Data Parallelism and Pei Equational Language

Guy-René Perrin and Eric Violard

ICPS, Université Louis Pasteur, Strasbourg
Pôle API, Boulevard Sébastien Brant, F-67400 Illkirch
{perrin,violard}@icps.u-strasbg.fr
URL: http://icps.u-strasbg.fr

Abstract. The theory Pei [13, 14] was introduced in order to address crucial questions of program design methodology through a straightforward generalization of equational notations [9, 3]. The relationship between a Pei statement and a data parallel program is quite obvious since Pei expresses, in an abstract way, parallel variables, global operations and communications, and data alignment. The paper illustrates this point and considers a few examples.

Keywords. Data Parallel Programming, Equational Languages.

1. introduction

Among the various parallel programming models that have recently emerged [11], the data parallel paradigm is of particular interest. Former data parallel languages, such as C* or MPL, were designed in order to program efficiently massively parallel architectures. But, in the programming point of view, the question is: how such low-level imperative languages can safely describe problems ? Obviously they seem not abstract enough for reasoning on programs.

The theory Pei [13, 14] was born from a generalization of the so-called polytope model [7] and equational notations [9, 3]. It was introduced in order to address crucial questions of program design methodology and transformations [5].

Equational notations associated with the polytope model have been of great interest in processor array synthesis or compiling techniques for static control programs. Compiling techniques, program rewriting and parallel program design obviously share some common knowledge about programs which serves as a basic tool for parallel program reasoning.

Since a decade, a wide range of research works on static analysis of programs, forms the foundation of parallelization techniques which improve the efficiency of the code: loop nest rewriting, advices to the compiler to align or distribute the data or the operations, etc. These techniques are of particular interest in data parallel programming.

They are based on geometrical transformations either of the iteration space or of the index domains of arrays, which assume conditions such as the affinity of dependences and of loop bounds. In some sense, this shows that, beside a classical functional point of view on programs, geometrical issues in parallel programming or parallelizing technique have to be considered of main importance for the mastery of efficient computations.

This geometrical approach entails an abstract manipulation of array indices, to define and transform
– the data dependences in the program,
– the way the data are, or are not, locally accessible,
– their expansion in a multidimensional space of virtual processors,
– etc.
This requires to be able to express, compute and modify the placement of the data and operations in an abstract discrete reference domain. Then, the programming activity may refer to a very small set of primitive issues to construct, transform or compile programs. They are the foundations of Pei.

The relationship between a Pei statement and a data parallel program is quite obvious: the paper illustrates this point. It is organized as follows: section 2 presents main features of Pei called *shaped data fields* and the equational notation and meaning of the language. Section 3 and 4 emphasize on two aspects to express or transform programs: modularity and transformation rules in Pei. They are applied on an example of a regular loop nest rewriting. The relationship between Pei and data parallelism is spread all over the paper and section 5 focuses on a very particular point in data parallel programming : the actual semantics of array indices in data parallel languages.

2. PRESENTATION OF PEI

2.1. Shaped data fields

Pei is an equational language. Its notations refer to mathematical objects called *shaped data fields*. Shaped data fields, like sets or bags, are collections of elements which are *values* of a given data type. When operations use two or more elements possibly belonging to several bags, it is necessary to define the way each element is addressed whithin its bag. This can be done for example by associating an index with each element. In the literature such indexed collections are referred to as *data fields* [3, 8].

A natural and interesting way to index elements of such a collection is to use a kind of geometry: therefore each element is associated with a point in a discrete geometrical space, say $\mathbb{Z}^n$. The *index set* of a data field is the set of points where its values are placed.

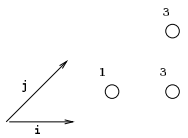

Fig. 1. Data field `A`

Example 1. Fig. 1 shows a data field `A` whose index set forms a triangle in $\mathbb{Z}^2$.

The way the values are combined in an operation can then be interpreted geometrically as *superimposing* values according to the points where they are placed.

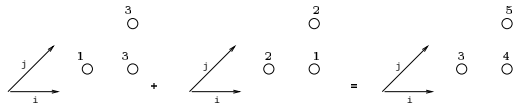

Fig. 2. Data field `A + B`

Example 2. For example, let us consider again the data field `A` and let `B` be another data field of same index set. Then, the expression `A + B` denotes the data field drawn on Fig. 2.

More generally, `A + B` denotes the data field whose values are the sum of values of `A` and `B` with the same index and whose index set is the intersection of `A` and `B` index sets.

Indexing is also natural to express *dependences* between the values of two objects as in a recurrence equation notation: dependence defines a partial function from indices to indices.

Example 3. Let `U` denote a data field whose index set is $\mathbb{N}$ and consider the data field, denoted as `V` whose values are shifted to the right. This relation between `U` and `V` can be denoted in PEI as: `V = U <|pre`
where `pre` is the name of a function which maps an index i to the index $i-1$. It means the value at index $i-1$ in `U` is moved to the index i in `V`. This is a classical feature in equational languages like LUSTRE [6] or $8_{1/2}$ [10].

Note that the function may not be injective and expresses a broadcast.

Example 4. Let `M` denote a data field whose index set is $[1..4] \times [1..4]$ in $\mathbb{Z}^2$: it represents a 4×4-matrix, say M. Let us consider the 4×4-matrix formed of the first column of M replicated four times. It can be represented by the data field `L` defined as: `L = M <|spread_col`
where `spread_col` is the name of a partial function which maps any point (i, j) with $j \in [1..4]$ to the point $(i, 1)$. It is defined in PEI as:

```
spread_col(i,j) = (1<=j<=4) .(i,1).
```

All these issues concern classical approaches in equational languages based on data fields. On the contrary, PEI is founded on a different concept of "shaped" data field. A *shape* expresses the relationship between the locality of values on a virtual architecture and the way they are indexed according to the problem terms: it associates indices of the index set with so-called *locations* [1]. A *shaped data field* is a data field associated with a shape.

[1] Locality is considered in a virtual meaning as a space and time expression

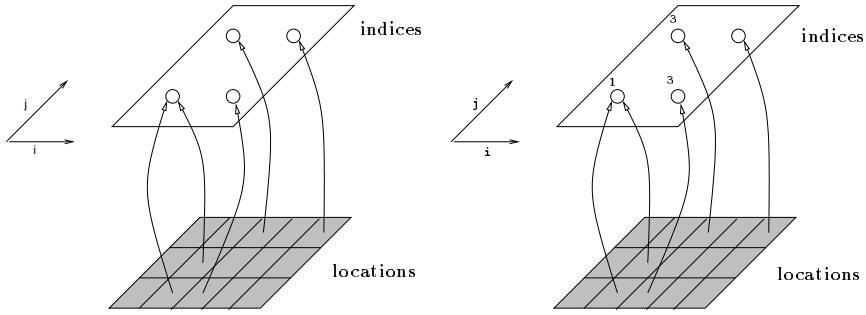

Fig. 3. (a) A shape (b) A shaped data field

Example 5. Fig. 3(a) shows a shape whose indices form a square in $\mathbb{Z}^2$: every index is associated with locations (grey painted). Fig. 3(b) shows a shaped data field which associates the data field `A` drawn on Fig. 1 with the previous shape. Each index of its index set i.e. a triangle $\{(i,j) \mid 0 \leq i+j \leq 1\}$, is associated with at less one location in the shape.

Let us consider the two shaped data fields `X` and `Y` drawn on Fig. 4. They are different data fields associated with different shapes, but the values associated with locations are the same: we say that the shaped data fields are equivalent. This relation is expressed in PEI through the equation: `X = Y ::align`
where `align` is the name of the partial function which maps any point (i,j) of the square $\{(i,j) \mid 0 \leq i,j \leq 1\}$ to $i+j$. This means the index $i+j$ in `Y` is related with the index (i,j) in `X`. The partial function named `align` is defined in PEI as:

```
align(i,j) = (0<=i,j<=1) .(i+j).
```

Note that, since locations express space items, associating indices and locations expresses data alignment on a virtual processor mesh. This facility exists in data parallel programming and allows to transform data placement on the machine in order to improve data locality and parallel code efficiency.

2.2. An introduction to PEI *Programming*

Previous section presented the main feature in PEI: the shaped data field. Let us see now how such items are expressed and what a PEI program means.

Any PEI program is composed of unoriented equations [2], each of them connecting two expressions of the same *shaped data field*. A program expresses the relation between some *input* shaped data fields and an *output* one. Here is a first example of PEI program:

[2] PEI means Parallel Equations Interpretor and pays homage to the architect of the Pyramide du Louvre.

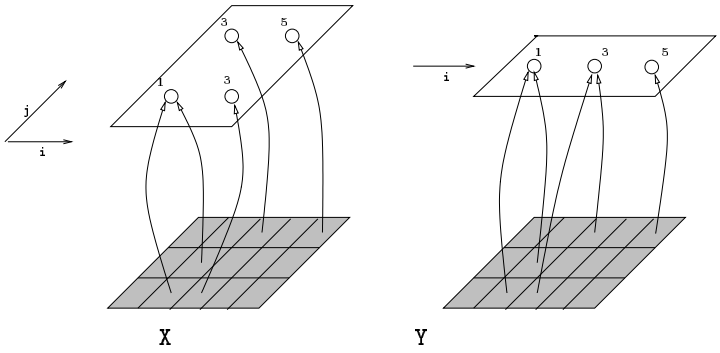

Fig. 4. Two equivalent shaped data fields

Example 6. Summation of two $N \times N$-matrices.

```
MatSum[n]: (A,B) -> C
  {
  A = A ::matrix
  B = B ::matrix
  C = A + B
  }

matrix(i,j) = (1<=i,j<=n) .(i,j)
```

- the two first equations define the index set of the input shaped data fields `A` and `B`: function `matrix` defined as the identity, means that their values are placed onto a square $[1..N] \times [1..N]$ of $\mathbb{Z}^2$ (This program is parameterized by `n` which denotes N).
- the operation `+` builds the pairs of value items of `A` and `B` which are placed at the same index: these values are added and form the output data field `C`.

Let us see a second example in order to carefully introduce the syntactic issues of Pei. It is a classical example of prefix-sum of N numbers.

Example 7. $x_k = \sum_{i \in 1..k} a_i, k = 1..N$.

```
PrefixSum[n]: A -> X
  {
  A = A ::dom
  X <|fst  = A <|fst
  X <|next = A + (X <|pre)
  }

dom(i) = (1<=i<=n) .(i)
```

```
pre(i) = (i-1)
fst(i)  = (i=1) .(i)
next(i) = (1<i<=n) .(i)
```

- the first equation defines the index set of the input data field `A`: its values are placed onto a line segment $[1..N]$ of $\mathbb{Z}$,
- Fig. 5 intuitively shows that data field `X` is a solution of the two last equations: its value items are the prefix computations of the sums of the value items in `A`. The former of these two equations says that the first values in `A` and `X` are the same. The second one defines others values of `X`. The expression `(X <|pre)` defines a data field resulting from `X` by shifting its values from left to right. They are then composed with the values of `A` in the expression `A + (X <|pre)` and added one another.

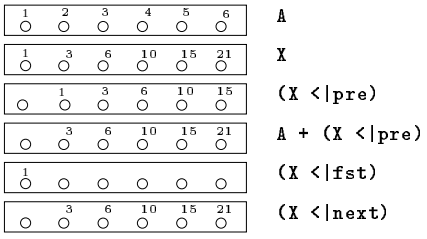

Fig. 5. `X` is a solution of the two last equations

2.3. Semantics of Pei *operations*

Expressions are defined by applying operations on shaped data fields. There are three operations in Pei:

- in section 2.1, the operation used in the expression `(Y ::align)` both modifies the index set (deletion of indices, change of dimension, ...) and the shape in such a way that locations are associated with the same values. As said before, writing `X = Y ::align` defines `X` and `Y` as equivalent shaped data fields. The operation is called *change of basis* and is denoted as `::`.
- another operation "moves" values in the index set. It is called *geometrical operation* (or routing), and is denoted as `<|`. For example, assuming `shift` is the function written in Pei as `shift(i) = (0<=i<4) .((i-1)%4)`, the expression `(V <|shift)` denotes a shaped data field whose values are shifted one place cyclically to the right.
  If the function is not injective the operation expresses a broadcast. For instance `(V <|spread)`, where `spread(i) = (0<=i<4) .(0)` means the value mapped at

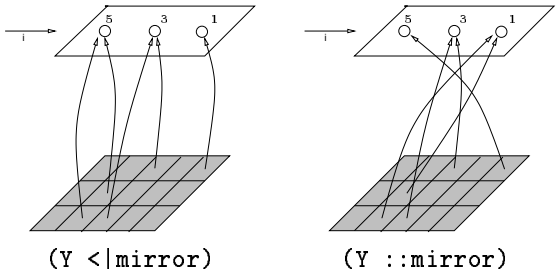

Fig. 6. Routing vs Change of basis

index point 0 in `V` is broadcasted to index points 0 to 3, to form the resulting shaped data field.
Note that change of basis modifies the index set and the shape, while only the index set is concerned with the geometrical operation. Fig. 6 outlines semantic differences between these operations if defined by the same function `mirror`, written in Pei as `mirror(i) = (2-i)`, applied to shaped data field `Y` of section 2.1.
This example can be enlightened in an intuitive way by considering that location and index sets look like a toy composed of two chips connected by a set of elastic strings : change of basis consists in twisting one chip round the other one in order to tangle or untangle the strings.

- the third operation computes the values of a shaped data field, and is called *global operation*. It performs an element-wise computation on the data field. Global operations are induced from classical n-ary operations ($-$, $+$ as examples). For example, `(V + 3)`, defines a data field whose values are computed from the `V` values having the same indices.

2.4. Example: Heat equation

Let us consider a thin uniform rod whose extremities are held to $0°C$. The temperature $U(x,t)$ of the rod at time t and distance x from one of its end, is defined by the following differential equation:

$$\frac{\partial U}{\partial t} = \frac{\partial U}{\partial x^2} \quad (1)$$

In order to simulate the diffusion of heat in the rod, a numerical solution uses a discretization in respect to time and space [12]. A finite-difference approximation of equation (1) is:

$$U_{i,j} = rU_{i-1,j-1} + (1-2r)U_{i,j-1} + rU_{i+1,j-1} \quad (2)$$

where r is a constant only depending on discretization parameters. Fig. 7 presents a framework of resulting values of U for $r = 5.10^5\,(m.s^{-2})$.

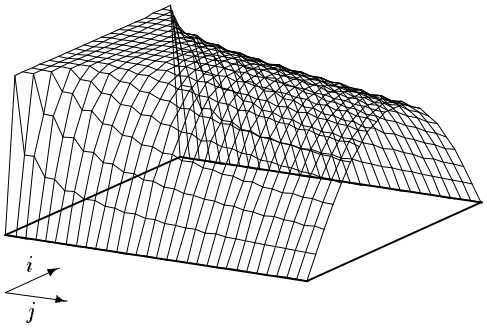

Fig. 7. Heat diffusion in a thin uniform rod

In Pei, U will be expressed by a shaped data field, say `U`, whose index set is a rectangle $[1..N] \times [0..T]$ in $\mathbb{Z}^2$ where N and T are discrete bounds of discretized space and time.

Dependencies in equation (2) are expressed by geometrical operations in Pei. These routings define four domains in the index set of `U` (cf. Fig. 8): the rod points (such that j=0), left and right borders and the domain inside (grey tinted). Values on borders are known: initial temperatures at rod points and 0 elsewhere. Values inside the rectangle are computed from values of shaped data fields `(U <|left)`, `(U <|middle)` and `(U <|right)` which are obtained by shifting `U` up-left, up, and up-right (according to the figure orientation) as this equation says:

```
U <|inside = r*(U <|left) + (1-2*r)*(U <|middle) + r*(U <|right)
```

with:

```
inside(i,j) = (1<i<n & 0<j<=T) .(i,j)
left(i,j)   = (i+1,j-1)
middle(i,j) = (i,j-1)
right(i,j)  = (i-1,j-1)
```

Let us consider the vector whose components are the initial temperatures of the rod. The vector is naturally associated with a shaped data field `V` placed on the line $[1..N]$ in $\mathbb{Z}$. In Pei, we can write the relation between `V` and `U` as follows:

```
U <|thinrod = V ::alignrod

thinrod(i,j)  = (1<=i<=n & j=0) .(i,j)
alignrod(i,j) = (1<=i<=n & j=0) .(i)
```

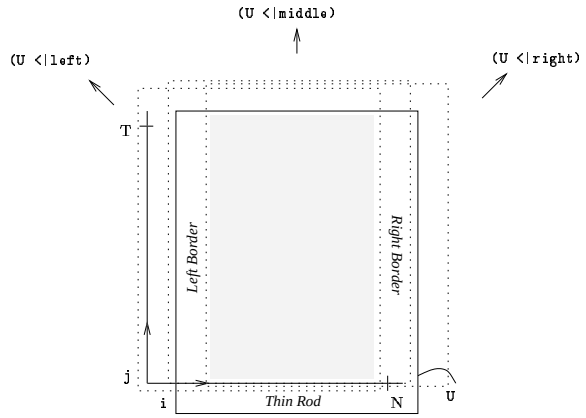

Fig. 8. Building `U`

Note that the equation implicitly defines both the index set of `V` and the way `V` is placed on `U`: this is the classical alignment issue in data parallel programming.

Here is a complete Pei statement for the problem:

```
DiffHeat[n,T,r] : V -> U
  {
  U <|bord    = 0
  U <|thinrod = V ::alignrod
  U <|inside  = r*(U <|left) + (1-2*r)*(U <|middle) + r*(U <|right)
  }

alignrod(i,j) = (1<=i<=n & j=0) .(i)
bord(i,j)     = ((i=1 | i=n) & 0<j<=T) .(i,j)
thinrod(i,j) = (1<=i<=n & j=0) .(i,j)
inside(i,j)  = (1<i<n & 0<j<=T) .(i,j)
left(i,j)    = (i+1,j-1)
middle(i,j) = (i,j-1)
right(i,j)  = (i-1,j-1)
```

3. PROGRAM STRUCTURATION

Modularity is an important issue in programming. It is achieved in Pei by *structuring* statements or objects.

3.1. Structuration of statements

In PEI, equation of the form: E_0 = P(E_1)
where P is the name of a program, and E_0 and E_1 are shaped data field expressions, may be written. Intuitively, such an equation means that the shaped data fields denoted by the expressions E_0 and E_1, satisfy the relation expressed by the program P.

Example 8. (The matrix product) The matrix product can be written in PEI by using the previous parameterized statement `PrefixSum[n]`:

```
MatProd[n] : A,B -> C
  {
  P = (A <|align_row)*(B <|align_col)
  C = PrefixSum[n](P)
  }

align_row(i,j,k) = (1<=i,j,k<=n) .(i,k)
align_col(i,j,k) = (1<=i,j,k<=n) .(k,j)
```

The given matrices denoted by input shaped data fields `A` and `B` are broadcasted along axes j and i respectively in a cube of size N. Elements of the matrices are then superimposed in order to compute all products $a_{i,k} \times b_{k,j}$. Last, products are added using `PrefixSum[n]` considering their projection along the k-axis as a N-vector. The N^2 summations result in the output `C`.

3.2. Structuration of objects

Shaped data fields express values mapped onto some index set which is a part of $\mathbb{Z}^n$. These objects are mainly characterized by operations (routing or change of basis) which do not involve values but only indices in $\mathbb{Z}^n$. In some sense, it means that objects are characterized by their structure, i.e. the way their index set is defined. For example, the previous 3D-data field `P` is defined as a 2D-array of N-vectors.

Example 9. Let us consider a $N \times N$-matrix, i.e. a data field whose values are placed on a square of $\mathbb{Z}^2$ and suppose we want to define an operation which shifts values one column right.

The way a shaped data field `A` expresses this matrix will induce different definitions for this operation:

- the matrix can be considered as formed of N^2 elements that are its values. In that case, the operation is written as `(A <|shift)` with `shift(i,j) = (i,j-1)`.
- the same operation can be written differently considering that it only involves the columns of the matrix. We can write: `(A <|shift')` with `shift'(j) = (j-1)`. In that case, the matrix is considered as formed of N columns: the values are placed on indices from 1 to N in $\mathbb{Z}$ and the index set of `A` is considered as a line in $\mathbb{Z}$.

Such a point of view can be used through a change of basis in order to express programs in a modular way.

Example 10. Let us consider a $(2N) \times (2N)$-matrix. It can be seen as a $N \times N$ block matrix whose "values" are blocks of size 2×2. Assuming the shaped data field `A` expresses the matrix, it can be block-organized by applying the change of basis defined by the bijection `by_block`:

```
by_block(x,y,i,j) = (1<=i,j<=n & 1<=x,y<=2) .(2(i-1)+x,2(j-1)+y)
```

Let `A0` be the shaped data field which satisfies `A0 = A ::by_block`. Then, `A0` can be used as a matrix whose index set is $[1..N] \times [1..N]$, any index identifying a 2×2 block. For example, this new layout of the matrix can be used to write the block-matrix product:

```
BlockMatProd[n,h] : A,B -> C
  {
  A0 = (A ::by_block) <|align_row
  B0 = (B ::by_block) <|align_col
  P ::step_wise = MatProd[h]((A0 ::elt_wise), (B0 ::elt_wise))
  C = PrefixSum[n](P)
  }

by_block(x,y,i,j) = (1<=i,j<=n & 1<=x,y<=h) .(h(i-1)+x,h(j-1)+y)
align_row(i,j,k)  = (1<=i,j,k<=n) .(i,k)
align_col(i,j,k)  = (1<=i,j,k<=n) .(k,j)
elt_wise(i,j,k,x,y) = (x,y,i,j,k)
step_wise(i,j,k,x,y) = (1<=i,j,k<=n & 1<=x,y<=h) .(h(i-1)+x,h(j-1)+y,k)
```

Matrices `A` and `B` of size $(hN)\times(hN)$ are block-organized by the change of basis operation `(::by_block)`. In the context of the program, we use names `(i,j)` to identify a block of size $h\times h$ and `(x,y)` to identify an element inside a block.

The blocks are then broadcasted into a cube by using operations `(<|align_row)` and `(<|align_col)` and form two cubes (whose third dimension is named `k`) of blocks. They are reorganized as two $h\times h$ matrices of elements by using the change of basis `elt_wise` which swaps `(i,j,k)` dimensions for `(x,y)` ones. Products of these two $h\times h$ matrices are then computed using a call to the program `MatProd[h]` which performs N^3 elementary-matrix products.

The intermediate result `P` is implicitly defined. By definition of `step_wise`, `P` is block-organized as a $(hN)\times(hN)$-matrix of N-vectors. Last, the call to the program `PrefixSum[n]` perform $(hN)^2$ summations and result in the output `C`.

4. PROGRAM TRANSFORMATIONS IN PEI

The ability to master efficiency of computations is of particular interest in data-parallel programming. It can be achieved by stepwise program transformations such as parallelization of a sequential program or generation of data alignment directives. Such transformations can be formally expressed in PEI.

4.1. Transformation rules of PEI

The semantics of PEI allows to define program transformations, as soon as a shaped data field is substituted for any equivalent one in an expression, or by applying some *algebraic law* on the operations. Here is a few examples of such laws (a detailed list is given in [15]).

$$
\begin{array}{lcl}
(\mathrm{uop}\ \mathrm{E}) <\!| f & = & \mathrm{uop}\ (\mathrm{E} <\!| f) \\
(\mathrm{uop}\ \mathrm{E}) :: f & = & \mathrm{uop}\ (\mathrm{E} :: f) \\
(\mathrm{E}_0\ \mathrm{bop}\ \mathrm{E}_1) <\!| f & = & (\mathrm{E}_0 <\!| f)\ \mathrm{bop}\ (\mathrm{E}_1 <\!| f) \\
(\mathrm{E}_0\ \mathrm{bop}\ \mathrm{E}_1) :: f & = & (\mathrm{E}_0 :: f)\ \mathrm{bop}\ (\mathrm{E}_1 :: f) \\
(\mathrm{E} <\!| f_0) <\!| f_1 & = & \mathrm{E} <\!| (f_0 \circ f_1) \\
(\mathrm{E} :: f_0) :: f_1 & = & \mathrm{E} :: (f_0 \circ f_1) \\
(\mathrm{E} <\!| f_0) :: f & = & (\mathrm{E} :: f) <\!| f_1 \quad \text{if} \quad f_0 \circ f = f \circ f_1
\end{array}
$$

where uop and bop express respectively unary and binary global operations. Some of these laws require hypotheses to be satisfied. This point is not detailed here.

Such transformation rules are very interesting either to improve program efficiency, or to parallelize a given code or to align data in a data parallel programming model. Here is an example dealing with parallelizing nested loops.

4.2. Time-space transformation

Among classical program transformations, the rewriting of loop nests is a parallelization technique a compiler could apply (see the wide literature on this topic [4, 1, 7], etc.). Let us consider a simple example.

Example 11. Nested Loops.

Assuming that $a_{i,1}$, $i \in [1..N]$ and $a_{1,j}$, $j \in [1..N]$ are input data, let us consider the following sequential loop nest:

```
do i=2,n
 do j=2,n
  a(i,j)=a(i-1,j)+a(i,j-1)
 enddo
enddo
```

From a dependence analysis an affine time×space transformation may apply (see Fig. 9), defined as:

$$\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ N \end{pmatrix}$$

The new iteration space can be scanned by a parallel loop which describes a computation front, within a sequential loop spending the time. Considering such a compiling

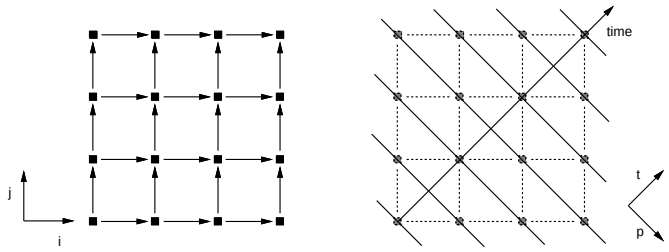

Fig. 9. Dependences and time-space transformation

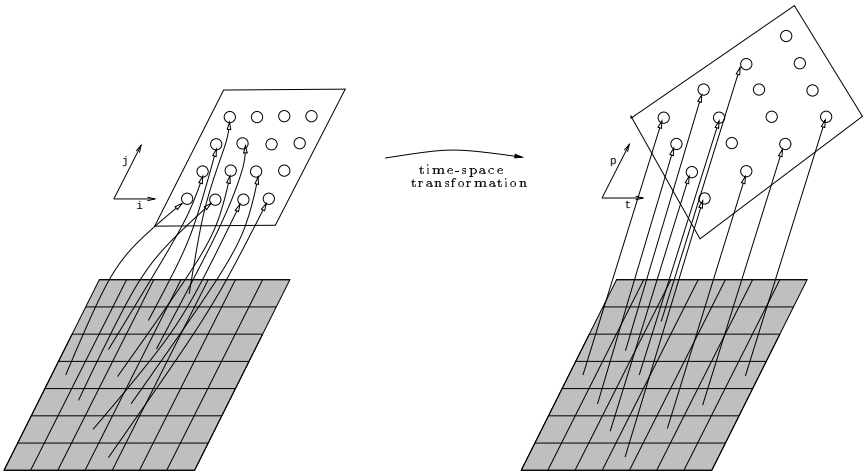

Fig. 10. Time-space transformation applied on shaped data fields in PEI

technique in the formalism PEI means: *apply a change of basis operation.* As said in section 2.3, this transformation leads to untangle the set of elastic strings which connect locations and indices. This is illustrated on Fig. 10.

Here is the example written in PEI:

```
NestedLoops[n]: D -> A
  {
  A <|bord = D ::bord
  A <|next = (A <|right) + (A <|up)
  }

bord(i,j) = (1<=i,j<=n & (i=1 | j=1)) .(i,j)
next(i,j) = (1<i,j<=n) .(i,j)
```

```
right(i,j) = (i-1,j)
up(i,j) = (i,j-1)
```

Let us consider this bijection for the change of basis operation:

```
time_space(t,p) = ((p+t-n+1)%2 =0) .((p+t-n+1)/2,(t-p+n+1)/2)
```

By definition of Pei, it expresses the inverse of the bijection which defines the time × space transformation above-mentioned. Needed hypotheses are satisfied: this change of basis can apply on the equations and come into the expressions:

```
(A <|bord) ::time_space = (D ::bord) ::time_space
(A <|next) ::time_space = ((A <|right) + (A <|up)) ::time_space
```

It rewrites as:

```
A' <|bord' = D' ::bord'
A' <|next' = (A' <|right') + (A' <|up')
```

with `A' = A ::time_space` and `D' = D ::time_space`, and where `bord'`, `right'` and `up'` are the functions the transformation leads to.
Expressions such as `(A <|right) ::time_space` are transformed by applying algebraic laws on operators. It can be rewritten as `(A ::time_space) <|right'` provided `right o time_space = time_space o right'` (idem for the other functions). One can define:

```
bord'(t,p) = (1<=p+t-n,t-p+n<=2n-1 & (p+t=n+1 | p-t=n-1)) .(t,p)
next'(t,p) = (1<p+t-n,t-p+n<=2n-1) .(t,p)
right'(t,p) = ((p+t-n+1)%2 =0) .(t-1,p-1)
up'(t,p) = ((p+t-n+1)%2 =0) .(t-1,p+1)
```

The predicates of these last functions can be simplified by using the Fourier elimination and yield the following Pei program:

```
ParallelLoops[n]: D' -> A'
  {
  A' <|bord' = D' ::bord'
  A' <|next' = (A' <|right') + (A' <|up')
  }

bord'(t,p) = (1<=t<=n & (p=t+n-1 | p=n-t+1)) .(t,p)
next'(t,p) = (1<=t<=2n-1 &
                  max(1,n-t+2,t-n+1)<=p<=min(2n-1,t+n-2,3n-t-1)) .(t,p)
right'(t,p) = ((p+t-n+1)%2 =0) .(t-1,p-1)
up'(t,p) = ((p+t-n+1)%2 =0) .(t-1,p+1)
```

Such a statement could be easily translated in a data parallel language in which a *masking instruction* determines active virtual processors among the array of $(2N-1)$ processors.

5. PEI AND DATA PARALLELISM

Previous examples have shown that the relationship between a PEI program and a data-parallel one is quite obvious since PEI expresses in an abstract way parallel variables, data alignments and global operations and communications.

In this section, we focus on a major semantic point in data-parallel languages: the semantics of indices in a variable. In languages like HPF, the indices in a program refer to indices of arrays, without any reference to data location. The association between indices and locations is a possibly not injective function: an array index can address several locations. In these languages, this allows to define data alignments through directives. Then, if the program expresses that a value is moved from an index to another one, it may involve any other relation between locations: we say that communications are hidden.

In other data parallel languages like C*, indices refer to virtual processors: $x[i]$ means the local value of x in the ith processor. The association between indices and locations is injective. Every index refers to only one location and any move of a value between two indices in a program, involves a similar move between the two locations of the variable. Therefore, parallel variables are distributed on the mesh and communications are explicit.

The language $\mathcal{L}$ [2] was proposed as a formal semantics for this last kind of languages. PEI proposes an extension to both semantics.

Example 12. The matrix product (continued). We illustrate this point by considering two versions of this problem. Let us first recall the previous program:

```
MatProd[n] : A,B -> C
  {
  P = (A <|align_row)*(B <|align_col)
  C = PrefixSum[n](P)
  }

align_row(i,j,k) = (1<=i,j,k<=n) .(i,k)
align_col(i,j,k) = (1<=i,j,k<=n) .(k,j)
```

According to the routing definition (see section 2.3), in this statement the shaped data fields `A` and `(A <|align_row)` have the same shape (which maps indices of both matrix and cube onto locations).
Moreover, since two different indices cannot be associated with the same location in a shape, an index of the cube cannot be associated with the same location as an index of the matrix. Last the shape of these data fields can be a bijection as shown on Fig.11.

Here is another PEI statement for the matrix product. It describes the same algorithm, but dependences (involving communications) have been replaced by references to locations (involving data alignment), i.e. change of basis is substituted for routing.

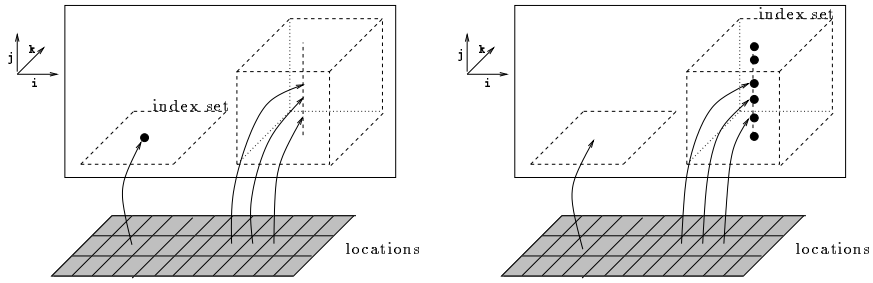

Fig. 11. Data fields `A` and `(A <|align_row)`

```
MatProd[n] : A,B -> C
  {
  P = (A ::align_row)*(B ::align_col)
  C = PrefixSum[n](P)
  }

align_row(i,j,k) = (1<=i,j,k<=n) .(i,k)
align_col(i,j,k) = (1<=i,j,k<=n) .(k,j)
```

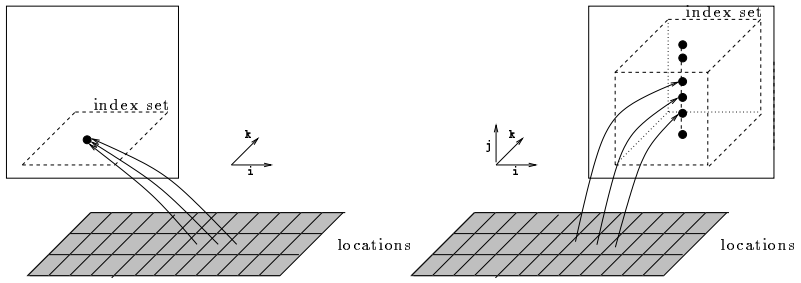

Fig. 12. Data fields `A` and `(A ::align_row)`

By definition, the shaped data fields `A` and `(A ::align_row)` have two different shapes, but the same values are associated with the same locations. On the example, it means that any index in `A` is necessarily attached to several locations: hence considering a vertical line of the cube (as shown on Fig. 12), every index on this line is attached to at least one location. Moreover, values on this line are the one located at an unique index in `A`. Since `A` and `(A ::align_row)` have the same values at the same location, any index in `A` is necessarily attached to all the locations associated with the index on

the line in `(A ::align_row)`. This means that the shape of `A` cannot be a bijection.

As said before, in the first statement, an index in `(A <|align_row)` can be attached to only one location. This statement can thus be interpreted in different intuitive semantics of data parallel languages, for example in C*.

On the contrary, in the second example, an index in `A` is attached to several locations: the association between indices and locations is not a bijection. It means that this statement expresses virtual indices in an array, which can be associated with different locations on a machine. This is typically what expresses data alignment in HPF, through an `ALIGN` directive.

These observations outline semantic differences between data parallel programming languages.

6. CONCLUSION

In the area of scientific programming, parallelism is now a major challenge. Besides low level programming languages, which may ensure an actual efficiency, general purpose or abstract high level languages are required in order to manage a safe programming.

This paradox can be solved if an abstract machine-independent language is associated with a sound transformation theory. So, a parallel program may be developed from an abstract statement until a detailed and optimized program, which takes data distribution and communications into account, according to given architectural constraints.

This was the aim of PEI, an equational language which is founded on the concept of shaped data field and an equivalence definition. Due to the unifying aspect of this theory, solutions that can be reached by these transformations are relevant to various parallel programming models, such as "systolic" processing or data parallelism. In this paper we particularly focused on data parallelism, and we have shown how the formal description of the language PEI can serve as a sound semantics for this programming model.

REFERENCES

1. U. Banerjee. *Loop Transformations for Restructuring Compilers: the Foundations.* Kluwer Academic Publishers, 1993.
2. Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *FGCS*, 8:363–378, 1992.
3. M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. Chapter 7.
4. P. Feautrier. Some efficient solutions to the affine scheduling problem, part 1, one-dimensional time. *Int. Journal of Parallel Programming*, 21(5):313–348, 1992.
5. Stéphane Genaud, Eric Violard, and Guy-René Perrin. Transformations techniques in PEI. *EUROPAR'95, LNCS*, 966:131–142, August 1995.
6. N. Halbwachs, P. Caspi, D. Pilaud, and J.A. Plaice. LUSTRE / a declarative language for programming synchronous systems. *P.O.P.L.*, 215:178–188, 1967.
7. C. Lengauer. Loop parallelization in the polytope model. *Parallel Processing Letters*, 4(3), 1994.

8. B. Lisper. *Data Parallelism and Functional Programming.* LNCS 1132-Tutorial Series, 1996.
9. C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.
10. O. Michel, D. De Vito, and J.-P. Sansonnet. 8 1/2 : data-parallelism and data-flow. *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming. World Scientific*, May 1996.
11. D.B. Skillicorn and D. Talia. Models and languages for parallel computation. Technical report, Queen's University, October 1996. also published in Computing Surveys.
12. G.D. Smith. Numerical solution of partial differential equations: finite difference methods. *Oxford Applied Mathematics and Computing series, Oxford University Press*, 1985.
13. E. Violard and G.-R. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.
14. E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694:500–516, June 1993.
15. E. Violard and G.-R. Perrin. Reduction in PEI. *CONPAR'94, LNCS 854*, pages 112–123, 1994.