What Really is Data Parallel Programming?

E. Violard

ICPS, Université Louis Pasteur, Strasbourg

Pôle API, Boulevard Sébastien Brant, F-67400 Illkirch

E-mail: violard@icps.u-strasbg.fr

Abstract

We developed a theory in order to address crucial questions of program design methodology. We think that it could unify two concepts of data-parallel programming that we consider fundamental as they concern data locality expression: the notions of alignment *in HPF and* shape *in* C^**. Our wish is to provide a semantic domain where data-parallel statements can be systematically proved as well as efficiently implemented.*

Keywords. Data Parallel Programming, Equational Languages, Parallel Programs Design Methodologies, Proof of programs, Shape in Parallel Programming.

Technical areas. Compilers and Languages, Programming Methodologies and Software.

1. Introduction

The interest of concepts beyond any programming language is that they express the relationship between what the programmer knows about the problem he wants to address and what the compiler knowns about the architecture on which the program will execute. These concepts state a connection between those two individual knowledges. They consist then in an abstract knowledge that the two entities (programmer and compiler) can share and can use to interact and finally perform the best implementation.

This is even more true when the architecture is a parallel one. The programmer and compiler both should bring their contribution: more exactly, they should influence each other in order to paliate the difficulties on each part. For example, based on these concepts, one could define a language in such a way that a program should be transformed by both entities in a dialog started by the programmer and ended by the compiler. The main point is then to find a good medium, i.e., the concepts that the two entities could handle.

As illustrated on Fig.1, any approach of parallelization establishes a level of collaboration between the programmer and the compiler workloads in order to reach efficiency. At one end of the sprectrum, a message passing programming model (such as PVM, or MPI) let the programmer be in charge of all the workload, whereas on the other side, automatic parallelization gives all the responsibility to the compiler. Data-parallelism permits a high degree of collaboration.

In particular, a major interest of the *data parallelism paradigm* is that it enables the programmer to describe its "parallel" algorithm by choosing a well-known "sequential" one and then focussing on the expression of *data locality*, while the compiler can be left in charge of the distribution onto physical processors, communications management, etc. In some sense, it reveals that the concepts for expressing data locality are of main interest in parallel programming and even may be viewed as the "essence of parallelism" in comparaison with sequential programming.

As examples, let us consider standard data parallel programming languages as HPF [7] and C* [13]: both allow data locality to be expressed and then exploited by the compiler. Locality in both of these languages is stated via a more or less explicit correspondance between indices of arrays and virtual processors.

However, these languages use two different concepts for expressing this correspondance:

- In HPF, the *alignment* between two arrays (using the directive `ALIGN`) can be considered as a typical notion for expressing data locality: it provides a way to inform the compiler that certain data elements should reside in the same virtual processor. By default an array is not aligned with any other one, so that the compiler must guess alone the best distribution onto physical processors.

- In C*, the *shape* notion is the way of logically configuring data: it is a kind of type which is associated with each object (in a declaration using the keyword `shape`). Typically, the corresponding elements of two arrays having the same shape reside in the same virtual processor. This notion of type entails some restrictions on the expressions which can be written: for example, the assignment of two (so called) *parallel variables* is forbidden if they are not of the same shape.

Thus, HPF and C* have different intuitive semantics for data locality. The following question then comes in mind: what really is data-parallel programming? HPF or C*? It is interesting to note that the two concepts *alignment* in HPF and *shape* in C* introduce an unbalanced choice for the programmer and the compiler:

- either the programmer provides sufficient effort to adapt its sequential algorithm and the compiler work is easier (C*),
- or the programmer just takes its sequential code again and a lot of work is left to the compiler to find the best placement given alignment constraints (HPF).

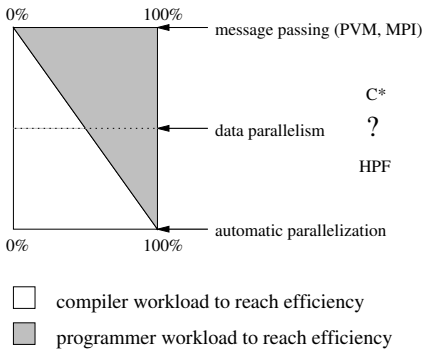


Fig. 1. Programming paradigms each state a collaboration level between programmer and compiler

We think that the programmer and the compiler must ideally have a fair share. Since HPF and C* each define a different share between the programmer and the compiler, it is certainly possible to design data-parallel languages where the main concepts of data-parallelism are more concentrated. Our aim is then to find these concepts.

In this article, we propose a theory which could bridge the gap between *alignment* and *shape* concepts: it defines a purer notion of data locality from which these two concepts could derive.

In order to carefully present this theory, the paper is divided into the following sections: the next section recalls the problem and illustrates it with a few simple examples in HPF and C*. Section 3 lists some related works. Section 4 introduces the theory with an informal meaning. Section 5 presents a sound model of our theory which provides a formal framework for demonstrating properties of data parallel programs. Proof of programs is the purpose of section 6 just before conclusion.

2. Some examples

Let us present some simple examples that are designed to show the lack of balance between programmer and compiler investments in C* and HPF.

• Let us consider the two following programs in HPF:

```
      REAL A(0:9,0:9),B(0:9,0:9)

!HPF$ ALIGN A(I,J) WITH B(I,J)

      ...
      B(1:9,1:9) = A(1:9,1:9) +
     &             A(0:8,1:9) +
     &             A(1:9,0:8)
      ...
```

```
      REAL A(0:9,0:9),B(0:9,0:9)

!HPF$ ALIGN A(I,J) WITH B(I,J)

      ...
      FORALL ( I=1:9, J=1:9 )
        B(I,J) = A(I,J) +
     &           A(I-1,J) +
     &           A(I,J-1)
      END FORALL
      ...
```

These two programs are equivalent: they only differ syntactically and could illustrate the dual macroscopic and microscopic views of data parallel programs [2]. Although the second syntactic form is not allowed in C*, there is no important difference between HPF and C* for expressing parallelism. Here is an equivalent program in C*:

```
      shape [10][10]matrix;
      real:matrix A, B

      ...
      where ((pcoord(0)>0) && (pcoord(1)>0))
         B = A + [.-1][.]A + [.][.-1]A;
      ...
```

The indices of variables have a different intuitive semantics; but in that case, no extra investment is in charge of the programmer in comparison with the HPF program. Just an introductory example then!

• Let us consider this new example in HPF:

```
      REAL A(0:7),B(0:7),C(0:7)

!HPF$ TEMPLATE X(0:14)
!HPF$ ALIGN A(I) WITH X(I+1)
!HPF$ ALIGN B(I) WITH X(2*I)

      ...
      C = A + B
      ...
```

where template X is used for defining the alignment of array A with array B, and the alignment of array C is left in charge of the compiler. In our sense, the HPF program could be encoded as follows in C*:

```
shape [15]vector';
real:vector' A',B',C';
...
C' = [.+1]A' + [.*2]B';
```

where A', B' and C' contain the same values as A, B and C, but at different indices: the vectors A, B and C have to be reindexed in order to describe a correspondance between vector values and virtual processors, which respects the alignment constraints of the HPF program. In particular, this redexing entails to modify the number of positions of vectors (to 15). Moreover, communications have to be made explicit.

Note that an interesting possibility in C* is to map a parallel variable to another shape (by using a get operation). This allows the correspondance between the indices of A, B and those of A', B' to be expressed. So that the program could be written:

```
shape [8]vector;
real: vector A, B, C
shape [15]vector';
real: vector' A',B',C';
...
A' = [.-1]A;
B' = [./2]B;
C' = [.+1]A' + [.*2]B';
C  = [.]C';
```

However, the operational behaviour of this program is different because it includes unnecessary assignment steps.

Translating this simple example from HPF to C* reveals some of the HPF compiler difficulties as much as those of the programmer in writing its program in C*.

The problem is much more apparent when we consider the expressiveness of these languages. For example, let us suppose that, independently of any syntactic restriction, we want to handle non linear alignments: in HPF, this will involve new techniques for the compiler, whereas in C*, the difficulty in writing an equivalent program will increase. Intuitively, it shows that increasing expressivness of these languages (by syntactic extensions) only emphasizes the lack of balance between the compiler and the programmer workload. Our ambition is then to find the equilibrium and to provide the programmer and the compiler with an ideal medium.

3. Related works

In [10], Björn Lisper gives a review of a new abstract programming formalism with the aim to generalize current data parallel constructs and to define a more abstract view of aggregate data. Like us, he has the desire to find a formalism where the "essence of data parallelism" is present in its purest form. The formalism he proposes is a minimal extension of a functional language, where objects called *data fields*, are partial functions with an *explicit restriction* operation. For example, in this formalism, expression $f \setminus \lambda i.(1 \leq i \leq n)$, where f is a partial function from integers to some data type, denotes a data field which is garanteed to be undefined for any integer i outside the interval $[1..n]$. Björn Lisper introduces a "parallel" evaluation mecanism which is different from the classical one for functional languages and which requires an *extent analysis* to be performed in order to tabulate data field values. This analysis uses these *explicit restrictions*. In this formalism, data locality is expressed via the classical λ-abstraction: for example, $\lambda ij.f(i)$ expresses the replication of data field f as columns into a matrix. Such expression could specifies an alignment in HPF as well as a cast – a form of shape declaration – in C*.

This approach is thus very close to ours in spirit. But in contrast with the Lisper's formalism which is associated with an intuitive meaning (different from the classical one for functional languages), we propose a model which could serve as a formal semantics for such formalisms.

Luc Bougé designed the language $\mathcal{L}$ [1] as a common kernel of data parallel languages like C*. The language $\mathcal{L}$ is quite simple and sufficient to express usual data parallel algorithms, but in comparison with a "sequential language", a few new constructs are introduced and proofs of correctness [3] yield unexpected complexity. As Luc Bougé said, "an interesting direction would be to extend this work to data parallel languages that take into account notions of data alignment and mapping".

The theory presented in this paper inherits from PEI [14]. PEI is a theory born from a generalization of the so-called Polytope model [9]. It was introduced to address crucial questions of program design methodology and transformations, through a straightforward generalization of equational notations [11, 5]. Its central concept is a notion of objects which are pairs of a partial function v from integer tuples and a bijection σ from the same integer tuples: v can be seen as a data field in the sense of Björn Lisper and σ can be interpreted as a space-time mapping. In this paper, we define a looser association between these two functions.

In the sequel, we precise the differences between these approaches and our theory.

4. Our theory

The theory defines a notion of objects, called *shaped data fields*, which are associated with specific operations.

4.1. Objects

A shaped data field is mainly a container of values. Containers without values are sometimes referred to as shapes – in [8], B. Jay defines a very general and abstract notion of shape via a categorical pullback –. The originality of our theory consists in its particular notion of shape.

In our theory, a shape is a system composed of two sets of points, respectively called *indices* and *locations*, and some arrows between them. Any point belongs to some "geometrical space", i.e., $\mathbb{Z}^n$.

A *location* expresses a position where one or possibly zero value can be placed. Each *index* of a shape is connected to one or more locations and allows all the values at these locations to be acccessed "as a whole": all these values are assumed to be equal. Conversely, every location is accessed by one index.

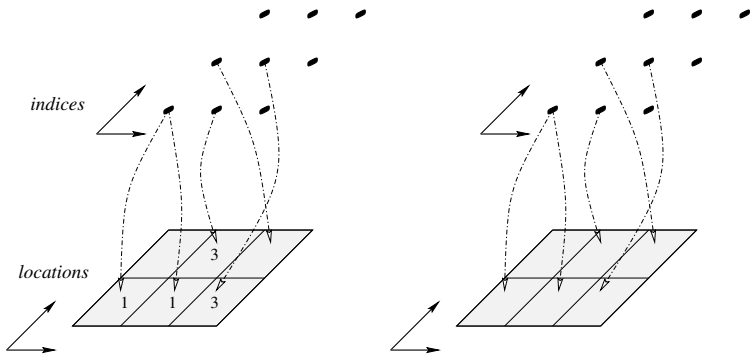


Fig. 2. A shaped data field and its shape

Example 1. Figure 2 shows an example of a shaped data field (and its shape on the right side). It contains four values: 1, twice, and 3, twice. Each value has its own location: for example, an instance of value 1 is at location $(0, 0)$ while the other one is at location $(1, 0)$. Locations are associated – generally, in a non-univoque fashion – with indices: so that, values are indexed (in the index set) by the points in $\mathbb{Z}^2$ of coordinates $(0, 0)$, $(1, 0)$ and $(1, 1)$: point $(0, 0)$ refers to two locations while

the two others are connected with only one. Note that point $(0, 1)$ is also connected with one location (it is thus a component of the shape), but this location is empty: no value is placed at this location.

◇

Thus, every value of a shaped data field has a location and is accessed via an index in the index set. In the litterature, indexed collections of values, detached from positions, are often referred to as *data fields* (see Alpha [11], Lisper formalism [6] as examples). It should be clear now that the theory defines a shaped data field as a data field associated with a shape.

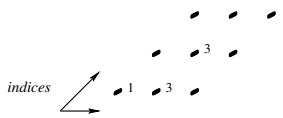


Fig. 3. A data field

Example 2. The shaped data field on Fig.2 associates a shape with the data field on Fig.3. This data field is composed of values 1, once, and 3, twice: these values are indexed by $(0, 0)$, $(1, 0)$ and $(1, 1)$, respectively. In other terms, the data field, in the sense of [11, 6], which is embedded in the shaped data field on Fig. 2, is the one drawn on Fig. 3.

◇

In the sequel, the index set of a shaped data field will naturally refer to the data field index set. Moreover, when we talk about values without more precision, it stands for the values of the shaped data field (placed on locations) and not the values of the data field (which forms a subset of).

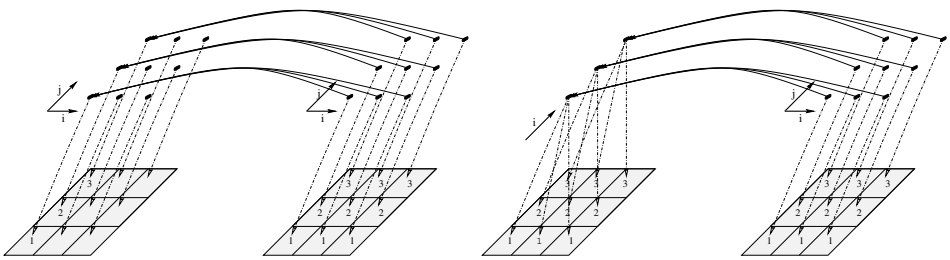
4.2. Operations

The theory defines three kinds of operations on shaped data fields:

- a *change of basis* concerns both the shape and the data field of a shaped data field: it changes the indices while keeping the values, and their locations unchanged.
- a *geometrical operation* does not affect the shape, but changes the indices of the values on the data field, and possibly deletes or duplicates some values.

  Any geometrical operation or change of basis is determined by a function from the indices of the new shaped data field to the indices of the old one.

- a *global operation* applies the same operation on all values. Any classical arithmetical or logical operation on values induces a global operation. In particular, any binary operation on values defines a global operation which combines two shaped data fields having the same shape.



(a) A geometrical operation (b) A change of basis

Fig. 4. Two examples of operation on shaped data fields

Example 3. Fig.4(a) shows two shaped data fields: the shaped data field on the right results from a geometrical operation applied to the shaped data field on the left. This geometrical operation is determined by the function which associates point of coordinates $(1, j)$ with any point (i, j) in the square $[1..3] \times [1..3]$ of $\mathbb{Z}^2$.

Fig.4(b) illustrates the change of basis defined by the function which associates point of coordinate i in $\mathbb{Z}$ with any point (i, j) in the same square. $\diamond$

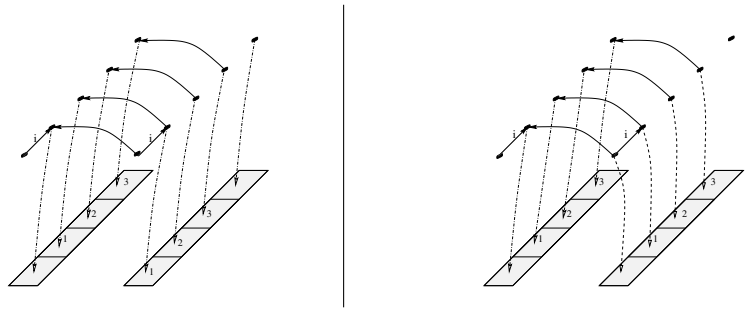


Fig. 5. Geometrical operation vs Change of basis

Example 4. Fig. 5 outlines the difference between a geometrical operation and a change of basis when they are determined by the same function: namely, the function which associates the point of coordinate $i+1$ with any point i in the line $[0..3]$ of $\mathbb{Z}$. In the case of the change of basis (on the right of the figure), the values keep their position; whereas they are "moved" in the case of the geometrical operation (on the left). In both cases, the resulting data field is the same, but associated with a different shape. $\diamond$

4.3. A minimal notation set

Here we introduce a minimal notation set for expressing problems or programs: both are particular cases of what we call *statements*.

A *statement* is a finite set of equations whose variables are shaped data fields. Each equation connects two *expressions* built using the operations on shaped data fields. A variable is an uppercase letter (e.g., A, B, X, ...). The classical arithmetical or logical operations are overloaded with global operations (e.g., $A + B$).

A notation is also needed for a geometrical operation and a change of basis: it is the only new notation we use. We will note $X.F$ where F is a pair (h, g) of partial functions, the result of applying first the change of basis determined by h, and then the geometrical operation determined by g, to X.

Therefore, a change of basis or a geometrical operation can be denoted by $X.F$, when either g or h is the identity.

In order to denote partial functions, we use the classical lambda-calculus notation $\lambda x.e$ and $f \setminus D$ for the restriction of function f to domain D.

Example 5. Let us consider the geometrical operation illustrated on Fig.4(a). The result of this operation applied to a shaped data field, say A, is a shaped data field which can be denoted by $A.Spread$, where $Spread$ is pair (h, g), h is the identity (on D) and g is partial function $\lambda(i, j).(1, j) \setminus D$, with $D = \{ (i, j) \mid 1 \leq i, j \leq 3 \}$. Then, if A and B are the shaped data fields on Fig.4(a), then they satisfy the equation $B = A.Spread$.

Similarly, the shaped data fields, say A' and B', on Fig.4(b) verify the equation $B' = A'.Spread'$, where $Spread'$ is the pair composed of partial function $\lambda(i, j).(i) \setminus D$ and the identity (on D), with $D = \{ (i, j) \mid 1 \leq i, j \leq 3 \}$. The expression $A'.Spread'$ denotes the result of a change of basis applied to A'.

$\diamond$

4.4. Theory adequacy

In this section, we consider several examples that can illustrate the adequacy of the previously defined theory: the next section provides a model to prove the following assertions.

Example 6. Let us consider the assignments $\mathtt{B}[i] = \mathtt{A}[i+1]$, for all $i \in [0..6]$, where we use the common notation for array elements and where $\mathtt{A}[i+1]$ and $\mathtt{B}[i]$ are legitimate elements for arrays A and B. In our theory, such dependence between arrays A and B can be expressed as follows:

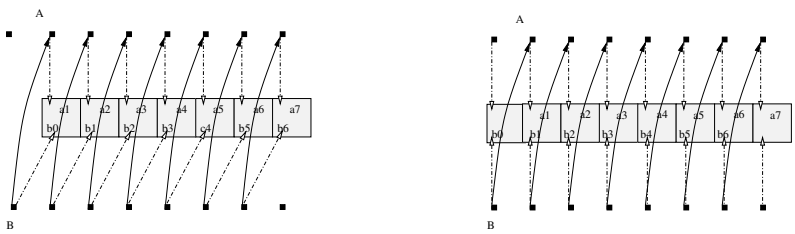
$$B = A.Shift \tag{1}$$

where A and B are shaped data fields which represent the arrays A and B, and $Shift$ is any pair of functions (h, g) satisfying $(h \circ g) = \lambda(i).(i+1) \setminus D$ with $D = \{ (i) \mid 0 \leq i \leq 6 \}$.

For example, Fig.6 represents two shaped data fields A and B which satisfy equation 1 for two different pairs (h, g) of functions: on the left, the case where $h = \lambda(i).(i{+}1) \setminus D$ and g is the identity (on D); on the right, the "symmetrical" case where $g = \lambda(i).(i{+}1) \setminus D$ and h is the identity (on $\{\ (i) \mid 0{\leq}i{\leq}7\ \}$).

In each of these two cases, the value of B at any location accessed by an index $i \in [0..6]$, is equal to the value of A at any location accessed by index $i{+}1$. Considering A and B as arrays, these two cases thus express the relation $\mathtt{B}[i] = \mathtt{A}[i{+}1]$.

These two cases also express a different placement of values onto locations. When g is the identity, there is no communication because the values $\mathtt{B}[i]$ and $\mathtt{A}[i{+}1]$ have the same locations. When h is the identity, it means that A and B have the same shape. In that case, $\mathtt{B}[i]$ and $\mathtt{A}[i{+}1]$ forcefully have different locations and a communication is required.



(a) g is the identity (b) h is the identity

Fig. 6. Array `B` obtained by shifting array `A`

These two particular cases could be written in HPF and C* respectively, as follows:

```
      REAL A(1:7), B(0:6)

!HPF$ ALIGN B(I) WITH A(I+1)

      ...
      B(0:6) = A(1:7)
      ...
```

```
shape [8]vector;
real:vector A, B;

...
B = [.+1]A;
...
```

In the case of the HPF program (when g is the identity), if the compiler takes the alignment into account, then there is no communication. In the case of the C* program (when h is the identity), the communications are explicit. ⋄

Example 7. Let us consider again the second example in section 2: arrays `A`, `B` and `C` each depend on the others according to relation $\mathtt{C}[i] = \mathtt{A}[i] + \mathtt{B}[i]$, for all $i \in [0..7]$. It is expressed as `C = A + B` in HPF independently of the placement of values.

According to the concepts of our theory, this relation could be expressed as the following statement:

$$C = A.Id + B.Id'$$

where Id (Id') is a pair (h, g) ((h', g')) of functions such that $h \circ g$ ($h' \circ g'$) is the identity on the domain $[0..7]$. In our theory, each such pair of functions expresses a particular placement of arrays.

For instance, the placement of arrays in the HPF program could be expressed by the following definitions [1]:

$h = g^{-1}$, $g = \lambda(i).(i{+}1) \setminus D$ and $h' = g'^{-1}$, $g' = \lambda(i).(2{\times}i) \setminus D$ with $D = \{\ (i) \mid 0{\leq}i{\leq}7\ \}$, as depicted on Fig.7.

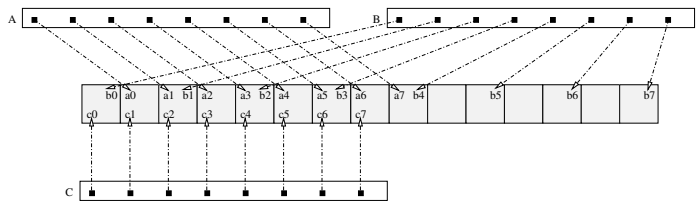


Fig. 7. Relative placements of arrays `A`**,** `B` **and** `C`

◇

Previous examples have shown that the relationship between a statement in our sense, and a data-parallel program is quite obvious since this theory defines in an abstract way parallel variables, data placement, global operations and communications. Here we confront our theory with 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 may not be one-to-one: 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 one-to-one. Every index refers to only one location and any move

[1] where f^{-1} stands for the inverse of f (provided f is injective)

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 $\mathcal{L}$ [4] was proposed as a formal semantics for this last kind of languages. A model of our theory would be an extension to both semantics.

Example 8. Suppose we want to compute all the products $\mathtt{A}[i,k] \times \mathtt{B}[k,j]$, for $1 \leq i, j, k \leq n$, in order to fill a three dimensional array `P` whose indices form a cube. Arrays `A` and `B` each contain the values of a $n \times n$-matrix. Here is a statement for this problem:

$$P = A.Spread_1 \times B.Spread_2$$

where $Spread_1 = (h, g_1)$, $Spread_2 = (h, g_2)$, h is the identity on domain $\{ (i,j) \mid 1 \leq i, j \leq n \} \cup D$, where D is domain $\{ (i,j,k) \mid 1 \leq i, j, k \leq n \}$, and functions g_1, g_2 are defined by $g_1 = \lambda(i,j,k).(i,k) \setminus D$, $g_2 = \lambda(i,j,k).(k,j) \setminus D$.

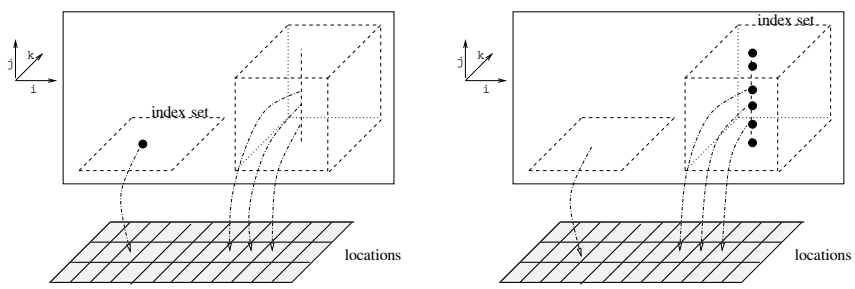


Fig. 8. Shaped data fields A and $A.Spread_1$

According to the geometrical operation, shaped data fields A and $A.Spread_1$ have the same shape. This shape 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 one-to-one correspondance between indices and locations, as shown on Fig.8.

Now, let us consider an other statement for the same problem. It describes a different algorithm as dependences (involving communications) have been modified and now refer to locations (involving data alignment), i.e., change of basis is substituted for geometrical operation:

$$P = A.Spread'_1 \times B.Spread'_2$$

where $Spread'_1 = (g_1, h')$, $Spread'_2 = (g_2, h')$ and h' is the identity on domain D.

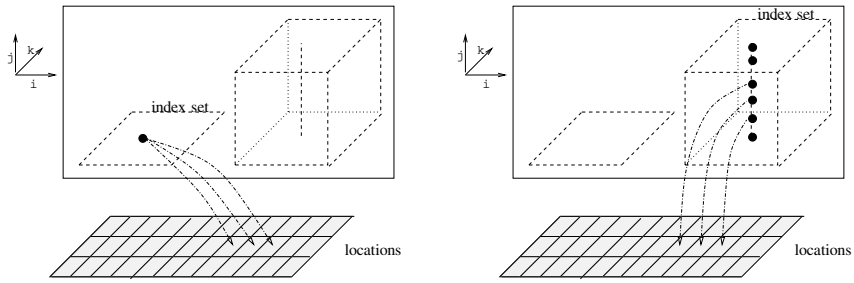


Fig. 9. Shaped data fields A and $A.Spread'_1$

By definition, the shaped data fields A and $A.Spread'_1$ 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 connected to several locations: hence considering a vertical line of the cube (as shown on Fig.9), every index on this line is connected to at least one location. Moreover, the values accessed by an index on this line, are the values which are accessed by an unique index in A. Thus, since A and $A.Spread'_1$ have the same values at the same locations, any index in A is necessarily connected to all the locations which are associated with the indices on the line in $A.Spread'_1$. This means that the shape of A cannot be a one-to-one correspondance between indices and locations.

As said before, in the first statement, an index in $A.Spread_1$ can be connected 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 connected to several locations: the correspondance between indices and locations is not one-to-one. 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. $\diamond$

These observations outline semantic differences between data parallel programming languages and we formalize them through a model of our theory.

5. A model of our theory

This section presents the mathematical definitions of shaped data fields and their associated operations. These definitions could serve as a semantic domain for data parallel languages.

5.1. Shaped data fields

The theory refers to mathematical objects called *shaped data fields*. This section includes their formal definition beginning with the related notions of *data field* and *shape*.

A *data field* models a collection of indexed values such that an index uniquely corresponds to a value. Any index is composed of the coordinates of a point in a "geometrical space", i.e., a given $\mathbb{Z}^n$: an index is thus an integer tuple. Note that, the points which serve for indexing the values of a given data field, may belong to different "geometrical spaces". For example, considering a data field with two values in it, one value may be indexed by a element of $\mathbb{Z}$ and the other one by an element of $\mathbb{Z}^2$.

In the sequel, we call I, the union of all $\mathbb{Z}^n$, for all $n \in \mathbb{N}$. A data field then associates a value in a given data type with some elements of I which form its *index set*. A data field is then formally defined as follows:

Definition 1. *Let V be a given data type. A* data field *(whose values are in V), is any partial function $\mathcal{X}$ from I to V.*

$$\mathcal{X} : I \rightharpoonup V$$

A *shape* expresses a set of arrows between two sets of points: indices and locations. These arrows describe a relation from indices to locations: indices form the domain of the relation and locations form its co-domain. Since the relation is injective (any element in its co-domain (a location) has at most one corresponding element in its domain (an index)), its inverse is a function. Therefore, a shape is defined by a function from locations to indices. Moreover, since locations and indices are two subsets of I, it is a partial function from I to I whose domain of definition is the set of locations, and whose image is the set of indices.

Definition 2. *A* shape*, is any partial function σ from I to I.*

$$\sigma : I \rightharpoonup I$$

A *shaped data field* X is a data field $\mathcal{X}$ associated with a shape σ. This association means that each index in the index set of the data field is an index of the shape: it belongs to the image of this shape, formally, $\mathrm{def}(\mathcal{X}) \subseteq \mathrm{img}(\sigma)$ [2]. In order to properly define this association, a shaped data field is defined via an high-order function as follows:

Definition 3. *Let V be a given data type. We call* shaped data field *(whose values are in V), a constant partial function X from shapes to data fields whose values are in V, that returns for every shape σ of its domain of definition, a data field $\mathcal{X}$, such that* $\mathrm{def}(\mathcal{X}) \subseteq \mathrm{img}(\sigma)$.

$$\begin{array}{ccc} X : (I \rightharpoonup I) & \rightharpoonup & (I \rightharpoonup V) \\ \sigma & \mapsto & \mathcal{X} \end{array}$$

$$(\forall \sigma \in \mathrm{def}(X) \,.\, \mathrm{def}(\mathcal{X}) \subseteq \mathrm{img}(\sigma))$$

5.2. Operations

From the previous definition of the shaped data fields, we can deduce the following definitions of the operations.

These definitions associate a mathematical meaning with the notation introduced in section 4.3. For sake of conciseness, the definitions are given using *semantic equations* [12]. Such equation defines the result of a given operation applied to arbitrary arguments. Moreover, the result of any operation is an high order function (a shaped data field): it is then defined itself by its image (a data field) when applied to an arbitrary argument (a shape). Last, in each of these semantic equations, it is implicit that the shaped data field applied to σ on the left handside, is undefined, iff all the values of the data field on the right side, are undefined.

[2] where $\mathrm{def}(f)$ stands for the domain of definition of f and $img(f)$ is a shorthand for the image of f, i.e., the set $\{\, f(x) \mid x \in \mathrm{def}(f) \,\}$.

Let $g, h, \sigma : I \rightharpoonup I$ and $G = (id, g)$, $H = (h, id)$, where id is the identity on I:

$$\textbf{Global operation: } (A + B)(\sigma) \stackrel{Def}{=} A(\sigma) + B(\sigma)$$

where the operation $+$ on the right handside of this semantic equation is the addition on data fields. This addition can be defined by the equation $(\mathcal{A} + \mathcal{B})(z) = \mathcal{A}(z) + \mathcal{B}(z)$, where the result is defined on the intersection of the domains of definition of the two arguments.

Intuitive explanation: For any two data fields $\mathcal{A}$ and $\mathcal{B}$ associated with an identical shape σ, and denoted by A and B, the result of the global operation is the data field $\mathcal{A} + \mathcal{B}$ associated with the same shape σ, and is denoted by $A + B$.

The previous definition easily generalizes for all the other arithmetical or logical operations.

$$\textbf{Geometrical operation: } (X.G)(\sigma) \stackrel{Def}{=} X(\sigma) \circ g \setminus img(\sigma)$$

Intuitive explanation: For any data field $\mathcal{X}$ associated with shape σ, and denoted by X, the result of the geometrical operation determined by g applied to X, is the data field associated with shape σ, which maps the value $\mathcal{X}(g(z))$ (if defined) to each index z of shape σ.

$$\textbf{Change of basis: } (X.H)(\sigma) \stackrel{Def}{=} X(h \circ \sigma) \circ h$$

Intuitive explanation: For any data field $\mathcal{X}$ associated with shape $h \circ \sigma$, and denoted by X, the result of the change of basis determined by h applied to X, is the data field associated with shape σ which maps the value $\mathcal{X}(h(z))$ (if defined) to each index z of shape σ.

Thus, X and $X.H$ have the same values placed at the same locations: consider a value of X at location l, since X has shape $h \circ \sigma$, its corresponding index is $(h \circ \sigma)(l)$, that is $h(z)$ with $z = \sigma(l)$. This value is thus equal to the value of $X.H$ at location l. This proof is summarized by the following commuting diagram:

$$\begin{tikzcd}
z \arrow[rr, "h"] & & h(z) \\
& l \arrow[ul, "\sigma"] \arrow[ur, "h \circ \sigma"'] &
\end{tikzcd}$$

Change of basis and then geometrical operation: $X.(h,g) \stackrel{Def}{=} (X.H).G$

Note that all these operations are deterministic in the sense that they associate a unique shaped data field with a given one. But, according to our association between data field and shape, the data field of the result is uniquely defined, whereas the shape which is associated with is generally not.

6. Proof of statements

The model presented in the previous section can serve as a base for demonstrating some properties of statements. In particular, it enables to prove *correctness* of a given statement, that is to check whether the statement expresses the expected relation between arrays elements or not, when we interpret each variable of the statement as an array.

Example 9. We aim to demonstrate that equation $B = A.Shift$ in example 6, where $h = \lambda(i).(i{+}1) \setminus D$ and g is the identity (on D) with $D = \{\ (i) \mid 0{\leq}i{\leq}6\ \}$, well expresses relation $\mathtt{B}[i] = \mathtt{A}[i{+}1]$, for all $i \in [0..6]$, where $[1..7]$ and $[0..6]$ are legitimate indices for arrays A and B, respectively. That was informally asserted until now. $\diamond$

In this section, we formalize our interpretation of shaped data fields as arrays and present a proof system which can demonstrate the assertions in section 4.4. This proof system applies to a class of statements in our theory for which shaped data fields can be interpreted as usual arrays. These statements are called *homogeneous* statements. An homogeneous statement is a *well-formed* statement with some additional properties.

6.1. Well-formed statements

Intuitively, a statement, say P, is well-formed if for each variable, say X, of P, a corresponding shape, say σ_X, can be inferred from P. Since this shape can be inferred, it is said to be canonical. In the following, a variable with its, so called, *canonical shape* can be viewed as a kind of array, to the extent that the indices refer to some locations. Well-formed statements are formally defined as follows:

Definition 4. *A* well-formed statement*, is a statement, say P:*

- *with an identified variable called* template *(and referred to as T),*

- *whose every equation is of the form:*

$$E_0 = \phi(E_1, E_2, \ldots, E_n)$$

  where ϕ is a global operation, each E_i ($i \in 0..n$) is of the form $X_i.F_i$, each F_i is a pair (h_i, g_i) of partial functions, X_i is a variable of P, and h_0 is the identity on I,

- *and such that, the following set of equations, denoted $S(P)$, whose variables are denoted by σ_X, for every variable X of P, has an unique solution for a given σ_T:*

  $S(P)$ is the set of all the equations:

$$\sigma_{X_i} = h_i \circ \sigma_{X_0}, \quad i \in 1..n$$

  induced from every equation of P.

Each equation $\sigma_{X_i} = h_i \circ \sigma_{X_0}$ (for any $i \in 1..n$) of $S(P)$, is deduced from the equation $E_0 = \phi(\ldots, E_i, \ldots)$ of P: from the definition of the change of basis determined by h_i in our model (cf. section 5.2), and since the other operations do not modify shapes, the equation of P says that the shape of X_i (which is equal to those of E_i) is equal to h_i composed with the shape the X_0 (which is equal to those of E_0).

The condition on $S(P)$ ensures that a unique (canonical) shape (namely σ_X) can be determined and finally associated with each variable (X) in the statement.

Checking this condition consists in solving $S(P)$. This can easily be achieved, thanks to the form of the equations in $S(P)$. Let us briefly describe the solving process. It consists in repeating the following treatment on $S(P)$:

> "For each variable σ_X (different from σ_T) which occurs on the right handside of an equation, say E, and on the left handside of an other one, say E'; substituting the expression on the right handside of E' for σ_X in equation E (provided σ_X does not occur in this expression)."

After a few steps, no substitution can apply. Then, one can conclude that the condition holds iff every equation rewrites as $\sigma_X = h_X \circ \sigma_T$ where h_X is a partial function (each σ_X is thus determined as a function of σ_T).
A particular case occurs when the process generates an equation with an irreductible cycle, that is of the form $\sigma_X = h \circ \sigma_X$ (where h is different from the identity on I): in that case, one can conclude that the statement is not well-formed, because either $S(P)$ has no solution, or $S(P)$ has an infinity of solutions, for a given σ_T.

This solving process can be automated and viewed as a kind of type inference where the inferred type is the canonical shape of variables. This notion of type is comparable to those of a language like C*: but on the contrary to C*, the shape of a parallel variable is not declared, but inferred. The shape of a variable in our theory only depends on the operations which are applied to the variable.

Given a well-formed statement, say P, the previous process determines partial function h_X, for every variable X (different from T) of P .

Example 10. Recall the statement in example 7. The statement matches the required form and induces the following equations on canonical shapes:

$$\begin{cases} \sigma_A = h \circ \sigma_C \\ \sigma_B = h' \circ \sigma_C \end{cases}$$

This set of equations has one and only one solution provided σ_C is known. Refering to the solving process, we have $h_A = h$ and $h_B = h'$. We conclude that the statement (with C as template) is well-formed.

$\diamond$

6.2. Shaped data fields as arrays

As previously said, each variable of a well-formed statement can be viewed as an array. In this section, we properly define this array.

The array represented by X, is named X and its indices are naturally defined from the indices of the canonical shape σ_X, i.e., $\mathrm{img}(\sigma_X)$, as they address some locations. In order to complete this definition of arrays, we first determine every canonical shape: let us recall that each σ_X depends on σ_T, according to the equation $\sigma_X = h_X \circ \sigma_T$. Therefore, we only need to determine σ_T.

From the form of the previous equation, σ_T defines the ultimate connection to locations: in other terms, the indices of shape σ_T could be viewed as locations for any other shape σ_X. Therefore, σ_T is defined as the identity on its domain of definition and this domain contains the locations of all the other shapes. According to this reasoning, the canonical shapes are defined as follows:

$$\begin{cases} \sigma_T \text{ is the identity restricted to the union of all the } \mathrm{def}(h_X) \text{ for all variables } X \text{ (except } T\text{)}, \\ \sigma_X = h_X, \text{ for every variable } X \text{ different from } T. \end{cases}$$

At this point it is necessary to outline some limitations of arrays. According to the usual sense, array indices form a (rectangular) non-empty finite subset of $\mathbb{Z}^n$ where n is the number of dimensions of the array. This allows the location of each array elements to be easily computed from its index. But, according to our notion of shape, the index set of a shape, $img(\sigma_X)$ for example, is possibly not a subset of $\mathbb{Z}^n$ (it may contain indices of several dimensions): in our theory, a shape is enough to retrieve locations from indices.

Therefore, we consider a subset of well-formed statements for which $\mathrm{img}(\sigma_X)$ defines a usual array. These statements are said to be *homogeneous*:

Definition 5. *A statement, say P, is* homogeneous *iff*

1. *P is well-formed (with T as template),*
2. *for every variable X of P (including T), domain $\mathrm{img}(\sigma_X)$ is a finite non-empty subset of $\mathbb{Z}^n$,*
3. *for every equation of P of the form described in definition 4, we have: $\mathrm{def}(h_i \circ g_i) \subseteq \mathrm{img}(\sigma_{X_0}), \forall i \in 1..n$.*

Given an homogeneous statement, for every variable X of P (including T), domain $\mathrm{img}(\sigma_X)$ may define a usual array, and the index of every value of X well belongs to this domain (by condition 3). Therefore, the set of indices of array X, denoted by $\mathcal{I}(\mathtt{X})$, is defined as the rectangular envelope of $\mathrm{img}(\sigma_X)$.

Example 11. Let us consider again the statement in example 7 with C as template. By definition:

$$\begin{cases} \sigma_A = h = g^{-1} = \lambda(i).(i{-}1) \setminus \{\, (i) \mid 1{\leq}i{\leq}8 \,\} \\ \sigma_B = h' = g'^{-1} = \lambda(i).(i/2) \setminus \{\, (i) \mid 0{\leq}i{\leq}14 \wedge i \bmod 2{=}0 \,\} \\ \sigma_C = id \setminus (\mathrm{def}(h) \cup \mathrm{def}(h')) = id \setminus \{\, (i) \mid 0{\leq}i{\leq}14 \wedge (i \bmod 2{=}0 \vee i{\leq}7) \,\} \end{cases}$$

and then: $\mathrm{img}(\sigma_A) = \mathrm{img}(\sigma_B) = \{\, (i) \mid 0{\leq}i{\leq}7 \,\}$, and $\mathrm{img}(\sigma_C) = \{\, (i) \mid 0{\leq}i{\leq}14 \wedge (i \bmod 2 = 0 \vee i{\leq}7) \,\}$. We verify that all these domains are subsets of a geometrical space ($\mathbb{Z}^2$). Moreover, the condition 3 is satisfied: $\mathrm{def}(h \circ g) = \mathrm{def}(h' \circ g') = \{\, (i) \mid 0{\leq}i{\leq}7 \,\} \subseteq \mathrm{img}(\sigma_C)$. We conclude that the statement is homogeneous and thus each variable defines a usual array. For example, A defines the array A whose set of indices is the rectangular envelope of $\mathrm{img}(\sigma_A)$. By definition, the arrays indices thus are: $\mathcal{I}(\mathtt{A}) = \mathcal{I}(\mathtt{B}) = [0..7]$ and $\mathcal{I}(\mathtt{C}) = [0..14]$.

As a comparison, see the declaration of arrays in the HPF program at the bottom of page 4. The declaration for C is different from the declaration which has been inferred: the HPF program is perfectly legal however, as directive ALIGN is only a comment for the compiler. This reveals an extra difficulty for the HPF compiler for dealing with such alignments. Now, compare with the corresponding program in C*, the size of this array has to be made explicit. ⋄

Now, considering an homogeneous statement, say P, we can state the relation between array elements, which is expressed by statement P. This relation is defined as a conjunction of equations. Each of them is induced by an equation of P as follows: every equation of P (of the form described in definition 4) induces the following equations:

$$\mathtt{X}_0[f_0(z)] = \phi(\mathtt{X}_1[f_1(z)], \mathtt{X}_2[f_2(z)], \ldots, \mathtt{X}_n[f_n(z)]),$$

$$\text{for all } z \in \bigcap_{i \in 0..n} \mathrm{def}(f_i), \text{ such that } \forall i \in 0..n \,.\, f_i(z) \in \mathcal{I}(\mathtt{X}_i)$$

$$\text{and where } f_i = h_i \circ g_i,\ i \in 0..n.$$

Example 12. Interpreting shaped data fields as arrays, the statement in example 7 expresses the following relation between arrays A, B and C: $\mathtt{C}[i] = \mathtt{A}[i] + \mathtt{B}[i], i \in [0..7]$ (this is true because $h \circ g = h' \circ g' = id \setminus \{\, i \mid 0 \leq i \leq 7 \,\}$). $\diamond$

Example 13. Let us consider the following new statement with X as template:

$$X = A.Id + B.Id' \tag{2}$$

$$C.Id'' = X \tag{3}$$

where Id and Id' are the pairs of functions in example 7, and $Id'' = (h'', id)$ with $h'' = id \setminus \{\, (i) \mid 0 \leq i \leq 7 \,\}$. This statement is homogeneous [3] and expresses the same relation (between the elements of A, B and C) as the statement in example 7. The arrays in this new statement are defined by $\mathtt{A}[0..7]$, $\mathtt{B}[0..7]$, $\mathtt{C}[0..7]$ and $\mathtt{X}[0..14]$. In particular, C is the same array as in the HPF program p.4. $\diamond$

6.3. Our proof system

Proof of a statement then sumarizes in the three following steps:

1. Check whether the statement is well-formed and compute every canonical shape σ_X.
2. Check whether the statement is homogeneous and compute the array indices $\mathcal{I}(\mathtt{X})$.
3. Last, state the relation between the elements of array X.

Example 14. The statements in the previous examples all can be proved by our proof system, except the first one in example 8 which is well-formed, but not homogeneous. This statement refers to a canonical shape (namely σ_P) whose indices belong to $\mathbb{Z}^2 \cup \mathbb{Z}^3$. Then, P cannot define a usual array.

In the following, we use our proof system for validating some statement transformations. We consider a simple example. The problem is adding a replicated n-vector V to a $n \times n$ matrix A. In our theory, this problem can be expressed as follows:

$$X = A.Id + V.Spread$$

[3] In particular, equation 3 is of the form $E_0 = \phi(E_1)$ with X for E_0 and $C.Id''$ for E_1.

where $Id = (h_1, g_1)$ and $Spread = (h_2, g_2)$, with:

$h_1 = id \setminus \{ (i,j) \mid 1 \leq i,j \leq n \}, \;\; g_1 = id,$

$h_2 = \lambda(i,j).(j) \setminus \{ (i,j) \mid i{=}1 \wedge 1 \leq j \leq n \}, \;\; g_2 = \lambda(i,j).(1,j) \setminus \{ (i,j) \mid 1 \leq i,j \leq n \}.$

The proof system demonstrates that this statement (with X as template) expresses relation $\mathtt{X}[i,j] = \mathtt{A}[i,j] + \mathtt{V}[j]$, $1 \leq i,j \leq n$ between the (so declared) arrays $\mathtt{X}[1..n, 1..n]$, $\mathtt{A}[1..n, 1..n]$ and $\mathtt{V}[1..n]$. This statement could be written as follows in HPF:

```
      REAL X(1:N,1:N),A(1:N,1:N),V(1:N)

!HPF$ ALIGN A(I,J) WITH X(I,J)
!HPF$ ALIGN V(J)   WITH X(1,J)
      ...
      FORALL ( I=1:N, J=1:N )
        X(I,J) = A(I,J) + V(J)
      END FORALL
      ...
```

Here is another statement for the same problem:

$$X = A.Id + V.Spread$$

where $Id = (h_1, g_1)$ and $Spread = (h_2, g_2)$, with:

$h_1 = id \setminus \{ (i,j) \mid 1 \leq i,j \leq n \}, \;\; g_1 = id,$

$h_2 = \lambda(i,j).(j) \setminus \{ (i,j) \mid 1 \leq i,j \leq n \}, \;\; g_2 = id.$

This statement expresses the same relation between arrays, but it defines another placement of values. Here is the HPF code for the new statement:

```
      REAL X(1:N,1:N),A(1:N,1:N),V(1:N)

!HPF$ ALIGN A(I,J) WITH X(I,J)
!HPF$ ALIGN V(J)   WITH X(*,J)
      ...
      FORALL ( I=1:N, J=1:N )
        X(I,J) = A(I,J) + V(J)
      END FORALL
      ...
```

Note that the previous version cannot easily be expressed in C* since the relation between indices and locations is not one-to-one.

Last, we can imagine the following rather simple version where locality of values cannot be easily expressed or exploited in HPF due to the compiler difficulties that are induced when the shape is declared apart and does not depend on the operations applied to variables. We can easily prove the correctness of this last version using the proof system previously presented:

$$X = A.Id + V.Spread$$

where $Id = (h_1, g_1)$ and $Spread = (h_2, g_2)$, with:

$h_1 = id \setminus \{ (i,j) \mid 1 \leq i, j \leq n \}, \quad g_1 = id,$

$h_2 = \lambda(i,j).(j) \setminus \{ (i,j) \mid 1 \leq i \leq j \leq n \}, \quad g_2 = \lambda(i,j).(min(i,j), j) \setminus \{ (i,j) \mid 1 \leq i, j \leq n \}.$

This statement could be associated with the following operational meaning (illustrated on Fig.10 for n=5): the components of the vector are assumed to be replicated above the diagonal of the matrix (change of basis h_2): the values on the diagonal are then broadcast below the diagonal (geometrical operation g_2) in order to put the components of the vector in the whole matrix before the addition is performed (by using a global operation).

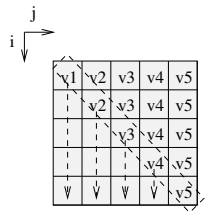


Fig. 10. Placement and broadcast of the vector components

7. Conclusion

In our opinion, in reference to automatic parallelization, the programmer must provide some additional information to the compiler in order that the compiler can reach more efficiency. In particular, data locality expression is of great importance in data-parallelism. Therefore, we proposed a theoretical framework which unify the two main concepts for expressing data locality: *alignment* of arrays in HPF and *shape* declaration in C*.

This framework allows the programmer and the compiler to share a minimum knowledge to reach efficiency. We think that it could help both the programmer and the compiler to transform the program in order to reach the best implementation.

References

1. L. Bougé. On the semantics of languages for massively parallel SIMD architectures. *PARLE'91, LNCS*, 506:166–183, 1991.
2. L. Bougé, D. Cachera, Y. Guyadec, G. Utard, and B. Virot. The data parallel programming model: A semantic perspective. In *LNCS Tutorial: The Data Parallel Programming Model*, volume 1132, pages 4–26. Springer-Verlag, 1996.
3. L. Bougé, D. Cachera, Y. Guyadec, G. Utard, and B. Virot. Formal validation of data parallel programs: Introducing the assertional approach. In *LNCS Tutorial: The Data Parallel Programming Model*, volume 1132, pages 197–219. Springer-Verlag, 1996.
4. Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *FGCS*, 8:363–378, 1992.
5. M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers*. Frontier Series. ACM Press, 1991. Chapter 7.
6. P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. *FPCA93, ACM Press*, pages 210–222, 1993.
7. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, January 1993.
8. C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
9. C. Lengauer. Loop parallelization in the polytope model. *Parallel Processing Letters*, 4(3), 1994.
10. B. Lisper. *Data Parallelism and Functional Programming*. LNCS 1132-Tutorial Series, 1996.
11. C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, U. Rennes, 1989.
12. R.D. Tennent. *Semantics of Programming Languages*. C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1991.
13. Thinking Machines Corp. *C* Programming Guide*, November 1990.
14. Eric Violard, Stéphane Genaud, and Guy-René Perrin. Refinement of data parallel programs in PEI. *IFIP TC2 Workshop on Algorithmic Languages and Calculi, Chapman & Hall*, February 1997.