CMAX: A Fortran Translator for the Connection Machine System

Gary Sabot and Skef Wholey
Thinking Machines Corporation
245 First Street
Cambridge, MA 02142

Abstract

The CMAX translator converts applications written in scalable Fortran 77 to parallel Connection Machine Fortran. The most obvious part of the translation problem, and one addressed by a number of previous translators as well as CMAX, is loop vectorization: the substitution of array syntax and intrinsics for Fortran 77 `DO` *loops. A less obvious (but equally important) part of the translation problem is the conversion of constructs arising from Fortran 77's linear memory model into code which does not rely on storage or sequence association. Most such constructs can only be detected and repaired through interprocedural analysis and transformation; CMAX is the first translator to perform such repairs. This paper describes the CMAX translator and some of the more important transforms that we have distilled from our users' porting experiences.*

1 Introduction

Fortran 77, which is available on almost all computing platforms, has served as the standard language of scientific computing. Its universality allows programs to be ported simply by recompiling, but Fortran 77 codes are often tuned to the details of a particular machine architecture and may not perform well after being recompiled for a new target machine. The wide variety of architectural features that appear in various high-performance target machines (highly pipelined functional units, multiple function units, massive parallelism, and distributed memories) make demands that often are not met by "dusty" Fortran 77 applications. The portability of a code's performance across different architectures—its *scalability*—is not guaranteed, and must be engineered in. Serendipitously, the process of modernizing a code to make it scalable often will improve its performance on the original target machine. One extreme example of this was encountered while modernizing a Fortran 77 astrophysics code in preparation for a port to a Connection Machine® computer (CM). Although the original authors believed that they were running as fast as was possible on the original target machine, a Cray Y-MP, modernizing the code and simplifying its loop structure (not changing the underlying algorithm) sped the code up over 40 times on the Cray [2]!

The CM Automated X-lator (CMAX) is a tool that converts standard Fortran 77 (F77) into CM Fortran (CMF), a modern Fortran which incorporates the array expression syntax of Fortran 90 (F90). When applied to a program written in scalable F77, the output of CMAX has good performance on distributed-memory, massively parallel systems like the CM.[1] Thus, CMAX provides a migration path for serial programs onto the CM.

CMAX can be used to assist in a one-time porting effort, at the conclusion of which the user discards the old F77 and maintains the CMF output. Alternatively, CMAX users can continue to maintain their scalable software in F77 for maximum portability to multiple platforms and can run CMAX as a preprocessor before regular compilation by the CMF compiler. Development of new scalable code might also be done in F77 using CMAX either for portability or because developers familiar with F77 prefer to continue using it rather than CMF.

This paper describes the CMAX translator and some of the experiences of initial CMAX users. Figure 1 illustrates the two separate code transformation steps involved in running a scalable F77 code on a Connection Machine system (a CM-2, CM-200, or CM-5). This paper focuses on the first step, in which code is converted into CMF by using CMAX. The second step of the conversion process involves compiling CMF for execution on the parallel target machine. That compilation process is described elsewhere [17, 18].

CMAX is based on Forge technology from Applied Parallel Research (APR) [12]. Thinking Machines Corporation (TMC) identified a number of capabilities lack-

ICS-7/93 Tokyo, Japan
© 1993 ACM 0-89791-600-X/93/0007/0147...$1.50

[1] While some contructs have slightly different performance characteristics across the Connection Machine family, CMAX does not distinguish between these different machines.

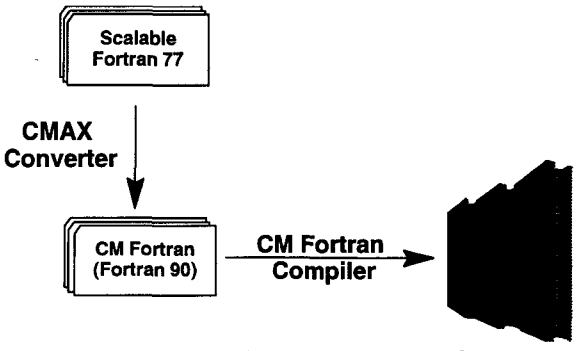

Figure 1: Conversion process using CMAX.

ing in existing vectorization tools and contracted with APR to implement them. TMC is currently continuing development of the code in-house with consulting help from APR. At present, the system consists of 340,000 lines of C code, of which over 60,000 lines are new and CMAX-specific. In addition, TMC's CMAX Quality Assurance test library contains over 100,000 lines of F77 code.

2 Scalable Fortran

A useful definition of a scalable code is one that attains respectable performance over a range of problem sizes and over a range of target machines (including conventional serial processors, vector computers, and massively parallel computers). Said another way, scalable code has performance which is portable between platforms. Clearly, scalability is becoming more important as various high-performance architectures proliferate, each making demands that break with the conventions of older serial models. Scalable Fortran is described in detail in [20]; these two rules capture the essence of the scalability conventions described there:

1. Scalable code operates on all (or nearly all) the data "at once." In F77, this means looping over as much of as many array axes as possible, with each iteration of an inner loop corresponding to an array element.
2. Scalable code does not assume a particular memory model. Multidimensional arrays should be declared as such and be declared consistently throughout the program. For example, it is not scalable to pass a 3D array to a routine that zeros out a 1D array, even if the linear sizes match. Scalable code does not make use of the linear memory model, and it therefore must avoid relying on:
   - Sequence association: The linearization of multidimensional values to unidimensional objects in column-major order.
   - Storage association: The sharing of storage between two or more variables or arrays, caused by EQUIVALENCE and by some usage of COMMON.

Non-scalable constructs in an application reduce the output code's performance on powerful target machines. For example, a use of storage association (if it is not recognized as an idiom and translated away) can force an array to be stored on a single processor to preserve correctness.

Even if a code does not contain non-scalable constructs, it may be that the basic algorithm is inherently serial. Unfortunately, rewriting scientific algorithms is beyond the capability of current automatic tools. What CMAX does is attempt to remove the drudgery of rewriting the syntax of a code that is already scalable in nature and is serial at only a superficial level.

Adhering to scalability guidelines can improve performance on serial and vector computers, as well as enabling translation to efficient parallel code. Consider the two program fragments in Figure 2. The first fragment, (a), uses a nest of four loops to compute a two-dimensional "stencil" operation. The second fragment, (b), uses a nest of two loops to compute the same stencil. The second is more scalable than the first, because each iteration of the inner loop corresponds to a unique element of the array B. The table, (c), shows times for the two loops on a serial computer and a vector computer in both scalar and vector mode. On the serial computer (a Sun 4/490), the scalable code is over three times faster; on the vector computer (a Cray Y-MP), the scalable code is over one hundred times faster in vector mode. Also note that while vectorization helps the scalable code, it actually hurts the non-scalable code. In addition to being faster on these platforms, the scalable loop is easily translated into parallel CMF whereas the non-scalable loop is not.

3 Translation

There are two major pieces to the problem of translating scalable F77 into CMF. The first piece involves substituting array operations for statements that operate on array elements within F77 loops [1]. This important and visible piece of the problem has been addressed by many translators, such as KAP and VAST [4, 13]. Most such translators operate on a subroutine-by-subroutine basis with no interprocedural analysis.

The equally important second piece of the problem is to translate constructs arising from F77's linear memory model so that they do not hinder performance on a distributed memory target machine. Most such constructs can only be detected and repaired through interprocedural analysis. Most vectorizing compilers and translators do not address this problem, which is a major focus of our CMAX effort.

The distributed memory of many parallel target machines is non-uniform in that it draws a distinction between the separate memory spaces of the Processing El-

```
DO J = 2,N-1
   DO I = 2,N-1
      B(I,J) = 0.0
      DO JJ = J-1,J+1
         DO II = I-1,I+1
            B(I,J) = B(I,J) + A(II,JJ)
         END DO
      END DO
   END DO
END DO
```

(a) Loop 1

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

(b) Loop 2

	Sun 4/490	Cray Y-MP (scalar)	Cray Y-MP (vector)
Loop 1	5.639	2.494	5.282
Loop 2	1.891	0.338	0.052

(c) Times

Figure 2: The two program fragments (a) and (b) perform the same stencil computation. Table (c) shows the times in seconds of these two fragments for $n = 1000$.

ements (PEs). For example, the CM-2, CM-5, MasPar MP-1, and I-WARP machines all consist of processors with their own local memories. Arrays on such machines can either be stored on a single processor or be distributed in some fashion among multiple processors. The variety of possible distributions means that when arrays are passed as arguments, the procedures must agree on how an array is stored. In CM Fortran, a mismatch between the array distribution of a dummy and actual argument is treated as a user error unless an interface block describes the dummy to caller mismatch. Even with an interface block, expensive communication is required to move arrays back and forth to cover up the mismatch. Interprocedural analysis allows CMAX to insert consistent array layout directives program-wide. Arrays used in vectorized loops are distributed across the PEs of the CM; others are stored in a single processor.

The interprocedural analysis required to address the linear memory model problem has a side benefit in that it can improve loop vectorization. For example, it enables CMAX to transform a loop with an embedded subroutine call into a call to a subroutine that contains the loop. In that form, it is possible that, subject to dependence analysis, the loop can be vectorized.

One ramification of interprocedural transformations that push loops and set array distributions is that a single subprogram might need to be called in multiple ways: one caller might pass a distributed array and another might not, one might push a loop down into the subprogram and another might not. The solution used by CMAX is to *clone* subprograms [7]. For each distinct transformed version of a subprogram that some caller requires, CMAX creates a renamed copy of the subprogram and modifies it appropriately.

In general, the process of compiling many cloned copies of a subroutine could lead to code explosion. However, other than loop pushing, the changes that lead to clones are tied to individual array arguments, and the number of transforms that can be applied to any one argument is small. In the case of loop pushing, the number of pushable loop is generally small. In practice, code explosion has not happened. For example, the F77 program X-PLOR started out with 698 subroutines. CMAX's output contained 83 additional routines due to variants. Using another measure, CMAX's input had 75,000 lines and its output had 96,000 lines. Of the output lines, 7,000 were CMF `LAYOUT` directives and 2,500 were descriptive comments inserted by CMAX.

The Convex Application Compiler performs interprocedural dependence analysis, inlining, and constant propagation, but because its target machine provides the illusion of a single flat linear memory space, it does not need to repair linear memory model constructs [6]. We believe that distributed memory architectures are much more amenable to scaling into thousands of processors than are shared memory architectures, which must expend hardware resources to provide the illusion of a single linear address space. In order to use large numbers of processors it is therefore necessary to replace constructs which rely on the linear memory model with those which do not.

The Convex Application Compiler clones subroutines in order to make possible interprocedural constant propagation (e.g., of loop bounds), but it does not clone them for other optimizations. It uses inlining in order to perform an optimization equivalent to CMAX's loop pushing [8].

The nature of the translation process and the large numbers of idioms recognized make it impossible to de-

scribe all CMAX transformations here. Therefore, we simply list the most important CMAX transformations and give a few small code examples to illustrate some of these.

3.1 Standard Vectorization Transformations

`DO` loops are vectorized using dependence analysis, scalar promotion, loop fissioning, and idiom recognition. Nested loops are vectorized into multidimensional array operations. Idiom recognition allows loops with certain types of dependences to be converted into parallel operations. Such operations include reductions, such as `SUM` and `MAXVAL`, and more complex computations such as `DOTPRODUCT`, `MAXLOC`, and `MATMUL`. See Figure 3.

The output program uses array syntax for elementwise computations, `WHERE/ELSEWHERE` for conditionalized parallel operations, and F90 intrinsics for array transformations. CMAX decision rules for conversions can be controlled by CMAX directives at the level of single loops or whole procedures, or globally through command-line switches.

Some location-dependent operations are hard to express in generic F90, and in some cases CMF's `FORALL` (an F90 "removed extension" which is a part of the High Performance Fortran (HPF) standard [10]) statements are generated to express succinctly the computation. Parallel prefix computations (or "scans") are recognized and expressed as calls to CMF Utility Library subroutines.

3.2 Interprocedural Transformations

Without interprocedural knowledge, a translator would have to make conservative assumptions about storage and sequence association. On Connection Machine systems, such conservatism tends to interfere with efficient execution. CMAX performs extensive interprocedural analysis and code transformations:

- CMAX generates `CMF$LAYOUT` directives to control array distribution. Arrays that are used in vectorized loops are distributed across the memories of the PEs; other arrays are not. The user may insert layout directives in the F77 source code to override the default behavior. Both user directives and automatically-inserted directives are propagated consistently throughout the program. In CM Fortran, a given array has a single layout throughout the life of the program. CMAX detects layout conflicts (which can only arise from user errors) and reports them to the user.
- CMAX detects uses of arrays that are incompatible with the distributed memory model. For example, some uses of `COMMON` (Figure 4) or `EQUIVALENCE` (Figure 5) statements can lead to an incompatibility. In such cases, CMAX generates `CMF$LAYOUT` directives to place arrays in the memory of a single processor to preserve the correctness of the code. Since the arrays are not distributed among multiple processors, all access to them will be serial rather than parallel. The user is informed of the possible loss of performance due to specific problem lines in the source code.
- CMAX pushes loops into subprograms. A loop with an embedded subroutine call can be transformed into a call to a subroutine that contains the loop, thus making it possible to vectorize the loop. See Figure 6. If dependence analysis indicates that it is legal, a loop can be distributed across each of the contained statements. As a result, even if one statement in a loop is not vectorizable, it will not inhibit vectorization or loop pushing involving the other statements.
  In the course of passing a number of codes through CMAX, we discovered that the loop pushing transformation increased CMAX compile time without a corresponding improvement in the output code's performance. As a result, by default we have turned loop pushing off in CMAX; users can turn it on where needed.
- CMAX passes entire arrays when possible. Calls are modified to pass whole arrays rather than elements or sections. This generally leads to more efficient execution, and often makes it possible to push loops into subroutines, which in turn makes vectorization possible. See Figure 7.
- CMAX passes array slices when appropriate. F77 users often pass n-minus-k-dimensional slices of n-dimensional arrays to subprograms in a way that assumes sequence association. For example, they might pass a two-dimensional slice of a three-dimensional array `A` as `A(1,1,n)`. CMAX recognizes when caller and callee are cooperating in this manner and produces CMF that does not depend on sequence association. In this case, the call would pass `A(:,:,n)`. See Figure 8.
- CMAX clones subprograms. Transformations such as those listed in this section can change the interface to a subroutine. CMAX generates copies of subprograms before modifying them in cases where multiple callers need to call a subprogram with different interfaces. For example, one caller might push two loops into a subroutine while another caller does not push any loops.
- CMAX recognizes dynamic memory allocation. Unlike F90 (and CMF), F77 lacks a dynamic memory facility, so users often construct their own idiosyncratic dynamic memory allocation facilities. Often these operate by carving up a large `COMMON` block, making use of the linear memory model (sequence and storage association) in a non-scalable way. CMAX defines a clean, portable F77 interface to a dynamic memory allocation facility. When passed through CMAX, these calls are translated into the appropriate F90 dynamic allocation constructs. We provide

```
REAL FUNCTION DOT(X,Y,N)                          REAL FUNCTION DOT(X,Y,N)
REAL X(N), Y(N)                                   REAL X(N), Y(N)
REAL RESULT                                       REAL RESULT
RESULT = 0                                  CMF$  LAYOUT X(:NEWS)
DO I=1,N                              ⇒     CMF$  LAYOUT Y(:NEWS)
   RESULT = RESULT + X(I) * Y(I)                  RESULT = 0
ENDDO                                             RESULT = RESULT + DOTPRODUCT(X,Y)
DOT = RESULT                                      DOT = RESULT
END                                               END
```

Figure 3: Dot product function in F77 and after translation by CMAX. The most notable difference between the input and output of CMAX is that CMF array operations are substituted for F77 loop iterations and that explicit directives are generated describing how to lay out arrays across the processing elements. `:NEWS` invokes a block array distribution in CMF.

```
SUBROUTINE KERNEL( TK)
...
COMMON /SPACE1/ U(1001), V(1001), W(1001),
1 X(1001), Y(1001), Z(1001), G(1001),
2 DU1(101), DU2(101), DU3(101), GRD(1001), DEX(1001),
3 XI(1001), EX(1001), EX1(1001), DEX1(1001),
4 VX(1001), XX(1001), RX(1001), RH(2048),
5 VSP(101), VSTP(101), VXNE(101), VXND(101),
6 VE3(101), VLR(101), VLIN(101), B5(101),
7 PLAN(300), D(300), SA(101), SB(101)
...
...
SUBROUTINE SUPPLY(i)
...
COMMON /SPACE1/ U(19977)
...
```

Figure 4: A major `COMMON` block used in the Livermore loops is declared inconsistently between subroutines in such a way to inhibit conversion. CMAX detects this inconsistency and forces the arrays to be allocated in a single processor.

a serial implementation of our interface so that users do not have to sacrifice portability when using this new interface.

3.3 Generating Maintainable Code

In addition to the code transformations described above, CMAX has a number of features that are important for producing CMF output that is readable and maintainable. These features are missing from most preprocessor-type vectorizers, but are important for users who plan to use CMAX to produce a maintainable output program:

- `PARAMETER`s are retained as named entities and are not substituted in-line.
- User statement labels are not renamed or removed.
- `INCLUDE` files are not expanded in-line unless they contain code that must be modified (which they rarely do).

```
COMMON /SPACE1/ U(1001), V(1001), W(1001),
1 X(1001), Y(1001), Z(1001), G(1001),
...
DIMENSION ZX(1023), XZ(1500), TK(6)
EQUIVALENCE ( ZX(1), Z(1)), ( XZ(1), X(1))
```

Figure 5: Since `X` and `Z` are only given 1001 elements, `ZX` and `XZ` use storage in `G` and `Y` as well as `X` and `Z`, respectively. The current CMF compiler does not support `EQUIVALENCE` on distributed arrays; even if it did, this kind of memory use is inherently nonscalable as it greatly constrains the distribution of the arrays involved. CMAX detects the requirement for contiguous allocation of `X`, `Y`, `Z`, and `G`, and forces these arrays to be allocated in a single processor.

- Output code contains comments describing interprocedural modifications made during conversion, such as adding arguments to a subroutine.

4 Debugging and Performance Analysis

CMAX is tightly coupled to Prism, a window-based debugger and program analysis tool. Within Prism, users can develop, execute, debug, and analyze the performance of programs written for Connection Machine systems [19]. Prism has been in use for several years with programs written in CMF and C*. CMAX piggybacks onto Prism's CMF support, and has been extended to accommodate users working with CMAX-translated applications. The new Prism environment provides facilities for examining CMAX CMF output and viewing it along side the input F77 program. All Prism features work in terms of either (or both) the original F77 or the CMF, and the user can change the active point from F77 to CMF with a single mouse click. Prism's new CMAX-related capabilities are unusual in that most vectorizers that produce source-level output code require the user to debug solely in terms of that output code.

CMAX delivers information about the transformations made to the input program through an auxiliary

```
...
DO K = 1,N
   CALL DOTP(A, B, C, K, N)
END DO
...
SUBROUTINE DOTP(A, B, C, K, N)
REAL A(N,N), B(N,N), C(N,N)
DO J = 1,N
   DO I = 1,N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
   END DO
END DO
END
```

⇒

```
      ...
      CALL DOTP_P1(A,B,C,N)
      ...
      SUBROUTINE DOTP_P1(A,B,C,N)
      REAL A(N,N), B(N,N), C(N,N)
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT C(:NEWS,:NEWS)
      A = A + MATMUL(B,C)
      END
```

Figure 6: CMAX pushes the outer loop into the DOTP subroutine, then vectorizes the subroutine, resulting in a MATMUL. The caller is modified to call the new subroutine, DOTP_P1.

```
REAL AA(NN)
...
CALL CRUNCH(AA(5), 57)
...
SUBROUTINE CRUNCH(A, N)
REAL A(N)
DO I = 1,N
   A(I) = SQRT(A(I)) + 2.0
END DO
END
```

⇒

```
      REAL AA(NN)
      ...
      CALL CRUNCH_V1(AA,5,NN,57)
      ...
      SUBROUTINE CRUNCH_V1(A,OS,LN,N)
      INTEGER OS
      INTEGER LN
      REAL A(1-OS+1:LN+1-OS)
CMF$  LAYOUT A(:NEWS)
      A(1:N) = SQRT(A(1:N)) + 2.0
      END
```

Figure 7: CMAX detects that when the calling routine passes AA(5) to the called routine, the programmer's intent is to pass a pointer into the middle of that array. Both caller and callee are modified so that both the whole array and the index into it are passed.

```
      SUBROUTINE CALLER(W,NX,NY,NT)
      INTEGER NX, NY, NT
      REAL W(NX,NY,NT)
CMF$  LAYOUT W(:NEWS,:NEWS,:SERIAL)
      CALL ZERO_2D_SLICE(W(1,1,8),NX,NY)
      RETURN
      END

      SUBROUTINE ZERO_2D_SLICE(W,NX,NY)
      INTEGER NX, NY, I,J
      REAL W(NX,NY)
      DO I=1,NX
         DO J=1,NY
            W(I,J) = 0
         ENDDO
      ENDDO
      RETURN
      END
```

⇒

```
      SUBROUTINE CALLER(W,NX,NY,NT)
      INTEGER NX, NY, NT
      REAL W(NX,NY,NT)
CMF$  LAYOUT W(:NEWS,:NEWS,:SERIAL)
      CALL ZERO_2D_SLICE(W(:,:,8),NX,NY)
      RETURN
      END

      SUBROUTINE ZERO_2D_SLICE(W,NX,NY)
      INTEGER NX, NY, I,J
      REAL W(NX,NY)
CMF$  LAYOUT W(:NEWS,:NEWS)
      W = 0
      RETURN
      END
```

Figure 8: CMAX recognizes that a two-dimensional slice of a three-dimensional array is being passed to the subroutine. By recognizing this idiom, CMAX removes the constraint of sequence association. Note that user has tuned the F77 input by inserting an explicit CMF layout directive. Because the dimension to be removed is serial rather than disributed across processors, no interprocessor communication will be needed when the output code is executed.

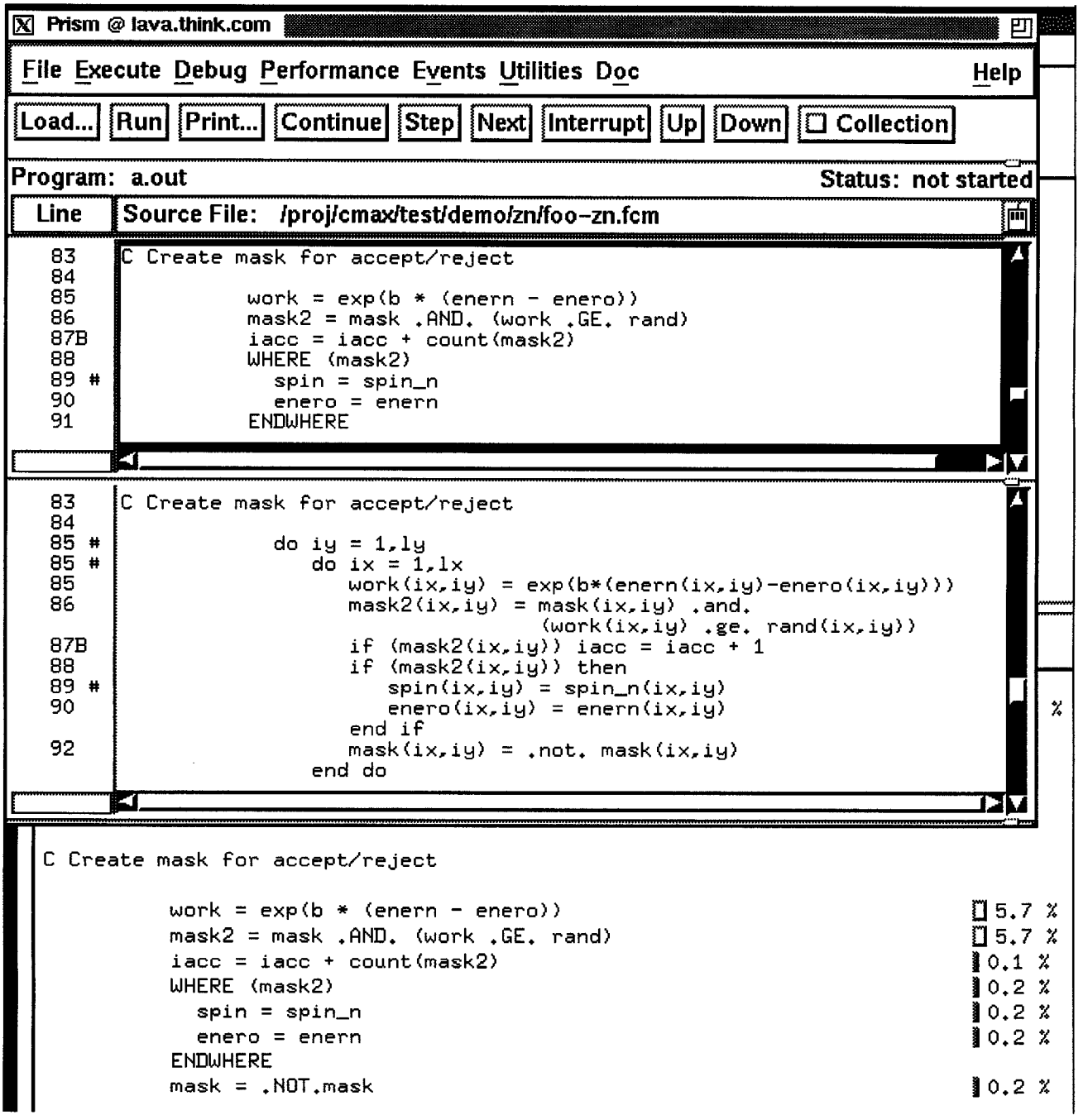

Figure 9: Prism screen when working on a CMAX-translated application. Note that there is a breakpoint set at line 87 which is shown with a "B" in both source panes. Also, the user has clicked on line 89 of the CMF source to find out which F77 lines gave rise to it. In response, Prism displays a "#" next to three F77 source lines.

file. The first part of this file details how lines from the F77 and CMF code map into each other. This includes line number mapping information, original names for renamed cloned subprograms, original names for variables such as arrays that are promoted versions of scalar variables, and so on. Using the line mapping table, Prism allows the user to point at a line in one code (either the F77 or CMF) and then can show what lines correspond to it in the other code. Note that this is not a one-to-one mapping. An array assignment may correspond to several `DO` statements and an assignment statement in F77. Similarly, when a loop body contains several assignment statements, the F77 `DO` statement may give rise to several array assignment statements.

With the line mapping information, the Prism capabilities work on either of the two source code windows. For example, a user can ask Prism to set breakpoints and create other events by pointing at lines in either the F77 or F90 window. The call stack can be viewed in terms of either source code. If a user scrolls through one source code, the other pane of source code can automatically scroll and display the corresponding code. Finally, all of Prism's performance profiling tools also work on either window. For example, a histogram of the time spent at each line of a program can be displayed for either version of the source code. Figure 9 shows how Prism displays a source window with a pane for each of the two source codes, as well as an underlying window at the bottom showing profiling information presented in terms of the original F77.

The second part of the auxiliary file contains CMAX-generated efficiency notes. By pointing at lines of the Fortran code or individual variables, the user can display these. Statement-by-statement efficiency notes explain what happened to each statement (e.g., "Turned into parallel assignment", or "DO statement vectorized away") or just why it did not vectorize. Notes on arrays explain how the array is allocated (on the control processor or distributed across the PEs) and why (e.g., "Not distributed because there was no vector usage").

5 The Porting Process

In a typical port, a CMAX user takes a working F77 code that has at its heart a parallel algorithm expressed in scalable Fortran. The user passes the code through CMAX and examines the CMAX-generated listing file to determine which arrays did not get distributed to PE memories and which loops and statements did not parallelize. The user modifies the F77 code by replacing non-scalable constructs with scalable constructs and by adding directives where necessary to more precisely direct what transformations CMAX should perform. The modified code can be executed and its performance profiled, and the user can iterate again to modify the F77 code to produce a better translation. If a portion of the program is inherently serial, perhaps a part of program initialization or data setup, the user can either leave it as serial or rewrite it, depending on how much it affects overall application performance.

When the goal of translating an application is a CMF program which will be maintained instead of the original F77, a highly interactive tool would be sufficient. However, when the goal of translation is a new F77 program, then conversion needs to occur repeatedly as part of the compilation process, and only a command-line interface callable from `make` or a similar tool is appropriate. Therefore, we chose to first provide a command-line interface to CMAX; a motif-based windowing interface is under development by APR. For debugging and performance tuning on the Connection Machine, CMAX users can use the window-based Prism programming environment (see Section 4).

6 Preliminary Results

CMAX recently went into beta release, and therefore only preliminary results are available.

The Livermore Fortran Kernels provide a simple measure of CMAX's vectorization capabilties as well as an example of how a program can dependend on a linear memory model [14]. Once the `COMMON` and `EQUIVALENCE` problems metioned in 3.2 were worked around, CMAX vectorized 14 of the 24 Livermore Fortran Kernels, including Kernel 11 (a `CMF_SCAN_ADD`) and Kernel 21 (a `MATMUL`). Kernel 24 is a `MINLOC` coded in a way that CMAX does not recognize; a small change to the code would make recognition possible.

Among the more notable real applications which have been or are currently being ported with CMAX are:

FLO67 This is a computational aerodynamics code developed by Antony Jameson of the Department of Aerospace Engineering, Princeton University. The program simulates three-dimensional airflow past a swept wing, and was originally implemented in approximately 5,000 lines of Fortran 77 code. As a proof-of-concept for the idea of scalable programming, the program was rewritten at TMC in scalable F77. The scalable F77 performs within 10% of the original code on serial machines (Sun 4/490 and IBM RS/6000), and the CMAX-translation of this scalable code into CMF performs within 10% of a hand-ported CMF version on a CM-200[20].

ARPS This atmospheric model from Kelvin Droegemeir at the Center for Analysis and Prediction of Storms, University of Oklahoma, contains over 50,000 lines of modern, well written F77 [9]. It was implemented by programmers who had a mental model of a parallel machine and avoided making assumptions about the underlying memory model. It took four days to port version 3.1 of ARPS from the RS/6000 to a Connection Machine system, at which

point the code ran. A partially tuned version has a kernel update time on a 32 processor CM-5 of 4 seconds, compared to to 90 seconds on the RS/6000. About 500 lines were changed or added in the initial port, most of which involved manually specifying the distribution of arrays. The CMAX user commented that 99% of the work was done by CMAX, and most of the user's time was devoted to iteratively running the application through CMAX and making improvements to the F77 based on the efficiency notes. Further performance tuning of the working code is in progress.

X-PLOR This software package for structure determination of biological macromolecules was written by Axel T. Brunger of Yale University [5]. The code consists of 70,000 lines of F77 that have been optimized for vector machines, and it can perform a wide range of the computations needed for x-ray crystallography and nuclear magnetic resonance studies. This code is only slightly larger than ARPS, but because of its age it presented a much greater challenge to CMAX. After 3 months of effort, the entire application has successfully passed through CMAX, and testing is in progress.

CYCLONE This data assimilation package from Andrew Bennet and Carl Hagelberg at the College of Oceanography, Oregon State University, iteratively produces an estimate of the initial state of the atmosphere based on meteorological observations, then integrates forward to forcast tropical cyclones [3]. It contains about 10,000 lines, and about 100 had to be changed in the initial port, and it is estimated that another 2,000 will have to be changed in order to get good parallel performance. The CMAX user commented that CMAX does 75% of the work involved in porting, leaving 25% to do by hand. The 25% is spent modifying the Fortran 77 input code in ways that do not hurt scalar performance, but are necessary for good CMAX performance.

7 Future Work

CMAX currently accepts standard Fortran 77 as input, along with directives to control CMAX operation. We are in the process of extending CMAX to accept a more general input language: CMF. Thus, the input can contain mixed DO loops and array syntax, and the output of CMAX can be fed back into CMAX iteratively.

CMF is expected to converge with HPF (High Performance Fortran), an emerging industry standard, so the CMAX's input and output languages will move toward HPF as well.

We also expect to add more transformations to assist in converting programs which rely on storage and sequence association. The particular transformations we add will be based on feedback from our users.

8 Related Work

There is a large body of research work on compilers that focus on restructuring loops. Good places to start include [1], [21], and [16]. In the commercial domain, KAP [11] and VAST [15] are the main programs of interest.

9 Acknowledgments

We would like to thank the developers at APR, in particular Shaul Sweed, Dennis Goodrow, and Gene Wagenbreth, and fellow employees at Thinking Machines involved with the CMAX effort, including Jonas Berlin, Gyan Bhanot, Rob Jones, Kirk Jordan, Rich Loft, Janet Marantz, Niraj Srivastava, Rich Title, and Yasunari Tosa. We thank Clifford Lasser for providing crazy ideas and convincing arguments.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] David Bacon. Performance-portable compilation for parallel architectures. Talk at Thinking Machines Corporation, August 1992.

[3] A. F. Bennett, L. M. Leslie, C. R. Hagelberg, and P.E. Powers. Tropical cyclone prediction using a barotropic model initialized by a generalized inverse method. *Monthly Weather Review*, 1993. To appear. Submitted July 1992.

[4] R. N. Braswell and M. S. Keech. An evaluation of vector Fortran 200 generated by Cyber 205 and ETA-10 precompilation tools. In *Proceedings of Supercomputing '88*, pages 106–113, 1988.

[5] Axel T. Brunger. *X-PLOR Version 3.0. A System for Crystallography and NMR.* Yale University, 1992.

[6] CONVEX Computer Corporation, Richardson, Texas. *CONVEX Application Compiler User's Guide*, first edition, April 1991. Document Order No. DSW-401.

[7] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Languages*, 1992.

[8] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[9] Kelvin Droegemeier, Ming Xue, Paula Reid, Joseph Bradley III, and Robert Lindsay. Development of the CAPS advanced regional prediction system (ARPS): An adaptive, massively parallel, multi-scale prediction model. In *Proceedings of the 9th Conference on Numerical Weather Prediction.* American Meteorological Society, October 1991.

[10] High Performance Fortran Forum, c/o Theresa Chatman, CITI/CRPC, Box 1892, Rice University, Houston, TX 77251. *High Performance Fortran Language Specification, Version 1.0 Draft*, January 1993. Available by

anonymous FTP from titan.cs.rice.edu in the directory public/HPFF/draft.

[11] Kuck and Associates, Urbana-Champaign, IL. *KAP Users Guide*, 1988.

[12] John Levesque and Richard Friedman. The state of the art in automatic parallelization. In *Proceedings of Supercomputing Europe*, February 1993.

[13] Glenn Luecke, James Coyle, Waqar Haque, James Hoekstra, Howard Jespersen, and Robert Schmidt. A comparitive study of KAP and VAST-2: two automatic vector preprocessors with Fortran 8x output. *Supercomputer (Netherlands)*, 5(6):15–25, November 1988.

[14] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, California, December 1986.

[15] Pacific Sierra Research Corporation, Los Angeles, CA. *VAST-2 Users Guide*, 1986.

[16] Constantine D. Polychronopoulos. *Parallel programming and compilers*. Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[17] Gary W. Sabot. A compiler for a massively parallel distributed memory MIMD computer. In H. J. Siegel, editor, *Frontiers of Massively Parallel Computation*, pages 12–20. IEEE Computer Society, October 1992.

[18] Gary W. Sabot. Optimized CM Fortran Compiler for the Connection Machine Computer. In *Proceedings of Hawaii International Conference on System Sciences 25*, pages 161–172. IEEE Computer Society, 1992.

[19] Thinking Machines Corporation. *Prism User's Guide*, version 1.0 edition, December 1991.

[20] Skef Wholey, Cliff Lasser, and Gyan Bhanot. Flo67: A case study in scalable programming. *International Journal of Supercomputer Applications*, 6(4), Winter 1992. To appear.

[21] Michael Wolfe. *Optimizing Supercompiler for Supercomputers*. The MIT Press, Cambridge, Massachusetts, 1989.