A Theoretical Framework of Data Parallelism and its Operational Semantics

Ph. Gerner, E. Violard

ICPS, Université Louis Pasteur, Strasbourg
Boulevard S. Brant, F-67400 Illkirch
{gerner,violard}@icps.u-strasbg.fr
Tel: (+33) 03 88 65 50 47, Fax: (+33) 03 88 65 50 61

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 measuring the efficiency of programs, and the impact of program transformations on efficiency. For this, we give a formal operational semantics to the statements in 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 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.

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 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 like HPF [4] and C* [10]: they both provide their own way of expressing data locality: *alignment* of arrays for HPF, and *shapes* of parallel variables for C*.

In a previous article [11], 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, in view of optimizing it. Estimating the efficiency of a program implies that we know its operational semantics.

This is how we proceed to present the operational semantics we use. In the next section, we cite some related works. In section 3, we briefly recall our theory. Section 4 is intended to link our theory with data locality expression in HPF and in C*. Section 5 presents the operational semantics for statements of our theory. This operational semantics is used in section 6 for discussing the respective efficiency of two versions of a classical parallel program to perform the Fast Fourier Transform.

2 Related Works

In [8], Björn Lisper gives a review of a new abstract programming formalism with the aim to generalize current data parallel constructs and 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, based on objects called *data fields*, are partial functions with an *explicit restriction* operation. For example, in this formalism, expression $f \setminus \lambda i.(0 \leq i \leq n)$, where f is a partial function from integers to some set of values, denotes a data field which is garanteed to be undefined for any integer i outside the range $0..n$. 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. However, the relation between the (parallel) evaluation mechanism, and the data locality which is introduced through λ-abstraction, is not formally defined. In the present paper, a theory and its formal operational semantics are presented, which aim at expressing the relation between data placement and parallel evaluation, and which can thus shed some complementary light on formalisms like Lisper's.

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. The operational semantics of $\mathcal{L}$ is defined with predicates and weakest preconditions. 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 [12]. PEI is a theory born from a generalization of the so-called Polytope model [7]. It was introduced to address crucial questions of program design methodology and transformations, through a straightforward generalization of equational notations [9, 2]. Its central concept is a notion of objects which are pairs of a partial function from integer tuples and a bijection σ from the same integer tuples: function σ can be seen as a space-time mapping. In this paper, we refer to a looser association between these two functions.

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

3 Our Theory

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

3.1 Objects

A shaped data field is mainly a container of values. Containers without values are sometimes referred to as shapes–in [5], 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.

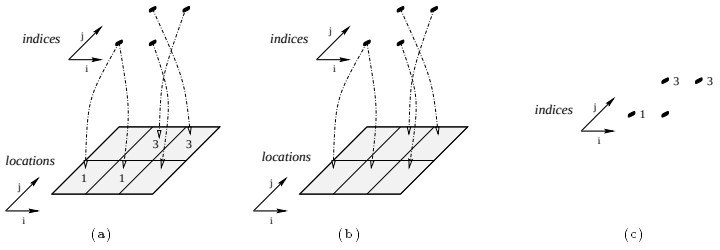

Fig. 1. (a) A shaped data field, (b) its shape and (c) its data field

Fig. 1(a) shows an example of a shaped data field. 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 by the points in $\mathbb{Z}^2$ of coordinates $(0,0)$, $(0,1)$ and $(1,1)$: index $(0,0)$ refers to two locations while the two others are connected with only one. Note that index $(1,0)$ 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, without locations, are often referred to as *data fields* (see Alpha [9] and Lisper's formalism [3], as examples).

Thus, our theory defines a shaped data field as a data field associated with a shape. For example, the shaped data field on Fig. 1(a) associates the shape on Fig. 1(b) with the data field on Fig. 1(c). This data field is composed of values 1, once, 3, twice: these values are indexed by $(0,0)$, $(0,1)$ and $(1,1)$, respectively. In other terms, the data field, in the sense of [9,3], which is embedded in the shaped data field on Fig. 1(a), is the one drawn on Fig. 1(c).

3.2 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 (partial) function from the indices of the new shaped data field to the indices of the old one.

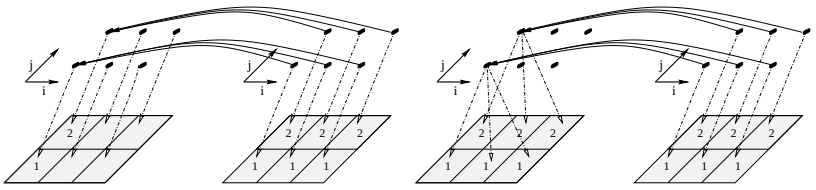

Fig. 2. (a) A geometrical operation and (b) a change of basis

For example, Fig. 2(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

$[1..3] \times [1..2]$ of $\mathbb{Z}^2$. Fig. 2(b) illustrates the change of basis determined by the same function.

- 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.

3.3 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: 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.

For example, assuming A and B are the shaped data fields on Fig. 2(a), 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 $D = \{ (i, j) \mid 1 \leq i \leq 3 \wedge 1 \leq j \leq 2 \}$ (since h is the identity, $Spread$ defines a geometrical operation).

4 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 between compiler and programmer. Some assertions are made, which are founded on an operational interpretation of statements in our theory. The validity of these assertions will be established in the next section, whose purpose is to formally define this operational meaning. We illustrate our point of view with two examples.

• 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'` are the result of having the values of `A`, `B`, `C`, placed in such a way that the alignment constraints are respected: for example, `B'[4]` contains the value of element `B[2]`.

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) could 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 *dependencies* 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 `C`$[i]$ depends on `B`$[i]$. In the HPF program, these dependencies 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:

$$h = \lambda(i).(i-1) \qquad h' = g'^{-1}$$
$$g = id \setminus [1..4] \qquad g' = \lambda(i).(2 \times i) \setminus [1..4]$$

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*, in our theory, the shapes are inferred from the statement, rather than being found

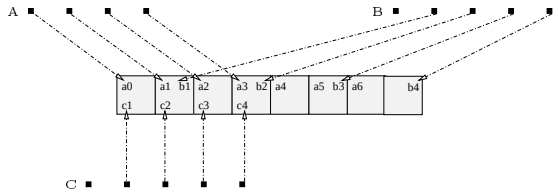

Fig. 3. The placement of values

by the programmer, as it was necessary for writing the C* program; and also, contrarily to C*, the indices are not names for locations.

Our theory then provides an intermediary level between HPF and C*, which we believe is better for the compiler because it provides it with a placement for all the variables, and better for the programmer with respect to the shape constraints of C*.

• Here we discuss an algorithm to perform FFT on a parallel computer. Our starting point is the iterative formulation of FFT p.380 in [6]: the Cooley-Tukey algorithm for one-dimensional, unordered, radix-2 FFT. 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 $n = 2^r$:

$$R[i,m] = X[i], \qquad \text{for all } (i,m) \in D_0, \quad (1)$$
$$R[i,m] = R[i{\downarrow}m, m-1] + R[i{\uparrow}m, m-1] \times W[i{\bowtie}m], \quad \text{for all } (i,m) \in D, \quad (2)$$
$$Y[i] = R[i,m], \qquad \text{for all } (i,m) \in D_r, \quad (3)$$

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:

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

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 in our theory from this 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, in our theory, 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:

$$R.Id_0 = X.Vector_0 \tag{4}$$

$$R.Id = R.Even + R.Odd \times W.Spread \tag{5}$$

$$Y.Vector_r = R.Id_r \tag{6}$$

where the definitions of all pairs of functions and dependence functions are reported in table 1, and $R.Id_0$, $R.Id$ and $R.Id_r$ each define a piece of R on subdomains D_0, D and D_r, respectively.

	(h, g)	$(h \circ g)$
Id_0	$= (id,\ id \setminus D_0)$	$id \setminus D_0$
Id	$= (id,\ id \setminus D)$	$id \setminus D$
Id_r	$= (id,\ id \setminus D_r)$	$id \setminus D_r$
$Vector_0$	$= (\lambda(i, m).(i) \setminus D_0,\ id)$	$\lambda(i, m).(i) \setminus D_0$
$Even$	$= (id,\ \lambda(i, m).(i{\downarrow}m, m{-}1) \setminus D)$	$\lambda(i, m).(i{\downarrow}m, m{-}1) \setminus D$
Odd	$= (id,\ \lambda(i, m).(i{\uparrow}m, m{-}1) \setminus D)$	$\lambda(i, m).(i{\uparrow}m, m{-}1) \setminus D$
$Spread$	$= (\lambda(i, m).(i{\triangleright\!\triangleleft}m) \setminus D,\ id)$	$\lambda(i, m).(i{\triangleright\!\triangleleft}m) \setminus D$
$Vector_r$	$= (\lambda(i, m).(i) \setminus D_r,\ id)$	$\lambda(i, m).(i) \setminus D_r$

Table 1. The definition of a particular placement.

By this statement, vector X (resp. Y) is aligned with the first (resp. last) line 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.

5 Operational Semantics

We define the class of programs, which we call *well-formed statements*, to which we shall give an operational semantics. This semantics defines the intended meaning of a program, that is, what it does and how it behaves at execution. In the next section, we shall exploit this semantics to express a program transformation that has an impact on efficiency at execution.

5.1 Well-Formed Statements

Informally, a well-formed statement is a statement in which the shape of each variable can be inferred uniquely from the shape of one of them; and that has a single-assignment form when the equations are read as assignment commands.

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

- *with some variables identified as* input variables, *and variables identified as* output variables, *and with a variable identified as the* 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_k ($k \in 0..n$) is of the form $X_k.(h_k, g_k)$, where X_k is a variable of P, and h_k and g_k are partial functions from I to I, where $I = \bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$, and $h_0 = id$, where id is the identity on I;*

- *such that, the following set of equations, denoted $S(P)$, whose variables, denoted by σ_X, for every variable X of P, has an unique solution for a given σ_T. Variables σ_X are partial functions from I to I. $S(P)$ 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;*

- *and such that, if a variable X appears in more that one E_0 term, and these terms have the forms $X.(id, g_1), \ldots, X.(id, g_n)$, then $\forall k, k' \in 1..n$, $k \neq k'$, $\mathrm{img}(g_k) \cap \mathrm{img}(g_{k'}) = \emptyset$.*

5.2 Placement

First, we need to explain the alignment definitions that a well-formed statement gives.

The system $S(P)$ associated with a well-formed statement P defines the placement of the values of each shaped data field of the statement 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 $\mathrm{loc}(X, i)$).

We know explain how $S(P)$ defines the functions σ_X. For the set of locations, we will use I. The values of the template variable T are considered placed on I with $\sigma_T = id$, so that by index i, we access at most one value, which is then located at location $l = i$. The system $S(P)$ admits a unique solution, giving the placement of all the other shaped data fields.

5.3 States and Transitions

We define the operational semantics for any well-formed statement P, with states and transitions between these states.

A *state* is a set of *stores*. We have one store for each location. The store associated with the location $l \in I$ is a function $\rho_l : Var \times I \rightharpoonup V$, where Var is the set of variables of the statement P, and V is a set of values. Intuitively, $\rho_l(X, i)$ is the current value of X, placed at location l, and indexed (in the index set of X) by i.

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 modeled by the operational semantics 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.

Transitions For every equation $X_0.(id, g_0) = \phi(X_1.(h_1, g_1), \ldots, X_n.(h_n, g_n))$, 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 $h_k \circ g_k$, $k \in 0..n$:

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

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.

Calculi: It is to be understood that, in the application of a transition, the calculus $\phi(\ldots)$ is made at location l_0. This is because we use the so-called "Owner-Computes Rule". 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 at location l_k, to location l_0, because the calculus to be done 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, and we refuse to make a communication when the value might become locally available.

The execution starts from the initial state: the transitions apply, in a non-deterministic way, and concurrently, until no more transition can apply.

It is possible, when some index sets are infinite, that the execution does not terminate. If an execution terminates, then all the other possible executions would terminate with the same state.

6 Example

Let us consider again the FFT example in section 4. A different algorithm could be described, characterized by another placement of values: the two-dimensional transpose algorithm. It is obtained by transforming the domain of R as sketched on Fig. 4.

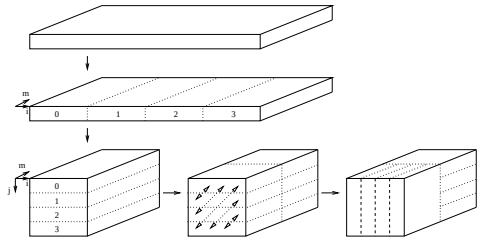

Fig. 4. A sketch of the transformation in the case n=16: the domain of R is first rearranged as a cube and then divided into two parts; last step consists in transposing the matrices forming the first of these two parts.

All these transformations reduce to a single change of basis applied to R and determined by the partial function *transf* defined by:

$$transf = \lambda(i,j,m).(if\ (m{>}\sqrt{n})\ then\ (i{+}j{\times}\sqrt{n},m)\ else\ (j{+}i{\times}\sqrt{n},m))\ \setminus\ D'$$

where $D' = \{\ (i,j,m)\ |\ 0{\leq}i,j{<}\sqrt{n} \wedge 0{\leq}m{\leq}r\ \}$.

The transpose algorithm can then be expressed by the following statement. It is the same statement as in section 4 except that we changed the template R to a new variable R', which is related to R by a new equation. Since the template has changed, the operational meaning is changed too, and it now corresponds to the transpose algorithm.

$$R.Id_0 = X.Vector_0 \tag{7}$$
$$R.Id = R.Even + R.Odd \times W.Spread \tag{8}$$
$$Y.Vector_r = R.Id_r \tag{9}$$
$$R' = R.Twist \tag{10}$$

where $Twist = (transf, id)$.

In both the binary-exchange algorithm and the transpose algorithm, we obtain the maximum parallelism by distributing the locations using a projection along the axis of the index m. Since we knew what were the communications before the projection, we know those of them which are left after the projection. This allows us to compare the efficacity of these algorithms on a given architecture. For example, on a hypercube-connected parallel computer, the binary-exchange and the transpose algorithm have a different efficiency depending on the ratio of communication bandwidth to the processing speed of the CPUs. For a detailed comparison of these two algorithms, see [6].

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. M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers*. Frontier Series. ACM Press, 1991. Chapter 7.
3. P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. *FPCA93, ACM Press*, pages 210–222, 1993.
4. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, January 1993.
5. C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
6. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.
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. Thinking Machines Corp. *C* Programming Guide*, November 1990.
11. E. Violard. What really is data parallelism? Technical Report RR 00-01, ICPS, Université Louis Pasteur, January 2000. Submitted for publication.
12. 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.