Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Parallel Computing 30 (2004) 139–161

PARALLEL COMPUTING

www.elsevier.com/locate/parco

A semantic framework to address data locality in data parallel languages

Eric Violard

LSIIT-ICPS, Université Louis Pasteur, Strasbourg, France and
Pôle API, Boulevard Sébastien Brant, F-67400 Illkirch, Strasbourg, France

Received 15 September 2001; received in revised form 19 May 2003; accepted 15 June 2003

Abstract

We developed a theory in order to address crucial questions of program design methodology. This theory deals with data locality which is a main issue in parallel programming. In this article, we regard this theory and its model as a minimum semantic domain for data parallel languages.

The introduction of a semantic domain is justified because the classical data parallel languages (HPF and C*) have different intuitive semantics: Indeed, they use different concepts in order to express data locality. These concepts are *alignment* in HPF and *shape* in C*. Consequently these two languages define their own balance between compiler and programmer investments in order to reach program efficiency. We present our theory as a foundation for defining a better balance.

Keywords: Data parallel programming; Equational languages semantics; Parallel programs design; Data locality

1. Introduction

Facilities in programming languages express virtual items (objects, functions, operations, ...) and serve as an intermediate toolbox, in one hand for the programmer to specify what he knows about the problem to be solved and, in the other hand, for the compiler to rely on the architecture on which the program will execute. These facilities then connect two domains and determine the knowledge the programmer

E-mail address: violard@icps.u-strasbg.fr (E. Violard).

0167-8191/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0167-8191(03)00089-9

and the compiler can share to interact each other in order to perform an efficient program.

Such a point of view leads to define a programming theory, supported by a relevant formalism, in which any program is a statement which can be transformed either by the programmer or by the compiler. The formalism and the features it expresses have to establish a fair *equilibrium point* between the programmer and the compiler investments in order to facilitate the collaboration between them.

Such a theory is even more useful if the architecture is a parallel one since programming paradigms, algorithm definitions and the share of architecture resources are more complex than those in sequential programming.

Any parallel programming paradigm expresses a specific balance between the programmer workload and the compiler one in order to reach program efficiency: On one side of the spectrum is the message passing model, as in PVM or MPI, which leaves the programmer in charge of the whole workload. Whereas, on the other side, the compiler has the whole charge in automatic parallelization. The programming paradigm associated with the emerging OpenMP standard [13] and data parallelism are located somewhere between these extrema.

Although OpenMP is more recent than standard data-parallel languages C* or HPF, in our opinion, it defines a collaboration degree between the compiler and the programmer which is not maximum, mainly because it has no features to express data locality. This can critically affect performance on machines which exhibit non-uniform memory access times. The research works which aim at extending OpenMP with some ideas inherited from the data parallelism [5,10] underline the importance of the expressions of data locality.

OpenMP lies on an execution model which is somewhat more machine dependent. Therefore the OpenMP user has much effort to reach efficiency while the OpenMP compiler has much less workload. This appears to be confirmed by recent works which aim to generate OpenMP programs from HPF programs [3] or to target the compilation of HPF programs to multithreaded runtime environment [2]. In some sense, these works show that data-parallelism may be a better medium.

Data parallelism is indeed a candidate to express an intermediate level of abstraction to be shared by the programmer and the compiler. A major interest of the data parallelism paradigm is that it enables the programmer to describe a parallel algorithm by choosing a well-known sequential one and then focusing on the expression of ''parallel variables'' and ''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 comparison with sequential programming.

This paper deals with these questions and aims to introduce a new theory, in particular because there exists different intuitive semantics for data parallelism and data locality. Hence let us consider standard data parallel programming languages such as HPF [9] or C* [15]. Both allow data locality to be expressed as a relationship between indices of arrays and indices of so-called virtual processors, but these languages differ for expressing this relationship:

- In HPF, *alignment* between two arrays, by using the `ALIGN` directive, provides a way to inform the compiler that some data elements should reside in the same virtual processor and then in the same physical one. By default an array is not aligned with any other one, so that the compiler must guess alone a best distribution onto physical processors.
- In C⋆, the *shape* concept is a way of configuring data: It is a kind of abstract data type which is associated with every parallel variable by using a `shape` declaration. Typically the corresponding elements of two arrays declared of the same shape reside in the same virtual processor. This concept of data type entails some restrictions on authorized expressions. For example, the assignment of two variables assumes they have the same shape.

Thus features for data locality have different meanings in HPF or in C⋆. Moreover it is interesting to note that these concepts yield an unbalanced workload for the programmer and for the compiler:

- In order to express a parallel program in C⋆ from a sequential one, the programmer has to provide some real effort. Then the compiler workload is weaker.
- Whereas in HPF the programmer may only consider the sequential code and the main workload is left to the compiler which has in charge to find a best data distribution from some alignment directives.

The question is then to define a data parallelism theory which introduces features such that the programmer and the compiler can fairly collaborate. In this article, we propose such a theory which could bridge the gap between *alignment* and *shape*.

In order to carefully introduce the 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 introduces the theory with an informal meaning. Section 4 presents a sound model of our theory which provides a formal framework for demonstrating properties of data parallel programs. Section 5 is devoted to a case study just before conclusion.

2. Some examples

Here are some very simple examples of programs in C⋆ or HPF.

2.1. Pascal summation on aligned arrays

```
      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-1,J) + A(I,J-1)
      END FORALL
      ...
```

This program in HPF obviously aligns matrices A and B and implicitly expresses communications at execution time to execute the assignment. Here is an equivalent program in C*, where A and B are declared of the same shape:

```
shape [10][10]matrix;
real:matrix A, B
...
where ((pc_coord(0)>0) && (pc_coord(1)>0))
  B = [.-1][.]A + [.][.-1]A;
...
```

Expressions of data distribution are very close on this example since facilities exist in these two languages to express that some index in two arrays refers to the same virtual processor.

We can conclude here that the balance between programmer and compiler investments in these two programs is the same. Just an introductory example then!

2.2. Summation of unaligned vectors

```
      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 the template X is used for defining the alignment of variable A relatively to variable B. The alignment of variable C is left in charge of the compiler.

The previous HPF program could be encoded as follows in C*:

```
shape [15]vector;
real:vector A,B,C;
...
where (pc_coord(0)<8)
  C = [.+1]A + [.*2]B;
...
```

where A, B and C are "re-indexed" in the assignment in order to describe a relationship between vector values and virtual processors which is equivalent to those described by alignment directives in the HPF program. In C* then, communications between virtual processors are expressed explicitly through the assignation statement. Moreover the "alignment" of C and the touched values are defined by the programmer who has to define *active* virtual processors.

Then translating this simple example from HPF to C* first reveals some difficulties the HPF compiler will ensure and those left in charge of the programmer while writing the C* program.

2.3. Matrix product

Here is a program in HPF. Since its code is really inspired by a sequential one, it illustrates a common way of HPF programming:

```
      REAL A(0:7,0:7), B(0:7,0:7), C(0:7,0:7)
!HPF$ ALIGN A(I,*) WITH C(I,*)
!HPF$ ALIGN B(*,J) WITH C(*,J)
      ...
      DO K=0,7
        FORALL (I=0:7, J=0:7)
          C(I,J) = A(I,K)*B(K,J) + C(I,J)
        END FORALL
      END DO
      ...
```

The directives in this program advise the compiler to collapse all the elements in row `I` of matrix `A` and all the elements in column `J` of matrix `B`, onto the same virtual processor than `C(I,J)`. This refers to a particular case of alignment, called *collapsing*.

In this example, since the alignment is defined for every element of `C`, any row `I` of `A` is replicated onto every virtual processor corresponding to an element in row `I` of `C`. Similarly any column `J` of `B` is replicated onto every virtual processor corresponding to an element in column `J` of `C`. Thus, this collapsing implies a large amount of *replication*; if the compiler follows the directives no communication will be performed. However, that will of course entail a probably excessive memory space usage because of columns and rows replication, unless a `DISTRIBUTE` directive is inserted and may reduce memory space on the physical architecture. Therefore the usage of such directives can be very tricky for the programmer since they have heavy consequences on the program behaviour according to their interaction and the way the compiler will implement them.

Without any further programmer's help, it is very difficult for the compiler to find a good trade-off between communications and memory space usage. Hence doing this requires to analyze not only the alignment directives, but also the scheduling of the summation of products by using commutativity and associativity properties of addition.

The programmer can help the compiler a little more by inserting an `INDEPENDENT` directive just before the `DO` loop in the program: This asserts that the iterations of the loop could be executed in any order. Even if this directive is added, a lot of workload remains for the compiler to produce a best distributed code.

Now, let us study the way language C⋆ can work on the matrix product example and the balance between programmer and compiler workload it implies.

Collapsing does not exist in C⋆ but the programmer is responsible for memory usage and communications which are explicit: The choice of shape for the matrices induces a program behaviour. For example, in order to compute products locally, the matrices have to be embedded into a three-dimensional shape: Their elements can then be broadcasted onto adequate places and products can be computed without any other communication. Another choice yields the following program, referred to as Cannon's algorithm [11], where the matrices are embedded into a two-dimensional shape:

```
shape [8,8]matrix;
real:matrix A, B, C
...
A = [.,(.+pc_coord(0))%%8]A;
B = [(.+pc_coord(1))%%8,.]B;
...
for(k=0;k<8;k++)
  {
  A = [.,(.-1)%%8]A;
  B = [(.-1)%%8,.]B;
  C += A*B;
  }
...
```

The previous program describes a memory-efficient version of the matrix product. Let us briefly recall this algorithm: First, elements of `A` and `B` are re-arranged in such a way that elements `A(I,(I+J)%8)` and `B((I+J)%8,J)` are placed on the same virtual processor. This arrangement is achieved by shifting all elements of `A` to the left, with wraparound, by `I` steps and similarly by shifting up all elements of `B`, with wraparound, by `J` steps. Then, at every iteration, elements of `A` (resp. `B`) are moved one step left (resp. up) so that products can be performed locally.

Much work has been done by the programmer on this program and especially the proof of program correctness. Moreover a load balancing has been performed so that the compiler can easily produce an efficient code.

This example clearly outlines different relationships between compiler and programmer. The question is now: Is a HPF programmer able to transform the previous program in order to help the compiler to find out Cannon's algorithm? This would require that non-linear alignments are allowed in HPF programs in order to express an initial arrangement of elements of matrices `A` and `B`. Such a statement could be then re-written as

```
      REAL A(0:7,0:7), B(0:7,0:7), C(0:7,0:7)
!HPF$ ALIGN A(I,J) WITH C(I,MODULO(J-I,8))
!HPF$ ALIGN B(I,J) WITH C(MODULO(I-J,8),J)
```

```
...
DO K = 0, 7
  FORALL (I = 0:7, J = 0:7)
    C(I,J) = A(I,MODULO(I+J-K,8))*B(MODULO(I+J-K,8),J)+
             C(I,J)
  END FORALL
END DO
...
```

where ALIGN pseudo-directives specify an initial arrangement of elements of matrices A and B as in Cannon's algorithm. Beyond syntactical differences in languages such as C* and HPF, this example again shows a different balance between the programmer and the compiler workload: Non-linear alignments, if allowed in HPF, should involve new compiling techniques [1], whereas the difficulty is in writing the program in C*.

In the rest of the paper we introduce a theory which defines features such that the programmer and the compiler can fairly collaborate to reach program correctness and efficiency.

3. An introduction for the theory

3.1. Objects

The theory lies on a notion of objects called *shaped data fields*. A shaped data field is mainly a container of values.

Containers without values are sometimes referred to as *shapes* (for example refer to [8] in which C.B. Jay defines a very general and abstract notion of shape via a categorical pullback). An original point of the theory we introduce here consists in a particular notion of shape: we consider a shape composed of two sets of points indeed, respectively called *indices* and *locations*, and by some arrows between them.

Any point in a shape belongs to some "geometrical space", i.e., $\mathbb{Z}^n$. A location expresses a place where at most one value can be placed. Each index of a shape is connected to one or more locations and allows all the values, assumed to be equal, which are placed at these locations to be accessed "as a whole". Conversely, every location is accessed by one index at most. Thus, every value of a shaped data field has a location and is accessed via an index in the index set.

In the literature, indexed collections of values, detached from locations, are generally called *data fields* (for example refer to Alpha [12] or Lisper formalism [7]). It should be clear now that the theory we introduce defines a shaped data field as such a data field associated with a shape. For example, the shaped data field on Fig. 1 associates a shape with the data field on Fig. 2.

In the sequel, the index set of a shaped data field naturally refers to the data field index set. Moreover when we talk about "values" it stands for the values within some shaped data field.

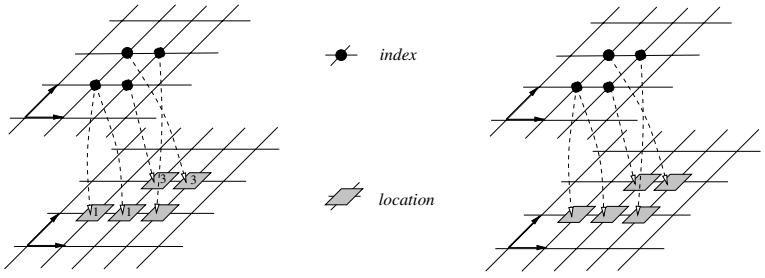


Fig. 1. A shaped data field (left) and its shape (right). It contains four values: 1 twice and 3 twice. Each value has its own location in $\mathbb{Z}^2$: For example, an instance of 1 is at location (1,1) while the other one is at location (2,1). The locations are connected with indices (1,1), (2,1), (1,2) and (2,2) in $\mathbb{Z}^2$: Index (1,1) refers to two locations while every other one is connected with only one own location. Note that index (2,2) is connected with one location but no value is placed in it.

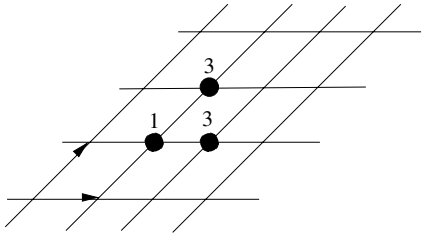


Fig. 2. The associated data field.

3.2. Operations

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

- A *change of basis* re-defines the relationship between indices and locations by changing the indices while keeping the values and their locations unchanged. It concerns both the shape and the data field of a shaped data field.
- A *geometrical operation* does not affect the shape but changes the indices of the values on the data field: it possibly moves, deletes or duplicates some values on locations. Any change of basis or geometrical operation is determined from a function on indices of the resulting shaped data field to the indices of the given 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.

3.3. A minimum notation set

Here we introduce a minimum notation set for expressing *statements*. A statement is a finite set of equations whose variables are shaped data fields. A variable is

denoted by an uppercase letter (e.g., $A, B, X, \ldots$). The classical arithmetical or logical operations are overloaded with global operations (e.g., $A + B$).

In order to simplify the notations, change of basis and geometrical operations on variables are denoted in an unified way as $X.f$ where f is considered as a pair (h, g) of partial functions: h defines the change of basis, g defines the geometrical operation and f stands for applying first the change of basis and then the geometrical operation on X.

This means that an operation which is really a change of basis is denoted as $X.f$, where $f = (h, g)$ and g is the identity and that an operation which is really a geometrical one is denoted as $X.f$, where $f = (h, g)$ and h is the identity.

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

Example 1. The shaped data field resulting from the geometrical operation illustrated on Fig. 3(b), applied on a shaped data field, say A, is denoted as A.spread, where spread denotes the pair (h, g): h is the identity and g is the partial function $\lambda(i, j).(1, j)_{\backslash D}$, with $D = \{(i, j) | 1 \leqslant i, j \leqslant 3\}$. Then, if B denotes the resultant shaped data field, we write the equation $B = A$.spread.

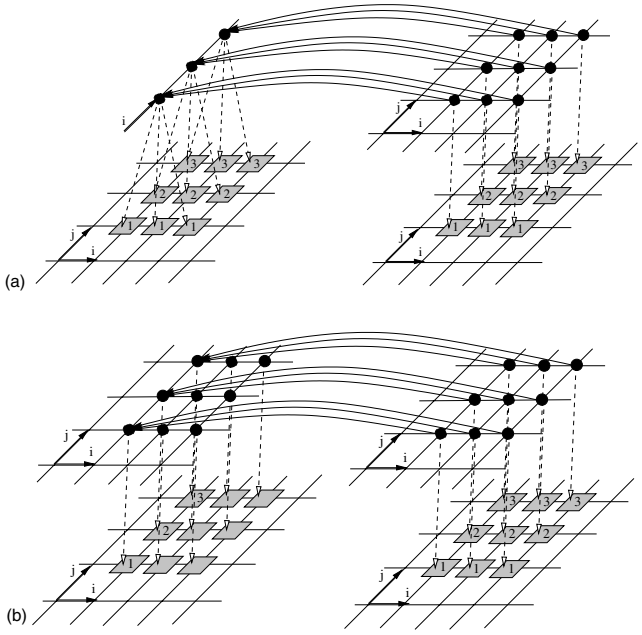


Fig. 3. Operations on shaped data fields. (a) The resulting shaped data field at the right is defined by a change of basis applied to the shaped data field on the left. The change of basis is defined from the function which associates point of coordinate i in $\mathbb{Z}$ with any point (i, j) in the square $[1, \ldots, 3] \times [1, \ldots, 3]$ of $\mathbb{Z}^2$. (b) Illustrates a geometrical operation defined from the function which associates point of coordinate $(1, j)$ with any point (i, j) in the same square.

Similarly, the shaped data fields, say A' and B', on Fig. 3(a) satisfy the equation $B' = A'.\text{spread}'$, where spread$'$ is the pair composed of partial function $\lambda(i,j).i_{\setminus D}$ and the identity.

3.4. Theory adequacy

In this section, we consider several examples which aim at illustrating the adequacy of the theory just defined. The next section will provide a formal model to prove the following assertions:

Example 2 (*Expression of uniform dependencies*). Let us consider the dependencies $\mathtt{B}[i] = \mathtt{A}[i+1]$, for any $i \in [0,\ldots,6]$. In our theory they can be expressed as

$$B = A.\text{shift}, \tag{1}$$

where A and B are shaped data fields representing arrays A and B and "shift" denotes any pair of functions (h,g) satisfying $h \circ g = \lambda i.i + 1_{\setminus D}$ with domain $D = \{i | 0 \leqslant i \leqslant 6\}$.

For example, Fig. 4 represents two shaped data fields A and B which satisfy Eq. (1) for two different pairs (h,g) of functions:

- The left part (a) represents the case of a change of basis where $h = \lambda i.i + 1_{\setminus D}$ and g is the identity,
- The right part (b) represents the case of a geometrical operation where $g = \lambda i.i + 1_{\setminus D}$ and h is the identity.

In both cases, any value of B accessed by an index $i \in [0,\ldots,6]$ is equal to the value of A accessed by index $i+1$: The dependencies between arrays A and B are then satisfied.
Nevertheless the two cases express different placements of values onto locations. In the first case there is no communication because the values $\mathtt{B}[i]$ and $\mathtt{A}[i+1]$ have the same location. The second case means that A and B have the same shape: Elements $\mathtt{B}[i]$ and $\mathtt{A}[i+1]$ forcefully have then different locations and communications are required.

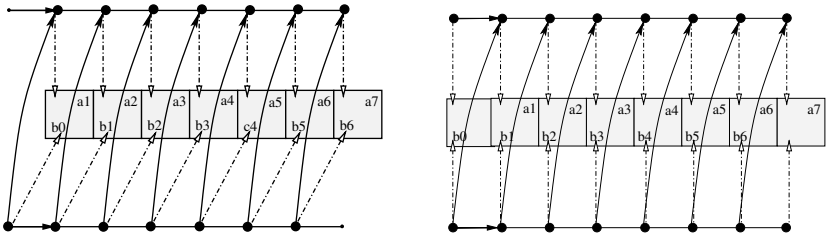


Fig. 4. Array B obtained by shifting array A.

These examples could be written, in HPF for the case of change of basis or in C⋆ for the case of geometrical operation, as follows:

```
      REAL A(1:7),B(0:6)            shape [8]vector;
!HPF$ ALIGN B(I) WITH A(I+1)        real:vector A, B;
      ...                           ...
      B(0:6) = A(1:7)               B = [.+1]A;
      ...                           ...
```

Example 3 (*Summation of unaligned vectors*). Let us consider again the Example 2.2: arrays A, B and C depend each other according to the relationship $C[i] = A[i] + B[i]$, for any $i \in [0, \ldots, 7]$.

It is expressed as C = A + B in HPF independently of the placement of values. In our theory this could be expressed by the following statement:

$$C = A.id_1 + B.id_2,$$

where id_1 and id_2 are pairs (h_1, g_1) and (h_2, g_2) of functions whose composition is equal to the identity on the domain $[0, \ldots, 7]$. These pairs of functions express a particular placement of the arrays. For instance, the placement of arrays in the HPF program could be expressed by the following definitions: [1]

$$h_1 = g_1^{-1} \quad g_1 = \lambda i.i + 1_{\setminus D}$$

$$h_2 = g_2^{-1} \quad g_2 = \lambda i.2 \times i_{\setminus D}$$

with $D = \{i | 0 \leqslant i \leqslant 7\}$, as depicted on Fig. 5.

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 parallel variables, data locations, global operations and communications. Here we confront the theory with a major semantic point in data parallel languages: the semantics of indices in a variable.

In languages such as HPF the indices in a program refer to indices of arrays without any reference to data location. Then, if a program expresses that a value in some index depends on the value in another one, it may involve any other relation between locations, which may imply or not some communication: In such cases we say that communications are hidden. Moreover the association between indices and locations may not be one-to-one: An array index can then address several locations. This is a particular, but powerful, case of data alignments through directives.

In other data parallel languages such as C⋆, indices refer to virtual processors: X[*i*] means the local value of X in the *i*th processor. The association between indices and locations is then one-to-one and any dependence between a value in some index and a value in some other one involves a similar dependence between the two associated

[1] Where f^{-1} stands for the inverse of f, provided f is injective.

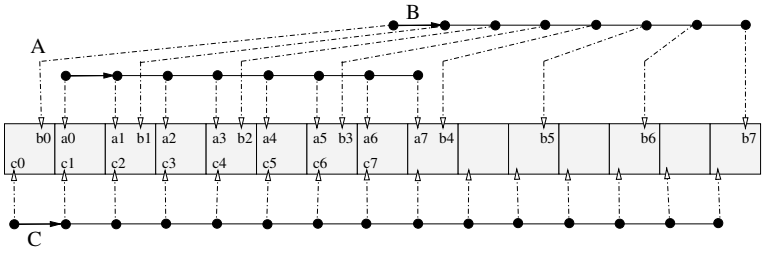


Fig. 5. Relative placements of arrays A, B and C.

locations. Therefore, parallel variables are distributed on the mesh and communications are made explicit.

Example 4 (*Matrix product*). Let us consider the computation of all products $\mathtt{A}[i,k] \times \mathtt{B}[k,j]$ for $1 \leqslant i,j,k \leqslant n$, mapped on a cubic three-dimensional array P. Here is a statement for this problem, by introducing geometrical operations

$$P = A.\mathrm{spread}_1 \times B.\mathrm{spread}_2,$$

where $\mathrm{spread}_1 = (h, g_1)$ and $\mathrm{spread}_2 = (h, g_2)$ with

- h the identity,
- functions g_1 and g_2 defined as $g_1 = \lambda(i,j,k).(i,k)_{\backslash D}$ and $g_2 = \lambda(i,j,k).(k,j)_{\backslash D}$, where $D = \{(i,j,k) | 1 \leqslant i,j,k \leqslant n\}$.

Shaped data fields A and $A.\mathrm{spread}_1$ have the same shape: it 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 may be a one-to-one correspondence between indices and locations. These points are shown on Fig. 6. This statement can thus be interpreted in different intuitive semantics of data parallel languages, for example in C*.

Let us consider now an other statement for the same problem by introducing changes of basis substituted for geometrical operations:

$$P = A.\mathrm{spread}_3 \times B.\mathrm{spread}_4,$$

where $\mathrm{spread}_3 = (h_3, g)$ and $\mathrm{spread}_4 = (h_4, g)$ with

- g the identity,
- functions $h_3 = g_1$ and $h_4 = g_2$.

Shaped data fields A and $A.\mathrm{spread}_3$ have two different shapes but the same values are associated with the same locations. Since every index on a vertical line of the cube, as shown on Fig. 7, is connected to at least one location and since the values accessed by an index on such a line are the values which are accessed by an unique index in A,

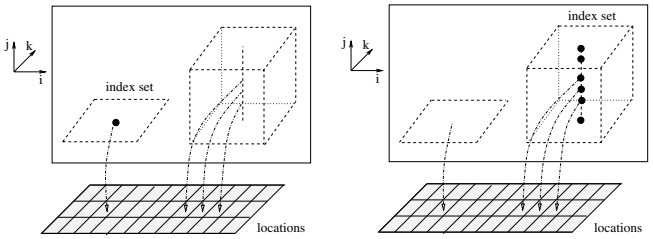


Fig. 6. Shaped data fields A and $A.\mathrm{spread}_1$.

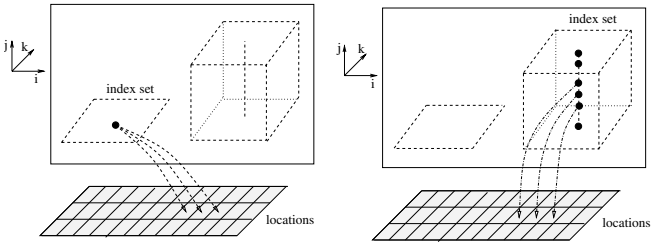


Fig. 7. Shaped data fields A and $A.\mathrm{spread}_3$.

this means that any index in A is necessarily connected to all the locations which are associated with the indices on the line in $A.\mathrm{spread}_3$. This means that the shape of A cannot be a one-to-one correspondence between indices and locations.

This statement expresses then virtual indices in an array which can be associated with different locations on a machine. This is typically what a data alignment expresses in HPF through an `ALIGN` directive. It describes then a different algorithm since dependences, involving communications in the previous expression, have been modified and now refer to locations, involving data alignment.

4. A formal model of the 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. Very few works are dedicated to semantics of data parallel languages: Let us mention the language $\mathscr{L}$ [4] and its semantics which is a model for languages such as C*. Our proposal is more general and meets the different cases we talked about just above.

4.1. 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* is a model for collections of indexed values such that any 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 index the values of a data field may belong to different geometrical spaces. 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 5. Let V be a given data type. A data field, whose values are in V, is any partial function $\mathscr{X}$ from I to V:

$$\mathscr{X} : I \rightharpoonup V.$$

We note $\mathrm{DF}(V)$, the set of data fields whose values are in 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 assumed to be injective (any location has at most one corresponding 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 6. A shape is any partial function σ from I to I:

$$\sigma : I \rightharpoonup I.$$

We note S, the set of shapes.

A shaped data field X is a data field $\mathscr{X}$ 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, $\mathrm{def}(\mathscr{X}) \subseteq \mathrm{img}(\sigma)$. [2]

In order to properly define this association, a shaped data field is defined via a partial function as follows:

Definition 7. 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, which returns a data field $\mathscr{X}$, such that $\mathrm{def}(\mathscr{X}) \subseteq \mathrm{img}(\sigma)$ for any shape σ in its domain of definition.

$$X : S \rightharpoonup \mathrm{DF}(V) \quad \text{such that } \forall \sigma \in \mathrm{def}(X)\ \mathrm{def}(X(\sigma)) \subseteq \mathrm{img}(\sigma).$$

[2] Where $\mathrm{def}(\mathscr{X})$ stands for the domain of definition of $\mathscr{X}$ and $\mathrm{img}(\sigma)$ is a shorthand for the image of σ.

4.2. Operations

These definitions associate a mathematical meaning of operations with the notation introduced in Section 3.3. For sake of conciseness, the definitions are given using *semantic equations* [14]: such an equation defines the result of a given operation applied on arbitrary arguments. Moreover, the result of any operation is a shaped data field, i.e., a function: It is defined itself by its image, i.e., a data field, resulting from an arbitrary argument, i.e., a shape.

Last, in every semantic equation, it is implicit that the shaped data field applied on σ, on the left side, is undefined if and only if all the values of the data field, on the right side, are undefined.

Here is some notations: $g, h, \sigma : I \rightharpoonup I$, $A, B, X : S \rightharpoonup \mathrm{DF}(V)$, $\mathrm{geom} = (id, g)$ and $\mathrm{basis} = (h, id)$, with id the identity on I.

Definition 8. A global operation, denoted as $+$, on two shaped data fields A and B is such that

$$(A + B)(\sigma) \stackrel{\mathrm{Def}}{=} A(\sigma) + B(\sigma).$$

The notation $+$ on the right side of this semantic equation is for the addition on data fields. This addition can be defined by the equation

$$(\mathscr{A} + \mathscr{B})(z) = \mathscr{A}(z) + \mathscr{B}(z),$$

where the result is defined on the intersection of the domains of definition of the two arguments.

Definition 9. A geometrical operation, denoted as "geom", is such that

$$(X.\mathrm{geom})(\sigma) \stackrel{\mathrm{Def}}{=} \begin{cases} X(\sigma) \circ g & \text{if } \mathrm{def}(X(\sigma) \circ g) \subseteq \mathrm{img}(\sigma) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This definition says that for any shaped data field X, whose data field $\mathscr{X}$ is associated with shape σ, the result of the geometrical operation, defined from g and applied on X, is the data field associated with shape σ, which maps value $\mathscr{X}(g(z))$, if defined, to each index z of σ.

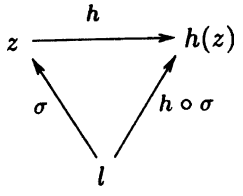
Definition 10. A change of basis, denoted as "basis", is such that

$$(X.\mathrm{basis})(\sigma) \stackrel{\mathrm{Def}}{=} X(h \circ \sigma) \circ h.$$

This definition says that for any shaped data field X, whose data field $\mathscr{X}$ is associated with shape $h \circ \sigma$, the result of the change of basis, defined from h and applied on X, is the data field associated with shape σ, which maps value $\mathscr{X}(h(z))$, if defined, to each index z of σ.

Thus, X and X.basis have the same values placed at the same locations: since the shape of X is $h \circ \sigma$, considering any value of X at location l, its corresponding index

is $(h \circ \sigma)(l)$, i.e., $h(z)$ with $z = \sigma(l)$. This value is then equal to the value of X.basis at location l. This proof is summarized by the following commuting diagram:



Definition 11. The composition of a change of basis and a geometrical operation is defined as

$$X.(h,g) \stackrel{\text{Def}}{=} (X.\text{basis}).\text{geom}.$$

Let us call Y the result of this operation. By definition, and for any σ such that $Y(\sigma)$ is defined, we have

$$\begin{aligned} Y(\sigma) &= ((X.\text{basis}).\text{geom})(\sigma) = (X.\text{basis})(\sigma) \circ g = (X(h \circ \sigma) \circ h) \circ g \\ &= X(h \circ \sigma) \circ (h \circ g). \end{aligned}$$

It means that, for any shaped data field X, whose data field $\mathscr{X}$ is associated with shape $h \circ \sigma$, the result Y is the data field associated with shape σ, which maps value $\mathscr{X}((h \circ g)(z))$, if defined, to each index z of σ, i.e., value $\mathscr{Y}(z)$ is equal to value $\mathscr{X}((h \circ g)(z))$.

5. A case study

In this section, we use the theory to deal with the matrix product problem.

Let us show how to write a first statement for this problem. This initial statement is built from the following set of *recurrent equations*:

$$\begin{aligned} &t_{i,j,k} = 0 \quad \text{if } (i,j,k) \in D_{-1}, \\ &t_{i,j,k} = a_{i,k} \times b_{k,j} + t_{i,j,k-1} \quad \text{if } (i,j,k) \in D, \\ &t_{i,j,k} = c_{i,j} \quad \text{if } (i,j,k) \in D_{n-1}, \end{aligned}$$

with

$$\begin{aligned} D_{-1} &= [0,\ldots,n-1] \times [0,\ldots,n-1] \times \{-1\}, \\ D &= [0,\ldots,n-1] \times [0,\ldots,n-1] \times [0,\ldots,n-1], \\ D_{n-1} &= [0,\ldots,n-1] \times [0,\ldots,n-1] \times \{n-1\}, \end{aligned}$$

where indexed variables a, b and c identify the $n \times n$ matrices, and t is an intermediate variable indexed on $[0,\ldots,n-1] \times [0,\ldots,n-1] \times [-1,\ldots,n-1]$.

Here is our first statement:

$$T.\text{init} = \cdots$$
$$T.\text{current} = A.\text{spread}_a \times B.\text{spread}_b + T.\text{prec}$$
$$T.\text{term} = C.\text{expand},$$

where A, B, C and T rely to variables in the previous recurrent equations set and "init", "current" and "term" operations restrict variable T on some sub-domains. They are defined as follows:

$$\text{init} = (id, id_{\setminus D_{-1}}) \quad \text{current} = (id, id_{\setminus D}) \quad \text{term} = (id, id_{\setminus D_{n-1}}).$$

And $\text{spread}_a = (h_a, g_a)$, $\text{spread}_b = (h_b, g_b)$, $\text{prec} = (h_p, g_p)$ and $\text{expand} = (h_e, g_e)$ express dependencies in the recurrent equations set.

They are such that

$$h_a \circ g_a = \lambda(i,j,k).(i,k)_{\setminus D},$$
$$h_b \circ g_b = \lambda(i,j,k).(k,j)_{\setminus D},$$
$$h_p \circ g_p = \lambda(i,j,k).(i,j,k-1)_{\setminus D},$$
$$h_e \circ g_e = \lambda(i,j,k).(i,j)_{\setminus D_{n-1}}.$$

Such dependencies determine the scheduling of computations. Without any other precision, the previous statement is the semantics of the following program written using a HPF-like notation:

```
REAL A(0:N-1,0:N-1), B(0:N-1,0:N-1), C(0:N-1,0:N-1)
REAL T(0:N-1,0:N-1,-1:N-1)
...
FORALL (I=0:N-1,J=0:N-1)
  DO K=0,N-1
    T(I,J,K)=A(I,K)*B(K,J)+T(I,J,K-1)
  END DO
END FORALL
FORALL (I=0:N-1,J=0:N-1)
  C(I,J)=T(I,J,N-1)
END FORALL
...
```

where the scheduling is explicit but the relationship between array indices and virtual processors is undefined. Note that this program is incorrect in HPF because it is not allowed to put a `DO` loop inside a `FORALL` body in HPF.

Setting pairs (h_a, g_a), (h_b, g_b), (h_p, g_p) and (h_e, g_e) to a particular instance attaches additional operational properties to the statement and preserves its correctness.

In the sequel, a line of reasoning is followed in order to find out some efficient solutions. The reasoning is based on both a proof system and an operational semantics associated with the considered statements. The proof system enables the user to prove the correctness of statements whereas the operational semantics enables him to weigh their efficiency. These issues are formally described in [16,6], respectively,

and defined for a class of statements called *well-formed* statements. Well-formed statements are statements in single-assignment form and such that the shape of each variable can be inferred uniquely from the shape of one of them, called *template*. Moreover, the partial functions (g and h) of a well-formed statement do not depend on the values of the variables. All the statements in this paper meet these conditions. Given a well-formed statement, a procedure determines one shape for each variable so determining, for each variable, its extent of indices, its extent of locations and the placement of its values onto locations. Then, each equation of the statement is viewed as both a relation between indexed values and a transition rule. The relation is stated by the proof system, whereas the transition rule is part of the operational semantics. In particular, here is what the operational semantics defines

- the virtual processors array for carrying out computations (induced from shapes of the shaped data fields involved in the statement, and induced itself from the changes of basis),
- the data placement on this array (also induced from shapes),
- the required memory size for storing data on each virtual processor (induced from the data placement),
- the computations to be performed by any virtual processor of the array (also induced from the data placement and from the "owner-computes" rule),
- the required (virtual) communications between the virtual processors of the array (also induced from the data placement),
- the computations scheduling (induced from the communications).

So, the previous statement can be transformed step by step by both the programmer and the compiler, for example in order to define a better trade-off between communications and memory usage. Some transformations may consist in substituting a pair of functions, say (h', g'), for some other, say (h, g), such that $h' \circ g' = h \circ g$. The so obtained statement expresses the same dependencies, i.e., remains correct, but a different relationship between indices and locations, i.e., is attached to a different operational meaning.

As examples, let us consider the following cases where we focus on the placement of matrix A values relatively to T ones. In each case, the data placement is determined by a particular pair (h_a, g_a).

Each case is illustrated with a figure. The figure shows the locations of shaped data field T. In order to simplify the drawing, every location of T is identified with its corresponding index. The locations of A values is a subset of the locations of T. These locations are grey painted. Last, an arrow between two locations specifies a (virtual) communication.

(1) Replication (Fig. 8). Let us define

$$h_a = \lambda(i,j,k).(i,k)_{\setminus D}$$

$$g_a = id$$

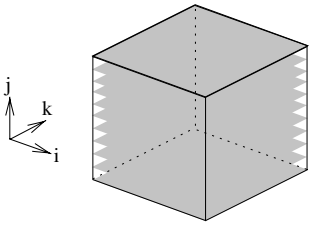


Fig. 8. Matrix replication.

It expresses that value $a_{i,k}$ is located at every location where some $t_{i,j,k}$, $j \in [0, \ldots, n-1]$, is located. Consequently, products computation does not require matrix A values communications. In return each value $a_{i,k}$ is (virtually) stored n times.

(2) Broadcast (Fig. 9). Let us define

$$h_a = id$$
$$g_a = \lambda(i,j,k).(i,k)_{\setminus D}.$$

It expresses that value $a_{i,k}$ has its own location which is different from the locations where T values are placed. Thus, value $a_{i,k}$ is broadcasted to all the locations where some $t_{i,j,k}$, $j \in [0, \ldots, n-1]$, is located.

(3) Alignment and broadcast (Fig. 10). Let us define

$$h_a = \lambda(i,j,k).(i,k)_{\setminus [0,\ldots,n-1]\times\{0\}\times[0,\ldots,n-1]},$$
$$g_a = \lambda(i,j,k).(i,0,k)_{\setminus D}.$$

It expresses that value $a_{i,k}$ is placed at the same location as $t_{i,0,k}$. As previously, value $a_{i,k}$ is broadcasted to all the locations where some value $t_{i,j,k}$, $j \in [0, \ldots, n-1]$, is located.

(4) Cannon's algorithm (Fig. 12).
The previous cases could be somewhere unsatisfactory for the compiler, for example if broadcast cannot be efficiently implemented due to some architecture limitations. In such cases, the programmer can be helpful by allowing to change the sequence of products accumulation and then by allowing an other

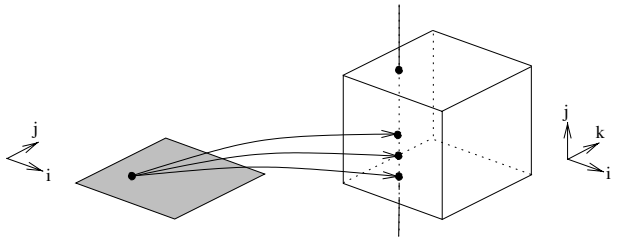


Fig. 9. Matrix broadcast.

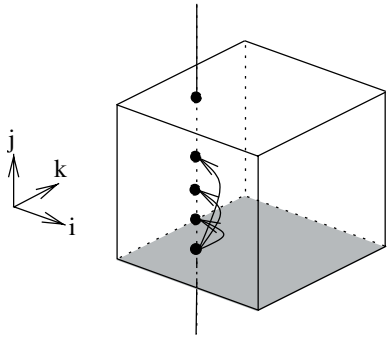


Fig. 10. Matrix alignment and broadcast.

communication-memory trade-off. This could be done in a HPF-like notation by inserting a `DO INDEPENDENT` directive in the previous program.
The semantics of the new program could be defined as

$$h_a \circ g_a = \lambda(i,j,k).(i, h_{i,j}(k))_{\setminus D},$$

$$h_b \circ g_b = \lambda(i,j,k).(h_{i,j}(k), j)_{\setminus D},$$

where $h_{i,j}$, $i,j \in [0,\ldots,n-1]$, is any permutation of $[0,\ldots,n-1]$.
Let us focus again on the placement of A. As seen previously it is possible to split dependencies into an alignment and a broadcast.
An interesting case is when the dependency allows A values to be aligned with values $t_{i,j,0}$ of T and then to be broadcasted along k axis, as drawn on Fig. 11. In this case the broadcast can be performed while products are computed. Such an alignment is allowed if and only if $\lambda(i,j).(i, h_{i,j}(0))$ and $\lambda(i,j).(h_{i,j}(0), j)$ are two permutations of $[0,\ldots,n-1] \times [0,\ldots,n-1]$. This reasoning may be conducted either by the programmer or by the compiler and may yield the following definition for $h_{i,j}$:

$$h_{i,j}(k) = (i + j - k)\%n.$$

It induces a new equivalent statement where the pair (h_a, g_a) is defined as follows:

$$h_a = \lambda(i,j,k).(i, (i+j)\%n)_{\setminus [0,\ldots,n-1]\times[0,\ldots,n-1]\times\{0\}},$$

$$g_a = \lambda(i,j,k).(i, (j-k)\%n, 0)_{\setminus D},$$

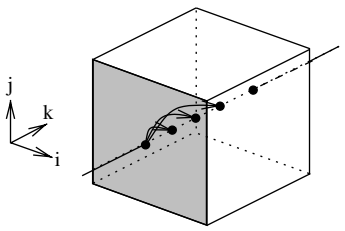


Fig. 11. Matrix alignment and broadcast along k axis.

in which h_a defines the initial placement of Cannon's algorithm and g_a expresses the communications to be performed.
From this new statement, the compiler can use a well-known *uniformization* technique to make communications uniform (Fig. 12) and implement these communications by using links of a parallel architecture.
Applying the same transformations for shaped data field B, i.e., defining (h_b, g_b) as

$$h_b = \lambda(i,j,k).((i+j)\%n, j)_{\setminus[0,\ldots,n-1]\times[0,\ldots,n-1]\times\{0\}},$$
$$g_b = \lambda(i,j,k).((i-k)\%n, j, 0)_{\setminus D},$$

yields Cannon's algorithm.

This case study illustrates a typical use of our framework to improve data locality. Statement transformations in our framework lie on composition of functions using formal calculus, either to compose functions (in order to prove correctness), or to find out part of them (in order to reach efficiency). The example outlines how both the programmer and the compiler can participate in performing this calculus. Part of the calculus (relying on analytic geometry) can be automated and performed by the compiler, part relying on some more complex or abstract algebraic properties can be incrementally achieved by the programmer.

Due to its unifying aspect, our framework enables the programmer and the compiler to overcome some difficulties on each side and to deal with some complicated issues such as non-affine index expressions or sparse computations. As examples, in [6], we consider the one-dimensional unordered radix-2 FFT (that exhibits non-affine dependencies) and use our framework to concisely transform a statement describing the binary-exchange algorithm into a statement describing the two-dimensional transpose one. This transformation consists in just adding one new equation for defining a new template. The new template results from a change of basis applied to the old one. This transformation can deeply modify the parallel algorithm. In [17], our framework is used to minimize communications when solving the Navier–Stokes equation. The applied transformations involve complex index functions (built from quotient, remainder and product of integer variables). In [18], it is shown how our framework can be used to systematically derive sparse formulations from dense programs. A particular sparse storage of a matrix is expressed by the application of a change of basis on a shaped data field. Thanks to the algebraic properties of the

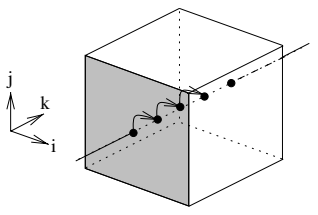


Fig. 12. Uniform communications.

operations on shaped data fields, this change of basis can be spread through the whole statement and the accesses to the sparse structure can be optimized.

6. Conclusion

Data locality expression is of great importance in data-parallelism. Therefore, a formal framework is necessary to formally define and to explain this notion. We proposed our theory as such a framework. It offers a notion of data locality in which those of HPF and C* could join up.

In comparison with the alignment concept in HPF, the compiler workload is lower since a statement defines a precise data placement, on the other hand, in comparison with the shape concept in C*, the user workload is lower since a statement can be incrementally improved.

The theory is supplied with a sound model on which both a proof system and an operational semantics can be based. The proof system enables the user to systematically prove the correctness of data parallel statements. The operational semantics enables him to weigh the impact of statement transformations on efficiency. These two issues were defined for a subset of statements called well-formed statements for which a procedure determines the shape of every variables from functions h (defining a change of basis). It may be interesting to extend the set of considered statements. For example, an interesting case is when functions g (defining a geometrical operation) depend on the values of some variables. In that case, the previously mentioned procedure may be reused and a proof system and an operational semantics may be defined. More precisely, it may be interesting to state which statements are computable.

Our framework is designed to be a medium which can be used by the programmer and the compiler in order to obtain an efficient program. For example, there is no limitation of complexity about the index functions that are used to access values. Although the compiler and the user have their own limitations (i.e., mathematical restrictions required to automatically find the optimal solution in the compiler side and problem understanding in the user side) the medium remains the same.

This theoretical 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 for reaching a best implementation.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, R. Keryell, A linear algebra framework for static hpf code distribution, 1993. in: Workshop on Compilers for Parallel Computers, Delft, The Netherlands, December 1993.

[2] G. Antoniu, L. Bougé, R. Namyst, C. Pérez, Compiling data-parallel programs to a distributed runtime environment with thread isomigration, Parallel Processing Letters 10 (2/3) (2000) 201–207.

[3] S. Benkner, T. Brandes, Exploiting data locality on scalable shared memory machines with data parallel programs, in: A. Bode, T. Ludwig, W. Karl, R. Wismüller (Eds.), Euro-Par 2000 Parallel Processing Conference, Munich, Germany, August 29–September 1, 2000. Lecture Notes in Computer Science, vol. 1900, 2000, Springer, Berlin, pp. 647–656.
[4] L. Bougé, J.-L. Levaire, Control structures for data-parallel SIMD languages: semantics and implementation, FGCS 8 (1992) 363–378.
[5] B. Chapman, P. Mehrotra, H. Zima, Enhancing OpenMP with features for locality control, Technical Report TR99-02, Institute for Software Technology and Parallel Systems, U. Vienna, February 1999. Available from <www.par.univie.ac.at>.
[6] P. Gerner, E. Violard, A theoretical framework of data parallelism and its operational semantics, in: EURO-PAR'2000, LNCS, vol. 1900, Springer-Verlag, Berlin, 2000, pp. 668–677.
[7] P. Hammarlund, B. Lisper, On the relation between functional and data parallel programming languages, in: FPCA'93, ACM Press, 1993, pp. 210–222.
[8] C.B. Jay, A semantics for shape, Science of Computer Programming 25 (1995) 251–283.
[9] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., M.E. Zosel, The High Performance Fortran Handbook, MIT Press, Cambridge, MA, 1994.
[10] F. Kuijlman, H.J. Sips, C. van Reeuwijk, W.J.A. Denissen, A unified compiler framework for work and data placement, In: Proceedings of the ASCI 2002 Conference, Lochem, June 2002, pp. 109–115.
[11] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin/Cummings, 1994.
[12] C. Mauras, ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. PhD thesis, U. Rennes, 1989.
[13] The OpenMP Forum. OpenMP Fortran Application Program Interface. Proposal Ver 1.0, SGI, October 1997. Available from <http://www.openmp.org>.
[14] R.D. Tennent, in: C.A.R. Hoare (Ed.), Semantics of Programming Languages, Prentice Hall, Englewood Cliffs, NJ, 1991.
[15] Thinking Machines Corp., C* Programming Guide, November 1990.
[16] E. Violard, What really is data parallelism?, Technical Report RR 00-01, LSIIT-ICPS, Université Louis Pasteur, January 2000.
[17] F. Voisin, Etude d'outils logiciels pour la parallélisation et la transformation de programmes dans les applications de calculs numériques (Software tools study for parallelization and transformation of programs in numerical computation applications), PhD thesis, Université Strasbourg I—Louis Pasteur, July 2001.
[18] F. Voisin, G.-R. Perrin, Sparse computations with PEI, International Journal of Foundations of Computer Science 10 (14) (1999) 425–442.