CODING FOR THE CRAY X-MP USING CFT77:
AN INTRODUCTION

R.A. Hendrickson and M. Metcalf
Coding for the Cray X-MP using CFT77: An Introduction

Michael Metcalf ¹ and Richard A. Hendrickson ²

Abstract

Data processing in HEP is about to move into the era of Vector Processing. The purpose of this note is to set out a number of guidelines enabling maximum advantage to be taken of a particular architecture. Program portability is inevitably affected.

¹) CERN, Geneva, Switzerland
²) Cray Research Inc., Mendota Heights, MN, USA
1. A New Opportunity

Until now, all of the computing systems we have used in HEP have been of the type known as scalar or serial computers, that is processors whose functional units carry out operations on at most two scalar operands at a time and which use compilers requiring that each such operation be described explicitly. Thus, to form the element-by-element product of two vectors of equal length requires a DO-loop which generates machine instructions which specify the exact operations to be performed on each element of the arrays, treating each element as a scalar entity at the point at which the operation on it is performed.

This note is concerned with a quite different type of computer architecture, capable under certain conditions of performing arithmetic operations at a rate an order of magnitude faster than the most powerful scalar machines — the true 'super-computers'. Computers with this performance are often referred to as array or vector processors, because of their ability to process arrays of operands efficiently. They are produced in a variety of architectures but we shall consider only an example of that type referred to as vector instruction processors, that is, computers which have the ability to perform operations on whole vectors with a single instruction.

The introduction of these computers has opened up whole new vistas of computing in scientific fields such as meteorology, plasma physics, weapons research, hydrodynamics and nuclear chemistry, and there is an active exchange of ideas and experience concerning the new computing techniques which are required (see, for instance, the journal *Supercomputer*).

These powerful machines do not always lend themselves easily to use for general computing tasks, and therefore they often require a so-called front-end computer to act as a job preparation station. This need is reinforced by the large costs of developing general purpose software; this would be difficult to amortize over the few dozen super-computers of any one type which are sold. This explains the increasing trend to install the portable UNIX operating system on these machines, *e.g.* UNICOS to replace COS on the Cray range. However, in spite of the relatively small numbers installed, since they can often be accessed through networks, they are now achieving considerable success amongst scientists engaged in large-scale scientific computing. Extensive details may be found, for example, in Hockney and Jesshope (1981). A short review was given by Levine (1982).
2. Optimization

In order to be able to obtain the highest possible degree of performance from these computers, the algorithms have to be selected and presented in a way that fully exploits their vector-processing capability. There are a number of levels at which this can be achieved on the Cray range of computers:

1. the program is written in a conventional way, and its vector nature is recognized automatically by the Fortran compiler, and translated into machine language vector instructions accordingly;

2. the compiler is assisted by compiler-directives;

3. the program code is restructured in an unconventional way, in order to take greater advantage of the automatic translation;

4. the program is written using specific vector extensions to the Fortran language, which are translated directly into machine language by the compiler;

5. the program is translated into a Fortran array-processing dialect by a pre-processor, and then compiled by a compiler recognising those extensions;

6. the kernel of the program, that part which is used most intensively, is written directly in vector machine language, (which can be simpler than the standard machine language as sometimes fewer instructions need to be coded).

Clearly, this work should be undertaken only in those areas of the program where large amounts of time are being spent — the so-called hot-spots. Means to identify these are available.

Because of the diversity of applications, general statements about the optimal approach to achieving high performance are difficult. Those programs which have to remain highly portable, on scalar machines as well as on vector processors, require a high level of automatic translation, in order to avoid the need to maintain different program versions corresponding to the various types of machine. Other applications may only be conceivable at all on a vector processor, and no effort will be spared to ensure the highest possible performance. For the former class of programs, a vector processor is of interest only if the compiler relieves the programmer of the bulk of the task of vectorizing the code although, interestingly, code which is straightforwardly optimized for a vector processor, for instance by simplifying subscripts or regularizing control flow, can have improved performance also on scalar machines. For the latter class, the processor is that much simpler to use if powerful syntactic extensions to the language are made available by the compiler. Some extensions of this type, which may become part of the Fortran language in the future, are now being included in new compilers, (see Metcalf, 1987). In the remainder of this note, the way in which this problem has been tackled by Cray Research Inc. will be briefly examined, and some features concerning code restructuring without
language extensions will be identified. However, first a basic principle of vector architectures, the pipeline, needs to be recalled (a detailed description is given by Ramamoorthy and Li, 1977).

![Diagram](image)

**Fig. 1** A pipeline with four sub-operations

In a pipeline (Fig. 1), the operation performed by a functional unit is segmented into a number of sections, each lasting one machine cycle, and these sub-operations proceed independently as operands pass through the unit, a new pair of input operands being accepted by the unit at every cycle. In the case of vector processors, it is possible to sustain a very high duty cycle in the functional units, as the operands are presented continuously as whole vectors, and the processing flow is not interrupted by the branches typically present when a Fortran DO-loop controls a vector operation. It is also unnecessary to load and decode the large number of instructions required for scalar operations, (although the loads are often reduced in number by an instruction stack.) This means, in principle, that when correctly programmed, the processor can provide a result every clock-cycle, and this provides a means of calculating the asymptotic performance of the processor – the number of floating-point operations which, theoretically, can be performed per second. This is usually measured in millions of floating-point operations per second, or megaflops, and the asymptotic value for a single functional unit is simply the reciprocal of the cycle-time, *i.e.* from the 9.5 nsec cycle-time of the CRAY X-MP we obtain an asymptotic value of 105 megaflops for a single processor. This figure can be exceeded for the whole processor because functional units can operate in parallel (giving up to 210 megaflops for the whole machine), but in practice substantially lower average rates are often obtained, as these features cannot easily be fully exploited.

A particular problem, as will be seen below, is the presence of IF-statements in loops. This may be dealt with either by compressing the elements of vectors into shorter vectors containing only those elements which should participate in the calculation, or by performing calculations on all elements and storing only those results which are actually required. Either way an overhead is incurred.

Before describing the CRAY X-MP, two basic and common points with respect to programming this type of machine must be made. The first is the need to ensure that the compiler actually has the opportunity to detect the vectors for which it is supposed to generate vector instructions. This means, that where a subroutine is called inside a loop, that that loop either needs to be *pushed* into the subroutine, or the subroutine should be *pulled* into the loop.
Secondly, each vector operation requires what is known as a start-up time. This is the time required to fetch and decode an instruction and to prime the registers and functional units with their first operands. On the Cray range, these operands are stored in vector registers, but nevertheless it does mean that the longer a vector can be, the less significant this time is with respect to the total time required for processing the instruction. The general principle of keeping the innermost loops the longest, in terms of number of iterations, stands particularly for this type of computer, and other general principles of optimization (as described in Metcalf, 1985) should usually also be applied.

One last problem should be mentioned, namely the importance of the speed of the scalar unit in achieving high speeds; tests have shown that this can be critical in the overall improvement with respect to a conventional serial computer, depending on the degree of vectorization attainable, as given by

\[ \text{performance} = \frac{p_V}{(\alpha + (1-\alpha)p_V/p_S)} \]

where \( p_V \) is the vector performance, \( p_S \) is the scalar performance, and \( \alpha \) is degree of vectorization. In general, a vector processor only begins to become cost-effective compared with a conventional architecture when about 70% of vectorization has been achieved.

3. The CRAY X-MP/48

The CRAY X-MP vector processor is one of the latest in a range of vector processors which began with the CRAY 1. Its basic hardware block diagram is shown in Fig. 2. An important feature of this machine is the fact that it has sets of vector registers, and that the vector operations are, therefore, able to proceed from register to register (rather than from memory to memory), resulting in a short start-up time, once the registers are loaded. This computer has two memory-to-register pipelines for such loading, and a third for storing.

The CRAY X-MP is a 64-bit word machine with a large, fast (38 nsec), main memory possessing a high bandwidth (315 Mwords/sec/pipeline), backed up by high-speed disc drives. It may have one, two or four CPUs, and this is indicated as the first digit of the numerical part of the serial code, e.g. the CRAY X-MP/48 has four CPUs. The second digit indicates that it has eight Mwords of main memory; this memory is organized into four sets of 16 banks. Each CPU possesses a set of scalar functional units, a set of shared floating-point functional units for either scalar or vector operands, and four units exclusively for vector operands — integer add, logical, shift and population count (to count the number of set bits in a vector). Divisions are performed by a reciprocal approximation of the divisor followed by a multiplication with the numerator. A 64-bit vector mask, or control vector, which may be created by vector compare instructions, is used to control the flow of vectors between registers. The scalar and address registers are of two types, those directly
Fig. 2 The hardware organization of the CRAY X-MP/48

accessible to the functional units, and another group eight times larger which acts as a buffer between memory and the main registers. The eight vector registers are loaded directly from memory and each may contain vectors up to 64 full words in length. All the functional units are fully segmented pipelines; the floating-point add requires six cycles for a complete single operation, the floating-point multiply seven, and the floating-point reciprocal fourteen cycles, but each can then produce one new result every cycle. The units are able to operate in parallel, bringing a further en-
hancement to the processing speed, which is augmented still further by an additional feature known as chaining. This is a procedure whereby the results of a vector operation may be fed directly from the target register into a second functional unit, without the whole vector operation having first to be completed. For instance, a vector add may proceed in parallel with a dependent vector multiply, as the results from the multiply are available almost immediately to the add unit. Program instructions are kept in a 128-word stack, which eliminates the need to fetch each instruction from memory if loops are shorter than the stack.

A solid state disc (SSD) buffer memory between the disc store and a main memory is an optional part of the configuration — the SSD at CERN will have 128 Mwords. The bandwidth with SSD attached is 2.4 Gbytes/sec. The SSD is addressed like a disc, and data sets may be kept there by a command in the job control language.

A single program may use all four CPUs concurrently.

4. Programming with CFT77

4.1 Introduction

A CRAY computer is essentially a Fortran engine, as the main software tools it offers are a program maintenance system and a compiler for the CFT77 language (Cray, 1986), essentially Fortran 77 with some extensions, some of them from Fortran 8x. This is a new compiler, written in Pascal to be portable across the whole range of Cray computers. It is able, under certain conditions, to recognize vectorizable loops, and it falls to the programmer to structure loops to take the best advantage of this feature. It is possible via a compiler directive to force the compiler to vectorize a loop which it would otherwise consider to be unsafe to vectorize because of some possible data dependence (see below). However, a check is always required that the use of such a directive is safe, and has not introduced an unexpected error.

The machine vector instructions are only of the simple vector/vector and scalar/vector add, multiply, divide and logic type, and accept only two operands. The fact that the vector registers are 64 words long and that vectors must therefore be broken into 64-word sections is handled by the compiler. For instance, a loop beginning

```
DO 1 I = 1,100
```

will generate two vector instructions, one with a Vector Length (VL) of 36 and the other with a VL of 64.

The scatter/gather hardware means that vectors need not be in contiguous words in memory — the elements can be separated by a constant stride or increment, or be addressable by a set indices thus making use of the scatter/gather instructions.
For the Fortran programmer this can imply a completely new way of structuring problems and their coded algorithms, in order to take advantage of the vector facilities and to avoid inadvertently rendering DO-loops or IF-loops non-vectorizable. This may happen particularly in any case where the execution of a vector operation may need as an operand a quantity modified in an earlier step of the same operation, a situation occurring in loops which are apparently or actually recursive, as in

\[
\text{DO } I \text{ } I = 2,100 \\
X(I) = X(I-1) + 1. \\
\text{I CONTINUE}
\]

where the value of \(X(I)\) depends on a value from the previous step. A loop such as

\[
\text{DO } I \text{ } I = 1,99 \\
X(I) = X(I+1) + 1. \\
\text{I CONTINUE}
\]

is, however, not recursive, as the value of \(X(I+1)\) is already available for use in the evaluation of \(X(I)\).

Currently vectorization is performed on an "all or none" basis for a loop. If there is anything which inhibits vectorization then nothing is vectorized. The major obstacles are dependencies, CALL statements and I/O statements. If a loop contains any of these inhibitors, then the loop will have to be split into two or three loops, some of which will vectorize and some will not.

4.2 Dependencies

For a loop to be vectorizable, the compiler must be able to evaluate the subscripts and analyze the data flow in the loop. This is called dependency analysis. The compiler examines the subscripts and variables used in subscripts to determine whether they are induction variables. These are variables which can be transformed into the form

\[
\text{variable} = \text{initial value} + \text{constant increment}
\]

These are sometimes called constant increment integers or constant increment variables. The expressions involved can be reasonably complex, involving multiplication and addition or subtraction by constants or "loop invariants", but not division, exponentiation or cross multiplication. In a loop like
DO 10 I = JJJ, KKK, LLL
J = 100 - I
K = K + 13
L = 2*I + 7
M = I*I
:

I, J, K and L are induction variables; M is not. Induction variables, and the restricted expressions described above, can be used as subscripts in vector array references.

Once the subscripting pattern has been analyzed, the compiler tests for non-vectorizable dependencies like

\[ X(I) = X(I-1) + 1.0 \]

where the needed result is in the vector pipe and cannot be used in the computation. There are several cases of dependencies:

1. no dependencies: the loop will be vectorized;
2. real dependencies: the loop will run in scalar mode;
3. removable dependencies as in

\[ X(I) = X(I-29) + 1.0 \]

in which the loop will be vectorized with a VL of 29;
4. ambiguous dependencies, the most common kind, as in:

\[ X(I) = X(I+N) + 1.0 \]

This is vectorizable if \( N > 0 \) or \( N < -64 \). If \(-64 < N < 0\), the loop can be vectorized with \( VL = \text{ABS}(N) \). The compiler will generate code to compute the safe value of VL (which might be 1) and run the loop in vector mode. If it is known that \( N > 0 \), then a CDIR$IVDEP compiler directive can be used to eliminate the small overhead of the loop prologue which performs the run-time checks on N. The case where \( N = 0 \) is really an error condition, and the control option ZEROINC must be used.

If the subscripts are complicated or do not use induction variables, the compiler will generate code which uses the hardware scatter/gather instructions. This code is slightly less efficient than the normal case with a constant stride. No dependency analysis is really possible. If an array is scattered into, then it cannot be used on a right-hand side, whether as a gather load or a normal load, e.g.

\[ \text{ARRAY(INDEX(I))} = \text{ARRAY(INDEX(I))} + 1.0 \]
will not automatically vectorize since two elements of ARRAY may be the same, and might then both be in the pipeline together. Where it is known that INDEX is a permutation vector (i.e. no two values are the same), an IVDEP directive can be used to force vectorization.

There is one special case of dependency: reduction. Examples are

\[
\begin{align*}
\text{PROD} &= \text{PROD} \times A(I) \\
\text{SUM} &= \text{SUM} + B(I) \\
\text{TEST} &= \text{TEST.\ OR.}(A(I) \land \lt B(I))
\end{align*}
\]

etc.

These will be compiled using vector techniques. In effect, an expression like the vector sum is evaluated as if SUM were an array with 64 elements. SUM(1) is set to B(1) + B(65) + B(129)\ldots, SUM(2) to B(2) + B(66) + B(130)\ldots, etc. After these partial sums are evaluated in vector mode, a separate reduction sums the 64 sums into a single total. Because the elements are evaluated in a different order from the equivalent scalar loop, there can be a small round-off difference when reduction loops are vectorized.

4.3 Run-time library

The run-time library supplies vector versions of all the interesting Fortran intrinsic routines — SIN, SQRT, LOG, EXP, etc. Simple functions — MAX, ABS, MOD, etc. — are expanded as in-line code. Loops which use these functions are vectorized. User supplied assembly language routines can also be used in vector loops if a VFUNCTION directive is issued. There is also an extensive library, $\text{SSCILIB}$, of optimized routines — matrix multiply, FFTs, searches and sorts, linear algebra, linear recursion, LINPACK, EISPAC, etc.

4.4 Recursion

Although linear recursion is not vectorizable, the compiler is able to generate very efficient code for a statement like

\[
A(I) = B(I) + C(I) \times A(I - 1)
\]

using loop unrolling (see below) and bottom loading (loading array elements at the end of a loop for use in the following iteration). Usually, the generated code is about as good as the assembly code in $\text{SSCILIB}$ for simple recursion.
4.5 Unrolling

In general, there is no need to distort loops by unrolling them in order to attempt to reduce overheads and to improve efficiency. Sometimes loops like

\[
\text{DO 10 I = 1,1000} \\
10 \quad A(I) = B(I) + C(I)
\]

are rewritten as

\[
\text{DO 10 I = 1,1000,2} \\
\quad A(I) = B(I) + C(I) \\
10 \quad A(I+1) = B(I+1) + C(I+1)
\]

the aim being to reduce loop overheads by halving the number of iterations. CFT77 will do this type of low-level unrolling based on instruction timing and register availability. It can make decisions based on detailed timing for a specific loop, and is more likely to succeed than a typical programmer.

4.6 Nesting

Because vectors work best with long strides, it is best to make the inner loop the longest. Loops should be nested as in

\[
\text{DO 10 I = 1,3} \\
\text{DO 10 J = 1,10000}
\]

rather than as in

\[
\text{DO 10 J = 1,10000} \\
\text{DO 10 I = 1,3}
\]

unless this would distort the physics or introduce dependencies, for instance in

\[
\text{DO 10 J = 1,10000} \\
\text{DO 10 I = 1,10} \\
10 \quad A(I,J) = A(I, J-1) + 1.0
\]

which is vectorizable (taking a column at a time), whereas

\[
\text{DO I = 1,10} \\
\text{DO J 1,10000} \\
10 \quad A(I,J) = A(I, J-1) + 1.0
\]

is not (due to recursion in the fastest-moving index). In the near future (late 1987?), CFT77 will begin to interchange loops to achieve long vector lengths and to remove dependencies.
4.7 IF statements

CFT77 vectorizes DO loops containing IF statements and IF blocks. If the IF statement is an exit from the loop (a so-called search loop), it must be in the first block of the DO loop (i.e. typically in the first sequence of straight-line code). A block-IF can be either an IF...THEN...ELSE construct or it can be formed from IFs and GOTOs. IFs can be nested to a reasonable depth (currently 20). The dependency analysis for nested IF blocks is more tricky. In general, dependencies between quantities defined outside the IF block and those defined inside the block are not resolved.

The IF statements are vectorized using a variant of the CFT77 CVMGT function, which performs an assignment under control of a logical mask. Thus

\[ \text{IF (A(I).NE.0) B(I) = 1.0/A(I)} \]

is compiled much like

\[ \text{B(I) = CVMGT(1.0/A(I), B(I), A(I).NE.0)} \]

There are 64 evaluations of 1./A(I), but "safe" values are used for the non-selected terms; this guarantees that no arithmetic exception occurs. This method is quite efficient if the IF-controlled expression is fairly simple and if the IF expression is .TRUE. a reasonable amount of the time. Given a loop containing an expression like

\[ \text{IF (PRICE(I).LT.0.10 X(I) = EXP(LOG(SIN(ACOS(TAN(SQRT(COSH(...

where the condition is rarely fulfilled, it is probably better to force scalar evaluation with a NOVECTOR directive so that only the few necessary computations are carried out.

4.8 Extensions

CFT77 supports a limited set of the Fortran 8x array syntax (Metcalf, 1987). Whole array assignment is permitted and array expressions can be used in assignment statements. Arrays can be used as arguments to intrinsic functions, and automatic arrays are implemented, e.g.

\begin{verbatim}
SUBROUTINE SUB(I,J,K)
REAL AUTARR(I,J,K)
\end{verbatim}

Here, AUTARR is a local array whose actual bounds, and hence storage, are defined only when SUB is called. Within SUB it may be used like any other array. Note, however, that array expressions cannot be used as actual arguments in a CALL.
CFT77 supports recursive function and subroutine references, which must be compiled with the ALLOC=STACK compiler option.

Stack allocation methods must also be used if *multiple processors* are applied to a single job. Currently there are two methods of *multi-tasking*:

1. *Macrotasking* where subroutines are restructured — for example in a weather-forecasting program one processor might deal with one hemisphere and a second with the other.

2. *Microtasking* where processor directives are inserted before DO loops (preferably outer ones), and different processors execute different iterations of the loop.

The former is often more efficient on a large program, but requires considerably more programmer intervention. The latter is easy to do but may not be as efficient unless there is a small number of time-consuming loops. Soon (late 1987?) there will be an automatic partitioning option for CFT77 which will use micro-tasking techniques on nested loops.

### 4.9 Modes

CFT77 supports a fast compile mode for debugging and the system provides a symbolic debugger. The debugger can be run either interactively or be used as a dump analyzer after a crash. CFT77 optimizes globally over an entire subroutine. This may cause problems for those used to traditional compilers, as dead code is eliminated and variables that are never used are never stored. Thus, in a subroutine like

```fortran
SUBROUTINE DEAD(A, N)
REAL A(N)
DO 10 I = 1,N
   T = SIN(A(I))
   A(I) = SQRT(T) + EXP(T)
10  CONTINUE
END
```

the variables T and I will be maintained entirely in registers, and not appear in memory. The debuggers will then fail to find any trace of them if there is a program crash.

### 4.10 Problems

Currently CFT77 has some problems with apparent dependencies. A loop like

```fortran
DO 10 I = 1,N
10  A(I,J) = (A(I, J-1) + 1.0)
```
does not have any dependencies, but nevertheless such loops sometimes fail to vectorize because the analyzer cannot handle the J and J − 1 conflict. This is being improved, but for the moment an IVDEP directive might be necessary.

4.11 Memory access and I/O

In general, even strides in loops should be avoided as the four-way memory interleaving factor is low enough to cause a reduction in the bandwidth if only one half or one quarter of the sets of banks is addressed. In addition, the rather low total I/O bandwidth of the computer can lead to bottlenecks which can only be relieved for a single program by use of asynchronous I/O features (BUFFER IN/ BUFFER OUT), in which the I/O takes place overlapped with computation. Small records are automatically buffered by the system. Using control options to make the file buffer size large can sometimes lead to greater efficiency at the cost of increased memory occupancy. Using buffered I/O with unblocked files is the fastest mode.

5. Summary

This computer system is straightforward to use, and program conversion usually presents no great problem. However, full advantage of its power can only be taken for certain types of code, and considerable effort is sometimes required to obtain really large gains compared to the performance of a serial machine, especially by avoiding recursive or conditional code in inner loops. This effort should clearly be expended only in the hot-spots.

REFERENCES


