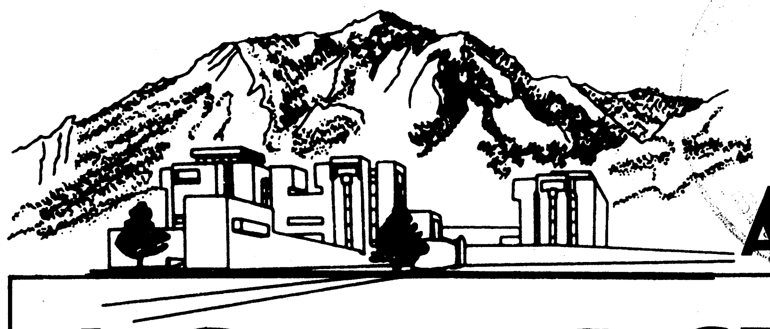
SPECIAL EDITION

The Record



SCD Announces A Second CRAY-1A

On December 20, 1982 a small group of staff members clustered on the Damon Room balcony to observe the delivery of NCAR's second CRAY-1 computer. Those in the group held their breath as the crane tipped when lifting the five-ton computer from the moving van. According to one onlooker, the rear of the crane rose a good two feet off the ground as it hoisted the CRAY-1's central processing unit. But the crane held its ground and safely lowered the computer into the air-intake tunnel of the SCD.

"Due to the coordination efforts of Physical Facilities Services, it was one of the smoothest moves we've ever had with a computer," said Robert Niffenegger (SCD manager of Operations). "Without their help, this move would not have been nearly as smooth or timely. The only problem was that the side road next to the High Altitude Observatory (HAO) addition was barely wide enough to accommodate the moving van. We spent some time filling in and grading the road," Bob said, "and as the van made the turn we all took some deep breaths, and hoped it wouldn't topple over into HAO's sunken telescope area." Thanks to the capable van driver, all went well, and there were about five or six inches of clearance. Before situating the second CRAY-1, several new holes had to be cut in the floor for the mechanical and power connections, and 24 extra steel pedestals were added beneath the floor for support. The new CRAY-1 now rests securely in the first basement of the Mesa Laboratory next to the old one.

The CRAY-1 is actually a used one--it belonged formerly to the Lawrence Livermore National Laboratory in California. However, it has been completely refurbished at CRAY headquarters in Chippewa Falls, Wisconsin, and has new refrigeration-condensor and motor-generator units. (Although plugged in directly to the Public Service power the CRAY-1 uses additional motor generators to convert the 60 Hz of Public Service power to the required 400 Hz. SCD will eventually have an uninterruptable power-supply system provided by a new storage battery unit. The batteries will supply 15 minutes of additional power to the IBM 4341 and the communications systems for a controlled shutdown should the Public Service power stop. The extra power provides enough time to alert users to log off gracefully and save their work before the system goes down.)

Except for its more subtle colors--sun gold and caramel--the new CRAY-1 is identical to the old one, with the same disk and memory capacity and speed. "We purchased the second CRAY-1 because we simply couldn't satisfy the demands of the users with the one we had," said SCD director Walter Macintyre. "We were turning people away, and some users were having difficulty completing their programs--we had reached an impossible load. This is an interim, short-term solution to our load problem," continued Walter. "We are asking for a next generation machine to be supplied sometime in the next two or three years. This future machine will have ten times the capacity of a CRAY-1."

SCD has had to do a lot of rearranging to accommodate the second CRAY-1 and its 16 disk drives, and will be busy for the next several months with rewiring and software production. The second CRAY-1 should be operational by the end of May. The National Science Foundation officially approved the additional CRAY-1 purchase shortly after Christmas, insuring a happier New Year for SCD and its computer users.

(Reprinted in part from Staff Notes, Vol. 18, No. 2).

VECTOR FORTRAN for NUMERICAL PROBLEMS ON CRAY-1

W. P. Petersen

31 March 1982

CONTENTS

1. INTRODUCTION 1

1.1 Segmentation and unrolling loops 1

1.2 Identical and independent operations 1

2. CRAY-1 VECTOR HARDWARE 2

2.1 Chaining and functional unit overlap 4

2.2 Memory access and timing 4

2.3 Reduction operations 5

3. PROGRAMMING THE LOOPS 6

3.1 Increments, dimensions - how to index the arrays 7

3.2 Conditional statements - IF and GOTO 10

3.3 Compiler directives 11

4. CHOOSING YOUR ALGORITHM 12

4.1 Outer products - a linear digital filter 12

4.2 Inner products, a symmetric Toeplitz matrix 13

4.3 Floyd's algorithm - removing the IF statements 13

4.4 An FFT - maximizing the inner vector length 13

4.5 Getting around dependencies 19

4.5.1 Gauss-Seidel Relaxation 19

4.5.2 A red/black ordering 20

4.5.3 Tridiagonal systems 21

4.6 Sparse matrices - nonlinear indexing 22

Bell Laboratories

VECTOR FORTRAN for NUMERICAL PROBLEMS ON CRAY-1

W. P. Petersen

31 March 1982

1. INTRODUCTION

Computations in vector mode on CRAY-1 can be an order of magnitude faster than in scalar mode. The present paper deals with the basics: vector hardware operations and how the CFT (Cray FORTRAN) compiler makes use of them. A little understanding of the vector hardware is useful because some operations on CRAY-1 are more efficient than others. Despite the hardware motivation, a modular 'each block does a vector computation' method usually works very well on any large computer.

1.1 Segmentation and unrolling loops

To CRAY-1, vectors are regularly spaced arrays of data that can be processed by segments. Regularly spaced data means that each element is the same number of memory locations from its predecessor. For example, the elements {A(N−1), A(N−3), A(N−5), A(N−7),... } are regularly spaced, while {A(1), A(2), A(4), A(8),... } are not.

In Cray FORTRAN the principal engines of vector operations are DO-loops. If there are n repetitions of the loop, executing as many as 64 at a time[*] gives

$$n = rsl + 64q$$

where $rsl \leqslant 64$ is the number in the residual segment, which is processed first, and q the number of additional segments of length 64. Machine instructions generated by CFT for vector DO-loops calculate q and rsl to 'unroll' the loop into segments of length $\leqslant 64$. A vector length register VL (ref. 1) is set to the number per segment. All unrolling of loops is transparent to the user, with CFT doing all segmenting and appropriate addressing.

1.2 Identical and independent operations

In writing vectorized codes, it is important to understand that vector elements really must be independent of one another, but treated identically. Vectorized conditional calculations have superfluous operations. Consider the summation† of N elements in an array A which skips the addition of null values, as shown in Figure 1.

[*] Vector registers having 64 words each are discussed in Section 2.

† See sections 2.4 and 4.2.

```
      SUM = 0
      DO 1 I = 1,N
         IF(A(I).EQ.0)GOTO 1
         SUM = SUM + A(I)
    1 CONTINUE
```

Figure 1
Non-Vector Summation

Plucking out the IF statement allows CFT to compile vector hardware instructions. Addition of zero elements may be superfluous, but do it anyway. The idea is that each A(I) must be treated exactly like every other. Conditional statements IF and computed GOTO imply that some data are very different than others and treated accordingly. Branching, which includes IF, GOTO and CALL statements, inhibits the use of the CRAY-1 vector hardware.

Furthermore, CFT considers an array to be a vector only if it is clear that no element of that array depends upon the previous computation of another. For example, in Figure 2, the natural order of I = 2, 3, 4, ... requires that A(I−1) has been replaced by A(I−2) for A(I) to be properly set.

```
      DO 1 I = 2,N
         A(I) = A(I-1)
    2 CONTINUE
```

Figure 2
Non-Vector Dependency Case

Thus, the A(I)'s in Figure 2 must be set one at a time rather than by segments.

Indexing arrays so that CFT compiles vector machine instructions is generally straightforward, but there are subtleties. Section 3 of this document and the CFT reference manual (ref. 2, part 3, section 2) deal with indexing in some detail.

2. CRAY-1 VECTOR HARDWARE

Only about one fourth of the CRAY-1 machine instructions use *vector registers.* The following notation will be used

$$V1 \leftarrow S1 + V0$$

to describe machine instructions. Here, the content of scalar register S1 is added element by element to the contents of vector register V0 and the results stored in vector register V1.

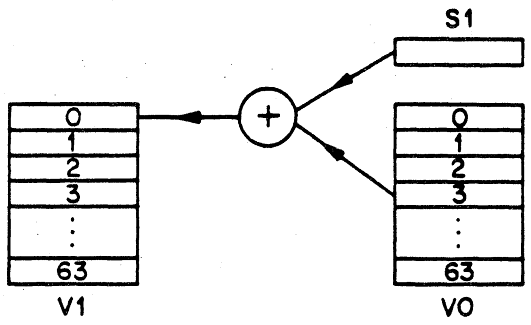


Figure 3
Register S1 Added to V0

Figure 3 is a pictorial representation of this instruction. Paths to each register are represented by arrows. The end of each path is a pointer indicating only one element at a time. Once this instruction is issued and addition begins, the pointer in V0 is incremented each clock period (12.5×10^{-9} seconds), until the the operands are exhausted. A similar pointer in result V1 is incremented when results begin to emerge from the adder, three clock periods later in this case.

The integer adder which may contain three separate pairs of operands concurrently in distinct stages of processing, is called a *pipelined functional unit*. It has some analogy to a short piece of pipe into which marbles are being pushed; until the pipe is full, no marbles emerge from the other end. Even though the first result does not emerge from the pipe until several clock periods later, successive results arrive one clock period apart. One 64-bit integer addition takes three clock cycles, but pushing a segment of 64 additions through only takes 67. Effectively, this is only slightly more than one cycle per addition - a factor of three faster than one at a time. Longer pipelines show even greater improvements - typically six or seven fold for floating point operations. Further, with *chaining* and *overlap*, described in the next section, several operations may run concurrently.

Machine instructions generated by CFT to execute a vectorized DO-loop control the segmentation (sec. 1.1), and involve some of eight vector registers V0, V1, ..., V7 of 64 words each. Scalar registers S0, S1, ..., S7 may also be used as operands in some vector operations, as in Figure 3 for example. Vector merging, that is, selecting vector elements word-by-word is implemented by a correspondence between the 64 bits of the S and VM registers and the 64 words of the V registers (see section 3.2). Operations may run concurrently if certain independence criteria are satisfied. In particular, each of seven functional units may run independently to perform the operations shown in TABLE 1.

VECTOR FUNCTIONAL UNITS		
unit	operations	purpose
memory	load	load register from memory
	store	store to memory from register
*	*F	f.p. (truncated) multiply
	*R	f.p. (rounded) multiply
	*I	f.p. (iterative) multiply[§]
/	/H	reciprocal approximation[§]
+	+F	f.p. add
	−F	f.p. subtract
logical	&	logical .and.
	!	logical .or.
	\	exclusive .or.
	VM	form vector mask
	merge	vector merge
+	+	integer add
integer	−	integer subtract
shift	>	right shift
	<	left shift

TABLE 1

Each vector functional unit is independent and results from one unit may be fed into another as operands - hence, operations may *chain* together.

NOTICE that there are vector hardware provisions only for processing of data, not for creation. In particular, the vector hardware will not create an array of integers. This means that arrays may not be generated directly from DO-loop index variables in vector mode: A(I) = FLOAT(I), for example.

§ Division uses a 30 bit reciprocal approximation and one Newton iteration, (ref. 3, pp. 5-53)

2.1 Chaining and functional unit overlap

If successive operations use different functional units, they may run concurrently. Division using a Newton iteration is an interesting example; we want a_i/b_i, and use the operations from TABLE 1 in Figure 4.

V0	←	B	b_i's
V1	←	/HV0	half precision $1/b_i$
V2	←	V0*IV1	2.0 - b_i/Hb_i
V3	←	V1*RV2	full precision $1/b_i$
V4	←	A	a_i's
V5	←	V4*RV3	a_i/b_i's

Figure 4
Division by Reciprocal Approximation

In a sense, there are only four separate operations in division, with the first pair linking together to form a chain, and instructions four and five overlapping. Briefly, separation occurs as follows:

1. Memory access is independent of reciprocal approximation; so, as soon as the first b_i arrives in V0 the second instruction is issued. As the b_i's are stored into V0, they are immediately copied by the reciprocal unit and start through that pipe, eventually emerging to be stored into V1. This linking to form a short chain is shown in Figure 5.
2. Although multiplication is independent of reciprocal approximation, the pointer in V0 has moved down before the first result arrives in V1. Thus, V0 is 'busy' as an operand and the third instruction must wait until the first pair are finished.
3. Instruction four uses the same functional unit (multiply) as the third. This unit is busy until the third is finished. Instruction five (load A) may issue after four begins, and will run concurrently. Operations begun by the issue of instructions four and five do not chain together, but 'overlap' and run concurrently.
4. Instruction six must wait until four (running concurrently with five) is finished because it uses the multiply unit again.

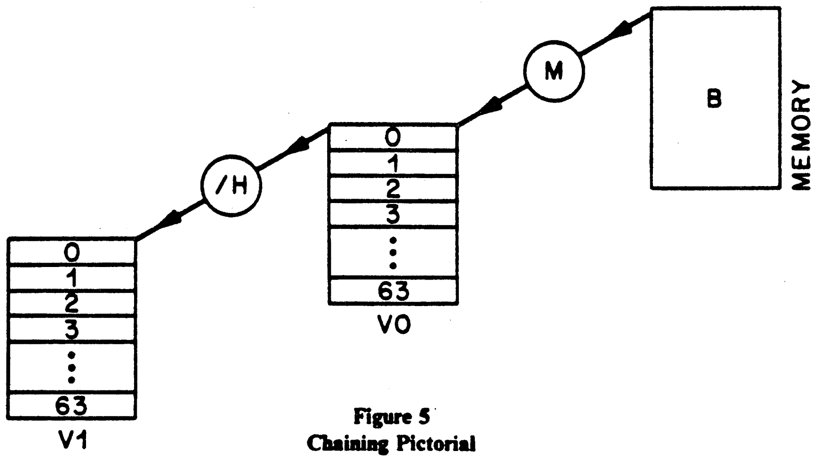


Figure 5
Chaining Pictorial

2.2 Memory access and timing

With some exceptions (division for example), the number of sequences which run concurrently is approximately the number of memory access instructions. Since there is only one port to memory, and in FORTRAN *all symbols are in memory*, this is usually the critical resource. A lower bound on any loop timing (in seconds) is

$$T \geqslant (12.5 \times 10^{-9}) \times \frac{8}{7} \times (\textit{number of memory references}),$$

where 8/7 accounts for pipeline overhead. Indexing - that is, segment counting, array offsets, setting VL, etc. - is independent of vector operations and runs concurrently with the last store into memory from the loop. Indexing by segments adds little to loop overhead, and is transparent to the CFT user. If there are no function calls, the bound is a reasonable timing approximation - roughly within a factor of two.

Memory is also the critical resource in INPUT/OUTPUT requests. Loading vector registers locks up memory. In principle, I/O runs concurrently with computation, but vector loading keeps memory pretty busy. Thus, for a particular user job, overlapping I/O with vector loops is not as helpful as might be expected. To make things more complicated, unless your job is running by itself, the operating system (COS - Cray Operating System) will start up another job while waiting for your I/O request to finish. However, a few things will help.

1. Avoid FORMATted records wherever possible. FORMAT statements are executed in an interpretative manner character-by-character, and are very slow.
2. Use of BUFFER IN and BUFFER OUT to transfer datasets will let you overlap your I/O with your computation, which may help sometimes.
3. Usually, sequential I/O using READ and WRITE statements works best because it's very simple and lets the operating system take care of all the concurrency headaches. Try to use long fixed-length records - somewhere around 500 words. Every data-set transfer involves at least one sector of 512 words, which includes some block and record control words. Also, try to avoid array references having multiple-of-eight increments which cause memory bank-conflicts.

Sequential array elements are stored in sequential *banks* of memory. There are 16 banks, each with a cycle time of four clock periods (see ref. 1, section 6-21). Fetching or storing by multiple-of-eight increments conflicts with this cycle. Memory bank conflicts also break chaining with independent operations. These are important considerations for FFT's and Cyclic Reduction, which are discussed in sections 4.4 and 4.5.3.

2.3 Reduction operations

Operations with a result vector having the same number of elements as the operands are usually more efficient than, say, dot-products — which reduce dimensionality. Consider a DO-loop which computes the product of N elements of array A:

```
      PROD = 1.0
      DO 1 I=1,N
         PROD = PROD*A(I)
    1 CONTINUE
```

Figure 6
Product of Vector Elements

Result PROD is of dimension unity, but A is an N-dimensional operand. Reductions use a curious property of the vector hardware.

To do reductions like this, three steps are necessary. For simplicity, let $N = 64q$, and divide A into q segments - each of length 64. In the first step, the following accumulation happens. Segment one and two are multiplied together, element-by-element, to yield a 64 element result. That result is now multiplied by the third - still a 64 element result, then the fourth, and so on until all q segments are used.

A second step reduces this 64-element partial result by a recursive hardware operation (ref. 1, pp3-14) similar to the following, if V0 contains the 64-element partial result,

$$V1 \leftarrow V1^*RV0$$

where the first element of V1 is set to 1.0. Result register V1 is also an operand. In this case, the pointer in V1 cannot advance until *functional unit time + 2* clock periods (7+2 = 9) later, when the first result is ready. When completed, the last nine elements of V1 are the product of the first element of V0 and every ninth element of V0. Use of identical registers for operands and results is deliberate and *recursive*, producing a useful reduction of the 64 products. An example of this recursive mechanism is shown in reference 1, but is too involved to reproduce here.

Finally, in the last of three steps, *f.u. time + 2* elements in V1 must be pulled out and the reduction completed. This remaining step, which uses the S-registers, can be scheduled to take about 45 clock periods, somewhat less than a vector segment operation on 64 elements. Such a macro is invoked by CFT to do reductions of *sums and products only*. If N is large, two additional steps after the initial accumulation add little extra time since each is executed only once. Compared to the *q* accumulation operations in the first step, the last two become unimportant. However, for moderate-length vectors (less than 100), reductions are less efficient than vector → vector operations. Sections 4.1 and 4.2 give examples which illustrate the point.

3. PROGRAMMING THE LOOPS

The appearance of

```
        VECTOR BLOCK BEGINS AT SEQ. NO.      n, P=   addr
```

at the end of a listing of a compiled subprogram means that CFT has generated a vector DO-loop in a particular block of code. A block is a basic unit of code which is locally optimized by CFT and is demarcated by register usage. Subroutine or user-defined function calls, GOTO statements, and inner DO-loops force blocking. *CFT only vectorizes inner loops*! Since blocking doesn't usually start exactly at the DO statement, to identify the vectorized loops in a compiled listing look for the first inner loop following the sequence number beginning the vector block. This will be the vectorized loop.

Many CFT intrinsic functions, SIN, EXP, SQRT for example, are vector mode (computation done in vector registers) and are used in DO-loops. Others, like ATAN, may be used in vector loops but are not really vector mode. These pseudo-vector functions pass segments of arguments *in-register* (V1, or V1 and V2), but process them element-by-element in scalar registers. Frankly, *pseudo-vector* (see ref. 2, Appendix B) routines exist because nobody has rewritten them. They do permit CFT to vectorize the rest of a loop, however. TABLE 2 summarizes the intrinsic CFT operations and functions which are vector-mode.

Neither DOUBLE PRECISION nor CHARACTER data computations are done in vector registers. The CRAY-1 has no double precision hardware. Data transfer and I/O for double precision does use vector register memory access, however. INTEGER type addition (subtraction) is 64 bit in vector mode, but multiplication is done by floating point hardware. Integer multiplication has only 48 bits of precision.

CFT VECTOR OPERATIONS AND FUNCTIONS BY DATA TYPE				
SINGLE	COMPLEX	INTEGER	DOUBLE	BOOLEAN
*	*	*		.AND.
+ −	+ −	+ −		.OR.
/	/	/		.XOR.
logical		logical		
SQRT				
EXP, **				
COS/SIN	CSIN/CCOS			
ALOG/ALOG10				
ABS	CABS	IABS	DABS	
DIM		IDIM[‡]		
INT		AINT	IDINT	
AMOD				
SIGN		ISIGN	DSIGN	
RANF				
MAX1/AMAX1		MAX0/AMAX0	DMAX1	
MIN1/AMIN1		MIN0/AMIN0	DMIN1	
	CONJG			
IFIX		FLOAT	SNGL	
CMPLX/DBLE	REAL/AIMAG			
SIGN		ISIGN	DSIGN	

TABLE 2

3.1 Increments, dimensions - how to index the arrays

Although many operations on regularly spaced data in memory can be done in vector mode, CFT doesn't recognize them all. Figure 7 shows several examples of array subscripts which CFT recognizes as vector indices. Two concepts are important here: invariants within the loop, and constant increment integers. Loop invariants are quantities unmodified by changes in the DO - variable (I in this case). Constant increment integers (CII's) 'follow the DO - variable' in that each time I is changed, the CII is incremented by a constant amount. In Figure 7, the variables I1, I3, I4, I5, I6, I8, I9, I10, I11, I12, I15, I17, I19, I20, I22, and X3(I13), IA(I14) are invariants. Variables I2, I7, I16, I18, I21, are constant increment integers (CII's).

[‡] Positive difference function DIM gives X = DIM(Y,Z) = Y - Z if Y > Z, X = 0 otherwise.

```
      DO 1 I = IL,IU,IS
C
C    CONSTANT INCREMENT INTEGER - I2
         Y1(I + I1)           = X1(I2)
         I2 = I2 + I3
C    REGULAR STEPS (I5,I8), WITH CII (I7)
         Y2(I4 + I*I5)        = X2(I6 - I7*I8)
         I7 = I7 + I9 + I10
C    REGULAR STEP (I12), INVARIANTS (X3, IA)
         Y3(I11 - I*I12)      = X3(I13)*X4(IA(I14) - I)
C    RECURSIVE SUMMATION OF ARRAY X6
         Y4(I15)              = Y4(I15) + X6(I16)
         I16 = I16 + I17
C    RECURSIVELY FORMING PRODUCT OF X6 ELEMENTS
         Y5(I17)              = Y5(I17)*X6(I18)
         I18 = I18 + I19
C    SETTING ELEMENTS IN ROW OF Y6
         Y6(I20,I21)          = X7(I)
         I21 = I21 + I22
C
    1 CONTINUE
```

Figure 7
Indexing of Vector Loops

We have the following rules.

RULES FOR VECTOR SUBSCRIPTS

1. The subscripts of a vector in an array may appear in only *one* dimension of that array. In Figure 7 only I21 varies in Y6.
2. These subscripts must be expressed as functions of the DO-loop variable or of constant increment integers. CII's are computed in the loop by

   CII = CII ± constant increment.

   The constant increment can be only an invariant integer or a sum of invariant integers. Expressions for CII's cannot contain parentheses, nor any operation but addition (subtraction). In Figure 7, index variables I2, I7, I16, I18, I21 are CII's, and I3, I9+I10, I17, I19, and I22 are the constant increments.
3. Subscripts must be one of the forms

   inv ± I*istep

   or

   inv ± CII*istep

   where *inv* is any invariant expression not containing a function reference (*inv* may be null) and *istep* must be a simple integer invariant. Step *istep* can be of either sign (see Y2(I4 + I*I5) and Y3(I11 - I*I12) in Figure 7).

WARNING

At the time of this writing, if istep = 0, *results may be erroneous. Be careful.*

As a final example, Figure 8 illustrates array references which will not vectorize, even as self-contained loops.

```
          DO 1 I = 1,N
C (1)
             Y1(I,I) = X1(I1 + I*(I2*I3))
C (2)
             Y2(I4)  = X2(IFCN(X) + I)
             I4 = I4 + 2*I5
C (3)
             Y3(I6 + I) = ATAN(X3(I))
             Y3(I7 + I) = FCN(X3(I))
C (4)
             Y4(I)     = X4(I + I*I9)
             Y5(I)     = Y4(I8 + I)
C (5)
             I10 = INDEX(I)
             Y6(I10) = FLOAT(I)
C
        1 CONTINUE
```

Figure 8
No Vectors Here

The reasons CFT will not vectorize these expressions are:

1. The spacing computations in Y1 and X1 are too complicated for CFT: a diagonal in Y1 (two dimensions - see rule 1), and the spacing (I2*I3) is not a simple invariant integer variable without parenthesis. Both these subscript expressions are easily changed:
   a. Set LD1 = LD + 1, where LD is the leading dimension of Y1(LD,.), then Y1(I*LD1 - LD) satisfies rule 3, making LD1 a loop invariant. Of course, LD1 must be calculated outside the loop.
   b. Call I23 = I2*I3, then X1(I1 + I*I23) satisfies rule 3.
2. The increment 2*I5 is not a sum of simple invariants (see rule 2), and the subscript expression for X2 contains a function reference:
   a. Changing 2*I5 to I5 + I5 fixes up the CII calculation of I4.
   b. Pulling the expression IFNX = IFCN(X) out of the loop permits X2(IFNX + I), with a valid subscript.
3. Reference to array Y3 represents a subtle dependency case if I7 ⩾ I6 and the storage overlaps. At compile time, this seems to be a dependency which would yield incorrect results; hence CFT will not attempt to use vector hardware for this calculation. See sections 1.2, 3.4, and 4.5. The reference to ATAN is really a pseudo-vector function (see TABLE 2). Unless FCN is declared by a VFUNCTION directive (see section 3.5), CFT does not recognize it as a vector function.
   a. Insertion of an IVDEP directive prior to the loop will permit the vector-mode access to Y3, at the user's risk if I7 ⩾ I6.
   b. Although ATAN is a pseudo-vector function, it still may be used in a vector loop.
   c. A CAL-coded (Cray Assembly Language) FCN with call-by-value linkage would have to be provided (see section 3.5)
4. References Y4 seem to CFT to be a dependency because if I8 is negative, Y5 requires an element Y4(I8 +I) which is computed at the same step. See sections 1.2, 3.4 and 4.5. In X4 the subscript is not of the form given in rule 3.

5. Array Y6 is referenced in a non-linear way indirectly through the set of pointers INDEX (see section 4.5). FLOAT(I) is a direct calculation on the DO - variable I (see section 2.0).

3.2 Conditional statements - IF and GOTO

At this time, CFT (version 1.09) permits no conditional statements IF or GOTO in vector inner loops. Because CRAY-1 is a simple machine, with relatively few instructions, superfluous calculations occur in processing conditionals. For example, if for each I we want

$$A(I) = \exp_1 \qquad \text{if } cond(I) = .false.$$
$$A(I) = \exp_2 \qquad \text{if } cond(I) = .true.$$

then both $\exp_1$ and $\exp_2$ are evaluated and the resulting vector A is selected from $\exp_1$ or $\exp_2$ on the basis of *cond* = *false/true* respectively. This procedure contrasts sharply with sequential machines where $\exp_2$ may not be evaluated unless *cond* is determined to be *true*. The hardware operations can be sketched (ref. 1, pp4-51 and pp4-71):

V0	←	*cond*	set V0 to logical conditions
VM	←	V0,M	set vector mask if *true*
V1	←	$\exp_1$	evaluate $\exp_1$
V2	←	$\exp_2$	evaluate $\exp_2$
V3	←	V2!V1&VM	select result

with the first two operations chaining together. Figures 9 and 10 illustrate the operations of setting the vector mask and merging. An element of V3 is selected from V2 if the vector mask (VM) bit corresponding to that element is set, from V1 otherwise.

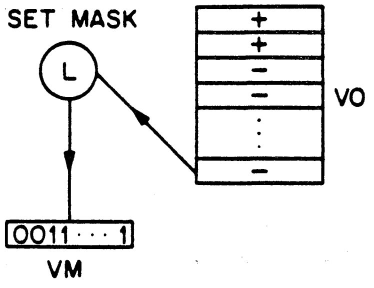


Figure 9
Set Mask

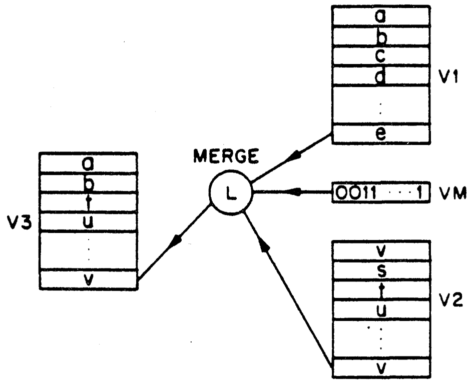


Figure 10
Merge Result

Macros which generate these instructions are invoked by MAX, MIN (see TABLE 2) and CVMG (see Figure 11) functions. The complex vector merge functions summarized in Figure 11 are not portable. Future versions of CFT will probably generate the same instruction sequences for IF statements which perform the same operations.

```
      DO 1 I = 1,N
C CVMGP selects X1 if TEST > 0 , X2 otherwise
      Y1(I) = CVMGP(X1(I),X2(I),TEST(I))
C CVMGM selects X1 if TEST < 0, X2 otherwise
      Y2(I) = CVMGM(X1(I),X2(I),TEST(I))
C CVMGZ likes X1 if TEST = 0, X2 otherwise
      Y3(I) = CVMGZ(X1(I),X2(I),TEST(I))
C CVMGN prefers X1 if TEST ≠ 0, X2 otherwise
      Y4(I) = CVMGN(X1(I),X2(I),TEST(I))
C CVMGT elects X1 if LTEST is truthful
      Y5(I) = CVMGT(X1(I),X2(I),TEST(I))
C selection for each CVMG is element by element
    1 CONTINUE
```

Figure 11
CVMG Functions

Despite their non-portability, these functions are attractive because their arguments can be any expression, vector or scalar.

Many conditionals can be expressed by AMAX1, AMIN1 etc. functions which use the mask/merge hardware instructions. Where applicable, the MAX/MIN conditionals are preferable to the CVMG functions because they are ANSII standard FORTRAN. An illustration of this can be seen in Floyd's algorithm in section 4.3.

3.3 Compiler directives

Figure 2 and Figure 8 have examples showing dependencies amoung array elements inhibiting the generation of vector instructions. A progammer who is wiser than the compiler may wish vector instructions to be generated anyway. Several 'comment line' directives exist for CFT to control vector compilation. Beginning with a 'C' in column 1, these directives are treated as comment lines by other Fortran compilers.

CDIR$ IVDEP tells CFT to ignore the apparent vector dependency in the next inner DO-loop.

CDIR$ NOVECTOR = n turns off generation of vector code for all loops with iteration counts less than n+1 if n is known at compile time. NOVECTOR is a switch and applies to all successive code compiled until this switch is toggled. If n is not specified, all vector code is suppressed. A variable n is not allowed.

CDIR$ VECTOR toggles NOVECTOR and NORECURRENCE (see below) directives, and causes resumption of vector code generation.

CDIR$ VFUNCTION *fname* instructs CFT that there exists a 'call-by-value' external vector function *fname* written in CAL with conventional linkage. See ref. 2, pp. (3)1-18 and Appendix F. Function *fname* is then used like any intrinsic (e.g. SIN, EXP).

CDIR$ NORECURRENCE = n switches off vectorized reduction operations for DO-loops with iteration counts less than n+1 if n is known at compile time. NORECURRENCE is a switch and applies to all successive compilation until toggled by a VECTOR directive.

CDIR$ NEXTSCALAR switches off vectorization for the next DO-loop only.

Compiler directives are not enabled unless the ON = E option is used in the CFT control statement (ref. 2, p. (3)1-1). This option is default at Murray Hill Computer Center. An example of IVDEP usage is illustrated in section 4.5.2.

4. CHOOSING YOUR ALGORITHM

In this section are a number of examples to illustrate the material discussed earlier. Unfortunately, the examples are microscopic and do not indicate global strategies for attacking large problems. A modular approach to larger calculations still demands attention to detail at the subroutine level, however. Some very useful timing information at a global level can be obtained using the FLOWTRACE option of CFT (ref. 2, pp. (3)1-5 and pp. (3)1-20,23). The total time spent in each subroutine, the percentage of the total time, and some overhead information is computed and printed by FLOWTRACE. The subroutines presented here frequently represent important kernels:

1. Section 4.1: Outer products versus inner products - a linear digital filter.
2. Section 4.2: Inner product formulations, discussed in 2.3 - a Toeplitz matrix solver.
3. Section 4.3: Conditional statements discussed in section 3.2, with an example of Floyd's minimum path algorithm.
4. Section 4.4: Maximizing the inner vector length to make the most efficient use of the vector capabilities - a sample Fast Fourier Transform.
5. Section 4.5: Eliminating vector dependencies, noted in Figure 2, and examples 3 and 4 of Figure 8.
    a. A Gauss-Seidel relaxation step.
    b. A Red-Black relaxation.
    c. A multiple tridiagonal solver.
6. Section 4.6: Indirect addressing noted in example 5 of Figure 8 - the problem with sparse matrices.

4.1 Outer products - a linear digital filter

A simple non-recursive digital filter is a convolution of time series data (d) with a small number of filter coefficients (a). Seismic data, for example, typically has about 2000 points per trace and requires 50 to 100 filter coefficients. Output f is

$$f_j = \sum_{k=1}^{m} a_k d_{j+k-1}$$

where $1 \leqslant j \leqslant n-m$, $1 \leqslant j+k \leqslant n$. Data d has $n+1$ points while the output f, containing less information, has $n-m+1$ points. There are two ways to go here: (1) for each point j sum the k elements, or (2) for each new k add all the j's from all the previous k's. Figure 12 and Figure 13 represent the dot-product method and outer-product method respectively.

```
  DO 1 J = 1,N-M+1
     F(J) = 0
  DO 1 K = 1,M
     F(J) = F(J) + A(K)*D(J+K-1)
1 CONTINUE
```

Figure 12
Dot-Product Method

```
  DO 1 J = 1,N-M+1
1    F(J) = 0
  DO 2 K = 1,M
  DO 2 J = 1,N-M+1
     F(J) = F(J) + A(K)*D(J+K-1)
2 CONTINUE
```

Figure 13
Outer-Product Method

In the above example, or that of matrix multiplication (ref.4), or Gaussian elimination (ref. 5) outer product procedures are generally more efficient. On the Murray Hill CRAY-1 (for N =2000, M =100), Figure 13 runs in half the time of Figure 12, precisely for the reasons given in section 2.3. Both examples compile to vector object code.

If the number of elements to be processed by an outer product method gives a short vector length while a dot product procedure gives a much longer one, the differences may be slight or the converse of the above argument may be true. Clearly, a dot-product is not efficiently done as an outer-product of length one. Some procedures are dominantly reduction operations as the next example shows.

4.2 Inner products, a symmetric Toeplitz matrix

An $n \times n$ symmetric Toeplitz matrix T with n independent elements has $T_{i,j} = t_{|i-j|+1}$, for $1 \leqslant i,j \leqslant n$. Each element of linear array t is used to form a pair of symmetric diagonals. The following solution (Figure 14) of the linear system $Tx = y$ is due to Levinson (ref. 6) and requires an operation count $\propto n^2$. The method is similar to conjugate gradient and uses a work space (C) of dimension $2n$.

In Figure 14 the bulk of computation is in dot-product operations computing local variables C1N, C1D and XN, XD. In subroutines CSOLV and XSOLV, re-labeling variables C(1) as C1 and −X(M +2) as SX eliminates apparent dependencies. The vector length of each loop increases each iteration $1 \leqslant m \leqslant n$. Segment overhead (see section 2.2) is constant, and for short segments becomes appreciable. In this example there is no know way to avoid functional unit overhead for short segments. In the example of an FFT in section 4.4, an increasing or decreasing vector length can be dealt with effectively, but not in the present case. Nevertheless, this $O(n^2)$ algorithm is very efficient because every loop is in vector mode. It inverts a 256 dimensional system in 17 milliseconds, but requires 77 milliseconds in scalar mode (i.e. OFF =V option of CFT, see ref. 2, p. (3)1-1).

4.3 Floyd's algorithm - removing the IF statements

The following shortest path through a network algorithm is due to Floyd (ref. 7). In Figure 15, the inner-loop IF and CVMG statements are 'commented out' to give alternate calculations of $m_{j,k}$ shown by the 'C --- ' lines. Initially, $m_{j,k}$ is the length of a direct link from point j of a network to point k. If no direct link exists, $m_{j,k}$ is assumed to be initially $10^{10} = \infty$. On exit, $m_{j,k}$ contains the length of the shortest path from j to k. In Figure 15 the array M is of positive integers, but this is not essential - substituting AMIN1 for MIN0 deals with positive reals. It turns out that the introduction of the integer variable T is necessary. If M(J,I) were used instead of T, the compiler would flag the inner loop as a dependency situation - M(J,I) might be overwritten when K = I in the J-th column - and the new M(J,I) would be used for K > I. This will not happen as a result of the semantics, not the syntax; namely, the positive array element M(J,I) will not be overwritten by M(J,I) + M(I,J) = 2•M(J,I). The CVMGM function is equivalent to MIN0, and vectorizes. Use of the IF-test replacement in lieu of CVMGM or MIN0 for N > 10 gives a non-vectorizing subroutine taking 5 times longer.

4.4 An FFT - maximizing the inner vector length

In their original paper, Cooley and Tukey (ref. 8) noted that their decimated procedure for computing discrete Fourier transform is a parallel algorithm. The following variant of a power-of-two transform on complex data is due to Temperton (ref. 9). It is ordered, requiring no bit-reversal because of decimation. The price paid for ordering is a work space the same length as the input array. A pipelined computer permits only a bit-reversed algorithm for in-place computation. A careful examination of the following signal flow diagram (Figure 17) shows that only simultaneous, distinct from asynchronous, storage of output at each of the $\log_2 n$ steps would permit an ordered out algorithm to be done in place. A bit-reversed eight point output vector has elements numbered 0, 4, 2, 6, 1, 5, 3, 7 - easily seen by writing out the three bit numbers.

```
      SUBROUTINE TOEPLITZ(N,X,Y,C,T)
      REAL C(2*N),T(N),X(N),Y(N)
C
      X(1) = Y(1)/T(1)
      C(1) = T(2)/T(1)
C
C   NEXT N-1 RECURSIVE STEPS INCREASE THE VECTOR LENGTH
C
      DO 1 M = 0,N-2
         IF((M.AND.1).EQ.1)THEN
            CALL CSOLV(M,C(N+1),C(1),T)
            CALL XSOLV(M,X,Y,C(N+1),T)
         ELSE
            CALL CSOLV(M,C(1),C(N+1),T)
            CALL XSOLV(M,X,Y,C(1),T)
         ENDIF
    1 CONTINUE
      RETURN
      END

      SUBROUTINE CSOLV(M,C,CC,T)
      REAL C(1),CC(1),T(1)
      IF(M.LE.0)RETURN
      C1N = T(M+2)
      C1D = T(1)
      DO 1 I = 1,M
         C1N = C1N - CC(I)*T(I+1)
         C1D = C1D - CC(I)*T(M+2-I)
    1 CONTINUE
      C(1) = C1N/C1D
      C1 = C(1)
      DO 2 I = 2,M+1
         C(I) = CC(I-1) - C1*CC(M+2-I)
    2 CONTINUE
      RETURN
      END

      SUBROUTINE XSOLV(M,X,Y,C,T)
      REAL C(1),T(1),X(1),Y(1)
      XN = Y(M+2)
      XD = T(1)
      DO 1 I = 1,M+1
         XN = XN - X(I)*T(M+3-I)
    1 CONTINUE
      X(M+2) = XN/XD
      SX = -X(M+2)
      DO 2 I = 1,M+1
         X(I) = X(I) + SX*C(I)
    2 CONTINUE
      RETURN
      END
```

Figure 14
Solution to Symmetric Toeplitz System

```
      SUBROUTINE FLOYD(M,N)
      INTEGER M(N,N),S,T
C  INF IS EFFECTIVELY INFINITE BUT DOESN'T OVERFLOW
      INF = 10**10
C
      DO 1 I = 1,N
      DO 1 J = 1,N
         T = M(J,I)
         IF((T.LT.INF).AND.(I.NE.J))THEN
            DO 2 K = 1,N
               S = T + M(I,K)
C ------------ IF(S.LT.M(J,K)) M(J,K) = S
C ------------ M(J,K) = CVMGM(S,M(J,K),S-M(J,K))
               M(J,K) = MIN0(S,M(J,K))
    2       CONTINUE
         ENDIF
    1 CONTINUE
      RETURN
      END
```

Figure 15
Minimum Path

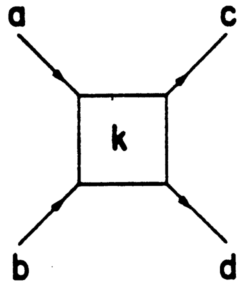


Figure 16
Computation Box

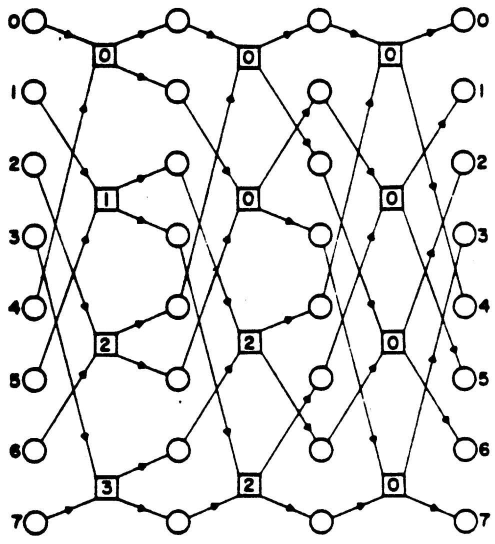


Figure 17
Signal Flow Diagram

Each *box* in Figure 17 calculates complex numbers *c* and *d* from inputs *a* and *b* as in Figure 16. For a box labeled *k*, the arithmetic is

$$c = a + b \qquad \text{(eqn. 1)}$$
$$d = w^k \cdot (a - b) \qquad \text{(eqn. 2)}$$

where *w* is the *n*-th root of unity. In fact, examination of Figure 17 shows there are two possibilities for vector computation of eqn. 1 and 2. The first can be seen at step one - the column of boxes to the left. Reading the elements w^k from memory as a complex vector gives

$$C = A + B \qquad \text{(eqn. 3)}$$
$$D = W \cdot (A - B) \qquad \text{(eqn. 4)}$$

where W contains $n/2$ elements $\{w^k, k = 0,1,2,3$ in the eight point example above$\}$. Vectors A and B are vectors of a's and b's respectively, C and D similarly. Vector multiplication in eqn. 4 is element-by-element. At step two, these vectors are now of half-length, with W containing every other w^k $\{(w^0, w^2)$ in Figure 17$\}$, but eqns. 3 and 4 are executed twice. Boxes one and three are computed first, then boxes two and four using the same W. At step three, the lengths of the vectors are halved again with eqns. 3 and 4 executed four times {vector length 1, $W = (w^0)$, in Figure 17} and so forth if n is larger.

An alternate method is to read the w^k elements as complex scalars

$$C = A + B \qquad \text{(eqn. 5)}$$
$$D = w^k \cdot (A - B) \qquad \text{(eqn. 6)}$$

which scale the vector $A - B$. At the first step, vector equations 5 and 6 are executed $n/2$ times {once for each $k = 0,1,2,3$ in Figure 17}; while on the second step the length of the vectors is doubled but only executed $n/4$ times. Thus, using equations 3 and 4 the vectors start with length $n/2$ and shrink by half at each step to length one; while using equations 5 and 6 the vector length is initially one and doubles at each step to $n/2$ at the last. The last step can be done in-place. A mixed procedure using equations 3 and 4 for the first few steps, and 5 and 6 for the last steps maximizes the inner vector length (ref. 10).

The computation in Figure 18 uses equations 5 and 6 to set

$$y_j = \sum_{k=1}^{n} \exp\left(\frac{2\pi i}{n}(j-1)(k-1)\right) x_k$$

for $1 \leqslant j \leqslant n$. Driver CFFT toggles between arrays X and Y. Complex vector W contains the $n/2$ elements $\{exp(2\pi ik/n)$ for $k = 0,1,2,\ldots,n/2-1\}$, which must be pre-computed. Using the vector SIN and COS functions, the calculation of W adds about 40% to the computation time.

Turning the DO 1 and DO 2 loops inside-out in STEP, with appropriate provisions for the increments I2 and I3, computes the transform using the eqns. 3 and 4. This alternate version of STEP is given in Figure 19. Notice here that W is read as a vector by steps of I2 in the inner loop. In this mode, when I2 is a multiple of 8, memory bank conflicts start to slow access (see section 2.2). Despite this technical difficulty, using the alternate form of STEP (Figure 19) for the first four steps of the transform improves the performance, as shown in Figure 20. This plot shows the execution time for a 1024-point complex FFT vs. the last step for which Figure 19 replaces STEP in Figure 18. That is, in Figure 20 IBREAK is the last step {of $\log_2 n$} for which eqns. 3 and 4 are used, with eqns. 5 and 6 used for the remainder. For transforms longer that 1024, IBREAK = 4 is still used. The 5 millisecond improvement shown in Figure 20 does not change for longer transforms.

To conclude this section, several remarks are appropriate concerning multiple FFT's and general radix transforms. Frequently one is interested in multiple transforms of the same length. For example, solving Poisson's equation on a square might involve calculating 128 independent transforms of length 128. This is easier to vectorize than the single transform case. Indeed, Figure 21 shows a variant of STEP for computing NT repetitions of step I+1 for NT transforms of length N. The important thing to notice is that the inner loop has a fixed repetition count - namely NT. The importance of this concept - *using the inner loop to run over the independent cases* - cannot be overstated for vector computing.

In Figure 21, the subscripts ranged by L treat the rows of X(NT,N) as independent. Similarly, to transform the columns independently, turn the (L,J)'s into (J,L)'s everywhere. Padding the leading dimensions of X and Y by one row will avoid memory-bank conflicts when the leading dimension of X and Y is a multiple of 8.

Although we have only discussed radix 2 transforms (N is a power of 2), it turns out that other radix transforms have some advantages. In part due to memory-bank conflicts, binary radix transforms are less efficient than those of radix 3 or 5 (see ref. 10), which in some cases have nearly twice the processing rates.

```
      SUBROUTINE CFFT(N,X,Y,W)
      COMPLEX X(N),Y(N),W(N/2)
      LOGICAL ITGLE
C  M is LOG2(N), computed using the leading zero count
      M = 63 - LEADZ(N)
      I2 = 1
      ITGLE = .TRUE.
C
      CALL STEP(N,I2,X(1),X(N/2+1),Y(1),Y(I2+1),W)
      DO 1 I = 1,M-2
         I2 = 2**I
         IF(ITGLE)THEN
            CALL STEP(N,I2,Y(1),Y(N/2+1),X(1),X(I2+1),W)
            ITGLE = .FALSE.
         ELSE
            CALL STEP(N,I2,X(1),X(N/2+1),Y(1),Y(I2+1),W)
            ITGLE = .TRUE.
         ENDIF
    1 CONTINUE
C
      I2 = N/2
      IF(ITGLE)THEN
         CALL STEP(N,I2,Y(1),Y(N/2+1),Y(1),Y(I2+1),W)
      ELSE
         CALL STEP(N,I2,X(1),X(N/2+1),Y(1),Y(I2+1),W)
      ENDIF
C
      RETURN
      END

      SUBROUTINE STEP(N,I2,A,B,C,D,W)
      COMPLEX A(1),B(1),C(1),D(1),W(1)
C
      I3  = 2*I2
      MI2 = N/I3
      DO 1 J = 0,MI2-1
         JW = 1 + J*I2
         JA = JW
         JB = JA
         JC = 1 + J*I3
         JD = JC
C
         DO 2 K = 0,I2-1
            C(JC + K) = A(JA + K) + B(JB + K)
            D(JD + K) = W(JW)*(A(JA + K) - B(JB + K))
    2    CONTINUE
C
    1 CONTINUE
      RETURN
      END
```

Figure 18
FFT using eqns. 5 and 6

```
      SUBROUTINE STEP(N,I2,A,B,C,D,W)
      COMPLEX APB,AMB,A(1),B(1),C(1),D(1),W(1)
C
      I3  = 2*I2
      MI2 = N/I3
C
      DO 2 K = 1,I2
C
         DO 1 J = 0,MI2-1
            APB = A(K + J*I2) + B(K + J*I2)
            AMB = A(K + J*I2) - B(K + J*I2)
            C(K + J*I3) = APB
            D(K + J*I3) = W(K + J*I2)*AMB
    1    CONTINUE
C
    2 CONTINUE
      RETURN
      END
```

Figure 19
Alternate Version of STEP

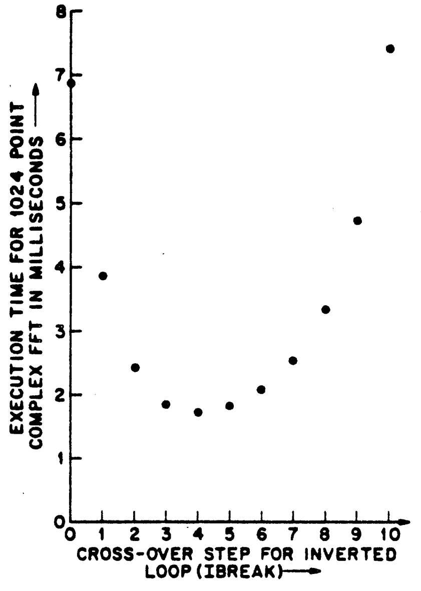


Figure 20
Execution Time vs. Crossover

```
          SUBROUTINE STEP(NT,N,I2,A,B,C,D,W)
          COMPLEX A(NT,1),B(NT,1),C(NT,1),D(NT,1),W(1)
C
          I3  = 2*I2
          NI2 = N/I3
C
          DO 1 J = 0,NI2-1
             J2 = J*I2
             J3 = 2*J2
             JW = 1 + J2
C
             DO 2 K = 1,I2
C
                DO 3 L = 1,NT
                   C(L,J3 + K) = A(L,J2 + K) + B(L,J2 + K)
                   D(L,J3 + K) = W(JW)*(A(L,J2 + K) - B(L,J2 + K))
3               CONTINUE
C
2            CONTINUE
C
1         CONTINUE
          RETURN
          END
```

Figure 21
STEP for NT Transforms of Length N

4.5 Getting around dependencies

When an updated element of an array is required for the subsequent calculation of an element in the same array, the elements may not be treated independently. In CRAY-1 documentation (ref.2), this situation is regarded as a 'dependency'. In fact, because of the register architecture, two types of dependencies exist. In Figure 22 both types are illustrated.

```
          DO 1 I = IL,IU
C       DEPENDENCE ON A PREVIOUS ELEMENT
             A(I) = A(I - I0)
C       OVERLAPPING STORAGE DEPENDENCY
             B(I1 + I) = 1.
             B(I2 + I) = 2.
        1 CONTINUE
```

Figure 22
Non-Vectorizing Dependencies

In the top example, if I0 > 0, then A(I) requires the previous value A(I - I0) to have been reset, as in Figure 2. Hence, there is a mandatory ordering and A cannot be regarded as a vector with independent elements. At compile time CFT cannot determine whether I0 is negative and inhibits generation of vector hardware instructions.

Storage of B in the second part of Figure 22 may overlap if I1 < I2. This kind of dependency is somewhat more subtle, a result of the register architecture of CRAY-1. For example, try setting I1 = 0, I2 = 1, IL = 1, IU = 3. The storage of a segment {B(2), B(3), B(4)} = {2., 2., 2.} over the segment {B(1), B(2), B(3)} = {1., 1., 1.} would give B = {1., 2., 2., 2.}, which is not the same as the desired B = {1., 1., 1., 2.}. At compile time, if I1 and I2 are not known, CFT flags this as a dependency case. In the following, examples are given which illustrate some ways to side-step these non-vectorizing dependencies.

4.5.1 Gauss-Seidel Relaxation Sometimes it is possible to find directions or subsets of multiply dimensioned arrays in which all the elements along those rays may be treated as independent. All the elements in one column can be regarded as independent of elements in other columns, for example. A simple relaxation step on the interior points of a rectangular grid is shown in Figure 23.

```
      SUBROUTINE GS(U,M,N)
      REAL U(M,N)
      DO 1 I = 2,M-1
      DO 1 J = 2,N-1
         U(I,J) = .25*(U(I-1,J) + U(I+1,J) +
     &                 U(I,J-1) + U(I,J+1))
    1 CONTINUE
      RETURN
      END
```

Figure 23
Gauss-Seidel Relaxation Step

In this figure, computation of the J th element in row I depends on the updated value of the (J −1) th element, inhibiting vectorization. Just drawing a picture of the grid of I,J elements shows that diagonals depend only on the updated elements of previous (lower) diagonals. Each of these diagonals may be treated as independent, as in Figure 24.

```
      SUBROUTINE VGS(U,M,N)
      REAL U(M-1,1)
      DO 1 I = 4,M+N-2
         L = I-M
         DO 1 K = MAX0(2,I-M+1)+1,MIN0(N-1,I-2)+1
            U(L,K) = .25*(U(L-1,K) + U(L+1,K) +
     &                    U(L-M,K) + U(L+M,K))
    1 CONTINUE
      RETURN
      END
```

Figure 24
Vector Gauss-Seidel Relaxation Step

Scanning by diagonals has the disadvantage that the vector length in the inner loop keeps changing. For very small iteration counts (1 or 2), the overhead to fill the functional unit pipelines is appreciable. Nevertheless, Figure 24 runs five times faster than Figure 23 if N,M > 100.

4.5.2 A red/black ordering A method asymptotically equivalent to the above is a red-black ordering. Figure 25 illustrates a simple red/black relaxation step, which has two additional features: use of the IVDEP directive, and parsing to minimize memory fetches. Within the DO 2 and DO 4 inner loops, the last pair of points in each equation is shared, eliminating two additional fetches. This example runs at 70 million floating point operations/second. A relatively easy modification of Figure 25 for Poisson's equation (not Laplace's equation, as above) performs floating point operations faster than the 80 megaHertz clock. This modification requires a relaxation parameter $\omega \neq 1$ and a 'source' term. Because of the additional computations, the operation rate actually goes up.

```
      SUBROUTINE RB(U,M,N)
      REAL U(M,N)
C         ---- RED POINTS ----
CDIR$ IVDEP
      DO 1 J = 2,N-2,2
C
        DO 2 K = 2,M-2,2
          U(K  ,J  ) = .25*(U(K-1,J  ) +(U(K  ,J-1)+
     &                      (U(K+1,J  ) + U(K  ,J+1))))
          U(K+1,J+1) = .25*(U(K+2,J+1) +(U(K+1,J+2)+
     &                      (U(K+1,J  ) + U(K  ,J+1))))
    2   CONTINUE
C
    1 CONTINUE
C         ---- BLACK POINTS ----
CDIR$ IVDEP
      DO 3 J = 2,N-2,2
C
        DO 4 K = 3,M-1,2
          U(K  ,J  ) = .25*(U(K  ,J-1) +(U(K+1,J  )+
     &                      (U(K-1,J  ) + U(K  ,J+1))))
          U(K-1,J+1) = .25*(U(K-2,J+1) +(U(K-1,J+2)+
     &                      (U(K-1,J  ) + U(K  ,J+1))))
    4   CONTINUE
C
    3 CONTINUE
C
      RETURN
      END
```

Figure 25
Red-Black Relaxation

4.5.3 Tridiagonal systems In both the forward elimination and back-substitution steps in the solution of a tridiagonal linear system, each element depends on its predecessors. To solve a single tridiagonal linear system by a parallel or vector algorithm represents a problem of some difficulty. There are several approaches: Buneman's variant of cyclic reduction (ref. 13), Stone's recursive doubling (ref. 14), and a parallel Cramer's rule method by Swarztrauber (ref. 15). The fastest of these methods on CRAY-1 is cyclic reduction (ref. 16). Unfortunately, each method relies on a recursive doubling of step size in memory, while halving the vector length of each operation, and vice-versa. Because of overhead introduced by short vectors, and memory-bank conflicts inherent at each stage, even cyclic reduction runs only twice as fast as a simpler scalar method (ref. 17) for very large problems. In fact, a simple scalar tridiagonal solver is faster than cyclic reduction for solution vectors of length less than 63. Since this not a delightful result, let's sidestep the issue and do another problem.

Multiple tridiagonal systems are much more tractable. Block tridiagonal equations, many line-relaxation methods, and three dimensional problems usually need solutions to many totally independent (unrelated) tridiagonal systems. So, the more the merrier - now the inner loops can be made to range over each unrelated system in turn.

```
      SUBROUTINE MANYSOL(M,N,L,D,U,Y,X)
      REAL L(N-1),D(N),U(N-1),Y(M,N),X(M,N)
C
C    SOLVES M SYSTEMS OF TRIDIAGONAL EQUATIONS
C
C                   A*X = Y
C
C    WHERE COMMON TRIDIAGONAL MATRIX A OF THE EQUATIONS
C    HAS LOWER DIAGONAL L, UPPER DIAGONAL U, AND
C    MAIN DIAGONAL D. MATRIX A IS N BY N. D AND Y
C    ARE MODIFIED.
C
      DO 1 I = 2,N
         T = L(I)/D(I-1)
         D(I) = D(I) - T*U(I-1)
C
         DO 2 J = 1,M
            Y(J,I) = Y(J,I) - T*Y(J,I-1)
    2    CONTINUE
C
    1 CONTINUE
C
      DO 3 J = 1,M
         X(J,N) = Y(J,N)/D(N)
    3 CONTINUE
C
      DO 4 I = N-1,1,-1
C
         DO 5 J = 1,M
            X(J,I) = (Y(J,I) - U(I)*X(J,I+1))/D(I)
    5    CONTINUE
C
    4 CONTINUE
C
      RETURN
      END
```

Figure 26
Many System Tridiagonal Solver

Once again, it must be emphasized that using inner loops to range over identical operations on independent data or systems is the key to successful vector processing. Indeed, using the above procedure to solve 100 tridiagonal systems of length 100 is 13 times faster than solving 100 one at a time. If each system has a different matrix, Figure 26 is easily modified for this purpose.

4.6 Sparse matrices - nonlinear indexing

Most general-purpose sparse matrix solvers (ref. 18) keep pointers to non-zero elements. Allocating storage only for non-zero elements and potential fill-in minimizes both memory requirements and the number of null operations. All six (ref. 19) forms of Gaussian elimination require a compression and decompression of rows or columns into indexed lists as the elimination proceeds. For example, the most efficient form of elimination (ref. 20) has the following reduction step on the working row.

```
      DO 1 I = IL,IU
         Y(INDEX(I)) = A*X(I) + Y(INDEX(I))
    1 CONTINUE
```

Figure 27
Sparse y := ax + y Operation

Here, Y is the working row, X is any lower packed row with INDEX an array of pointers to the positions of X in its expanded form. For an arbitrary sparse matrix, INDEX will not point to regularly spaced elements.

This is a difficult problem with no vector hardware solution. At present, only CAL-coded modules can manage to approach vector-mode floating point operation rates. The following operations are available for sparse matrices on CRAY-1 with the 1.10 CFT software release.

```
      DO 1 I = 1,N
C  SCATTER OPERATION
         Y(INDEX(I)) = X(I)
C  GATHER OPERATION
         Y(I) = X(INDEX(I))
C  SPAXPY OPERATION
         Y(INDEX(I)) = A*X(I) + Y(INDEX(I))
C  SPARSE DOT-PRODUCT
         SPDOT = SPDOT + Y(INDEX(I))*X(I)
    1 CONTINUE
```

Figure 28
Sparse Operations

Respectively, the calling sequences for these modules are as follows.

```
      CALL SCATTER(N,Y,INDEX,X)
      CALL GATHER(N,Y,X,INDEX)
      CALL SPAXPY(N,A,X,Y,INDEX)
      DOT = SPDOT(N,Y,INDEX,X)
```

Figure 29
CAL Sparse Operations

For N > 5, all these subprograms are more efficient than in-line Fortran. For large N these modules give a factor of 3 timing improvement.

W. P. Petersen

MH-45231-WPP-wpp

W. P. Petersen

Atts.
Attachment 1 - List of References
Attachment 2 - Index

LIST OF REFERENCES

1. Cray Hardware Reference Manual, Cray Res. Publ. 2240004.
2. Cray Fortran (CFT) Reference Manual, CRI publ. SR-0009 version 1.10, revision H (ANSII standard).
3. CAL Assembler Reference Manual, CRI publ. SR-0000. (Cray Assembly Language)
4. W. Petersen, "MXM - unit spaced fast matrix multiply," CRI technical note SN-0213, Dec. 1980.
5. K. Fong and T. L. Jordan, "Some linear algebraic algorithms and their performance on CRAY-1," Los Alamos Scientific Laboratory Report LA-6774 (University of California report #UC-32), June 1977.
6. N. Levinson, *Journal of Mathematics and Physics*, Vol. XXV, No. 4, January 1947, pp. 261-278.
7. Robert W. Floyd, "Shortest path," Algorithm 97 - Communications of the Association for Computing Machinery, 5, 345(1962).
8. J. W. Cooley and J. W. Tukey, "An algorithm for machine calculation of complex Fourier series," *Math. Computation*, vol. 19, pp. 297-301, April 1965.
9. Clive Temperton, "Mixed-radix fast Fourier transforms without reordering," European Centre for Medium Range Weather Forecasts - Technical Report #3, February 1977.
10. W. Petersen, "CFFT2 - complex fast Fourier transform binary radix subroutine," Cray Research Inc. Technical Note 2240203, March 1978.
11. Clive Temperton, "Fast Fourier transforms on CRAY-1," European Centre for Medium Range Weather Forecasts - Technical Report #21, January 1979.
12. D. M. Young, "Iterative Solutions of Large Linear Systems," Academic Press, New York (1971).
13. O. Buneman, "A compact non-iterative Poisson solver," Stanford Univ. Institute for Plasma Research - Report #294, (1969).
14. H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal system of equations," *J. Assoc. Computing Machinery*, vol. 20, 1973, pp. 27-38 (1973).
15. P. N. Swarztrauber, "A parallel algorithm for solving general tridiagonal equations," *Math. Computation*, vol. 33, No. 145, pp. 185-199, January 1979.
16. P.N. Swarztrauber, private communication.
17. R. Penumalli, private communication.
18. B. Dembart and K. Neves, 'Sparse triangular factorization on vector computers', in *Exploring Applications of Parallel Processing to Power System Analysis*, Electric Power Research Institute Report EL-566-SR, October 1977.
19. S.C. Eisenstat, M. H. Schultz, and A. H. Sherman, "Considerations in Design of Software for Sparse Matrix Computation," in *Sparse Matrix Computation*, J. R. Bunch and D. J. Rose editors, Academic Press, 1976.
20. D. S. Dodson and W. Petersen, "Sparse triangular factorization on CRAY-1," Cray Research Inc. Technical Note SN-0217, June 1981.

INDEX

KEYWORD	PAGES	KEYWORD	PAGES
ANSII standard	11	I/O	5
block	6	I/O block sizes	5
BUFFER IN/BUFFER OUT	5	IVDEP	11,21
CAL	9,11,13	loop invariant	7-9,13
CALL statements	2	loops	1,6-9,12,13,19
CDIR$	11,21	machine instructions	2-4,10
CFT control	11,12,13	masks	3,10,11,13
chaining	3,4,5,10	MAX/MIN functions	7,10,11,13,15
character data	6	memory access	1,3-5,13,21
CII	7,8,9	merges	10,11,13,15
clock period	2,3,5,6,20,21	network	13,15
compiler directives	11,20	outer products	12
concurrent operations	3,4,5	overlap	3-5
conditional statements	10,11,13,15	pipelined functional units	3-6
constant increment integer	7,8,9	products	6
Cray Assembly Language	9,11,23	reciprocal approximation	3,4
CVMG functions	10,11,13,15	recursive doubling	21
cyclic reduction	21	red black methods	12,20,21
dependencies	2,19,20,21	reductions	5,6,11,12,14
digital filter	12	regular spacing	1,7
division	4	relaxation	12,19-22
DO loop	1,6-9,12,13,19	residual segment	1
dot products	5,12-14	S registers	2-6
double precision	6,7	scalar mode	1,2,9,11,13,22,23
FFT	12,13-19	scalar registers	2-6
floating point	3,6,20	SCATTER	23
floating point functional units	3	segments	1,19
FLOWTRACE	12	sparse matrices	22
Floyd's algorithm	12,13,15	SPAXPY	23
formatted I/O	5	subscripts	7-9,22,23
Fourier transform	12,13-19	sums	2,5,6
GATHER	22,23	superfluous calculation	1,2,10-11,13,15
Gauss Seidel relaxation	19,20	Toeplitz matrix	12-14
GOTO	2,6,10	tridiagonal systems	12,21,22
IF	2,10,13,15	unrolling loops	1
independent elements	1-3,19	V registers	2-6,10
independent operations	3,4,19	VECTOR BLOCK	6
indexing	5,7,8,9	vector length	1,3,12-19
inner products	5,6,8,12,13,14	vector mode	1-3
inner vector length	6,13-19	vector registers	2-6,10
integer data	3,6,13,15	VL register	1
intrinsic functions	6,7	VM register	3,10

Non-Profit Organization
U.S. POSTAGE PAID
Boulder/Colorado
Permit No. 558

The Record

Issued by the
National Center for Atmospheric Research
Scientific Computing Division
P.O. Box 3000
Boulder, Colorado 80307

PRINTED MATTER

SUMMERS, BARBARA
Mesa Lab
NCAR
Boulder, CO (MAIL ROOM)