A Theoretical Framework of Data Parallelism and its Operational Semantics

Ph. Gerner and E. Violard

LSIIT-ICPS, Université Louis Pasteur, Strasbourg
{gerner,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*. In this article, we aim at exploring the impact of program transformations on efficiency. For this, we define a formal operational semantics associated with the aforementioned theory.

1 Introduction

The interest of the concepts that lie behind 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 consist then in an abstract knowledge that the programmer and the compiler share and can use to interact and eventually perform the best implementation.

This is even more true when the architecture is a parallel one, when both the programming and compiling tasks are more complex. The programmer and compiler should influence each other in order to paliate these difficulties. 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 can handle.

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 focusing 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 data parallel programming.

As examples, let us consider standard data parallel programming languages like HPF [1] and C* [6]: they both provide their own way of expressing data locality: *alignment* of arrays for HPF, and *shape* of parallel variables for C*.

In a previous article [7], we proposed a theory which offers a notion of data locality in which those of HPF and C* could join up. The purpose of the article was, to provide a model of this theory, to serve as a formal basis for proving the correctness of data parallel programs.

In the present article, we focus on the execution of data parallel programs, with the objective of comparing the efficiency of the different versions of a program. Estimating the efficiency of a program needs to define its operational semantics.

2 Our Theory

2.1 Objects

Our objects are called *shaped data fields*. A shaped data field is mainly a container of values. Containers without values are sometimes referred to as shapes–in [2], 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.

A *shape* is a relation σ between a set of *indices* and a set of *locations*. Both sets are subsets of $\bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$. The indices of a shape serve to access values, and if a value v is accessed through index i, then value v is said to be *located* at all the locations which are related to i by σ. This relation is such that a location is related to at most one index. In Sect. 4, this relation will serve to describe a placement of the values onto a set of virtual processors.

A shaped data field results from the association of a shape with a *data field*. In the literature, indexed collections of values, without locations, are often referred to as data fields (see Alpha [5] and Lisper's formalism [4], as examples). The values of our data fields are indexed by integer tuples.

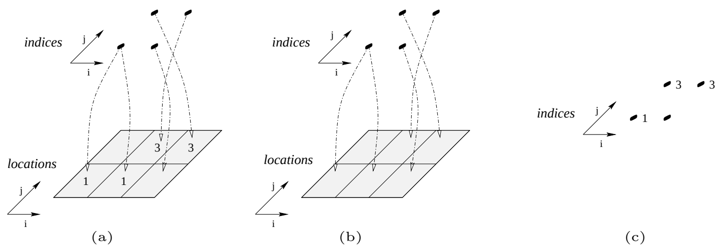


Fig. 1. (a) A shaped data field, (b) its shape and (c) its data field

2.2 Operations

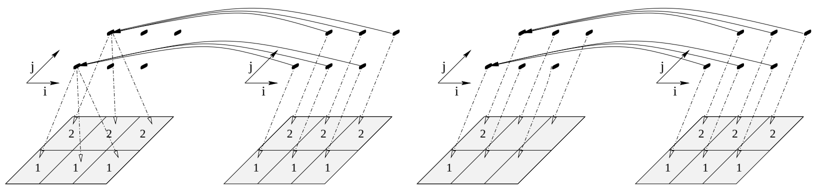
The theory defines three kinds of operations on shaped data fields:

– a *re-indexing* 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 *re-locating* does not affect the shape; it changes the locations of some of the values of the data field (with possible duplication or deletion). This change of locations is expressed through a change of indices.

  Any re-indexing or re-locating is determined by a (partial) function from the indices of the new shaped data field to the indices of the old one. For example, Fig. 2(b) shows two shaped data fields: the shaped data field on the right results from a re-locating applied to the shaped data field on the left. This re-locating is determined by the function which associates point of coordinates $(1, j)$ with any point (i, j) in the rectangle $[1..3] \times [1..2]$ of $\mathbb{Z}^2$. Fig. 2(a) illustrates the re-indexing determined by the same function.

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



(a) (b)

Fig. 2. (a) A re-indexing and (b) a re-locating

2.3 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 computations (e.g., $A + B$). We use the following notation for expressing re-indexing and re-locating operations, and also more general cases of dependences: $X.F$, where F is a pair (h, g) of partial functions, is the result of applying first the re-indexing determined by h, and then the re-locating determined by g, to X. When g is the identity, $X.F$ is a re-indexing, and when h is the identity, $X.F$ is a re-locating. 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.

For example, assuming A and B are the shaped data fields on Fig. 2(b), then they satisfy the equation $B = A.Spread$, 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 $D = \{\,(i,j) | 1{\leq}i{\leq}3 \wedge 1{\leq}j{\leq}2\,\}$ (since h is the identity, *Spread* defines a re-locating).

3 Theory Adequacy

This section is intended to link our theory with data locality expression in HPF and in C* and also to show that our theory makes a better compromise than these languages between compiler and programmer.

3.1 Example 1

Let us consider the two following programs in HPF and C*:

```
      REAL A(0:3), B(1:4), C(1:4)

!HPF$ TEMPLATE X(1:8)
!HPF$ ALIGN A(I) WITH X(I+1)
!HPF$ ALIGN B(I) WITH X(2*I)

      ...
      FORALL ( I=1:4 )
        C(I) = A(I-1) + B(I)
      END FORALL
      ...
```

```
shape [9]vector';
real: vector' A',B',C';
...
C' = A' + [.*2]B';
...
```

In the HPF program, template `X` is necessary for aligning arrays `A` and `B` the way we want. The C* program is equivalent to the HPF program to the extent that it expresses a correspondance between array elements and virtual processor that respects the alignment constraints of the HPF program. In the C* program, arrays `A'`, `B'` and `C'` have the same shape, so that for every `i`, value `A'[i]` is on the same virtual processor as values `B'[i]` and `C'[i]`. `A'`, `B'` and `C'` are the result of having the values of `A`, `B`, `C`, placed in such a way that the alignment constraints are respected: for example, `A'[4]` (resp. `B'[4]`) contains the value of element `A[3]` (resp. `B[2]`), so that `A[3]` and `B[2]` are on the same virtual processor.

In the HPF program, the array `C` is not aligned: the placement of values of `C` is left in charge of the compiler. The C* program is more prescriptive as it explicits a particular placement of values and the required communications. In some sense, translating the HPF program to C* reveals the difficulties of the HPF compiler in handling the alignment directives as much as those of the programmer in writing its program in C*.

In our theory, these programs (and their operational meaning) can be expressed by the following statement:

$$C = A.\mathit{Shift} + B.\mathit{Id}$$

where A, B and C are shaped data fields and represent arrays `A`, `B` and `C`, and *Shift*, *Id* are pairs (h, g), (h', g') of partial functions such that $h \circ g$ equals $\lambda(i).(i{-}1) \setminus [1..4]$ and $h' \circ g'$ equals the identity on $[1..4]$. This is enough to express the dependences between the values of arrays `A`, `B` and `C`: the functions $h \circ g$ and $h' \circ g'$ express that, for any $i \in [1..4]$, `C`$[i]$ depends on `A`$[i{-}1]$ and `B`$[i]$. In the HPF program, these dependences are expressed by the `FORALL` block.

Now we precise the functions h, g, h', and g', for expressing the placement used in C* program (which respects the alignment constraints of the HPF program), which is depicted on Fig. 3:

$$\begin{array}{ll} h = \lambda(i).(i{-}1) & h' = {g'}^{-1} \\ g = id \setminus [1..4] & g' = \lambda(i).(2{\times}i) \setminus [1..4] \end{array}$$

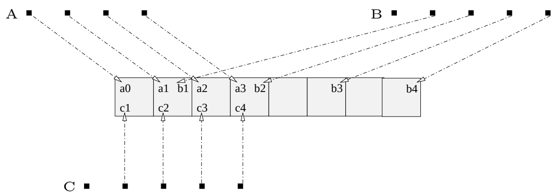


Fig. 3. The placement of values

In comparison with HPF, in our theory a shape is associated with each variable, so that each value of any variable is placed on a set of locations, including the values of variable C in the above statement. And in comparison with C*, the shapes are inferred from the statement, rather than being found by the programmer, as it was necessary for writing the C* program; and also, contrarily to C*, the indices are not names for locations.

3.2 Example 2

Here we discuss the Cooley-Tukey algorithm for the one-dimensional, unordered, radix-2 FFT. Our starting point is the iterative formulation of FFT in [3], p.380. The algorithm is defined by the following set of recurrent equations, where vector $Y[0..n{-}1]$ is the Fourier transform of vector $X[0..n{-}1]$, and $n = 2^r$:

$$\begin{array}{lll} R[i,m] = X[i], & \text{for all } (i,m) \in D_0, & (1) \\ R[i,m] = R[i{\downarrow}m, m{-}1] + (R[i{\uparrow}m, m{-}1] \times W[i{\bowtie}m]), & \text{for all } (i,m) \in D_{1..r}, & (2) \\ Y[i] = R[i,m], & \text{for all } (i,m) \in D_r, & (3) \end{array}$$

where R is a temporary $n{\times}r$-matrix, W is a given n-vector that contains the powers of the primitive root of unity in the complex plane (also known as *twiddle* *factors*), and the following subdomains are considered:

$$D_0 = \{\, (i,m) \mid 0{\leq}i{<}n \wedge m{=}0 \,\},$$
$$D_{1..r} = \{\, (i,m) \mid 0{\leq}i{<}n \wedge 1{\leq}m{\leq}r \,\},$$
$$D_r = \{\, (i,m) \mid 0{\leq}i{<}n \wedge m{=}r \,\},$$

where "$\downarrow$", "$\uparrow$" and "$\bowtie$" are infixed bitwise operations on the binary representation of index i. For example, $i{\uparrow}m$ is obtained from i by forcing the m^{th} most significant bit to 1. These operations are defined as follows:

$$(b_0 b_1 ... b_{r-1}){\downarrow}m = (b_0 b_1 ... b_{m-1} 0 b_{m+1} ... b_{r-1})$$
$$(b_0 b_1 ... b_{r-1}){\uparrow}m = (b_0 b_1 ... b_{m-1} 1 b_{m+1} ... b_{r-1})$$
$$(b_0 b_1 ... b_{r-1}){\bowtie}m = (b_m b_{m-1} ... b_0 0 ... 0)$$

In order to build a first statement from the above equation set, each array is represented by a shaped data field and each dependence is modeled by a partial function. For example, X and R in equation (1) are considered as shaped data fields, and R depends on X according to the function which maps every (i,m) in D_0 to i, i.e., the partial function $\lambda(i,m).(i) \setminus D_0$. Then, each dependence, say d, is split into two partial functions, say (h,g), such that $d = h \circ g$ in order to define a particular placement of values. This yields the following statement:

$$R.\mathit{Id}_0 = X.\mathit{Vector}_0 \tag{4}$$
$$R.\mathit{Id}_{1..r} = R.\mathit{Even} + (R.\mathit{Odd} \times W.\mathit{Spread}) \tag{5}$$
$$Y.\mathit{Vector}_r = R.\mathit{Id}_r \tag{6}$$

where the definitions of all pairs of functions and dependence functions are reported in Table 1. $R.\mathit{Id}_0$, $R.\mathit{Id}_{1..r}$ and $R.\mathit{Id}_r$ each identifies a piece of R on subdomains D_0, $D_{1..r}$ and D_r, respectively.

Table 1. The definition of a particular placement.

	(h,g)	$(h \circ g)$
Id_0	$= (\mathit{id},\ \mathit{id} \setminus D_0)$	$\mathit{id} \setminus D_0$
$\mathit{Id}_{1..r}$	$= (\mathit{id},\ \mathit{id} \setminus D_{1..r})$	$\mathit{id} \setminus D_{1..r}$
Id_r	$= (\mathit{id},\ \mathit{id} \setminus D_r)$	$\mathit{id} \setminus D_r$
Vector_0	$= (\lambda(i,m).(i) \setminus D_0,\ \mathit{id})$	$\lambda(i,m).(i) \setminus D_0$
Even	$= (\mathit{id},\ \lambda(i,m).(i{\downarrow}m, m{-}1) \setminus D_{1..r})$	$\lambda(i,m).(i{\downarrow}m, m{-}1) \setminus D_{1..r}$
Odd	$= (\mathit{id},\ \lambda(i,m).(i{\uparrow}m, m{-}1) \setminus D_{1..r})$	$\lambda(i,m).(i{\uparrow}m, m{-}1) \setminus D_{1..r}$
Spread	$= (\lambda(i,m).(i{\bowtie}m) \setminus D_{1..r},\ \mathit{id})$	$\lambda(i,m).(i{\bowtie}m) \setminus D_{1..r}$
Vector_r	$= (\lambda(i,m).(i) \setminus D_r,\ \mathit{id})$	$\lambda(i,m).(i) \setminus D_r$

By this statement, vector X (resp. Y) is aligned with the first (resp. last) column of matrix R. Moreover, the twiddle factors have been replicated onto the matrix so that the dependence defined by *Spread* does not involve any communication. This statement describes the *binary-exchange algorithm* if the indices of variable R are identified with locations. In the following, R will be referred to as the *template* of this statement.

4 Operational Semantics

All the statements we have seen in the previous section are computable. They are examples of what we call *well-formed statements*. This class of statements is associated with a proof system [7] and an operational semantics.

4.1 Well-Formed Statements

Informally, any well-formed statement is such that the shape of each variable can be inferred uniquely from the shape of one of them. Moreover, we restrict ourselves to statements in single-assignment form.

Definition 1. *A* well-formed statement*, is a statement, say P:*

- *with some variables declared as* input variables*, other ones declared as* output variables*, and with a single one declared as the* template *of the statement (and referred to as T);*
- *whose every equation has the form*

$$X_0.(h_0, g_0){=}\phi(X_1.(h_1, g_1), \ldots, X_n.(h_n, g_n)) \ ,$$

  where ϕ is a global computation, and for all $k \in [0..n]$, X_k is a variable of P, but X_0 is not an input variable, h_k and g_k are partial functions from I to I, where $I = \bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$, $g_0 = id|_D$, where id is the identity on I, and either $h_0 = id$ (case 1), or $h_0 \neq id$, and $\forall k \in [1..n], h_k = id$ (case 2);
- *such that, the following set of equations, denoted $S(P)$, has a unique solution for any σ_T. Its variables are partial functions from I to I, and we have one variable σ_X for each variable X of P. $S(P)$ is the set of all the equations $\sigma_{X_k} = h_k \circ \sigma_{X_0}$, $k \in 1..n$, induced from every equation of P of case 1, and equations $\sigma_{X_0} = h_0 \circ \sigma_{X_k}$ induced from equations of case 2;*
- *and such that, if a variable X appears in more that one left-hand side, in terms $X.(h_1, id|_{D_1}), \ldots, X.(h_q, id|_{D_q})$, then $\forall k$, $h_k = id$, and $\forall k, k' \in 1..q$, $k{\neq}k' \Rightarrow D_k \cap D_{k'} = \emptyset$.*

Relative Placement of Variables. The system $S(P)$ associated with a well-formed statement P defines the placement of the values of each shaped data field on a set of *locations*. For each variable X, this placement is defined by σ_X the following way: the values of X accessed by index i are placed at locations ${\sigma_X}^{-1}(i)$ (in the sequel, this set is denoted $\mathrm{loc}(X, i)$).

$S(P)$ defines the functions σ_X as follows. The set of locations is I, and the values of template variable T are considered placed on I with $\sigma_T = id$, so that at most one value is accessed through index i, and this value is then located at location $l = i$. The placement of all the other variables are uniquely inferred from this one.

4.2 States and Transitions

A *state* is a set of *stores*. We have one store for each location. The store associated with

location $l \in I$ is a partial function $\rho_l : \mathit{Var} \times I \rightharpoonup V$, where Var is the set of variables of statement P, and V is a set of values. Intuitively, $\rho_l(X,i)$ is the current value of X, indexed (in the index set of X) by i, and placed at location l. A *transition* describes a change in the store of a location. So, a transition induces a change from one state to another. The transitions can occur concurrently.

The execution begins with an *initial state*. In the initial state, if X is an input variable, and $\rho_l(X,i)$ is defined, then $l \in \mathrm{loc}(X,i)$, and $\forall l' \in \mathrm{loc}(X,i), \rho_{l'}(X,i) = \rho_l(X,i)$; and if X is not an input variable, then $\forall l \in I, \forall i \in I,\ \rho_l(X,i)$ is undefined. From the initial state, the transitions apply, in a non-deterministic way, and concurrently, until no more transition can apply.

Transitions. For every equation $X_0.(h_0,g_0) = \phi(X_1.(h_1,g_1),\ldots,X_n.(h_n,g_n))$, of P, we have the following transition rule, where $i \in I$ is any index, $l_0, l_1, \ldots, l_n \in I$ are any locations, and d_k is for $h_k \circ g_k$, $k \in 0..n$:

$$\frac{\begin{array}{c}\mathrm{UNDEF}_0 \wedge p \in \mathrm{loc}(()X, g_1(i)), p' \in \mathrm{loc}(()X, g_1(i)), p'' \in \mathrm{loc}(()Y, g_2(i))\\ \forall k \in 1..n, \mathrm{DEF}_k \wedge (l_0 = l_k \vee l_0 \notin \mathrm{loc}(X_k, d_k(i)))\end{array}}{\rho_{l_0} \longrightarrow \rho_{l_0}[(X_0, d_0(i)) \leftarrow \phi(\rho_{l_1}(X_1, d_1(i)), \ldots, \rho_{l_n}(X_n, d_n(i)))]} \tag{7}$$

where $\rho[(X,i) \leftarrow v]$ is the store ρ except that it maps (X,i) to value v, UNDEF_0 means that $d_0(i)$ is defined and $\rho_{l_0}(X_0, d_0(i))$ is undefined, and DEF_k ($k \in 1..n$) means that $d_k(i)$ and $\rho_{l_k}(X_k, d_k(i))$ are defined.

Computations: Assuming the Owner-Computes Rule, it is to be understood that, in the application of a transition, the computation $\phi(\ldots)$ is made at location l_0. Note that, when $\mathrm{loc}(X_0, d_0(i))$ contains more than one location, the calculus will be made several times. Such a case is useful when duplicating the calculus avoids some communications.

Communications: When $l_0 \neq l_k$, there is a *communication* for sending the value from location l_k, to location l_0, because the computation on l_0 needs this value. The transition does not apply if $l_0 \neq l_k \wedge l_0 \in \mathrm{loc}(X_k, d_k(i))$, because in this case the value at location l_k will be available locally, that is, at location l_0.

5 Example

Let us consider again the FFT example in Sect. 3.2, with $n = 2^4$. Suppose we are given 4 physical processors. Equation (5) is the core of the algorithm. R is the template of the statement, so $\sigma_R = \mathit{id}$, which means that if R contains a value at index (i,m), then this value lies on location (i,m). So, the transition rule associated to this equation breaks down to the following rule:

$$\frac{\begin{array}{c}(i,m)\in D_{1..r}\ \wedge\ \rho_{i,m}(R,(i,m))\text{ is undefined}\\ \wedge\ \rho_{i\downarrow m,m-1}(R,(i\downarrow m,m-1))\text{ is defined}\\ \wedge\ \rho_{i\uparrow m,m-1}(R,(i\uparrow m,m-1))\text{ is defined}\end{array}}{\begin{array}{l}\rho_{i,m}\longrightarrow\rho_{i,m}[(R,(i,m))\leftarrow\rho_{i\downarrow m,m-1}(R,(i\downarrow m,m-1))\\ \qquad\qquad\qquad\qquad +(\rho_{i\uparrow m,m-1}(R,(i\uparrow m,m-1))\times\rho_{i,m}(W,i\bowtie m))]\end{array}} \tag{8}$$

This rule is the only rule, among those induced by the statement, that generates communications. The communications come from the fact that indices (i,m), $(i\downarrow m,m-1)$ and $(i\uparrow m,m-1)$ represent different locations (because R is the template), so that for doing the computation at (i,m), 2 communications are needed for getting the required values which sit at locations $(i\downarrow m,m-1)$ and $(i\uparrow m,m-1)$. We get a total of $2\times\mathrm{card}(D_{1..r})=2\times n\times r=2\times16\times4=128$ communications.

The rule also shows that the axis of index m follows the direction of time during execution of the program. So, an obvious projection onto physical processors is along this axis. For each (i,m), either $i\downarrow m=i$ or $i\uparrow m=i$, so that this projection reduces the number of communications by a half, to 64. But we have only 4 processors, not 16, and in this case what is classically done is, distributing 4 contiguous elements of the array (on the first dimension of R) onto one processor. The number of communications is then reduced further, from 64 to $64-(2\times16)=32$ communications (because the dependences in the last 2 steps require no communications).

A different algorithm could be described, characterized by another placement of values onto the processors: the two-dimensional transpose algorithm. This new algorithm can be expressed by the same statement as in Sect. 3.2, except that we re-index variable R, obtaining a new template variable R'. This is expressed by just adding to the statement the following equation:

$$R.(\mathit{transf},\mathit{id})=R'\ , \tag{9}$$

where $\mathit{transf}(i,j,m)=\begin{cases}(j+i\times\sqrt{n},m), \text{ if } 0\le i,j<\sqrt{n}\wedge 0\le m<r/2\\ (i+j\times\sqrt{n},m), \text{ if } 0\le i,j<\sqrt{n}\wedge r/2\le m<r\end{cases}$

This transformation is sketched on Fig. 4.

Since we have only changed the template, the number of (virtual) communications is the same for this new algorithm. The interpretation of the new template is the following one: the third axis of R' corresponds to time, and the first two axes give a square of locations of dimensions $\sqrt{n}\times\sqrt{n}$.

In this version, (virtual) communications occur only on planes i (except between time steps $m=r/2-1$ and $m=r/2$), so that it is interesting to project onto physical processors along axis i. Also, projecting along time axis m is natural, so that we end up projecting along axes i and m.

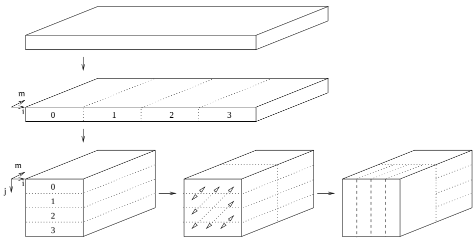


Fig. 4. A sketch of the transformation in the case n=16: the domain of R is first rearranged as a parallelepipede and then divided into two parts; last step consists in transposing the matrices forming the first of these two parts.

If one look at slices $m = r/2-1$ and $m = r/2$ as matrices, the communications beween these slices are those of a transposition of values, hence the name of the algorithm. If given 4 processors again, if we project this way our $n = 2^4$ values onto them, then the total number of communications is the number of communications required for the transposition, which is $(\sqrt{n} \times \sqrt{n}) - \sqrt{n} = 16 - 4 = 12$.

6 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] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, January 1993.
[2] C.B. Jay. A semantics for shape. *Sc. of Computer Programming*, 25:251–283, 1995.
[3] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.
[4] B. Lisper. *Data Parallelism and Functional Programming.* LNCS 1132-Tutorial Series, 1996.
[5] C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.
[6] Thinking Machines Corp. *C* Programming Guide*, November 1990.
[7] E. Violard. What really is data parallelism? Technical Report RR 00-01, ICPS, Université Louis Pasteur, January 2000.