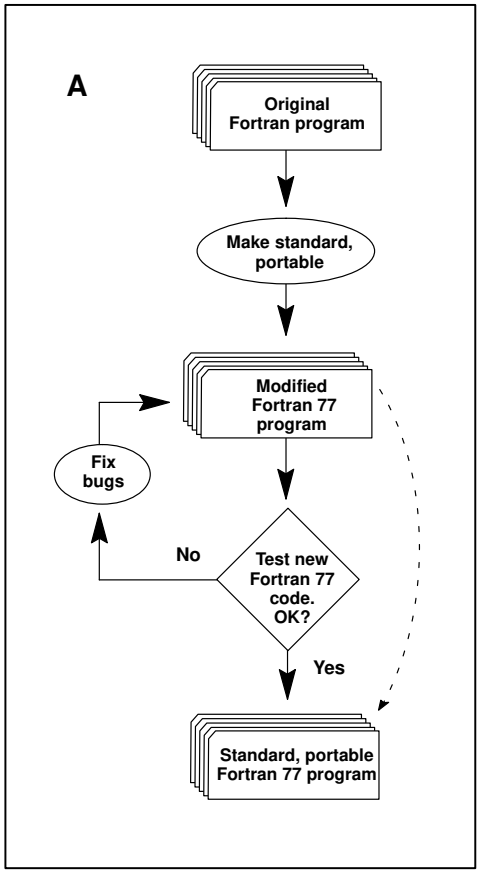
Consider the cross-product of two factors:

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

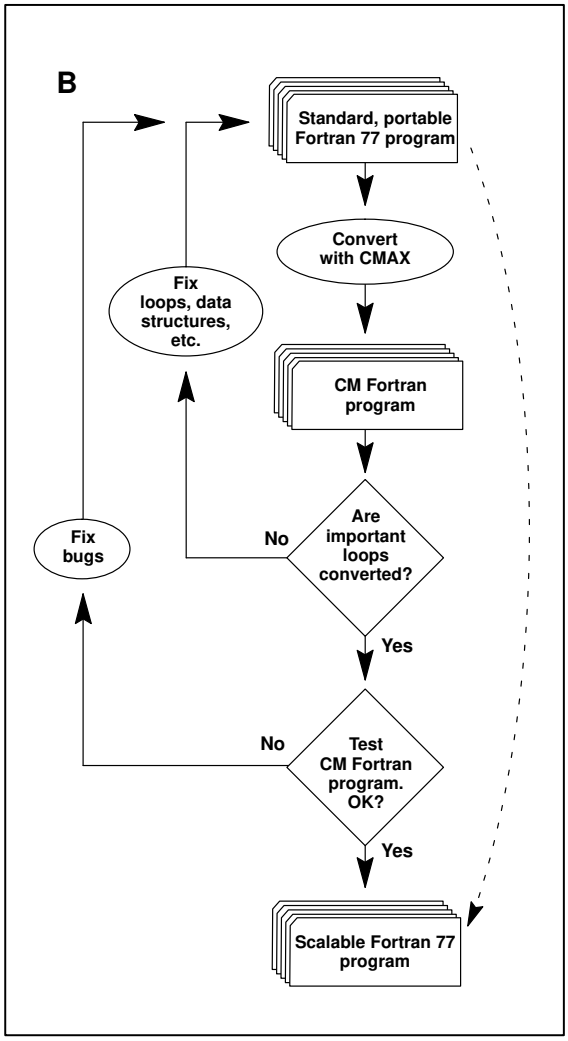
Users in groups 1 and 2 are porting existing code to the CM system, with the intention either of maintaining the code in Fortran 77 for portability to other architectures (group 1) or of maintaining it in CM Fortran for the CM only (group 2). All these users need to perform the tasks shown in flowcharts 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.

Copyright © 1994 Thinking Machines Corporation

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.

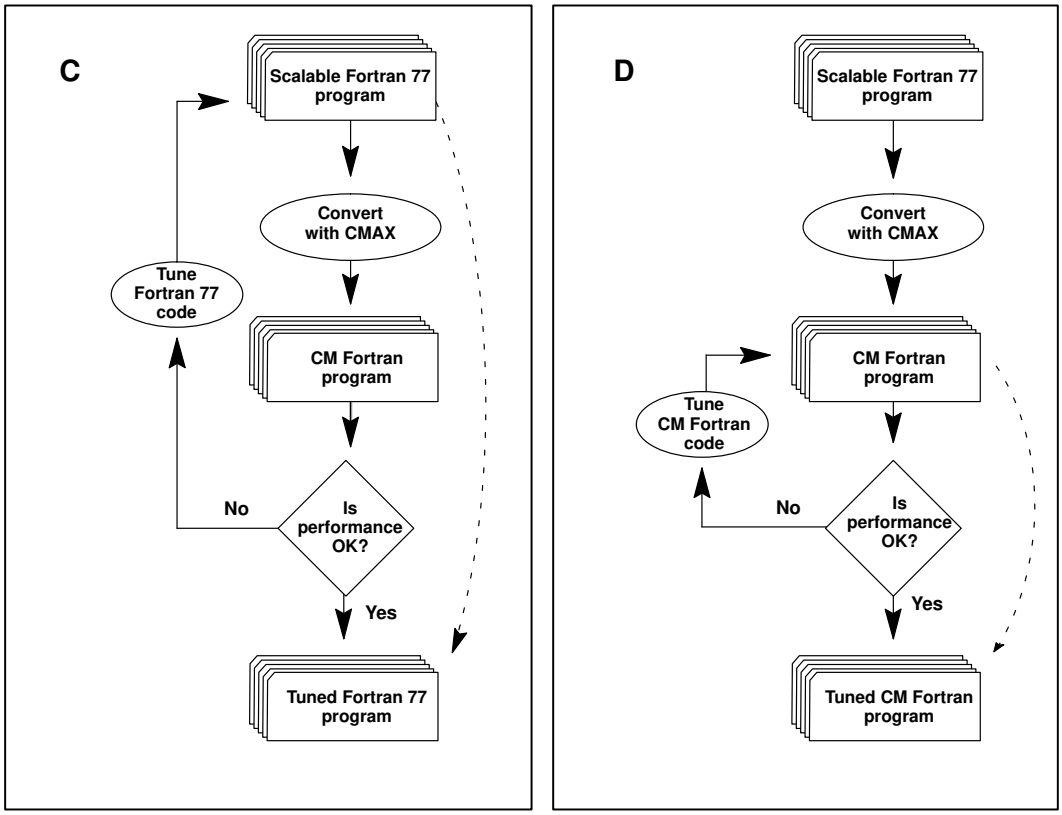


Older Fortran to standard, portable Fortran 77



Standard, portable Fortran 77 to scalable Fortran 77

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation



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.

Copyright © 1994 Thinking Machines Corporation

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.

Version 2.0, May 1994
Copyright © 1994 Thinking Machines Corporation