Computer Physics Communications 37 (1985) 27–38
North-Holland, Amsterdam

AN APPLICATION OF PROGRAM TRANSFORMATION TO SUPERCOMPUTER PROGRAMMING

Alain BOSSAVIT

Electricité de France, Direction des Etudes et Recherches, Service IMA, 1 Avenue du Général de Gaulle, 92141 Clamart, France

and

Bertrand MEYER *

Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

We show how a sequence of systematic program transformations can be used to derive an efficient, vectorizable program (to be used on vector computers such as the Cray machines) from an initial version which is mathematically simple but recursive and very inefficient.

The example chosen is that of cyclic reduction. We start with a description of the algorithm which follows directly from a mathematical analysis of the problem and is expressed in terms of operations of the "vector machine", specified as an abstract data type; we end up with an Ada package.

We discuss the advantages and limitations of Ada with respect to vector programming and raise some issues concerning the use of program transformations in software design methodology.

1. Background

In previous work, we have investigated the application of modern software engineering techniques to the design of vector programs (e.g. refs. [15,5,6,7], etc). Our general approach has been to investigate supercomputer programming not as a set of recipes designed to yield maximum performance on some or other specific machine architecture, but rather as a systematic design activity, in which the concern for efficiency must not offset other important software qualities such as correctness, reliability, extensibility, portability and others.

Techniques which can be applied towards this goal include assertion-guided stepwise program development [10] and the use of abstract data types for the specification of "virtual vector machines" as models of actual vector processing hardware.

This paper continues our previous efforts by studying the application of another well-known program construction method, program transformation, to the development of an efficient vector program corresponding to an important algorithmic concept, cyclic reduction. We start from a correct but very inefficient program, obtained as a straightforward implementation of the basic mathematical idea and expressed in terms of high-level operations of the abstract "vector machine"; we then perform a series of transformations, each aimed at removing some of the inefficiency while preserving the semantics of the program. The final version, for which we offer an Ada implementation, is an efficient, readily vectorizable program.

2. The total reduction problem

2.1. Statement of the problem

Consider a set S with a binary operation, written $\oplus$, which gives S the structure of a monoid, i.e. $\oplus$ is associative and has a zero element, writ-

* On leave from EDF, Clamart, France.

0010-4655/85/$03.30

ten 0. Note that $\oplus$ is *not* required to be commutative. Elements of S will be called **scalars** *.

We define $V = VECTOR[S]$, the set of finite sequences of elements of S. An element v of V, called a **vector**, is of the form

$$v = \langle v_1, v_2, ..., v_n \rangle,$$

where $v_i \in S$ for $i = 1, 2, ..., n$. The number of elements of a vector v is written $|v|$.

We define the **shift** operation

$$\tau : V \to V$$

such that

$$\tau(\langle v_1, v_2, ..., v_n \rangle) = \langle 0, v_1, v_2, ..., v_n \rangle.$$

The total reduction problem is, given a vector $a \in V$, to find another vector $x \in V$ such that

$$x = a \oplus \tau x \tag{1}$$

which can also be written, in scalar terms:

$$x_1 = a_1, \quad x_i = a_i \oplus x_{i-1},$$

or equivalently:

$$x_i = a_i \oplus a_{i-1} \oplus a_{i-2} \oplus \ldots \oplus a_1$$
for $i = 1, 2, ..., |a|$.

2.2. Applications

The total reduction problem, as defined by (1) above, has several applications. The most obvious ones are the sum of the elements of a, obtained by taking ordinary addition for $\oplus$, and linear recurrences, which may be written as

$$\begin{vmatrix} x_i \\ 1 \end{vmatrix} = \begin{vmatrix} a_i & b_i \\ 0 & 1 \end{vmatrix} \oplus \begin{vmatrix} x_{i-1} \\ 1 \end{vmatrix}$$

which is an instance of the total reduction problem obtained by taking for $\oplus$ the product of 2×2 matrices.

But some classes of non-linear recurrences fall into the same model; a straightforward generalization is

$$x_i = \frac{a_i * x_{i-1} + b_i}{c_i * x_{i-1} + d_i}$$

which can be put into the form of (1) by again taking for $\oplus$ the product of 2×2 matrices and writing the equation as

$$x_i = u_i / v_i,$$

where

$$\begin{vmatrix} u_i \\ v_i \end{vmatrix} = \begin{vmatrix} a_i & b_i \\ c_i & d_i \end{vmatrix} \oplus \begin{vmatrix} u_{i-1} \\ v_{i-1} \end{vmatrix}.$$

A useful particular case where this is applicable is **Cholesky factorization**: consider a symmetric matrix with diagonal

$$\langle d_1, d_2, ..., d_n \rangle$$

and subdiagonal

$$\langle s_1, s_2, ..., s_{n-1} \rangle.$$

The recurrence to be solved for Cholesky factorization is

$$b_{i-1}^2 + a_i^2 = d_i, \quad b_i * a_i = s_i,$$

i.e. by eliminating b_i:

$$a_i^2 = d_i - s_{i-1}^2 / a_{i-1}^2$$

which is a problem of the above form if we take $x_i = a_i^2$.

3. The vector machine

3.1. Vector operations

Eq. (1) does not seem to lend itself naturally to efficient solution on vector processors such as the Cray-1 or Cray-XMP, which favor the execution of "extension" operations [15,5]. Roughly speaking, extension operations are those which can be executed in parallel on all the elements of a vector (or more generally, in the case of the Cray machines, on whole vector slices). A typical extension

* This use of the word "scalar" does not quite conform to standard mathematical usage, but is common in discussions of vector programming.

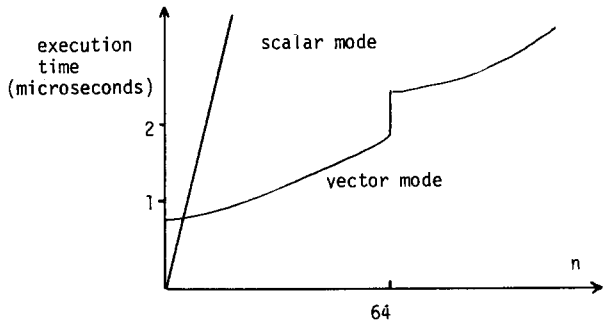

Fig. 1. Performance of vector addition in scalar and vector mode on the Cray-1.

operation is the addition of two vectors, element by element *.

Such operations on vectors may be executed by vector hardware much more efficiently than by just applying repetitively their non-vector, or "scalar" counterparts. More precisely, a scalar operation which takes time S when applied to one element will take time

$$t_{\text{scal}}(n) = n * S$$

when applied to a vector of n elements. A true vector operation, when applied to this vector, will take a time approximately equal to

$$t_{\text{vect}}(n) = U + n * V,$$

where U is the start-up time and V is the asymptotic unit vector time. On a vector machine, of course, V must be significantly less than S.

The performance of vector addition in both scalar and vector mode on the Cray-1 is illustrated in fig. 1. Vector mode becomes better than scalar mode for vector lengths $n > U/(S - V)$. The non-linearity of actual vector processing time, which is apparent on the figure, is due to the fact that the Cray processes vectors by slices of maximum length 64, hence the discontinuity at $n = 64$ (and also 128, 192, etc.).

The performance of an operation executed in vector mode may be characterized by two parameters [13]:

- the asymptotic vector speedup $p = S/V$;
- the "half-performance length" $n_{1/2} = U/V$, defined as the value of n for which the per-element performance is half the asymptotic one, i.e. $(U + n * V)/n = 2 * V$; this parameter gives an idea of the minimum length for which the benefits of vector mode offset the penalty incurred for short vectors because of the startup time.

On a Cray-1, depending on the operation, p varies between 7 and 10 and $n_{1/2}$ between 20 and 30.

Only those parts of a program which conform to certain rules may be executed in vector mode and thus achieve high performance. For Fortran programs on the Cray-1, the rules are the following [15]:

1. only "DO" loops are "vectorizable";
2. these loops may only contain "primitive" operations such as assignment and arithmetic or boolean operations (no jumps, etc.);
3. the data elements accessed during successive loop iterations must be regularly spaced in memory, i.e. array indexes must be linear functions of the loop index;
4. no "backward dependency", in which a statement updates an array value $a(i)$ and uses a previous value of the same array, $a(i - p)$ (for some $p > 0$), is permitted;
5. no "cross dependency", in which an array value may be updated by one statement of the loop and used by another, is permitted.

In the last two cases, vectorization is inhibited by the compiler not because the hardware could not carry out the computation in vector mode, but because the vector semantics of the program may be different from the standard (sequential) semantics implied by Fortran and other common languages. If, on the other hand, one feels certain that the dependency is only apparent, for instance if the element updated in a loop with index i is $a(2 * i + 1)$ and the value used is that of $a(2 * i)$ (so that the array slices updated and used are in fact disjoint), then one may force vectorization; the Cray Fortran compiler will accept a special directive, *IVDEP*, to that effect.

* It should be noted that on the Cray machines or on the CDC Cyber 205 vector operations are not actually performed on all elements in parallel, but rather use pipelining. For most practical purposes, however, pipelining may be considered as a form of parallelism.

The above rather stringent rules seem to preclude the vectorization of many simple algorithms; for example, the formula which we have given for total reduction, i.e. (1) above, clearly implies repeated backward dependencies.

In order to obtain vectorizable versions of this and other algorithms, more perspective is needed on the "vector machine" and the operations it may perform.

3.2. An abstract model

Rather than studying at the scalar (e.g. Fortran) level what can be vectorized and what cannot, it is preferable to provide a formal model of the machine at the appropriate level of abstraction. Here we consider a vector computer as a virtual machine associated with an abstract data type, type $V = VECTOR[S]$, and capable of performing a certain number of operations.

There is in fact probably no such thing as *the* vector machine, but rather various models adapted to various applications. We thus tailor our specification to the problem at hand. Rather than giving a complete formal description of the abstract data type "vector", we concentrate on some useful operations and their essential properties.

On a vector computer such as the Cray-1, all the operations in table 1 (except for "length" and "access to element" which require constant time) are "extension operations" which can be executed in vector mode. It should be noted, however, that some vector computer architectures may be more restrictive: the CDC Cyber 205, for instance, requires array elements to be contiguous not just equally spaced, so that operations such as "odd part", "even part" and "merge" do not qualify.

The above list of operations is by no means exhaustive; more complete lists may be found in e.g. refs. [6,7]. It should also be noted that for some applications it may be useful to introduce operations extracting other "slices" than just the odd and even parts. The operations given here will suffice, however, for our purposes.

Table 1
Operations and their properties

Operation	Type	Notation	Properties
zero vector	V	0	all elements zero
length	$V \rightarrow Integer$	$\lvert v \rvert$	
access to elements	$V \times Integer \rightarrow S$	v_i	
extension of a scalar operation $\oplus$	$V \times V \rightarrow V$	$v \oplus w$	let $z = v \oplus w$: $\lvert z \rvert = min(\lvert v \rvert, \lvert w \rvert)$: $z_i = v_i \oplus w_i$ $(i \in 1 \ldots \lvert z \rvert)$
shift	$V \rightarrow V$	τv	$\lvert \tau v \rvert = \lvert v \rvert + 1$; $(\tau v)_i =$ $v_i - 1$ for $i > 1$, 0 for $i = 1$
odd part	$V \rightarrow V$	Ov	$Ov_i = v_{2i-1}$
even part	$V \rightarrow V$	Ev	$Ev_i = v_{2i}$
merge into odd and even parts	$V \times V \rightarrow V$	*alternate* (v, w)	let $z =$ *alternate* (v, w): $Oz = v$; $Ez = w$

Among the abstract properties of these operations which are particularly interesting are the following (for any vectors v, $w \in V$):

$$E\tau v = Ov, \quad \text{(i)}$$
$$O\tau v = \tau Ev, \quad \text{(ii)}$$
$$O(v \oplus w) = Ov \oplus Ow, \quad \text{(iii)}$$
$$E(v \oplus w) = Ev \oplus Ew, \quad \text{(iv)}$$
$$\tau(v \oplus w) = \tau v \oplus \tau w. \quad \text{(v)}$$

4. Cyclic reduction

The above properties, expressed at the vector rather than scalar level, provide the key to an efficient solution of the total reduction problem (1) by a vector algorithm. The idea to be applied here is a very fruitful heuristics, using the concept of recursion and close to techniques such as "red–black ordering" which can be applied to the development of several efficient vector algorithms.

In the "total reduction" equation

$$x = a \oplus \tau x, \quad (1)$$

let us try to reduce the problem size by a factor of 2 by applying operators O and E (odd and even parts) to both sides, yielding:

$$Ox = O(a \oplus \tau x),$$
$$Ex = E(a \oplus \tau x),$$

i.e. by applying properties (i) to (iv):

$$Ox = Oa \oplus \tau Ex, \quad (2)$$
$$Ex = Ea \oplus Ox, \quad (3)$$

The interesting fact here is that by substituting the value of Ex, as obtained from (3), into (2), and using the associativity of $\oplus$ combined with property (v) above, we obtain a new equality:

$$Ox = (Oa \oplus \tau Ea) \oplus \tau Ox \quad (4)$$

which is a new instance of the total reduction problem, applied to the new vector variable Ox, a being replaced by $Oa \oplus \tau Ea$. This new instance uses vectors of approximately half the size of the original ones.

We thus have the essential ingredients for an efficient recursive algorithm, known as **cyclic reduction**:

- for vectors length 0 or 1, the result x will be just a;
- for larger vectors, we apply the algorithm recursively, using formula (4), to obtain Ox; formula (3) then yields Ex;
- we obtain x by merging these two vectors (*alternate* operator).

5. Program development

5.1. First procedural version

The first version of the procedure is a direct translation of the basic mathematical definition. We use an Ada-like notation.

```
procedure total_reduction1
  (a: in VECTOR; x: out VECTOR)
  var oddpart, evenpart: VECTOR
begin
  if |a| ≤ 1 then
    x := a
  else -- |a| > 1
    total_reduction1 (Oa ⊕ τEa, oddpart);
    evenpart := Ea ⊕ oddpart;
    x := alternate(oddpart, evenpart)
  end if
end procedure -- total_reduction1
```

The above version is correct but grossly inefficient for several reasons:

- the procedure is recursive;
- it has local vector variables (*oddpart* and *evenpart*) which must be allocated anew for each recursive instance of the procedure;
- it uses two parameters, an input a and an output x, whereas in practice one usually prefers to work on a single vector, which is initially the input and will gradually be "transformed" so as to become the output (the initial value being saved if necessary).

We shall get rid of these sources of inefficiency through a stepwise process. To make the successive program transformations clearer, we underline in each version the elements which have been changed from the previous version.

5.2. Removing extra variables

Our first transformation is a straighforward one, which gets us a little closer to our aim of working on a single object (x): we note that it is harmless to begin the procedure by the assignment $x := a$ in all cases, not just when $|a| \leqslant 1$ (in the other case, this assignment will be overridden by the assignments to the odd and even parts of x).

```
procedure total_reduction2
  (a: in VECTOR; x: out VECTOR)
  var oddpart, evenpart: VECTOR
begin
  x := a;
  if |a| > 1 then
    total_reduction2 (Oa ⊕ Ea, oddpart);
    evenpart := Ea ⊕ oddpart;
    x := alternate(oddpart, evenpart)
  end if
end procedure -- total_reduction2
```

The next simplification is to get rid of the local variables *oddpart* and *evenpart* by extending the notation a little: we now allow assigning vector values directly to the slices Ox and Ex of a vector x. For example, to change the even part of x to y, we shall just write

$Ex := y$

instead of

$x := alternate\ (Ox, y)$.

With this new notation, the procedure can be simplified as follows:

```
procedure total_reduction3
  (a: in VECTOR; x: out VECTOR)
begin
  x := a;
  if |a| > 1 then
    total_reduction3 (Oa ⊕ τEa, Ox);
    Ex := Ea ⊕ Ox;
  end if
end procedure -- total_reduction3
```

The next obvious step towards the goal of working with only one vector variable is to replace all occurrences of a with x after the initial assignment $x := a$. We have to be very careful here: in the procedure resulting from such a transformation, the same vector x will be used as both an **in** and **out** actual parameter of the recursive call. It should be noted that Hoare's specification of the semantics of recursive procedures [12] specifically excludes this case.

The replacement will be correct, however, if for the time being we assume a copy mechanism for parameter passing. In other words we take **in** to mean "parameter passed by value", i.e. copied upon each procedure call into a variable local to the procedure instance; and we take **out** to mean "parameter passed by result", i.e. copied back on procedure return, from the local variable. To avoid any confusion resulting from the fact that we are using an Ada-like notation, it should be noted that this mode of parameter passing is *not* the normal Ada mechanism for **in** and **out** parameters.

```
procedure total_reduction4
  (a: in VECTOR; x: out VECTOR)
begin
  x := a;
  if |x| > 1 then
    total_reduction4 (Ox ⊕ τEx, Ox);
    Ex := Ex ⊕ Ox;
  end if
end procedure -- total_reduction4
```

5.3. Isolating the recursion

It is useful now to separate the procedure into two parts: one which uses the initial vector a and one which does not. To this effect, we transform the procedure into a set of two mutually recursive procedures, only the first of which depends on a; the second one, called *internal_part*$_5$, has only x as a parameter, of mode **in out**. Again, this is correct only if we assume a copy mechanism for parameter passing, i.e. an **in out** parameter is copied to (at call time) and from (at return time) a variable local to the procedure instance.

```
procedure total_reduction5
  (a: in VECTOR; x: out VECTOR)
begin
  x := a;
  internal_part5 (x)
end procedure -- total_reduction5
```

```
procedure internal_part5 (x: in out VECTOR)
begin
  if |x| > 1 then
    total_reduction5 (Ox ⊕ τEx, Ox);
    Ex := Ex ⊕ Ox;
  end if
end procedure -- internal_part5
```

We can now isolate the recursion by expanding the call to *total_reduction* in *internal_part*. The effect of this call is to assign the value of the first parameter to the second and to call *internal_part* recursively. By carrying out this expansion, we get rid of the mutual recursion introduced in the previous step: in the new version, only *internal_part* will be (directly) recursive; *total_reduction* remains useful for initialization only.

```
procedure total_reduction6
  (a: in VECTOR; x: out VECTOR)
begin
  x := a;
  internal_part6 (x);
end procedure -- total_reduction6

procedure internal_part6 (x: in out VECTOR)
begin
  if |x| > 1 then
    Ox := Ox ⊕ τEx;
    internal_part6 (Ox);
    Ex := Ex ⊕ Ox;
  end if
end procedure -- internal_part6
```

5.4. Introducing an integer parameter

The remarkable feature of the recursive scheme which we have obtained is that the recursive call now has a single and simple actual parameter, Ox, where the formal parameter was x. Thus the sequence of actual parameters in successive recursive calls, starting with the initial call from *total_reduction*$_6$ will be

$$x = a,\ Ox,\ O^2x,\ ...,\ O^mx,$$

where O^kx $(k \geqslant 0)$ is the kth iterate of O. The value of the exponent for the innermost call is

$$m = 1 + \lfloor \log(|a| - 1) \rfloor,$$

(here and in the sequel, logarithms are in base two; for any real number x, $\lfloor x \rfloor$ denotes the floor of x, i.e. the greatest integer n such that $n \leqslant x$).

This remark suggests a new version in which the explicit parameter to the recursive part is not x itself any more, but k, the number of times operator O must be iterated. Of course all instances of the recursive procedure must be able to work on x; thus, we make x a variable global to the recursive procedure. To this end we make procedure *internal_part* local to the non-recursive procedure *total_reduction*.

```
procedure total_reduction7
  (a: in VECTOR; x: out VECTOR)
  var m: NATURAL -- i.e. non-negative integer;
  procedure internal_part7 (k: in NATURAL)
    -- local to total_reduction7
  begin
    if k ⩽ m then
    O^k x := O^k x ⊕ τE O^(k-1) x:
    internal_part7 (k + 1);
    E O^(k-1) x := E O^(k-1) x;
    end if
  end procedure -- internal_part7
begin -- total_reduction7
  x := a;
  m := 1 + ⌊log(|a| - 1)⌋;
  internal_part7 (1); -- initial parameter is one
end procedure -- total_reduction7
```

5.5. Removing the recursion

These procedures can be further simplified. The body of procedure *internal_part*$_7$ is of the form

```
if k ⩽ m then
  U_k;
  internal_part7 (k + 1);
  D_k
end if
```

where U_k is the statement

$$O^kx := O^kx \oplus \tau\, EO^{k-1}x$$

and D_k is the statement

$$EO^{k-1}x := EO^{k-1}x \oplus O^kx.$$

Thus, the execution of the successive recursive

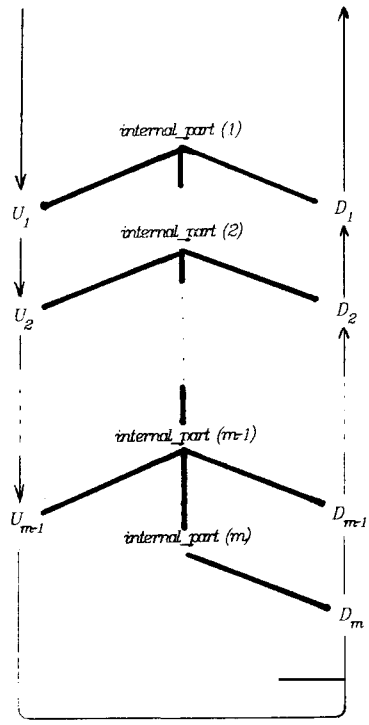

Fig. 2. Procedure *internal_part*$_7$.

calls amounts to a traversal of the above tree in the order indicated by the dotted line (see fig. 2), i.e. the successive execution of

$U_1, U_2, ..., U_{m-1}, D_m, D_{m-1}, ..., D_2, D_1$.

where $m = \lfloor 1 + \log(|a| - 1) \rfloor$. Note that there is one more instance of D_k than of U_k since U_m is a null statement.

Thus, no recursion is needed after all: the body of procedure *total_reduction*$_7$ may be readily represented by

up ; *down*

where *up* and *down* are two simple loops:

```
-- up:
   for k := 1 to m - 1 do
      U_k
   end for;

-- down:
   for k := m downto 1 do
      D_k
   end for;
```

(the mnemonics used for the loops reflect the fact that the index k goes up in the first loop and down in the second one).

It is particularly interesting to note that, although the recursion initially seemed quite necessary, it has been completely removed. The above version is truly non-recursive in that it does not seem to contain any hidden recursive feature, for example a stack lurking in the guise of an integer representing an array of binary values as in some iterative implementations (see e.g. ref. [14]) of the Tower of Hanoi, Quicksort, the Deutsch–Schorre–Waite tree traversal algorithm, etc.

6. A scalar vectorizable version

6.1. The program

It is useful to write U_k and D_k in a form which is closer to how they would be expressed in an ordinary (scalar) programming language, but still easily amenable to automatic vectorization. We define

slice (*low*, *high*, *step*),

where *low*, *high* and *step* are integers such that *low* $\leqslant$ *high* and *step* > 0, as the set of all integers of the form

low + *k* * *step*

which fall into the range *low..high*. Then U_k and D_k can be written as follows:

```
-- U_k (i.e. O^k x := O^k x ⊕ τEO^{k-1} x):
   forall i in slice (1 + 2^k, |a|, 2^k) do
      x[i] := x[i] ⊕ x[i - 2^{k-1}]
   end forall

-- D_k (i.e. EO^{k-1} x := EO^{k-1} x ⊕ O^k x):
   forall i in slice (1 + 2^{k-1}, |a|, 2^k) do
      x[i] := x[i] ⊕ x[i - 2^{k-1}]
   end forall
```

We have used the notation **forall**...**in**... to emphasize the fact that the above are parallel loops: on a vector processor, all the vector operations corresponding to an instance of U_k or D_k can be performed simultaneously.

Note that the backward dependencies in these loops are only "apparent" in the sense of section 3.1: since both loops are low-level translations of vector operations (U_k and D_k, kept as comments in the above code), the expected interpretation is the vector one (which anyway turns out to be identical to the sequential loop semantics in this case). Thus, if a conservative vectorizer such as the Cray Fortran Translator inhibits vectorization of these loops because of the apparent dependencies, the programmer should override the inhibition.

Below is a non-recursive version of *total_reduction* which integrates the various improvements achieved so far. This version would be readily vectorizable by any simple vectorizer (such as CFT, The Cray Fortran Translator, on the Cray-1). A further simplification is obtained by using variables *step* and *half_step*, corresponding to 2^k and 2^{k-1}, respectively, in lieu of k.

```
procedure total_reduction8
  (a: in VECTOR; x: out VECTOR)
  var step, half_step: NATURAL;
    size: NATURAL; -- size will stand for |a|
begin
  size := |a|;
  forall i in slice (1, size, 1) do
    x[i] := a[i];
  end forall;
  step := 2; half_step := 1;
      -- This corresponds to k := 1
  while step < size do -- Uk
    forall i in slice (1 + step, size, step) do
    x[i] := x[i] ⊕ x[i − half_step]
    end forall;
    half_step := step; step := 2 * step
  end while;
    -- here {1 ⩽ half_step − size − step =
    -- 2 * half_step }
  while step > 1 do -- Dk
    forall i in slice (1 + half_step, size, step) do
      x[i] := x[i] ⊕ x[i − half_step]
    end forall;
    step := half_step; half_step := half_step/2
  end while
end procedure -- total_reduction8
```

6.2. A timing diagram

The diagram in fig. 4 may be helpful in visualizing the operations performed on x during an execution of the procedure. It applies to the case $|a| = 9$. The elements are represented horizontally; the vertical axis represents time. Execution of the operation

$$x[i] := x[i] \oplus x[j]$$

at time t is pictured as fig. 3.

The two main loops ("*up*" and "*down*") appear clearly on the diagram: the first one is executed in steps 1 to 3, the second one in steps 4 to 7.

It is interesting to note that this diagram follows directly from the non-recursive version of the procedure; it can also be deduced from the initial recursive version (by expanding the call graph), but the deduction is much more difficult.

Note that there is a minor possibility for extra parallelism, between steps 4 and 5, that our development method has not captured.

The time needed for total reduction of a vector a using cyclic reduction on the Cray is approximately

$$t_{CYCL} = 2*(r-1)*U + (2*(n-1)-r)*V,$$

where $r = \lfloor \log(|a|) \rfloor$. This time should be com-

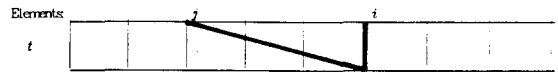

Fig. 3.

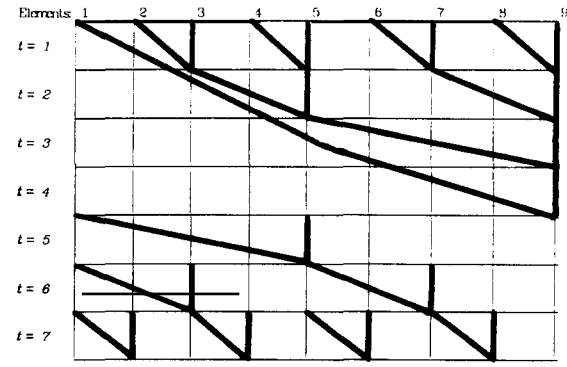

Fig. 4.

pared to $t_{SCAL} = (n-1) * S$ for the trivial algorithm (constants U, V and S were introduced in section 3.1). For the Cray, the cutoff point at which cyclic reduction becomes more efficient is approximately $|a| = 40$.

7. An Ada version

Below is an implementation of the algorithm as an Ada function, embedded in a generic package. The following points are worth nothing:

- the generic mechanism of Ada provides a way to write the package so that it can be applied to various cases; the same generic package can have many instances depending on what the type *SCALAR* and the "+" operation, which corresponds to the operation written ⊕ above, are chosen to be: for instance the type *INTEGER* and integer addition, a matrix type and matrix multiplication, etc.
- The Ada generic mechanism is flexible but strictly syntactical: the language provides no way to specify that the actual generic parameters must have predefined semantic properties, for instance that "+" must be associative. A language such as LPG (Language for Generic Programming [4]) makes it possible to impose such conditions on generic parameters.
- Procedure *ADD_TO_VECTOR* is the one which performs the vector operations (corresponding to U_k and D_k as defined above). These operations must be expressed in scalar form, using loops (**for**...**in**...**loop**...**end loop**). Thus, on a vector computer an Ada program such as this one will require the intervention of a vectorizer, similar to those which exist for Fortran (e.g. CFT on the Cray-1), in order to take advantage of the vector computation facilities of the hardware.
- The loop in procedure *ADD_TO_VECTOR* seems to involve a backwards dependency. However, this is only an apparent dependency, as defined in section 3.1, since the loop updates s and uses $s - offset$, but these two slices are disjoint whenever $offset \neq s.step$, which is the case for the two calls to *ADD_TO_VECTOR* in the package. This implies, however, that a vectorizing Ada compiler would still have to provide some kind of "vectorize at any risk" directive similar to Cray Fortran's *IVDEP*.

The fact that vector programmers should still resort to such low-level and error-prone techniques in Ada is all the more disappointing that Ada comes close to providing adequate notations for true vector programming: it has vector operations such as vector assignment (used below in the initializing statement $x := a$ of function *TOTAL_REDUCTION*) and the notion of slice; however, an Ada slice must be a contiguous subarray, whereas the slices which we need here are not contiguous, which is why we must use loops.

On the other hand, a language such as Actus [16], explicitly designed for use on vector computers, readily allows for non-contiguous slices, but lacks the generic facility of Ada.

```
generic
  type SCALAR is private;
  with function "+" (X, Y: SCALAR)
    return SCALAR is ⟨⟩;
package CYCLIC_REDUCTION is
  type VECTOR is
    array (NATURAL range ⟨⟩) of SCALAR;
  function TOTAL_REDUCTION (a: VEC-
      TOR)
    return VECTOR;
private
  type SLICE is
    record low, high, step: NATURAL end;
end CYCLIC_REDUCTION;

package body CYCLIC_REDUCTION is
  procedure ADD_TO_VECTOR
    (x: in out VECTOR;
    s: in SLICE;
    offset: in NATURAL)
    -- x(s) := x(s) + x(s − offset)
  is
    bottom: constant NATURAL := s.low;
    top: constant NATURAL := s.high;
    stride: constant NATURAL := s.step;
    last: constant NATURAL :=
      (top − bottom)/stride;
  begin
    for i in 0..last do
```

```
      x(bottom + i * stride) :=
        x(bottom + i * stride)
        + x(bottom + i * stride - offset)
    end for;
  end ADD_TO_VECTOR;

  function TOTAL_REDUCTION (a: VECTOR)
  return VECTOR is
    initial: constant NATURAL := a' FIRST;
    final: constant NATURAL := a' LAST;
    size: constant NATURAL := initial - final + 1;
    x: VECTOR := a;
    step: NATURAL := 2;
    half_step: NATURAL := 1;
  begin
    UP:
      while step < size loop
        ADD_TO_VECTOR (x,
          (initial + step, final, step),
          half_step);
        half_step := step; step := 2 * step;
      end loop UP;
        -- here {1 ⩽ half_step < size ⩽
        --    step = 2 * half_step}
    DOWN:
      while step > 1 loop
        ADD_TO_VECTOR (x,
          (initial + half_step, final, step),
          half_step);
        step := half_step;
        half_step := half_step/2;
      end loop DOWN;
    return x;
  end TOTAL_REDUCTION;
end CYCLIC_REDUCTION;
```

8. Conclusion

Transformational programming has been advocated by several authors (e.g. refs. [1,2,9,3,8]) whereas other researchers in software design methodology prefer a more direct approach to the synthesis of programs from specifications [10,11]. Although we do not wish to enter this debate here, the derivations obtained in this paper may bring some interesting elements.

Even though the sequence of transformations needed to produce the final program may seem overly long and complex, we do not know of any other rigorous way to derive that program. We would be interested to learn of a more direct argument, if there is one.

On the other hand, it is not clear to us whether any of the existing program transformations systems (where the term "system" is taken to denote coherent sets of tools and/or methods) may indeed support the transformations described here.

In any case, we feel that the development presented here is another example of the need for applying systematic techniques to the design of vector programs. Effective supercomputer programming requires a wide range of modern software engineering techniques; program transformation may be one of them.

Acknowledgement

We are grateful to Alan Wilson for the useful comments he made as a referee for this paper.

References

[1] J. Arsac, Commun. ACM 22 (1979) 43.
[2] R. Balzer, N. Goldman and D. Wile, in: Proc. Second Intern. Conf. on Software Engineering (1976) p. 223.
[3] F.L. Bauer, M. Broy, W. Dosch, R. Gnatz, F. Geiselbrechtinger, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing and H. Wössner, The Munich Project CIP, Technische Universität München, Munich (December 1983).
[4] D. Bert, Rapport R-408, IFIAG, IMAG Institute (Grenoble University), Grenoble (December 1983).
[5] A. Bossavit and B. Meyer, in: Algorithmic Languages, eds. J. de Bakker and R.P. van Vliet (North-Holland, Amsterdam, 1981) p. 99.
[6] A. Bossavit, in: Proc. IFIP TC2 WG 2.5 (Numerical Software) Working Conf. on PDE Software: Modules, Interfaces and Systems, Söderkoping, Sweden (August 1983).
[7] A. Bossavit, in: Proc. Conf. on The Use of Supercomputers in Theoretical Science, Antwerpen, Belgium (30 July–1 August 1984).
[8] J.M. Boyle and M.N. Muralidharan, IEEE Trans. on Software Eng. SE-10 (1984) 574.
[9] J. Darlington and R.M. Burstall, Acta Informatica (1976) 41.

[10] E.W. Dijkstra, A Discipline of Programming (Prentice-Hall, Englewood Cliffs, New-Jersey, 1976).
[11] D. Gries, The Science of Programming (Springer-Verlag, Berlin, 1981).
[12] C.A.R. Hoare, in: Symp. on the Semantics of Programming Languages, Lecture Notes in Mathematics, vol. 188 ed. Erwin Engeler (Springer-Verlag, Berlin, 1971) p. 103.
[13] C.W. Hockney and C.R. Jesshope, Parallel Computers (Adam Hilger, Bristol, Great Britain, 1981).
[14] B. Meyer and C. Baudoin, Méthodes de Programmation (Eyrolles, Paris, 1978).
[15] B. Meyer, Atelier Logiciel no. 24, HI-34552/01, Electricité de France (4 June 1980).
[16] R. Perrott, ACM Trans. on Programming Languages and Systems (1979) 177.