EVA : an Explicit Vector lAnguage.
An Alternative Language to Fortran 90 (former 8x).

J.-L. Dekeyser, Ph. Marquet, and Ph. Preux

Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Techniques de Lille
59655 Villeneuve d'Ascq - France
dekeyser@frcitl81.bitnet,
marquet@lifl.lifl.fr.eunet

Abstract

Fortran is the main language used on supercomputer today. Indeed, all supercomputers compilers have extensions, providing language features for explicit vector handling, to Fortran 77. These extensions are different on each machine and their functions are limited. Even with the next standard Fortran 8x, vector syntax is incomplete. EVA is an explicit vector language with powerful vector handling tools. Taking into account the billion of dollars invested in productive Fortran 77 programs, EVA has been designed keeping in mind Fortran and C interface facilities.

Keywords supercomputer, vector processing, Eva, language design, Fortran 8x, Fortran 90.

Introduction

For ten years, a fast expansion of the population of supercomputers has been observed (from Cray-1 and Cyber 205 in early 80's, to Cray Y-MP, Nec SX-2, ... in these early 90's). The CPU power of these machines came on from a few hundreds of MegaFlops to a few GigaFlops (150 Mflops for the Cray 1, 1300 Mflops for the SX-2 of NEC, 2600 for the Cray Y-MP). At the same time, concepts about software quality have been developed [DAV 77]: reusability, modularity, portability, ... which are not always taken into account in scientific programming.

In the vector mono/multi processors with shared memory environment, programming suffers from several disadvantages:

- Non-portable applications: a program using a vector notation depending on the machine is neither compilable nor runnable on an other machine. A vector language version, unchanging and available on a large population of (super)computers is still awaited: "Fortran 8x" [MET 89], [SMI 88].
- Scalar languages inadequate to vector programming: in order to be the efficient, supercomputers have to process vectors. Programs written in scalar Fortran and then converted in vector code are only efficient if they are thought with a vectorial mind. Let's write straight vector code!

They are two different vector programming technics:
The first one uses a standard scalar language (Fortran 77) associated with a vectorizer generating vector code from scalar code [ALL 77], [BOS 88]. The interest of such an approach resides in the portability, either on scalar or vector machines. Nevertheless, the lack of vector syntactic tools is made difficult because of such a programming.
The second one uses a language based on a vector syntax [PER 86]. The portability of applications suffers from the lack of standard between supercomputer constructors and

the missing of vector syntax languages on scalar computers. Then, the debugging must absolutely be processed on supercomputers. It does not seem judicious to use such machines for development.
Furthermore, the existing vector languages such as Fortran CFT on Cray machines [CRA 86], Fortran 200 on Cyber 205 and Eta10 [CDC 86], Fortran 77/VP on Fujitsu VP [MIU 83] are only offering rudimentary syntactic tools for vector handling.

Studies concerning vector programming have already been realized. Vectran in 1975 by Paul and Wilson at IBM [PAU 75]. Actus, an extension of Pascal, at the beginning of the 80's by Perrott [PER 79], [PER 83] introduces tools to specify the concurrent processing of array elements. Hellena in 1987 by Jégou [JEG 87] defines the notion of *access pattern* which allows the selection of actually processed vector elements. Other studies are leading to the definition of Ada packages [VOL 89] or to the definition of C++ classes in order to process vectors.

In this context, we propose the language EVA. EVA includes tools for explicit vector handling: we defined a notion of vector and a certain number of vectorial operations. EVA's target is constituted by a whole collection of Unix machines, from workstations to supercomputers. The EVA compiler produces a vector intermediate code called DEVIL. Different packages for code generation ensure portability (fig. 1). In order to keep access to the large pre-existing libraries, EVA allows subroutines/functions calls or sharing of data with Fortran or C (Fortran for the scientific programming, C for the Unix environment). Furthermore, calls to EVA functions from Fortran or C allows the rewriting of the sole vector parts of scalar programs. The development on workstations allows an interactive, user-friendly debugging of applications. Then, numerous efficient executions of this code are ensured on the targeted supercomputer.

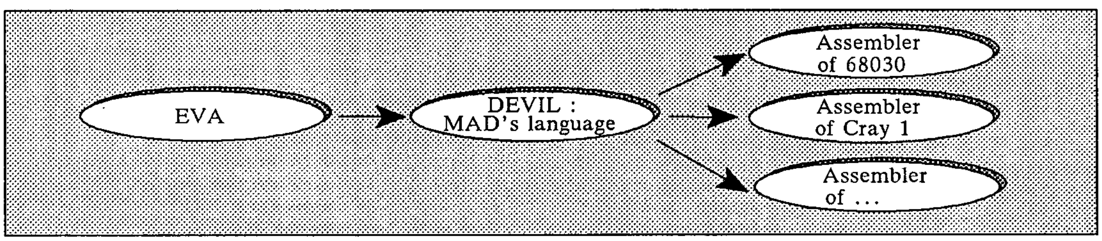

Figure 1 – Program development system

MAD, a virtual vector machine, has been defined "beyond" existing architectures (scalar, pipe-line and array machines). The machine MAD is the target of the EVA compiler. DEVIL (the MAD's language) is used as an intermediate language. A translation of DEVIL into the effective target machine language constitutes the last compilation pass.

After a short comparison of EVA with Fortran 8x approaches of vector programming, we present the different objects handled by EVA, especially vectors. We also introduce new syntactic vector tools. Vector handling examples are proposed in appendix.

Vector programming : the EVA's approach

The EVA's development has mainly been motivated:

- by proposing a vector programming language alternative to Fortran 8x,
- by proposing syntactic vector tools to ensure a higher legibility and a better maintenance of programs,
- by decreasing programmers' work in developing vector algorithms.

We chose to adopt the same approach as Fortran 8x as regards with the following items [ANS 89]:

- The specificity of the target machine has not to be taken into account to write programs using EVA.

- EVA' targets are as well supercomputers as workstations. Development and production can be realized on different machines.
- Explicit vector notations facilitates efficient code generation.
- Functions with vector results generalize vector operators.

Three main points make the difference between EVA and Fortran 8x:

- With Fortran 8x, a vector is obtained from a possibly described array. A description is either a list of triplets (lower : upper : step), or an integer vector expression. Only an array can be associated with a name (*parent–array* [ANS 89]).
  We propose vector objects with more powerful description functions. Furthermore, every vector can be associated with a name.
- Some vector operations require calls to functions/libraries (intrinsic functions *pack/unpack*, *shape/reshape*, or *merge*) which lead to less legibility programs.
  We propose to take the semantic of such operations into account in the syntax.
- Fortran 8x was defined as a universal language, its complexity slowing down its implementation.
  EVA is a language specialized in vector handling. EVA does not include some of the usual programming language features, such as pointers handling or input/output management. EVA's learning is made faster thanks to the simplicity of the language. Yet, interfaces with other languages (mainly C and Fortran) endow EVA with the universal aspect of existing scalar languages.

EVA : an high–level vector language

EVA is a high–level vector language: each algorithm handling vectors uses a vector syntax. Vectors are explicitly declared and used in vector expressions and instructions. Using an iterative flow control structure will never produce vector code.

An EVA program can be constituted from several *modules*. At linkage time, the list of modules containing external objects is specified. A module contains the definition of one or several EVA functions, and a declaration area visible by all the functions of the module. A special function called `main()` specifies the entry point of the module, it can be found anywhere in the module. A module which does not declare a `main()` function is a library module. External variables may be imported from Fortran, C, or EVA. Arguments are usually passed by reference, but they can be passed by value for the C interface.

EVA is a block structured language. Each block is made of a declaration area and an instruction area. The usual rules of block structured language are applicable.

A source line can be made of several instructions, and an instruction can be written on several lines without any constraints. After an instruction, an optional semicolon may increase the legibility. The EVA's compiler inputs sources produced by `cpp`, the standard Unix preprocessor. EVA's "header files" contain complements to the language (predefined constants and macros, standard packages: input/output, ...).

EVA's objects

EVA handles two kinds of objects: scalars and vectors from different predefined types (`bool`, `char`, `int`, `float`, and `double`). Structured objects are obtained by construction. All these objects are operated by EVA instructions.

Scalars

A scalar declaration is built with the following pattern:

```
int a, b                                -- I am a comment !
double r1, r2                           -- Me too !

int a, b [10]                           -- ERROR: array not allowed
```

An initial value may be supplied with the declaration:

```
int i = f (12) + param                  -- any expression is allowed
```

Vectors

❑ An EVA vector is made of two areas: an *allocation zone* and a *description zone*. The description zone selects elements from the allocation zone to constitute the vector (fig. 2). Vectors without any description zone are made of all the elements of their allocation zone.

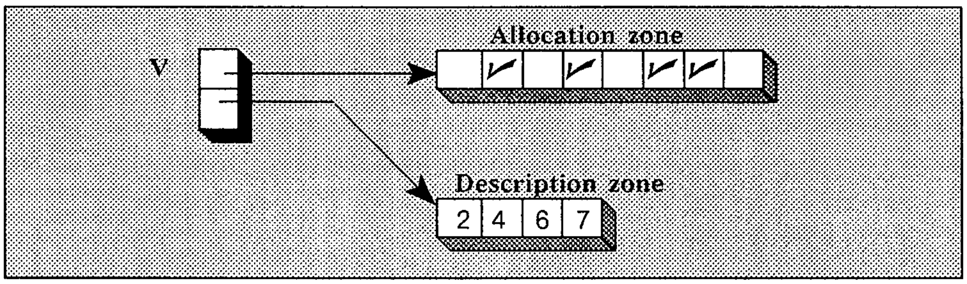

Figure 2 – An EVA vector V

The type of the vector is the type of the allocation zone elements. The description zone contains:

- either a sequence of indexes (integers). The selected elements from the allocation zone are those indexed by the sequence elements. The same element of the allocation zone can be selected many times by repeating the same index.
- or a sequence of bits. Vector elements are obtained by masking the allocation zone.

The *length* of a vector is equal to the number of selected elements.

Declaration example:

```
vect int v, v1                          -- declaration of vectors of integers
```

❑ Allocation and description zones are linked to a vector by the association operator '<–'. A vector declaration statement can occur with an association.

Association – assignment

Different association and assignment operators allow to allocate, initialize, and modify EVA objects.

Association

The association operator '<–' links a target vector to its allocation and description zones. Three types of association are available:

- *Dynamic association*: a scalar expression is associated to the vector. The integer value of the expression is the size allocated to the allocation zone. The description zone is empty. The elements of the allocation zone are not initialized. The fig. 3 describes the following vector declaration:

```
vect int vtarget <- 8
```

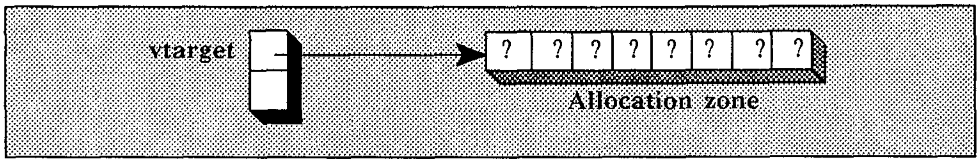

Figure 3 – A simple vector vtarget

- *Sharing association*: a vector name *W*, with or without a description *D*, is associated to the vector *V*.

  $$V <- W \text{ or } V <- W\ [D]$$

  W shares its allocation zone with *V* (fig. 4). The description zone of *V* is built from the one of *W* and the eventual description *D* (fig. 5).

```
vect int vodd, veven
      vodd  <- vtarget [1|3|5|7]              -- 1|3|5|7 is a literal vector
      veven <- vtarget [2 :[2] 8]             -- lower :[step] upper, same as 2|4|6|8
```

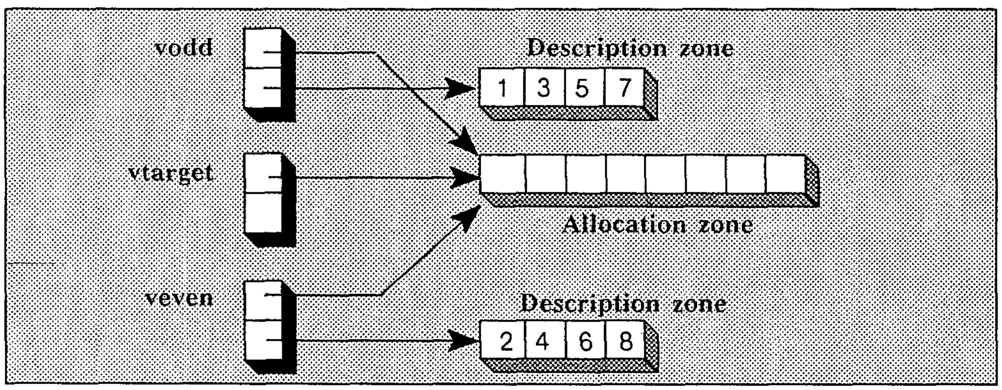

Figure 4 – Sharing of the same allocation zone

```
vect int vgath <- veven [ b'1011' ]           -- b'1011' is a literal bit vector
```

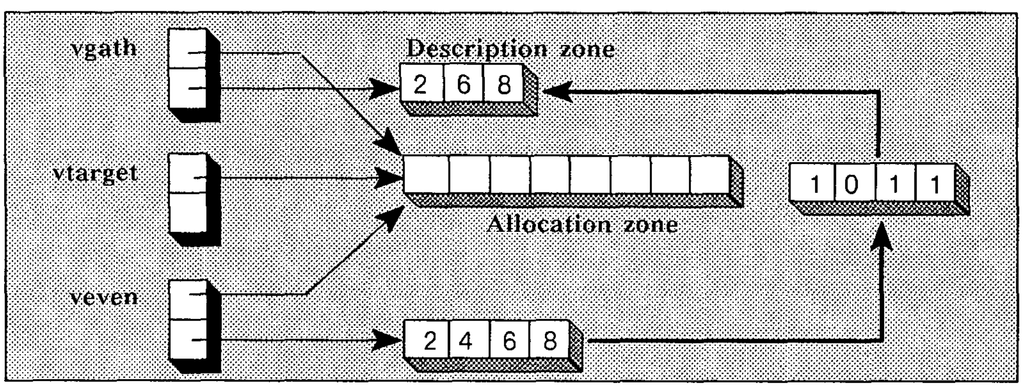

Figure 5 – Gathering the description zone

- *Initialized dynamic association*: a vector expression, different from a simple name or a described name (refer to *Sharing association*), is associated to the vector. This expression is evaluated. The size allocated to the allocation zone is the length of the resulting vector. The description zone is empty. The resulting vector value is copied in the allocation zone (fig. 6).

```
vect int vinit1 <- 1 : 8,                     -- 1 : 8 is a literal vector
       vinit2 <- vinit1 + 1
```

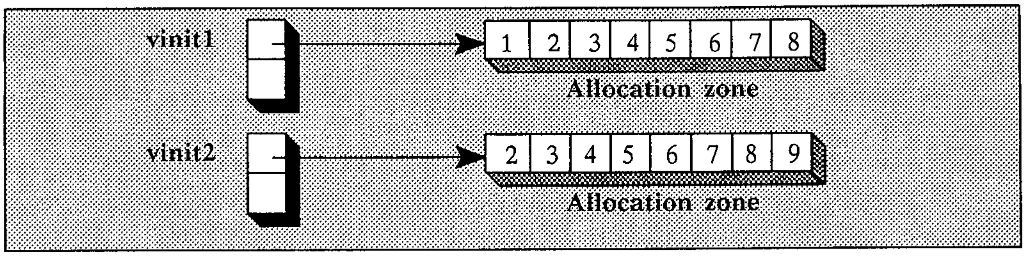

Figure 6 – A simple vector vtarget

❑ Remarks: several vectors may share the same allocation zone. A description zone belongs properly to a vector. For instance elements of `vtarget` allocation zone are reachable with different ways (fig. 4 and 5).

Before any association, the former allocation and description zones of the target vector are automatically freed. However, an allocation zone still referenced by an other vector will not be freed. The operator '<–' is the only operator forbidden in any expressions. It can only appear in a vector declaration or in an association instruction.

Assignment

Applied to a vector, the assignment operator '=' assigns values to its allocation zone elements. The assigned elements are those selected by the description zone. The assignment of a scalar to a vector stores the scalar value in all the vector elements. Example (fig. 7):

```
veven = 4
```

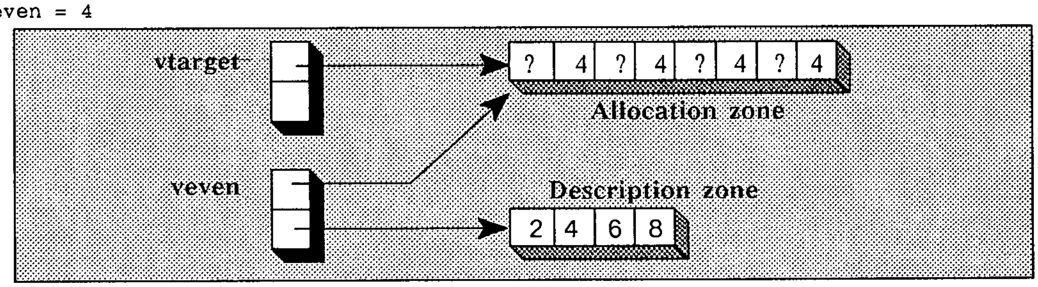

Figure 7 – Equal (=) effects

Assignment ignoring vector dependencies

For a vector assignment '=', the right expression is evaluated then stored in a temporary before effective assignment. It allows to take into account dependencies between the right and the left part of the '='. In order to optimize assignment without dependencies, the assignment ':=' can be used to avoid the storage in a temporary. For scalars, ':=' and '=' are equivalent. Example:

```
vect real v <- 1 : 100,              --          FORTRAN 77
       vi <- v [ 1 : 99 ],           --          ----------
       viplus1 <- v [ 2 : 100 ],     --              REAL V(100) , TEMP(99)
       temp <- 99                    --
-------
-- with control of dependencies      --
      viplus1 = vi + viplus1         --              DO 10 I = 1, 99
-- is equivalent to                  --          10  TEMP(I) = V(I) + V(I+1)
      temp := vi + viplus1           --              DO 20 I = 1, 99
      viplus1 := temp                --          20  V(I+1) = TEMP(I)
-------
-- with no dependencies we can write
      vi := vi + viplus1             --              DO 30 I = 1, 99
                                     --          30  V(I) = V(I) + V(I+1)
-------
-- unpredictable result
      viplus1 := vi + viplus1
```

Some assignment operators '+=', '*=', ... '+:=', '*:=', ... are available:

```
v += vi
-- is equivalent to
v = v + vi
```

EVA's objects - complements -

Constants

An EVA constant is a variable with unique assignment/association. Its declaration must be preceded by 'const' and must own an initial value. This value is not necessarily computable at the compile-time. For a scalar, an expression with a scalar result initializes the variable. Then, this variable will not be allowed to be modified. For a vector, an association '<–' definitively sets the allocation and description zones. Only updating the vector elements is still possible by an assignment ('=' or ':='). The declaration of such variables follows the pattern:

```
const int   i = 12,
            j = i + 1
const vect int v <- 1: 100,              -- v [1] = 1, v [2] = 2, ....
               w <- v [f (j): 50]        -- f (j) is a function call
-- we can modify the contain of allocation zone
        v = 4                            -- v [1] = v [2] = ... v [100] = 4
```

Allocation / disallocation of vector zones

Most of vectors are dynamic; allocation and description zones are allocated on the heap at the run-time. For each new association and at the end of a block for all declared vectors in the block, description zones are freed, and allocation zones are not referred any longer. Allocation zones are automatically freed when no more vector referenced them.

Constant vectors which length are compile-time computable are allocated during the compile-time.

```
const vect int vect_static <- 1 : 100            -- NO dynamic allocation
```

Shape projection

❑ By construction, a vector is uni-dimensional. To allow multi-dimension projection on a vector, *shape* characteristics are associated to it. Declarations of multi-shape vectors are allowed:

```
vect int multi [100] [10, 3, 4] [12, 8]
        multi [10] = ...                  -- one rank access
        multi [1:10, 1:5] = ...           -- two ranks access
        multi [1:10, 2, 1|3|4 ] = ...     -- three ranks access
```

The vector multi is accessible through 1, 2 or 3 dimensions. Dimension declarations have not any effects on allocation and description zones. The effective function to access the elements takes into account the shape characteristics fitted the access.

```
-- multi [1] is equivalent to multi [1, 1] and multi [1, 1, 1]
-- multi [2]                  multi [2, 1]     multi [2, 1, 1]
-- multi [3]                  multi [3, 1]     multi [3, 1, 1]
...
```

The sizes of each dimension are obtained from integer expressions. These expressions are not necessarily during the compile-time. For a given number of dimension, an only shape declaration is allowed.

A dimension is defined by the upper bound and the lower bound. By default, the lower bound is equal to 1. Example:

```
vect int v [2 .. 12],
         w [f (14) + 1]                        -- is equivalent to w [1 .. f(14)+1]
```

❑ The last dimension of shapes is not always an expression. The symbol '?' can be found. In this case, the size of the last dimension depends on the length of the vector. Whenever the length is modified, the last dimension is updated. The vector is said to have an *assumed shape*.

```
vect real vass [20, ?] [?] <- 110          -- vass is "[20, 5] [110]"
vass <- 1000                               -- now vass is "[20, 50] [1000]"
```

The uni-dimensional shape is assumed by default.

```
-- the declaration
vect int vass1 [12, 10]
-- is equivalent to
vect int vass1 [?] [12, 10]
```

Structures

❑ The declaration of structured objects requires a previous type definition. A structure is made of several *fields*. Each field is either a scalar, or a vector, or a structure. Example of type definition:

```
typedef complex
       real real_part,
            imag_part
end
```

After this declaration, we can declare variables of this type:

```
type complex I
```

❑ A *structure element* is any scalar or vector object member of a structure.

```
-- "complex. real_part" is a structure element
-- while "complex" is not
```

❑ The declaration of structure vectors requires for all structure elements to be scalar. Vectors of vectors does not exist in EVA. Example:

```
vect type complex vcmplx
```

For a structure vector, one allocation zone and one description zone are associated to each structure element. Multi-shape is available for structure vectors.

❑ Aggregates are extended structures. The '<<' and '>>' indicate the beginning and the end of a list of expressions. Each expression specifies the value of one field.

```
I = <<0.0, 1.0>>
```

The description of vectors

The association between a vector expression and an inter-brackets ('[' and ']') expression is called a *description of vector*. The inter-brackets expression is called an *expression of description*. Any reference to the name of a vector *V* indicates the whole set of its elements. In order to access a selection of these elements, it is possible to associate to *V* a description *D*. The resulting vector *V[D]* shares the same allocation zone as *V* but has its own description zone. This one is obtained by selecting elements in the description zone of *V* according to the description *D* (fig. 5). Example:

```
vect real vbase [10, 10]  <- (1 : 100) * 0.5
-- [100] is assumed
-- vbase [1] = 0.5, vbase [2] = 1.0, .. vbase [100] = 50.0
--
const vect int index <- 1 :[2] 100

main ()
      vbase [ index ] = 1.0             -- store 1.0 to odd elements
      vbase [ index + 1 ] = 0.0         -- store 0.0 to even elements
....
```

An expression of description is either integer or boolean.

```
vect real va <- ....
       va [ va > 0.0 && va < 10.0 ] +:= 1.0              -- bit mask
```

When the expression of description is scalar, the obtained result is a scalar value. Example:

```
int i = 10, j
j = index [ i + 1 ]
```

A vector described by a vector expression is a vector expression. Thus, another description can be applied to it. The evaluation of the descriptions goes from left to right. Only a vector name can be described by a multi-dimensional description. Once described, the vector is mono-dimensional. Example:

```
vbase [1 :[2] 10, index [1 : 5]] [1 :[2] 25] = 2.0
-- vbase [1, 1] = vbase [5, 1] = vbase [9, 1] = vbase [3, 3] = vbase [7, 3] =
-- vbase [1, 5] = vbase [5, 5] = vbase [9, 5] = vbase [3, 7] = vbase [7, 7] =
-- vbase [1, 9] = vbase [5, 9] = vbase [9, 9] = 2.0
```

Any vector expression can be described. The expression of description selects some elements of the vector result. For the majority of the operators, the distribution of the description to each vector operand produces the same result. Choosing one of both will depend on the rate of selected elements or the complexity of the expression.

```
va = vb [1 : * - 1] + vc [1 : * - 1]                -- two descriptions
va = (vb + vc) [1 : * - 1]                          -- only one
```

The instanciables

An expression of description can depend on the shape characteristic informations of a vector (lower bound, upper bound, ...). Such an expression is said instanciable. Instanciable expressions are written with several *instanciable primitives*. The *instanciation* of those expressions are effective at vector description time. Then, the instanciable primitives are substituted by the values of the described vector shape characteristic informations.

The star '*'

At the instanciation time, the upper bound value of the dimension is assigned to the star. On the left part of the vector constructor ':', the star takes the value of the dimension lower bound. The star can appear in an expression: the associated value, at the instanciation time, is used to evaluate the expression. Example:

```
vect int vtarget <- 12,                         -- vtarget assumed to [12]
         veven   <- vtarget [2 :[2] *]          -- same as [2 :[2] 12], assumed to [6]
         vodd    <- vtarget [* :[2] *],         -- same as [1 :[2] 12], assumed to [6]
last_odd = vodd [*]                             -- last_odd = vodd [6] is a scalar
all_odd_except_first_and_last <- vodd [* + 1: * - 1]    -- same as vodd [1 + 1 : 6 - 1]
```

The instanciables primitives

Apart from the star, three instanciable primitives are available: `lower`, `upper` and `size`. At the instanciation time, these functions are respectively assigned with the integer value of the lower bound, the upper bound and the size of the dimension where they are used. Example:

```
first_odd = vodd [lower]                        -- vodd [1]
vreverse    <- vtarget [upper :[-1] lower]
vfirst_half <- vtarget [* : size/2 + lower]
```

Multi-dimensions instanciables

The instanciable notions defined for mono-dimension shapes of vectors can also be applied to vectors multi-dimension shapes. At the instanciation time, `upper`, `lower`, `size` and the '*' are assigned with a value depending on the dimension where they appeared. Example:

```
vect real multi [2 .. 12, 30]
vect real first_row <- multi [lower, * : *]        -- multi [2, 1 : 30]
```

For a given dimension, the instanciables linked to the other dimensions can also be referenced. The primitives `upper (n, d)`, `lower (n, d)`, and `size (n, d)` are parameterized by the number of dimensions `n` and the referenced dimension `d`. Example:

```
int tmp
vect real multi  [12, 8],
          biggest_square_matrix <- multi [* : tmp = min (*, upper (2, 2)), * : tmp ]
-- same as multi [1 : tmp = min (12, 8), 1 : tmp ]
```

Instanciables objects

Instanciable objects can be defined. As for the expressions, these objects are instanciated for each expression of description where they appear. Only unique assignment or association objects (`const`) can be initialized with an instanciable expression. An instanciable object can never appear on the left side of an assignment. Example:

```
const vect int voddindex <- 1 :[2] *
vect int       vodd      <- vtarget [voddindex],
               veven     <- vtarget [voddindex + 1]
const int second = lower + 1

voddindex = 0                                -- ERROR
```

As unique assignment/association objects, the initial values of instanciable objects are evaluated at declaration time (and not at instanciation time).

```
int i = 4
const vect int vindex <- max (* , 3) :[2] i + 1
vect  real v [2 .. 20]

v <- f(i)
i = 10
v [vindex] = 0                               -- equivalent to v [ 3 :[2] 5 ]
```

Multi-dimension instanciable objects are structures. Their initial value is obtained from an aggregate. Example:

```
-- DIM_VV and DIM_SV are types defined in the include file "instype.h"
const type DIM_VV corner2d   = <<lower | upper, lower | upper>>
const type DIM_SV first_line = <<lower, * : *>>
vect real twoD [param1, param2]
....
twoD [corner2d] = 0.0
-- twoD [1, 1] = twoD [1, param2] = twoD [param1, param2] =  twoD [param1, 1] = 0.0
twoD [first_line] +:= 1.0
```

Objects manipulation

The EVA's expressions are either scalar or vector. Scalars are manipulated via arithmetical, comparison, and boolean operators. These operators are extended to vectors. Other vector specific operations are available.

Any operator is applicable to two vectors of any size. Three rules permit to know the length on which vector operators will be computed. One of the rule is necessarily applied. An other rule application inhibites temporarily the current rule.

Rule of the length by default

For a given expression, vector operations are computed on the same length. This length is the one of the first vector of the expression. Example:

```
vect int v10 <- 10,
         v20 <- 20,
         v30 <- 30
v10 = v30 + v20 * v10        -- the operators +, * and = are computed on 10 elements
```

In an expression, a function call, an association, a description, and a concatenation triggers the local application of the rule of the length by default. For a function call, the rule is applied to each parameter. For an association (resp. a description), the rule is applied to the right operand of the association (resp. the expression of description). For the concatenation, the rule is applied to each operand.

```
v20 = f (v10 + v30 + v20) * v30     -- the operators + are computed on 10 elements,
                                    -- the * and = on 20 elements
```

The attribute `'length` references the current length on which vectors are computed.

Explicit length

The rule of the length by default is well suited to expressions with vectors of the same length. A scalar integer expression can be associated to any expression, instruction or block of instructions. The value of this expression specifies the length on which vectors have to be computed. This length becomes the length by default. That mechanism is well suited to expressions with vectors of different lengths. Function calls, descriptions, associations, and concatenations are still triggering the application of the rule of the length by default.

```
% 5 % v10 = v20 + v30        -- the + and the = are computed on 5 elements

% 5 % {                              -- for the entire block, the default length is 5
   vect double v <- 'length          -- 'length is the length by default (here 5)
   v = v20 + v30                     -- % 5 % v = v20 + v30
   v = f (v20 - v30)                 -- the - is computed on 20 elements, the = on 5
   v10 [v] = v30 [% 10 % v20 + v10]  -- the + is computed on 10 elements, the = on 5
}
```

Specifying the length for a block of instructions allows to increase the efficiency of the produced code.

Any expressions, instructions and blocks of instructions may be preceded by '% ? %'. This inhibites the cast of the length and "re–triggers" the rule of the length by default.

The rule of the minimum

For this rule, the length is specific for each operator. Operations are computed on the minimum of the length of their vector operands. This rule is obtained by '% < %'.

```
% < % v20 = v30 + v10        -- the + is computed on 10 elements, the = on 10
```

Generation of code is required to compute the minimum of the lengths for each operator.

Arithmetic operators

Arithmetic operations between two vectors are processed on paired elements.

```
vect real va, vb
.....
...va + vb ....
```

is semantically equivalent to:

```
...va [1] + vb [1]....
...va [2] + vb [2]....
.......
...va [default_length] + vb [default_length]....
```

With one scalar operand, the operation is distributed to each vector element.

```
vect real va
real r
...
...va + r ...
```

is semantically equivalent to:

```
...va [1] + r....
...va [2] + r....
........
...va [default_length] + r....
```

Comparison operators - Boolean operators

Boolean and comparison operators work as previously. Resulting vectors are boolean.

```
vect bool vbit <- 100
vect real vr   <- vrandom (100)          -- vector of 100 random numbers
vbit = vr > 0.0 and vr < 0.5
```

Structure and vector of structures

Scalar and vector operations are available for structures and structure vectors. The result is obtained by distribution of the operation to the structure elements.

```
vect type complex vc1 <- 100,
                  vc2 <- 100
vc1 = <<0, 1>>                       -- vc1 [i] is << 0, 1>>
vc2 = vc1 - 1                        -- vc2 [i] is <<-1, 0>>
vc2 [1 :[2] *] = vc2 + vc1           -- vc2 [2 * i + 1] is << -1, 1>>
```

The concatenation

The operator '|' realizes the concatenation. The operands are either vector or scalar. The result is a vector built from the left operand elements followed by the right operand elements. The operator '//' of multiple concatenation accepts a scalar expression as left operand. Let *N* be this value, the result is obtained by *N* concatenations of the right expression. Example:

```
const vect int  rotate_1 <- (* + 1 : *) | lower
const vect bool vmask    <- (N/3) // b'011'
```

These operators are also literal vector constructors.

The constructor ':'

From three scalar expressions, the operator ':' builds a vector. The expression to evaluate the step is optional, its default value is 1. Example:

```
int low, up, step
vect int vi <- low : [step] up        -- same as vi <- low | low + step |
                                      -- ... | low + n * step (<= up)
```

Conditional expression

A conditional expression is built around symbols '?', '#', and '!'. It follows the pattern

<expression 1> ? <expression 2> ! <expression 3>

or <expression 1> # <expression 2> ! <expression 3>

The expression 1 results a boolean, either scalar or vector.

- For a scalar result, if it's TRUE, the result of the expression is the result of the expression 2 (expression 3 is not evaluated) else it is the result of the expression 3 (expression 2 is not evaluated).

- For a vector, the result is obtained by the MASK ('?') of the results of the expressions 2 and 3. For a MERGE ('#') the result is obtained by a shuffle of the values of expressions 2 and 3. The result is a vector. One of the rules of the length is applied to the three expressions.

Example of conditional scalar expression:

```
int factorial (i)            -- definition of a function factorial
       int i;
       return (i == 1) ? 1 ! factorial (i- 1) * i
end
```

Example of MASK:

```
vect int min_by_element (a, b)      -- definition of a function min_by_element
       vect int a, b                -- min_by_element [i] = min (a [i], b [i])

       return (a < b) ? a ! b
end
```

Example of MERGE:

```
vect int  a, b, c, b1, c1
vect bool bit
...
a = bit # b ! c
-- is equivalent to               --              remarks :
b1 <- a [bit]        b1 = b      -- two instructions on the same line
c1 <- a [! bit];     c1 = c;     -- two instructions followed by an optional semi-colon
```

Block constructors

`if else endif` and `loop exit endloop` instructions control the flow of instructions. `while for` and `repeat` statements are also available.

An extra block constructor `with` is proposed. A with-block is built on the following pattern:

```
with ( < description expression > )
       < list of instructions >
end
```

The description expression (integer or boolean) is the *description by default*. This description controls the <u>first</u> vector assignment operator ('=', ':=', ...) of instructions and the eventual inner blocks instructions. Any EVA instruction can appear in a with-block. The two following blocks are equivalent:

```
with (descr_exp)
       v1 = v2 [1 :[2] *] + v3
       w1 = w2 = w3
       if (any (v1 > 0))
              puts ("positive(s) value")
              v1 += 1
       end
end
```

and

```
{                                                          -- beginning of block
       temp = descr
       v1 [temp] = (v2 [1 :[2] *] + v3) [temp]
       w1 [temp] = (w2 = w3) [temp]
       if (any (v1 > 0))
              puts ("positive(s) value")
              v1 [temp] += 1
       end
}                                                          -- end of block
```

With-block nested is possible. The two following blocks are equivalents:

```
with (d1)
        with (d2)
                v1 = v2 [1 :[2] *] + v3
        end
end
```

and

```
{
        temp1 = d1
        {
                temp2 = d2 [temp1]
                v1 [temp2] = (v2 [1 :[2] *] + v3) [temp2]
        }
}
```

Furthermore, successive with–blocks can be unified in a unique with–block by using the derivated constructor `otherwith`. Example:

```
with (1 :[2] N)
        a = ...
otherwith (2 :[2] N)
        b = ...
end
```

is equivalent to:

```
with (1 :[2] N)
        a = ...
end with (2 :[2] N)
        b = ...
end
```

The description expression of the `otherwith` can reference the previous description by default by the attribute `'with`. The description expression of the `otherwith` becomes the description by default. The `'with` attribute is usable in a with–blocks or an otherwith–block.

When the description expression is a boolean vector, the `with` can act as a Fortran 8x WHERE:

```
with (pressure <= 1.0)                  -- WHERE (PRESSURE <= 1.0)
        pressure +:= inc_pressure       --     PRESSURE = PRESSURE + INC_PRESSURE
        temp -:= 5.0                    --     TEMP = TEMP - 5.0
otherwith (not 'with)                   -- ELSEWHERE
        raining = true                  --     RAINING = .TRUE.
end                                     -- END WHERE

-- a cpp macro increases legibility:
#define elsewith      otherwith (not 'with)

                                        -- TEMP = A >= 0
with (a >= 0)                           -- WHERE (TEMP)
        a += 1                          --     A = A + 1
                                        -- END WHERE
        with (a <= 10)                  -- WHERE (TEMP .AND. A <= 10)
                a += 10                 --     A = A + 10
        end                             -- END WHERE
elsewith                                -- WHERE (.NOT. TEMP)
        a += 2                          --     A = A + 2
end                                     -- END WHERE
```

A description expression may be an instanciable expression. Its value is instanciated for each implicit description by default. Any non–instanciable part of the expression is evaluated before the block execution. The `'with` attribute is then only usable in a description expression.

```
with (1 :[2] N)
         v = 1                  -- v [1: [2] N] = 1
        ~w = 1                  -- w = 1
end
```

Miscellaneous

Multi-vectors

To make the handling of a large number of vectors easier, *multi-vector* are introduced. It permits to reference a vector by a name and an index. A multi-vector is not directly handled by EVA. Its size (number of vectors constituting the multi-vector) must be compile-time computable. A shape or an initial value is applied to each vector of the multi-vector.

```
vect int (100) multi-vect [20, 12]          -- multi-vect is 100 vectors
multi-vect (i + 1) <- 240                   -- i must be a scalar
multi-vect (i + 1) [1 : *] = 0

multi-vect [1 : *] = 0                      -- ERROR: no operation allowed on multi-vect
```

Functions

Parameters and the return value of a function are of any type except multi-vector. Calls and returns are pass-by-reference convention. A formal parameter may be constant. Type checking of arguments is not ensured.

The shape characteristics of an vector actual parameter are not transmit to the called function. When vector formal parameters are declared, local shape characteristics are defined. An assumed shape definition ensures the coherence between length and shape characteristics. Example:

The return of local variables (allocated on the stack) implies to copy their values on the dynamic area (heap). Such return are detected by EVA, the copy mechanism is triggered. At the end of a function all the no more referenced vectors are automatically freed.

```
vect real square_spread (nb, V)
       int nb
       vect real V [nb],
                Local_vector [nb, nb]                   -- the result
       for (i, 1, nb)
             Local_vector [* : *, i] = V
       endfor
       return Local_vector
end
```

Built-in functions

EVA proposes several built-in functions. Numerical (`abs`, `sqrt`, ...) and mathematical (`sin`, `cos`, `log`, ...) functions accept either scalars, or vectors as parameters. The returned result is either a scalar, or a vector. Some vector reduction functions (`count`, `sum`, `all`, `any`, `minval`, `minloc`, `firsttrue`, ...) return a scalar value. Other functions return a vector (`random`, ...).

Declaration of externals

❑ Definition of *common* makes variables available in different EVA modules. This also ensures the transfer of data towards or from other languages (Fortran for scientific environment and C for Unix access).

Sharing data with Fortran subroutines is achieved by such a common declaration:

```
const vect real v <- 100          --    REAL V
int i, j                          --    INTEGER I, J
common /MY_COMMON/ i, j, v        --    COMMON /MY_COMMON/ I, J, V (100)
```

The variables listed in the common are either scalars, or unique association vectors with *dynamic association*. These variables have to be previously declared at the same level as the common declaration. The allocation zone of such external vectors is allocated on the common area. Their description zone is empty.

In order to share data between different EVA or C modules, the following common declaration is required:

```
-- in an eva file
vect int evavector <- 100
common evavector

/* in a C file */
#include "eva.h"
/* INT_VECT is a type defined in the C include file "eva.h" */
extern INT_VECT evavector;          /* an eva vector */
```

An external variable must be previously declared at the same level. Any type is usable for external variable.

❑ External function calls need explicit declarations in global area. By default, calls and returns are pass-by-reference. To obtain pass-by-value, and interface all the C libraries, EVA proposed the following extension :

```
extern int Ffact                          -- all parameters by address
extern int @ Cfact ()                     -- all parameters by value
extern int Cfunc (@, )                    -- first parameter by value,
                                          -- second and successors by address
```

Attributes

To obtain more informations about vectors, attributes are available. The syntax of an attribute is the following:

< name of vector > ' < name of attribute >

Two kind of attributes are available:

Effective attributes: They are depending on the effective vector allocations.

```
vect int v_copy <- v'length
v_copy = v
```

attribute name	type of the result		function
length	scalar	int	length of the vector
allocsize	scalar	int	size of the allocation zone
descrsize	scalar	int	size of the description zone
described	scalar	bool	the vector is described
bitnotindex	scalar	bool	the description is a bit vector (not an index one)
alloc	vector	same	copy of elements of the allocation zone
descrint	vector	int	copy of element of the description zone
descrbit	vector	bool	copy of element of the description zone

Table 1 – A non-exhaustive list of attributes for vectors

Declaration attributes: They are depending on the vector declarations. These informations may not correspond to the allocation ones. Examples:

`lower()`, `upper()`, `size()` are depending on the shape. They accept zero or two parameters (number of dimensions, and dimension index in the shape). Example:

```
vect int v [12, 14] [200]
int i
i = v'upper (1, 1)          --> 200
i = v'upper (2, 1)          --> 12
i = v'upper (2, 2)          --> 14
i = v'upper (3, 1)          --> 0  because not declare with 3 dimensions

vect int v [10, ?]
i = v'upper (2, 2)          --> 0  because not yet associated
```

Conclusion

We described our ongoing project of EVA language implementation to program vector supercomputers. Compared with vector extended Fortran 77 language [KAR 88], EVA should have the advantage of allowing machine independent explicit access to vector instructions. In addition, vector codes can be developed, tested and run on scalar machines and workstations.

Now, EVA programs compile and run correctly on Motorola 680x0. EVA compiler has been designed by using high level language development tools, Lex and Yacc [JOH 75] on Unix; thus to ensure an easy portability on Unix machines. Future works include, from this prototype, the development of a fully reliable and complete system (including specific EVA libraries). Our aim is to generate vector as well as scalar codes from EVA source codes for various target architectures such as Cray, IBM VF and vector accelerators (FPS, Intel 860, ...). An other aspect of our works consist in improving Eva's portability on distributed memory multi-processors machines (Transputers, Hypercube iPSC, ...).

Bibliography

[ALL 87] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form", *ACM TOPLAS*, vol. 9, n°4, Oct 1987.

[ANS 89] ANSI, System Programming Languages, FORTRAN 8x, X3J3 draft S8.112, June 1989.

[BOS 88] P. Bose, "Interactive Programm Improvement via EAVE: An Expert Adviser for Vectorization", *Proceedings of the 1988 International conference on Supercomputing,* pp 119-130, Saint-Malo, France, July 1988.

[CRA 86] Cray Research Inc., "CFT 77 Reference Manual", SR-0018,1986, Mendota Heights MN 55120, USA.

[CDC 86] Control Data Corp., "Fortran_200 Reference Manual", 60480200, 1986.

[DAV 77] C.G. Davis and C.R. Vick, "The Software Development System", *IEEE Trans. Software Eng.*, vol. SE-3, n°1, pp 69-84, Jan. 1987.

[JEG 87] Y. Jégou, "Le langage vectoriel Hellena", *INRIA-Publication interne*, n°368, June 1987. (in French)

[JEG 87] Y. Jégou, "Acces Pattern: A Useful Concept in Vector Programming", *Proc. of the 1st International Conference on Supercomputing,* Athens, Greece, June 8-12, 1987, pp 377-391. in: *Lectures Notes in Computer Science n°297, ed. Springer-Verlag.*

[JOH 75] S.C. Johnson, "Yacc : Yet Another Compiler Compiler", *Comp. Sci. Tech. Rep. 32, AT&T Bell Labs.*, Murray Hill (NJ).

[KAR 88] A.H. Karp, R.G. Babb II, "A Comparison of 12 Parallel Fortran Dialects", *IEEE Software*, pp 52-67, September 1988.

[MET 89] M. Metcalf and J.K. Reid, " Fortran 8x Explained", Revised version, *Oxford University Press*, 1989.

[MIU 83] K. Miura, K. Uchida, "FACOM Vector Processor VP 100/200", *Proceedings ot the NATO Advanced Research Workshop on High Speed Computing*, Jülich, W. Germany, June 20-22, 1983.

[PAU 75] G. Paul, M. W. Wilson, "The VECTRAN Language: An Experimental Language for Vector/Matrix Array Processing", *IBM Palo Alto Scientific Center*, Report 6320-3334, August 1975.

[PER 79] R.H. Perrott, "A Language for Array and vector Processor", *ACM TOPLAS*, vol. 1, n°2, pp 177-195, February 1979.

[PER 83] R.H. Perrott, D. Crookes and P. Milligan, "The Programming Language ACTUS", *Software - Practice and Experience*, vol. 13, pp 305-322, April 1983.

[PER 86] R.H. Perrott and A. Zarea-Aliabadi, "Supercomputer Languages", *ACM Computing Surveys*, vol. 18, n°1, pp 5-22, March 1986.

[SMI 88] edited by B.T. Smith, "A Review and Analysis of Fortran 8x", *ACM SIGNUM Newsletter*, vol. 23, n°2, pp 29-60, April 1988.

[VOL 89] G. Völksen and P. Wehrum, "Ada for Scientific Computation on Vector Processors", *Journal of Pascal, Ada & Modula 2*, pp 16-32, Nov/Dec 1989.

Appendix – Examples

Prime Numbers

Those programs find all the primes under an input value by the sieve method. A function returns a logical vector with primes marked as true.

EVA program

```
# include <eva_io.h>         -- input / output
# include <control.h>        -- extend control constructs (repeat, ..)

vect bool findprimes (N)
   int N,
       nextprimes,
       Nsqrt = sqrt (N)
   vect bool primes     <- N,
             candidates <- N
   -- initialisations
   primes = false
   candidates = true; candidates [1] = false
   -- loop of the sieve
   repeat
      -- find the nextprimes and mark it
      primes [nextprimes = candidates´ firsttrue] = true
      -- its multiples aren´t candidates
      candidates [nextprimes : [nextprimes] *] = false
   until (any (candidates) and nextprimes < Nsqrt)
   -- the result
   primes [nextprimes + 1 : * ] := candidates [nextprimes + 1 : *]
   return (primes)
end -- of the function
main ()
   vect bool primes_number
   int i
   geti (i)
   primes_number <- findprimes (i)
end
```

Fortan 8x program

```
      PROGRAM MAIN
         LOGICAL, ALLOCATABLE :: PRIMES_NUMBER (:)
         READ *, I
         ALLOCATE (PRIMES_NUMBER (I))
         CALL FINDPRIMES (PRIMES_NUMBER, I)
      CONTAINS
         SUBROUTINE FINDPRIMES (PRIMES, N)
         INTEGER N, NEXTPRIMES, NSQRT
         LOGICAL PRIMES (N)
         ALLOCATABLE :: CANDIDATES (:)
C        INITIALISATIONS
         ALLOCATE (CANDIDATES (N))
         PRIMES = .FALSE.
         CANDIDATES = .TRUE.
         CANDIDATES (1) = .FALSE.
C        LOOP OF THE SIEVE
         DO
C           FIND THE NEXT PRIME AND MARK IT
            NEXTPRIMES = MINLOC ( (/ (I, I = 1, N) /), CANDIDATES)
C           ITS MULTIPLES AREN´T CANDIDATES
            PRIMES (NEXTPRIMES) = .TRUE.
            CANDIDATES (NEXTPRIMES : : NEXTPRIMES) = .FALSE.
            IF (.NOT. ANY (CANDIDATES).OR.NEXTPRIMES.GT.NSQRT)  EXIT
         END DO
         PRIMES [NEXTPRIMES + 1 : ] = CANDIDATES [NEXTPRIMES + 1 : ]
         RETURN
         END
      END PROGRAM MAIN
```

The Ising Model

The Ising model is a well-known Monte Carlo simulation in 3-dimensional space. There is three differents phases to program the Ising model

(1) Counting the nearest neigbors that have the same spin;
(2) Generating an array of random numbers;
(3) Determining which gridpoints are to be flipped.

Eva version

```
# include <control.h>              -- extend control constructs
Transition (N, Iterations, Ising, P)
   const int N, Iterations, size  = N * N * N
   const vect bool  Ising
   const vect real  P [6]
      -- instanciables
      const vect int  all <- * : *, l_shift <- 2 : * | 1,
         r_shift <- upper | 1 : * - 1

   for (i, 1, Iterations)
      const vect bool Flips <- size
      const vect int  Ones  [N, N, N] <- size, Count <- size
      const vect real Threshold <- size,
      Ones = 0
      Ones [Ising] = 1
      Count := Ones [l_shift, all, all] + Ones [r_shift, all, all]
             + Ones [all, l_shift, all] + Ones [all, r_shift, all]
             + Ones [all, all, l_shift] + Ones [all, all, r_shift]
      with (! Ising)
         Count := 6 - Count
      end
      Threshold := (Count >= 4) ? P [Count] ! 1.0
      with ((Flips := random (N)) <= Threshold)
         Ising := ! Ising
      end
   endfor
end
```

The Fortran 8x version of this program is proposed in [ANS 89] § C.13.2.3

```
SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)
   LOGICAL ISING (N, N, N), FLIPS (N, N, N)
   INTEGER ONES (N, N, N), COUNT (N, N, N)
   REAL THRESHOLD (N, N, N), P (6)

DO I = 1, ITERATIONS
   ONES = 0
   WHERE (ISING) ONES = 1
   COUNT = CSHIFT (ONES, 1, -1) + CSHIFT (ONES, 1, 1) &
         + CSHIFT (ONES, 2, -1) + CSHIFT (ONES, 2, 1) &
         + CSHIFT (ONES, 3, -1) + CSHIFT (ONES, 3, 1)
   WHERE (.NOT. ISING) COUNT = 6 - COUNT
   THRESHOLD = 1.0
   WHERE (COUNT .EQ. 4) THRESHOLD = P (4)
   WHERE (COUNT .EQ. 5) THRESHOLD = P (5)
   WHERE (COUNT .EQ. 6) THRESHOLD = P (6)
   FLIPS = RAND (N) .LE. THRESHOLD
   WHERE (FLIPS) ISING = .NOT. ISING
END DO
END
```