Algorithmic Languages, de Bakker/van Vliet (eds.)
© IFIP, North-Holland Publishing Company, 1981, 99–114

The Design of Vector Programs

Alain Bossavit and Bertrand Meyer

Direction des Etudes et Recherches, Electricité de France, Clamart, France

Current vector computers such as the Cray-1, Cyber 205 S1, DAP or BSP pose a special challenge to the software designer as the available software tools and techniques are far behind the hardware developments, and the goals of efficient vector programming seem to conflict with some of the basic principles of good software engineering. After studying some properties of these computers, with particular emphasis on the Cray-1, we purport to show that a systematic approach to vector programming is possible and fruitful; the proposed methods are applied to the systematic, proof-oriented derivation of several vector algorithms. Language aspects are also considered.

1. Introduction

The advent of 'second-generation' vector processors [8] such as the Cray-1, CDC Cyber 205, Lawrence Livermore Laboratory S1, ICL DAP and Burroughs BSP, is one more piece of evidence for the fact that software lags far behind hardware as far as practical industrial usage is concerned. These computers, built with the latest LSI or VLSI technology in highly optimized architectures, are capable of achieving speeds which were unheard of before: for example, a Cray-1 computer will in good conditions carry out more than 100 million 'actual' operations, excluding control, per second. On the other hand, a look at the software provided with these 'super-computers' will show them to be what may be called **Fortran machines:** even though processors for other languages may exist, these computers are obviously tailored to a philosophy of programming which has the static array as its only data structure and the DO-loop as its main control structure. Recipes given for writing efficient programs in that framework [6], seem at first glance to be very far from modern ideas about programming, if not incompatible with them.

Vector programming thus appears as a challenge for the software specialist. Areas where advances are needed include the following interrelated topics:

(1) algorithmics (algorithms for vector processing, and methods for finding such algorithms);

(2) program design (how to find program and data structures which will lead to efficient use of supercomputers while ensuring other program qualities such as reliability, clarity, portability, modularity, etc.);

(3) program transformation (methods for adapting existing programs to efficient execution on vector computers);

(4) languages for vector programming;

(5) proof methods.

The aim of this paper is to lay some foundations for a systematic treatment of vector programming. It is mostly concerned with (1) and (2), with a brief discussion of (4).

The particular machine which motivated this study is the Cray-1 computer, which seems to be the most widely available among the 'second generation' vector machines, and is quoted as the fastest currently available computer, even in scalar mode [4, 8]. Most of the discussion is, however, also valid for the other machines.

In Section 2, we give a software interpretation of the rules which must be obeyed by a computation in order to be able to use the vectorization capabilities of the hardware. In Section 3, we give a more abstract interpretation of these rules in terms of the data types involved. Section 4 discusses language problems. Section 5 is devoted to a study of systematic program construction techniques applied to vector programming; several algorithms, in particular a 'vector Cholesky', are derived.

2. Rules for Vectorization

Vector machines require that a program satisfy certain conditions in order to be vectorizable, i.e. amenable to processing in vector, as opposed to scalar, mode. The study of these conditions is particularly interesting in the case of vector computers such as the Cray-1 or BSP which accept standard FORTRAN, so that vectorization rests with the compiler rather

than the programmer. Abstracting from machine peculiarities, five basic conditions appear as necessary and sufficient:
- repetitive series of operations;
- primitive operations only;
- regularity;
- no backward dependency;
- no cross dependency.

These conditions are studied in [12] for the Cray case. We shall outline them here in general terms.

2.1. *Repetitive series of operations*

The only sequences amenable to vectorization are loops, and, more precisely, ***for*** loops, i.e. counter loops with a number of executions known at the outset. The ***for*** loop control structure, associated with the array data structure, is the software representative of the so-called SIMD (*Single Instruction stream, Multiple Data stream*) mode of restricted parallelism.

2.2. *Primitive operations only*

With some slight extensions, only assignments and numerical or boolean operations are allowed in a vector loop. This precludes in particular jumps, thence conditional statements other than conditional assignments. The Cray-1 Fortran compiler (CFT) will also inhibit vectorization of a loop containing a subprogram call (except the subprogram is known to CFT as having a vector version) or another loop (thus restricting vectorization to the innermost loops).

2.3. *Regularity*

For a loop to be vectorizable, it must involve only 'regular' array elements, i.e. elements whose indices follow a strictly defined pattern, so that they can be fetched in advance for vector operations. On the Cyber 205, the only regular elements are those which are stored contiguously; on the Cray-1, a sequence is regular iff the distance between successive elements is constant (but not necessarily 1). Thus only certain types of subarrays may be processed in vector mode.

2.4. *No backward dependency*

Let a loop with i as a counter contain the following array element assignment:

$$a[f_0(i)] := \boldsymbol{op}(b_1[f_1(i)], b_2[f_2(i)], \ldots, b_m[f_m(i)])$$

where ALGOL-like brackets are used for array elements, ***op*** is some numerical or logical operation, the f_k's are linear functions (from the regularity rule), and all arrays are considered as one-dimensional (which is always possible on a machine with a linear store).

This assignment has a backward dependency, which will inhibit vectorization, iff for some k $(1 \leq k \leq m)$ b_k is a, and for some pair of values p, q in the range of i, the following holds:

$$p < q \ \textbf{\textit{and}}\ f_k(p) = f_0(q).$$

In other words, the computation of $a[f_0(q)]$ will use the value of another element of a, which was fetched for updating in some previous iteration. For example, the assignment $a[i] := a[i-1]+1$ introduces a backward dependency.

The reason for this rule is that the vector interpretation of such a computation would use the old value of the array element, not the new one as in the standard (sequential) interpretation of the loop.

Note that the vector interpretation makes perfect sense; it is only different from the sequential one.

On the Cray-1 the condition is less stringent; a backward dependency will actually arise only if the above condition holds together with

$$q - 64 < p$$

where 64 is the length of the vector registers, which on the Cray must be used for the operands and results of vector operations (in contrast, the Cyber 205 and BSP work directly on vectors stored in memory). Vector processing on the Cray-1 may be considered, for all practical purposes, as successive processing of 64-element vector slices, all elements in a slice being processed in parallel.

An important case of backward dependency occurs when the dependency affects a simple variable (which may be considered as a one-element array, whose index is constant through the loop), i.e. when the loop contains an assignment of the form

$$x := \boldsymbol{op}(x, b_1[f_1(i)], b_2[f_2(i)], \ldots).$$

Such an operation is called a **reduction**; it is particularly unfortunate that it should not vectorize, since it corresponds to the very common case of accumulating a result into a variable, as in the computation of the sum of the elements of a vector, or of the scalar (inner) product of two vectors. In practice, techniques exist for reducing the loss of efficiency of reductions as compared to truly vectorizable operations; reductions may thus be thought of as 'pseudo-vectorizable' operations who execute more slowly than vectorizable operations but faster than scalar ones.

2.5. *No cross dependency*

Let a loop contain the following assignments:

$$a[f_0(i)] := \boldsymbol{op}(\ldots);$$

$$c[g_0(i)] := \boldsymbol{op}'(\ldots, a[g_l(i)], \ldots).$$

They induce a cross dependency, which will inhibit vectorization, iff for some pair of values p, q in the range of i, the following holds:

$$g_l(p) = f_0(q)$$

with $|q-p| < 64$ (on the Cray-1).

For example, the following statements in a loop on i will cause a cross dependency:

$$a[i] := 1; \qquad c[i] := a[i+1].$$

The rule stems from the fact that, due to the limited size of the instruction buffers, long loops may have to be split into several shorter ones in order to be vectorized (by slices of 64 on the Cray); thus the two assignments might end up in two different loops, giving a different semantics for the program. In our example, assuming a was initially all 0, then c would receive the previous null values in the sequential case and the new unity values in the vector case.

3. Basic Thoughts for a Vector Programming Methodology

Considering the preceding rules, even though they do not include many details which may be found in manufacturers' documentation, it is quite tempting to dismiss them as too low-level and machine-dependent, and assert that vector programming is just programming with objects of data type 'vector'. Although we will use this definition as the basis for our approach to vector program construction, it should be pointed out that it is not quite sufficient and that the previous rules, especially the last ones on dependency, must also be taken into account for practical purposes.

Let us illustrate this point with an important vector algorithm: matrix multiplication. Assume c is initialized to zero; a, b, c have dimensions (m,n), (n,p) and (m,p) respectively. The ordinary algorithm will not vectorize (notations are mostly taken from [11]):

```
for i in 1.. m do
    for j in 1.. p do
        for k in 1.. n do                                   (3.1)
            $c[i,j] := c[i,j] + a[i,k] * b[k,j]$
```

In terms of the preceding rules, we may say that $c[i,j]$ has a backward dependency on itself (the last line is a reduction). Now if we reverse the loops on j and k, the program becomes vectorizable. This in fact means that instead of the 'element' formula which forms the basis for algorithm (3.1):

$$c[i,j] = \sum_{k=1}^{n} a[i,k] * b[k,j]$$

one relies on the 'vector' formula

$$c[i,*] = \sum_{k=1}^{n} a[i,k] * b[k,*]$$

(where $x[i,*]$ and $x[*,j]$ respectively denote the ith line and jth column of matrix x).

However, if we applied a purely functional view of vector programming, i.e. obtained a program directly from an 'abstract data type' specification of matrix multiplication, the initial version of our program, as deduced from the last formula, would require, for each line i, n vector variables:

$$c_1[i,*] := a[i,1] * b[1,*];$$
$$c_2[i,*] := a[i,2] * b[2,*] + c_1[i,*];$$
$$\dots$$
$$c_m[i,*] := a[i,m] * b[m,*] + c_{m-1}[i,*];$$
$$c[i,*] := c_m[i,*].$$

For practical reasons (storage) this is excluded; the same variable $c[i, *]$ has to be used all along. This programming simplification is correct because it does not conflict with the no backward dependency rule, as every operation of the form

$$c[i, *] := op(c[i, *])$$

will be implemented as a counter loop whose body is $c[i,j] := op(c[i,j])$ without any reference to $c[i, l]$ for $l \neq j$ (note that the loop counter here is j). This condition guarantees that the vectorized form of the new version (i.e. the standard program where loops on j and k have been interchanged) is indeed semantically equivalent to the standard program.

Such a condition, which is more restrictive but conceptually simpler than the no backward dependency rule, may be used as a replacement for it in a systematic approach. It can be formalized in the following way, inspired from the presentation of sequences in the specification language Z [1]. Let $VEC\ X[(n)]$, for $n \in \mathbb{N}$ (the set of n-vectors of elements of X) be defined as the set of all total functions from $1, \ldots, n$ to X. Let & be the functional binary operator such that, if f and g are two functions with the same domain Y, then $f \& g$ is the function h such that, for any $y \in Y$, $h(y)$ is the pair $(f(y), g(y))$. Then for any binary operation p on X ($p : X \times X \rightarrow Z$ for some Z) we may define a vector **extension** of p, $ext(p) : VEC[X](n) \times VEC[X](n) \rightarrow VEC[Z](n)$, whose value for any two vectors v and w in $vec[X](n)$ is

$$ext(p)(v, w) = p \circ (v \& w)$$

where $\circ$ is functional composition; in other words, for any $i \in 1, \ldots, n$,

$$ext(p)(v, w)(i) = p(v(i), w(i)).$$

It is possible to define in the same way (at least if p is associative) a vector **reduction** of functionality

$$red(p) : VEC[X] \rightarrow X$$

where $red(+) = \sum$, etc.

We shall interpret the rules of Section 2 as implying that, in designing programs for vector computers, one should work on objects of data type vector, restricting oneself to extension operations as much as possible. When an extension operation cannot be applied, a reduction will still be preferable to operations which would perform arbitrary shifting of indices (e.g. $p \circ ((v \circ pred)\&w)$, where *pred* is the predecessor function on integers, which would give $p(v(i-1), w(i))$ for any i); such operations would introduce hopeless backward dependencies.

The situation may be depicted using a hierarchy of abstract machines (Fig. 1). At the matrix level, machine MAT offers the operations of matrix algebra: multiplication, inversion, etc. At the vector level, several machines are available to implement these operations: the extension machine EXT, the reduction machine RED, and others. Choosing one of them will lead to a definite algorithm , the scalar machine SCAL, which corresponds to conventional programming languages. It is clear that the standard matrix multiplication algorithm given above (3.1) stems from the RED machine, while its vectorizable counterpart will come out naturally if one uses the EXT machine.

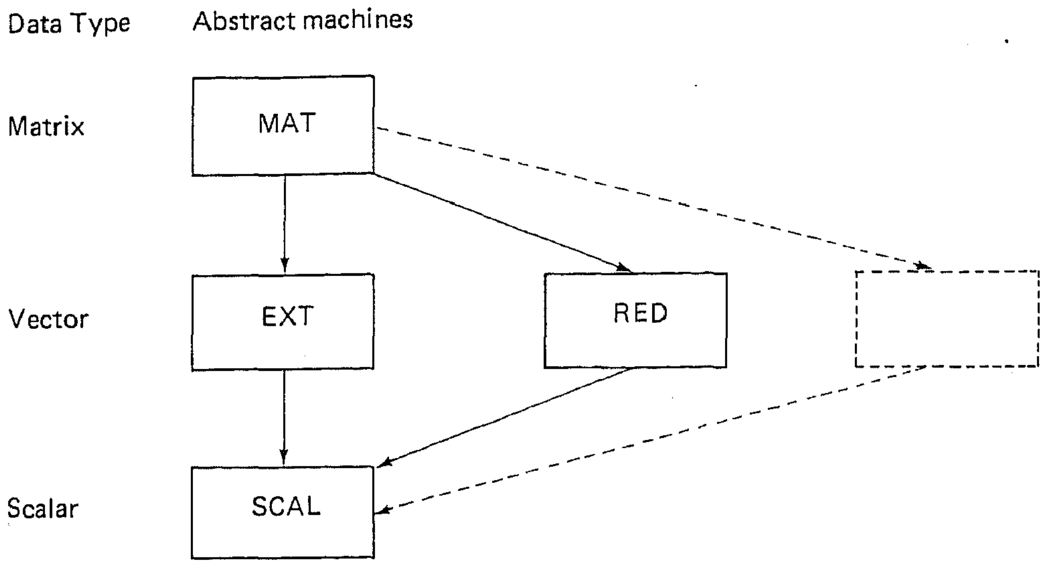

Fig. 1. Hierarchy of types and virtual machines.

Using the above approach, we will derive vector algorithms by working on vector objects from the beginning. This should lead to programs which are both properly structured and efficient on a vector processor. This should be contrasted with the results obtained through more 'ad hoc' methods. For example Higbie [6], in a paper on how to write code which will vectorize on the Cray, warns that 'overly modular or structured programs' will not be vectorizable (because of the rule which we called 'primitive operations only', precluding subprogram calls inside a vectorizable loop). If this were true, the situation might be considered quite sad for the programmer, forced to choose between structure and vectorization. On the other hand, if one agrees that a program is 'structured' at least as much from its proper adequation of control structure to data structure as from its observance of rules regarding control structure only (e.g. many subprograms, etc.), then the answer is clear: rather than in-line expansion of subprogram calls in loop bodies, one should strive to write subprograms working on entire arrays (to use expressions found in Cray publications, "put the loop in the subroutine rather than the subroutine in the loop"). This will, in effect, implement the 'vector' data type abstraction. If the program is indeed vectorizable, i.e. if it does have vectors as its principal objects, there is a good chance that the version thus 'vectorized' will be clearer and better 'structured' independently of any machine consideration.

4. Language Considerations

Before we turn to the derivation of a few vector algorithms, we must pay some attention to language issues. The Cray approach uses a standard language, FORTRAN, and places the task of detecting vectorizable portions of code upon the compiler. The BSP also has a 'vectorizer' for standard FORTRAN code (an introduction to the techniques used for such program transformations may be found in [10]). Other methods have been used or suggested (see [9] or [14] for a survey); for example, the Cyber 205 supercomputer only vectorizes calls to special array processing subroutines. Perrott [14, 15, 16] has argued repeatedly in favor of using a language designed specifically for vector programming; he describes such a language, ACTUS, based on PASCAL. This approach can be justified on several grounds:

- In the Cray and BSP approach to optimization, the programmer has to present his code in a 'favorable' way so that the compiler will be able to detect vectorizable pieces of code; he thus has to know the compiler's idiosyncracies in this respect. This, however, has to be balanced with the considerations on program structuring expressed above.
- The search for vectorizable code amounts to de-compilation (reconstructing higher-level vector constructs, such as they might be expressed in ALGOL 68, PL/I or APL, from lower-level FORTRAN scalar operations), which is a rather silly activity;
- It is quite natural to specify the amount of allowable parallelism in

connection with the data structure definition rather than with the description of the operations performed on it.

On the other hand, the 'vector language' approach seems extremely difficult to implement in the context of a large scientific computing center (the typical target for supercomputers), where it is not realistic to imagine that programmers will turn to a new language for every new kind of application and every new machine – especially at a time when concerns for portability are at last making their way into the scientific programming community.

Given the failures experienced by all previous efforts to impose languages other than FORTRAN to this community, it is doubtful that a proposal applying to vector computers would succeed. In view of the current state of the art, the Cray approach seems sensible as far as program **coding** is concerned. Languages such as ACTUS may, however, be very useful as intermediary notations for vector program **design,** and we shall use similar ways of expression in the examples which follow.

5. Examples of Systematic Vector Program Construction

We turn now to the application of the principles expounded in Section 3 to the construction of some practical programs. We shall use a method and set of heuristics for constructing programs from specifications which were exposed in [13]. A similar approach was applied to classical (scalar) numerical algorithms in [2].

The following notation will be used in addition to the ones defined in Section 3:
- $VEC(n)$ stands for $VEC[REAL](n)$, the set of vectors of n real elements;
- $MTR(m,n)$ is the set of (m,n) real matrices;
- $P_l v$, where $v \in VEC(n)$ and $l \leq n$, is the projection of v on $VEC(l)$.

For a matrix $s \in MTR(m,n)$, if $i \leq m$ and $j \leq n$, we will consider line $s[i, *]$ and column $s[*, j]$ as vectors in $VEC(m)$ and $VEC(n)$ respectively.

5.1. *Triangular systems*

We saw in Section 3 a vector algorithm for matrix multiplication. Let us proceed with the inverse operation: solving linear systems. We first examine triangular systems. This will be a simple example of top-down synthesis of a numerical algorithm.

The first step in the design of the program (called *trisolv*) is to express it as a matrix algorithm (which could run on the virtual machine MAT):

in s: $MTR(n,n)$, b: $VEC(n)$; ***out*** x: $VEC(n)$;

(P) $\{1 \le i \le n \Rightarrow P_{i-1} s[*, i] = 0$ ***and*** $s[i,i] \neq 0\}$

trisolv

(Q) $\{sx = b$, i.e. $\sum_{k=1}^{n} s[*, k] * x[k] = b\}$

We must refine *trisolv* into a predicate transformer (on the vector machine EXT) from the precondition (P) to the postcondition (Q). Let us try twice the heuristic called 'uncoupling' [13], i.e. add an auxiliary vector variable y, and an integer one l, noticing that

$$\begin{aligned}(Q) \Leftrightarrow b = \sum s[*, k] * x[k] &\Leftrightarrow (y + \textstyle\sum_{k \le n} s[*, k] * x[k] = b \;\boldsymbol{and}\; y = 0)\\ &\Leftrightarrow (y + \textstyle\sum_{k \le l} s[*, k] * x[k] = b \;\boldsymbol{and}\; P_l y = 0)\\ &\quad \boldsymbol{and}\; l = n.\end{aligned}$$

So (Q) $\Leftrightarrow (I(l)$ ***and*** $l = n)$ if we set $I(l) =$ the first term of the ***and*** above. Here, $I(l)$ is a 'weakening' of the exit condition (Q) (which is $I(n)$). We notice that $I(0)$ can be trivially obtained. Thus a refinement of *trisolv*, using $I(l)$ as an invariant and $l = n$ as the goal (exit condition) will be:

```
var $l$: Integer;
$l := 0$; $y := b$ $\{I(l)\}$
while $l < n$ do
  $l := l + 1$;
  reestablish $I(l)$;
$\{l = n$ and $I(l)\}$
```

This program is correct (by construction): $I(l)$ being a loop invariant, it is true after the completion of the loop, and the exit condition $l = n$ is also true, hence $I(n)$. The statement *reestablish* is now (just as *trisolv* was, one step backwards) a specification for what is to be done.

Next step: develop *reestablish*. One must go from $I(l-1)$, i.e.

$$y + \textstyle\sum_{k<l} s[*, k] * x[k] = b \;\boldsymbol{and}\; P_{l-1} y = 0$$

to $I(l)$, i.e.

$$y + \textstyle\sum_{k<l} s[*, k] * x[k] = b - s[*, l] * x[l] \;\boldsymbol{and}\; P_l y = 0.$$

Without modifying b, which is part of the input, we must use the assignment $y := y - s[*, l] * x[l]$ after an $x[l]$ such that $P_l(y - s[*, l] * x[l]) = 0$ has been found. But $P_{l-1}s[*, l] = 0$ by hypothesis, and $P_{l-1}y = 0$ also. The equation thus becomes $y[l] - s[*, l] * x[l] = 0$, thence $x[l]$. The final version of the program is:

```
$l := 0$; $c := b$; $I(0)$
**while** $l < n$ **do**
    $l := l + 1$;
    {*reestablish* $I(l)$ :}
        $x[l] := y[*, l]/s[l, l]$
        $y := y - s[*, l] * x[l]$
```

Starting from a matrix specification and aiming at the EXT vector target machine, we have just synthesized a program which must be, by construction, vectorizable.

5.2. *Vectorized Choleski*

We shall now introduce a more difficult algorithm, Choleski factorization: given a symmetric positive-definite matrix A, find a lower triangular S such that $SS^t = A$ (in view of the resolution in two easy steps, using e.g. the above program, of the linear system $Ax = b$). What follows is also valid for the LU factorization.

We again apply systematic top-down synthesis. Here are the successive steps. First the specification, expressed in terms of MAT objects:

```
        **in** $a$: $MTR(n, n)$; **out** $s$: $MTR(n, n)$;
(R)     {*symmetric*($a$) **and** *positive-definite*($a$)}
            *Choleski*
            {$1 \le i \le n \Rightarrow P_{i-1}[*, i] = 0$}
(S)         {$A = SS^t$, i.e. $a = \sum_{k \le n} s[*, k] * s[*, k]$}
```

As before, we uncouple (S), after introducing the auxiliary variable c of type $MTR(n, n)$:

$$(S) \Leftrightarrow ((c + \textstyle\sum_{k \le l} s[*, k] * s[*, k] = a \textbf{ and } P_l c = 0) \textbf{ and } l = n)$$

$$\Leftrightarrow \qquad (I(l) \textbf{ and } l = n).$$

The next refinement is, quite naturally:

```
$l := 0$; $c := a$; $\{I(0)\}$
**while** $l < n$ **do**
    $l := l+1$; $\{c + \sum_{k<l} = a$ **and** $P_{l-1}c = 0\}$
    *reestablish* $I(l)$;
    $\{c + \sum_{k<l} = a - s[*, l] * s[*, l]$ **and** $P_l c = 0\}$.
```

To *reestablish* $I(l)$, one must perform the assignment $c := c - s[*, l] * s[*, l]$ once an $s[*, l]$ such that $P_{l-1}s^l = 0$ and

$$P_l(c - s[*, l] * s[*, l]) = 0$$

has been found. As $P_{l-1}c = 0$, row l is the only one concerned, and must satisfy *l-column*$(c - s[*, l] * s[*, l]) = 0$, that is to say $c[l, *] - s[l, l] * s[*, l] = 0$, which implies ($l$ component)

$$c[l, l] = (s[l, l])^2.$$

Thence the two instructions for *reestablish* $I(l)$:

$$s[l, l] := sqrt(c[l, l])); \; s[*, l] := c[l, *]/s[l, l].$$

As c is symmetric (this fact is itself a loop invariant), $P_{l-1}c[*, l] = 0$ implies $P_{l-1}c[l, *] = 0$, therefore $P_{l-1}s[*, l] = 0$.

The final version will thus be:

```
$l := 0$; $c := a$;
**while** $l < n$ **do**
    $l := l+1$;
    $pivot := sqrt(c[l, l])$;
    $s[*, l] := c[l, *]/pivot$;
    $c := c - s[*, l] * s[*, l]$
```

A FORTRAN translation appears on Fig. 2 and 3. It exhibits some of the nice properties of programs resulting from top-down design (high-level built-in documentation, etc.) and the safety guaranteed by the systematic synthesis method.

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE
     *                           C H O V E C
     *                         (N, A, S, NDP)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                      C
C                              PURPOSE:                                C
C                                                                      C
C     FACTORIZATION OF A SYMMETRIC MATRIX, VECTORIZABLE VERSION.       C
C                                                                      C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C     INPUT
C
          INTEGER     N
C                                  Order of the matrix A
          REAL        A    ( 1 )
C                                  Array of the entries of A. Aij is at
C                                  the position ((J - 1)(2N - J) + 2I)/2.
C                                  ('column-symmetric storage mode')
C     OUTPUT
C
          REAL        S    ( 1 )
C                                  Array of the entries of A. Aij is at
C                                  the position ((J - 1)(2N - J) + 2I)/2
C                                  On exit, if NDP = N, A = S tr(S)RT
          INTEGER     NDP
C                                  Number of columns actually taken into
C                                  account during the factorization.
C                                  If NDP < N, a non-positive radix ap-
C                                  peared in the treatment of column
C                                  NDP + 1
C-----------------------------------------------------------------------
C
C     LOCAL VARIABLES:
C
          INTEGER L, NNP1S2, ADRLL, ADRJL, ADRJJ, I, J, LP1
          REAL PIVOT, MUL, RADIC
C
C     ARITHMETIC FUNCTION:
C
          INTEGER ADDRESS
          ADRESS(I, J) = ((J - 1)*(2*N - J) + 2*I)/2
C
```

Fig. 2. Head of the vectorizable Choleski program (FORTRAN).

6. Conclusion

The field of numerical and scientific programming, although the oldest and one of the best established among the application domains of computers, has shown strong resistance to the practical implementation of software research and advances in programming methodology. With the

```
C------------------------------------------------------------------------
C
          NDP = 0
C     i <--- 0 ;
          L = 0
C     C <--- A ;
          NNP1S2 = (N*(N + 1))/2
          DO 1 I = 1, NNP1S2
1              S(I) = A(I)
C              -- The array S contains both C and A.
C     while i < n do
2     IF (L 'GE' N) GOTO 7
C         i <--- i + 1 ;
               L = L + 1
               ADRLL = ADRESS(L, L)
C         pivot <--- sqrt(C11) ;
               RADIC = S(ADRLL)
               IF (RADIC 'LE' 0') GOTO 7
C                   -- Exception if A is not positive definite
               PIVOT = SQRT(RADIC)
               NDP = L
C         S1 <--- C1/pivot ;
               DO 3 I = L, N
3                   S(ADRLL + I - L) = S(ADRLL + I - L)/PIVOT
C         C <--- C - S1 * S1 ;
               LP1 = L + 1
               IF (LP1 ' EQ' N) GOTO 6
               DO 5 J = LP1, N
                    ADRJJ = ADRESS(J, J)
                    ADRJL = ADRESS(J, L)
                    MUL = S(ADRJL)
                    DO 4 I = J, N
                         S(ADRJJ+I-J) = S(ADRJJ+I-J) - MUL*S(ADRJL+I-J)
C                        -- This loop is the only vectorizable one
4                        CONTINUE
5                   CONTINUE
6              CONTINUE
          GOTO 2
7     RETURN
      END
```

Fig. 3. Body of the Choleski program.

popularization of new 'number-crunching' machines, there is again a strong temptation to go back to low-level, machine-dependent, programming techniques, and to dismiss any attempts at better software engineering as incompatible with the efficient use of these very fast computers. We hope to have shown that such an attitude has no justification, and that systematic methods can be applied for the rational and efficient use of this new technology.

References

[1] J.R. Abrial, S.A. Schuman and B. Meyer, Specification language, in: Proceedings Summer School on Program Construction, Belfast (September 1979).

[2] A. Bossavit and B. Meyer, On the constructive approach to programming: the case for partial Choleski factorization (a tool for static condensation), in: Vichnevetsky and Stepleman (Eds.), Advances in Computer Methods for Partial Differential Equations III (IMACS, 1979).

[3] Cray-1 Computer System, FORTRAN (CFT) Reference Manual, Cray Document No. 2240009, Version E (1981).

[4] M. Dungworth, The Cray 1 computer system, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited papers (Maidenhead, 1979) pp. 51–76.

[5] P.M. Flanders, FORTRAN extensions for a highly parallel processor, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited Papers (Maidenhead, 1979) pp. 117–134.

[6] L. Higbie, Vectorization and conversion of FORTRAN programs for the Cray-1 (CFT) compiler, Cray Document No. 2240207 (June 1979).

[7] Infotech State of the Art Report on Supercomputers, Volume 1: Total Systems Issues; Volume 2: Invited papers (Maidenhead, 1979).

[8] E.W. Kozdrowicki and D.J. Theis, Second-generation of vector supercomputers, Computer (IEEE), Special Section on Sypersystems for the 80's, 13 (11) (1980) 71–83.

[9] D.J. Kuck, Languages and compilers for parallel and pipeline machines, in: CREST Conference on Design of Numerical Algorithms for Parallel Processing, Bergamo, Italy (June 1981).

[10] D.J. Kuck, Automatic program restructuring for high-speed computation, in: W. Händler (Ed.), CONPAR 81, Nürnberg, June 1981, Lecture Notes in Computer Science 111 (Springer, Berlin, 1981) pp. 66–84.

[11] B. Meyer and C. Baudoin, Méthodes de programmation (Eyrolles, Paris, 1978).

[12] B. Meyer, Un calculateur vectoriel: Le Cray-1 et sa programmation, EDF Report HI/3452-01, Atelier logiciel No. 24 (May 1980).

[13] B. Meyer, A basis for the constructive approach to programming, in: S.H. Lavington (Ed.), Information Processing 80 (North-Holland, Amsterdam, 1980).

[14] R.H. Perrott, Parallel languages, in: Infotech State of the Art Report on Supercomputers, Volume 1: Total Systems Issues (Maidenhead, 1979) pp. 117–149.

[15] R.H. Perrott, A standard for supercomputer languages, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited Papers (Maidenhead, 1979) pp. 291–308.

[16] R.H. Perrott, A language for array and vector processors, TOPLAS (Transactions on Programming Languages and Systems, ACM) 1 (2) (1979) 177–195.