The Status of Non-Mandatory Directives

September 30, 2003

Philippe Gerner
VERIMAG, Centre Équation, 2, av. de Vignate, 38610 Gières, France [1]

and

Éric Violard
LSIIT-ICPS – UMR 7005, CNRS-ULP, Bd. S. Brant, F-67400 Illkirch, France [2]

Abstract

Directives to the compiler seem to be a good idea for having both high-level programming and efficient programs in parallel programming. The ease of programming with directives is illustrated by the language HPF. However, HPF is not widely used today. OpenMP is another directives-based approach, aimed in part at being more effective than HPF. But MPI, though a lower-level programming solution, is still often preferred against OpenMP—for efficiency reasons. This paper addresses the question of the relative failure of these directive-based approaches. We argue that a better definition of the status of the directives could lead to directives being a solution of choice when designing high-level parallel languages with efficient implementations—especially when the problem of maintenance is considered. We show how to define clearly the status of the directives. We do this by applying the discussion to the case of data parallelism and its alignment directives.

1 Introduction

The primary motivation for parallel programming is efficiency at execution (in terms of execution time). The problem with parallel programming is that parallel execution

[1]e-mail: `gerner@imag.fr`
[2]e-mail: `violard@icps.u-strasbg.fr`

environments change frequently. And most often than not, the architecture of the new machine differs form the architecture of the old one, so that the majority of choices of implementation which were made for good performance on the old machine are not relevant any more on the new one. Today one even has to migrate one's parallel programs onto the computational grid [1].

In response to this problem, the traditional approach has been to look for an execution model which would be abstract enough to cover as large a set of architectures as possible, while reflecting as faithfully as possible the effective cost of execution of these architectures. These execution models must also come with a corresponding programming model.

Several models have been proposed. The message-passing programming model MPI [9] focuses on organizing a cooperation of communication processes. The "skeletons" approach consists in identifying "patterns" of parallel algorithms, and then proposing implementations for them on different architectures or execution models—see for example [7]. The BSP (Bulk Synchronous Parallelism) approach [20] is an execution model based on super-steps and synchronizations. It serves an an intermediary between the programmer and the machine: some languages are adapted to include the BSP model, and the BSP model is implemented onto several machines. An example is the BSMLlib library [11] for the Caml language. Data parallelism, as embodied by the languages C* [18] and High Performance Fortran (HPF [5]), focuses on global operations on arrays. The OpenMP parallel programming model (e.g., [2]) focuses on threads and shared memory.

Each of these approaches has its particular way of dealing with differing architectures. MPI is meant for distributed-memory architectures, but can be executed on machines with a shared-memory architecture as well—though the return on programming investment can then be less satisfying. Symmetrically, the purpose of OpenMP is to efficiently exploit shared memory architectures, which currently are the predominent architectures, but it can be made to run on a distributed memory architecture as well—with, agin, a possible lowering of the return on programming investment. BSP is an execution model with a notion of communication between tasks, so it is close to a message-passing programming model, but this is also independant of the effective implementation. The directives of HPF are meant to reduce the communications which would be generated by an execution on a distributed-memory machine, so these directives are less relevant for a shared-memory architecture.

With respect to adaptation to different architectures, the "directives" approach is interesting. This is because programs that need parallelism typically last, being very often big and complicated codes: they become "legacy code". Thus, changing the code when the current predominant architecture goes to far from the execution model the program is ideally suited for, is a task which ideally should be avoided. So we wish to reuse the existing code as much as possible, so as to capitalize on the large effort which has been put into the existing application. In this perspective, ideally, a clear distinction should be visible in the code between the functional part of the code, and the part which is aimed only at providing efficiency. And in this context, directives to the compiler may provide an interesting compromise between optimal performance and ease of system updating: for addressing a big change in the architecture, one would only have to change the directives which have been added to the functional code.

Directives have found a new dimension in high-level parallel programming with HPF and OpenMP. But what about the status of the directives? In OpenMP, directives are either all mandatory (to the compiler), or all ignored (by the compiler)—the choice is made by the programmer through an option to the compiler. When they are all mandatory, they do not allow the compiler the possibility of providing a better solution, in case it could have found one. This can be a problem given that OpenMP directives are lower-level than those of HPF, so with changes of architectures the compiler might want to have so latitude w.r.t. the directives. Of course it is possible to ignore all directives, and thus ignore the associated execution model, but in this case we are left with no help at all for the compiler. In contrast to that, the HPF approach, where directives are not mandatory, seem good.

However HPF or OpenMP programming is not the most frequent practice in parallel programming. This is because MPI, being lower-level, allows for fine tuning w.r.t. a particular architecture, and then is still the choice when high performance is crucial—which is bad from the "legacy code" point of view, as in an MPI code it is not easy to separate the MPI part form the functional part of the program, and so maintenance of such code is not an easy task. OpenMP provides some good and promising performance, but its directives present some problem which will be shown in the next section. We will argue also that HPF is a good language; but it suffered from being ahead of its time in terms of compiler technology (see, e.g., [4]) (it still is). That is, some of its directives can be so difficult to handle for the compiler, that the compiler might simply ignore them, not knowing how to handle them properly. That is why HPF programmers use debuggers or profilers for seeing "how this compiler has used our directives".

In this article we are interested in the problem posed by non-mandatory directives of the likes of HPF directives. The problem is that since these directives are only indications to the compiler, the programmer currently has no precise definition of what they "mean" to the program, and so he does not know which directives are relevant and which are not, in his program. There is no well-defined feedback from the compiler to the programmer, and the programmer has to resort to debuggers and profilers in order to try to infer the effect of his directives onto the program. But such inference is not precise, and is mostly "guessing what the compiler has done". One should obviously have a more precise contract between the programmer and the compiler, and writing this contract necessitates having a way to define precisely the semantics of directives—and in particular, when they are taken into account by the compiler, and how. In this article we aim at showing how it is possible to define the semantics of directives.

The following of the article in structured in four sections. In the following section, we motivate the study of the semantics of directives through several examples. In section 3, we propose a clean way of defining the semantics of directives. This is done through an example, by defining a small language, defining some directives to go with it, and defining the status of these directives. Section 4 presents some related works. The conclusion summarizes and describes perspectives.

2 A Motivating Example

In this section we show which problems are exhibited by the current types of directives in parallel languages.

We discuss directives in the two following programs that describe a standard 5-point stencil calculation used when solving Laplace's equation in two dimensions: first an OpenMP implementation, where directives are mandatory, and second an HPF implementation, where directives are not mandatory.

In both cases, the programmer wants to indicate to the compiler that, assuming both matrices `A` and `B` are divided column-wise into matrix blocks, each processor should compute its own matrix block. The OpenMP implementation is as follows:

```
        DOUBLE PRECISION :: A(N,M), B(N,M)

        ...

!$OMP   PARALLEL SHARED(A, B) PRIVATE(I,J)
!$OMP   DO SCHEDULE(STATIC)

        DO J = 2, M-1
          DO I = 2, N-1
            B(I,J) = (A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))/4
          END DO
        END DO

!$OMP   END DO
!$OMP   END PARALLEL
```

The directives in this program declare that (1) matrices `A` and `B` must be shared among threads (so each processor has access to its own matrix block), (2) a copy of variables `I` and `J` must exist on each process (so there is a local storage associated with these variables on each processor), (3) iterations of the outer `DO` loop must be divided among threads (so each processor executes the iterations corresponding to the computation of a matrix block).

Note that, for sake of efficiency, the programmer swapped the `DO` loops. Thus, one sees with this example that in OpenMP it is sometimes necessary to modify the existing code in order to add some directives to the compiler. The drawback, in this case, if that in case the OpenMP directives are later abandoned, the code modifications stay, and become an hindrance in later maintenance.

Practice of OpenMP seems to become more widespread than practice of HPF: the reason is probably that compiling OpenMP directives is easier than compiling HPF directives.

On the other hand, HPF draws a better line between the functional aspect and the efficiency aspect of the program. In the above program, from the functional point of view, the order in which assignments are executed is not relevant. In OpenMP, as in most imperative programs, the programmer has to decide on a schedule, without this schedule being important in itself. In HPF, by contrast, it is possible to abstract from a particular schedule. Here is an HPF version of the above program:

```
        DOUBLE PRECISION :: A(N,M), B(N,M)

!HPF$   DISTRIBUTE A(*,BLOCK)
!HPF$   ALIGN B(:,:) WITH A(:,:)

        ...

        FORALL(I = 2:N-1, J = 2:M-1)
          B(I,J) = (A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))/4
        END FORALL
```

The `FORALL` construct provides the programmer with a convenient means to express that the array assignments can be performed in any order. The directives thus do not have to take into account an arbitrary schedule. In the HPF program the directives describe an execution which is equivalent to the execution described by the OpenMP program. The directives declare that (1) matrix `A` should have its columns distributed across processors by slicing them uniformly into contiguous blocks and matrix `A` should not to be distributed along its first axis. (2) each of matrix `B` elements should be mapped in the same way as the corresponding matrix `A` elements (so `A(I,J)` and `B(I,J)` reside on the same processor and consequently each processor computes its own matrix block).

Thus, HPF directives are less intrusive than OpenMP directives, which is of importance from a software engineering of view.

The norm documentation for HPF states: "HPF directives appear as structured comments that suggest implementation strategies [. . .] about a program to the compiler" [5, p. 4]. Thus, "the explanations given [in the HPF norm] do not necessarily reflect the actual compilation process" [5, p. 6]. By contrast, the norm of OpenMP says that either all directives are ignored, or all directives are taken into account (by the compiler) (and the choice is made by the programmer with an option to the compiler). We prefer the HPF solution, because we think that it is important to let the compiler ignore a directive if it knows a better solution than the solution proposed by the programmer through this directive.

However non-mandatory directives pose some problems. Given a HPF program with directives, how is it possible to know which directives are relevant? Which are not? What is the placement schema chosen by the compiler? What is the impact of the set of directives?

For example, consider the case where the HPF programmer replaces one of its directives, d, by another one, d', and observes that execution of the program takes a shorter time. What does it mean about d and d'? We must consider the following possible cases (where "used" means "the compiler has followed the advice from this directive"). In each case, we report the conclusion the programmer could make if he knew that it is this case that does occur.

&	d was used	d was not used
d' is used	d' is better than d	d' is better than not d'
d' is not used	not d is better than d	–

But because the HPF compiler gives no indication on its treatment of directives, it is not possible for the programmer to know which case occurs, and therefore he can not assert whether d' is a good directive or not.

In the following section we aim at providing a method for introducing directives in a programming language in such a way that the programmer can use them efficiently.

3 Semantic Framework for Directives

We want to define a framework which allows for a clean definition of the semantics of directives. We want the directives to benefit from a first-class status in the definition of the programming language.

We choose to illustrate our proposed approach through the definition of the semantics of directives in the case of a data parallel formalism. In this case, the execution model which is addressed through the directives is execution on a parallel machine with a distributed memory architecture.

The structure of this section is as follows: first we define our formalism; then its semantics "stripped of directives" is given: in our framework it is important that a program has a meaning when directives are not considered. And then the semantics that takes the directives into account is given.

3.1 The formalism $\mathcal{D}$

We provide a small formalism, which we call $\mathcal{D}$, for modeling the concepts of a data parallel language. This formalism has no concrete syntax: we follow the view of Bjarne

Stroustrup, about "the syntax being the user interface of a language" [17, p. 208], and we focus here on the semantics.

For the semantics without directives and data distribution we are inspired by the Alpha language [23], which was designed for expressing systems of affine recurrent equations. We extend this to non-affine recurrent equations.

Programs and Variables In a $\mathcal{D}$ program variables are names which denote *data fields*, in the sense of Björn Lisper [15] and Éric Violard [22]. They can be seen as sparse arrays: values in a data fields are accessed via their *index*, taken in a set *Index*. We call *VarName* the set of variable names in a program. *Input* variables are identified, with a set $Input \subseteq VarName$—and similarly, output variables, with a set *Output*. The values associated with a data field are elements of a set *Val*, and can be seen through a function

$$\boxed{v : VarName \times Index \to Val_\perp} \ ,$$

where $Val_\perp = Val \cup \{\perp\}$, where *Index* is a set of indices, and where $\perp$ represents absence of a datum.

A program in $\mathcal{D}$ consists in a set of *assignments* and a set of *directives*, both involving data fields. Assignements and directives are now described. (The semantics of directives will be addressed in Sec. 3.3.)

Assignments An assignment is an n-tuple which denotes a relation between some data fields. We present the case where the relation is between three data fields: other cases are similar.

Such a ternary relation is noted with a 7-tuple $(Z, A, X, g_1, Y, g_2, op)$, where:

- $X, Y, Z \in VarName$, $Z \notin Input$;
- $A \subseteq Index$;
- g_1, g_2 are functions of type $Index \to Index_\perp$, such that $\forall i \in A,\ g_1(i) \neq \perp$ and $g_2(i) \neq \perp$;
- and op has type $Val \times Val \to Val$.

A function v is said to *satisfy* such a relation if:

$$\forall i \in A, v(Z,i) = \begin{cases} op(v(X,g_1(i)), v(Y,g_2(i))), & \text{if } \quad v(X,g_1(i)) \neq \perp \\ & \text{and } v(Y,g_2(i)) \neq \perp \\ \perp, & \text{otherwise,} \end{cases}$$

Directives A directive is described by a 4-tuple (X, A, Y, g), with $X, Y \in VarName$, $A \subseteq Index$, and $g : Index \to Index_\perp$. Intuitively, it describes an alignment between data fields X and Y: any $X(i)$ should be placed on the same processor as $Y(g(i))$.

Example We give the transcription in $\mathcal{D}$ of the stencil operation given in section 2, with only the alignment directive, for the sake of simplicity.

- $\mathit{Val} = \mathbb{R}$, $\mathit{Index} = \mathbb{N}^2$. $\mathit{VarName} = \{\mathtt{A}, \mathtt{B}\}$, $\mathit{Input} = \{\mathtt{A}\}$, $\mathit{Output} = \{\mathtt{B}\}$.
- Parameters : $N, M \in \mathbb{N}$.
- Assignments: $\{(\mathtt{B}, \mathit{comp\text{-}domain}, \mathtt{A}, \mathit{get1}, \mathtt{A}, \mathit{get2}, \mathtt{A}, \mathit{get3}, \mathtt{A}, \mathit{get4}, \mathit{stencil\text{-}op})\}$, with :

$$\mathit{comp\text{-}domain} = \{(i,j) \in \mathbb{N}^2 \mid 2 \leq i \leq N-1 \ \wedge \ 2 \leq j \leq M-1\}$$

$$\begin{array}{lcl} \mathit{get1} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i-1, j) \\ \mathit{get2} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i+1, j) \\ \mathit{get3} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i, j-1) \\ \mathit{get4} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i, j+1) \end{array}$$

$$\mathit{stencil\text{-}op} \ : \ \mathbb{R}^4 \to \mathbb{R}^*, \quad (v_1, v_2, v_3, v_4) \mapsto (v_1 + v_2 + v_3 + v_4)/4$$

- Directives: $\{(\mathtt{A}, \mathbb{N}^2, \mathtt{B}, \mathit{id})\}$, where id is the identity on $\mathbb{N}^2$.

3.2 Semantics without the Directives

In this section the semantics of $\mathcal{D}$ while directives are not yet not taken into account yet: only the assignments are considered. The semantics of the program is a set of chains of transitions, which represent the possible *executions* of the program. The transitions are from states to states, and a state is a function v. The execution goes from an initial state to a final state.

Initial State An *initial state* v_0 for the execution of the program is defined as containing values only in input variables:

$$\forall (X, i) \in (\mathit{VarName} \setminus \mathit{Input}) \times \mathit{Index}, \ v_0(X, i) = \perp \quad .$$

Transitions The assignments can be seen as equations (possibly recursive). The program is then a system of equations. In effect, execution is the construction of a solution to the program: we use the fixed-point construction [10]. We require that programs be in single-assignment form: the same value of a same data field can not be defined by two different assignments. (This is a natural restriction.) Thus the program can be seen as a single equation $v = \psi(v)$, where ψ is defined from the assignments. Then the operational semantics of the program corresponds to following the transitions that correspond to the repeated application of function ψ, beginning from the initial state v_0: $\psi(v_0), \psi^2(v_0), \psi^3(v_0), \ldots$. If each of the index domains in the assignments (the "A"s) is finite, then the fixed point is reached in a finite number of steps, meaning that execution terminates.

Final State A *final state* is the least fixed point thus defined—we note it "v_0^*" in the following. The outputs of the program are the values to be found, in state v_0^*, in the output variables.

3.3 Semantics with the Directives

Now directives are addressed. This implies that we now take into account the fact that the program is meant to run on a parallel machine with distributed memory. The data (of the data fields) are to be distributed onto the memories attached to the processors.

A set *Proc* of processors is considered. The model considers that each value from a data field has some *representatives* mapped onto the processors. The location of the representatives is fixed at compile time, and is constrained by the programmer by means of directives. A (compile-time) choice for the location of the representatives is modeled by a function

$$\boxed{\ell : \mathit{VarName} \times \mathit{Index} \to \mathcal{P}(\mathit{Proc})} \ ,$$

where $\mathcal{P}(\mathit{Proc})$ is the set of subsets of *Proc*, and $\ell(X, i)$ is the set of processors where the value $v(X, i)$ has a representative, when computed (we use the so-called "Owner Computes Rule"). We consider that one datum can have several representatives—on several processors—because duplicating some computations can serve to eliminate some inter-processor communications, which in some cases can decrease execution time.

Transitions are the transitions of a "distributed" state. The distribution of a state v through a function ℓ is the distributed state $v \triangle \ell : \mathit{VarName} \times \mathit{Index} \to (\mathit{Val}_\perp \times \mathcal{P}(\mathit{Proc}))$, with $(v \triangle \ell)(X, i) = (v(X, i), \ell(X, i))$. This is extended to programs: program $\psi \triangle \ell$ operates on states distributed through ℓ.

Every ℓ can not go with any ψ and initial state v_0, however: we have to make sure that locations are booked for the data produced by the program. That is, one must have: $v_0^*(X, i) \neq \perp \implies \ell(X, i) \neq \emptyset$. The problem of booking enough space is a compilation problem which has been studied elsewhere (see, e.g., [14] for the affine case, [16] and [21] for the general case), and we address this issue in the present paper.

The important point is that we consider that communications are generated by the distributed execution. If $v(X, i) = \perp$ and $\psi(v)(X, i) \neq \perp$, then on every $p \in \ell(X, i)$, the computation of $v(X, i)$ must have occurred (following the Owner Computes Rule), requiring communication of a representative of each of the needed argument values, coming from other processors if there does not exist a representative on p.

Let us give an example. Let a program ψ be distributed following function ℓ. Let in this program the computation of the value of X at index i require the value of Y at index j. Suppose this value of Y at j is available for the first time, during execution, at state $(\psi \triangle \ell)^n (v_0 \triangle \ell)$, for some $n \in \mathbb{N}$—that is, $\psi^n(v_0)(Y, j) \neq \perp$—, and that all other argument values for the computation of the value of X at i are already available. And let $p \in \ell(X, i)$. Then the transition from state $(\psi \triangle \ell)^n (v_0 \triangle \ell)$ to state $(\psi \triangle \ell)^{n+1} (v_0 \triangle \ell)$ implies the computation of the value of X at i, on processor p. If $p \in \ell(Y, j)$, a representative of the value of Y at j is available on this processor, and no communication is required. Otherwize, the value must be fetched from another processor $p' \in \ell(Y, j)$, and a communication is generated to this effect. Thus, the total number of communications generated by the dependency from (X, i) to (Y, j) is $\mathrm{card}(\ell(X, i) \setminus \ell(Y, j))$.

3.3.1 Semantics of Directives

The aim is to minimize the number of inter-processor communications, because they take time, and having too many of them can seriously affect total execution time. The ideal would be to have, for every dependency from a (X, i) to a (Y, j) : $\mathrm{card}(\ell(X, i) \setminus \ell(Y, j)) = 0$. In order to help the compiler to minimize communications, the programmer can help it by giving, through the means of directives, some constraints on the distribution of data, which play the role of hints.
A directive (X, A, Y, g) (let us call it d) is said to be *satisfied* by function ℓ ($\ell \models d$) if:

$$\forall i \in A,\ \ell(X, i) = \ell(Y, g(i)) \ .$$

By extension, a distributed program $\psi \triangle \ell$ satisfies d ($\psi \triangle \ell \models d$) if and only if $\ell \models d$.

Now we want to establish the semantics of such a directive, when the compiler is allowed to not satisfy it. We do this with the notion of a *policy* concerning how directives must be taken into account by the compiler. Below we present three such policies. The first is aimed at being alike to the policy which is followed by HPF compilers—though this is mainly guessing, since this policy is written nowhere for HPF. The second is more elaborate, adding an exception to the first one. The third presents yet another policy, which could be better. In the three cases, the principle is to define the set of allowed implementations, and the preference over these allowed implentations: the allowed implementations are defined by the allowed set of models for the program, and the preference is defined by a preorder relation over these and to define an order relation on this set, which models the preference of the programmer among the possible implementations.

Semantics *à la* HPF First we define the semantics of directives for $\mathcal{D}$, with a policy which is meant to reflect that of HPF concerning directives. That is, the directives are not mandatory. We however assume that the programmer wishes his directives to be taken into account if possible, and we also assume that the implementer of the compiler feels commited to act accordingly.

Than this kind of non-compulsiveness of the directives is modeled as follow. The set of possible implementations for a program ψ is the set $\mathcal{E}$ of (execution chains for) all programs $\psi \triangle \ell$, for *all* possible distributions ℓ of the data. The preference of the programmer is expressed as an order $\preccurlyeq_D$ on these possible implementations. Since the programmer prefers its directives to be taken into account, we set—where D is the set of directives of program ψ:

$$\psi\triangle\ell \preccurlyeq_D \psi\triangle\ell' \iff (\forall d \in D \;:\; \psi\triangle\ell \models d \implies \psi\triangle\ell' \models d) \ .$$

That is, an implementation $\psi\triangle\ell'$ is preferred to an implementation $\psi\triangle\ell$ if it satisfies *at least* all directives that $\psi\triangle\ell$ satisfies. In short, if $\psi\triangle\ell'$ satisfies at least one more directive than $\psi\triangle\ell$, the programmer is supposed to be happier. The compiler then should act accordingly. That is, if it can have the choice between implementing $\psi\triangle\ell$ and $\psi\triangle\ell'$, he should choose implementing $\psi\triangle\ell'$. Thus, in the end, if it does not implement all directives, the reason is only that he does not know how to do it in the current state of the art in compiling.

Notice that we did not choose the formulation

$$\begin{gathered}\psi\triangle\ell \preccurlyeq_D \psi\triangle\ell' \\ \iff \\ \mathrm{card}(\{d \in D \mid \psi\triangle\ell \models d\}) \;\leq\; \mathrm{card}(\{d \in D \mid \psi\triangle\ell' \models d\}) \ .\end{gathered}$$

This is because the order relation must reflect the preference of the programmer, and in practice the directives do not all have the same potential impact on minimizing communications. In this view the first formulation is better because is does not establish equality of any two directives in terms of programmer preference.

A More Elaborate Policy We now show how to express the following policy: the programmer prefers its directives to be satisfied, but will gladly allow the compiler and exception if the compiler knows that it is a bad idea, and knows a better implementation.

Let us first define the following auxiliary preference relation $\preccurlyeq_{\mathrm{nbComm}}$, on the set $\mathcal{E}$ of allowed implementations:

$$\begin{gathered}\psi\triangle\ell \preccurlyeq_{\mathrm{nbComm}} \psi\triangle\ell' \\ \iff \\ \mathrm{nbComm}(\psi\triangle\ell) \geq \mathrm{nbComm}(\psi\triangle\ell') \ ,\end{gathered}$$

where $\mathrm{nbComm}(\psi\triangle\ell)$ is the number of communications generated by execution of ψ when the data are distributed following ℓ.

This preference relation states that implementation $\psi\triangle\ell'$ is prefered by the programmer over implenentation $\psi\triangle\ell$ if it generates less communications. This looks as if it is exactly what the programmer wants, but it is naïve: it supposes that the compiler knows how to compare between two possible implementations in terms of the communications generated at execution. Such an ability is not realistic in the current state of the art in static analysis. Moreover this preference relation does not mention the directives—which is natural since it does not need them. In fact directives are needed because $\preccurlyeq_{\mathrm{nbComm}}$ cannot (yet ?) be implemented as such.

Now we define the new policy with the following order relation $\preccurlyeq_{D,\text{nbComm}}$ on the set $\mathcal{E}$ of allowed implementations:

$$\psi\triangle\ell \preccurlyeq_{D,\text{nbComm}} \psi\triangle\ell' \\ \Longleftrightarrow \\ \psi\triangle\ell \preccurlyeq_D \psi\triangle\ell' \ \vee\ \psi\triangle\ell \preccurlyeq_{\text{nbComm}} \psi\triangle\ell' \ ,$$

This preference relation is interesting because it takes into account that the compiler may, with some couples $(\psi\triangle\ell, \psi\triangle\ell')$, be able to verify that adding an additional directive would be detrimental to the total number of generated communications. And this ability of the compiler might grow with time, as state of the art evolves.

A Better Policy We now show how to formulate the following policy: directives are mandatory *unless* the compiler has a better solution than the solution proposed by the directives.

Notice that if a compiler follows this policy and does not implement a directive, this means it can garantee that it is providing a better solution—whether it would have been able to effectively implement this directive or not.

The fact that the directives are mandatory by default implies a behavior of the compiler which differs from the behavior of HPF compilers: if the compiler does not know how to satisfy one of the directives, then it must ask the programmer to modify or suppress this directive. Thus, every directive which is present in a code which is compiled is one that the compiler knows how to handle—even if it is for determining it can do better.

The point in this policy is that the evolution of compilers is foreseen, as with the previous policy: a directive which is used by the current compiler can be ignored by a more sophisticated compiler in the future which knows how to do better without.

In order to define formally this policy, we must define the set of authorized implementationsm and the preference relation on it. Clearly the set of authorized implementations is a strict subset of the set $\mathcal{E}$ which was used for $\preccurlyeq_D$ above, where the compiler was allowed for example to implement zero directives if it didn't know how to implement any of them. It can be constructed in two steps. First, we must include the set $\mathcal{A}$ of all implementations which satisfy *every* directive in D. Then to $\mathcal{A}$ we can add the set $\mathcal{B}$ of all "better implementations". A "better implementation" $\psi\triangle\ell$ is an implementation which does not satisfy a certain subset C of D, but is such that there exists a $\psi\triangle\ell'$ in $\mathcal{A}$ with $\text{nbComm}(\psi\triangle\ell) \leq \text{nbComm}(\psi\triangle\ell')$. Now we can define the set of authorized implementations to be $\mathcal{E}' = \mathcal{A} \cup \mathcal{B}$.

The preference relation over $\mathcal{E}'$ is defined as:

$$\preccurlyeq_{\text{bet}} = \preccurlyeq_{\text{nbComm}|_{\mathcal{E}'}} \ .$$

That is, when having the choice between two implementations $\psi\triangle\ell$ and $\psi\triangle\ell'$, the compiler must compare the number of communications generated by both, and choose the best. It is placed in this position in two cases. The first if when both implementations are in $\mathcal{A}$. This is the case when the directives are not involved in the comparison. The second case is when $\psi\triangle\ell$ is in $\mathcal{A}$ and $\psi\triangle\ell'$ is in $\mathcal{B}$. This is when the compiler is in the process of looking for a possible alternative to the satisfaction of one or several directives.

4 Related Works

4.1 Semantics of Data Parallelism

Our approach to modeling distribution is inspired by previous work [6]—in which however distribution was not yet clearly separated from the program itself. Other formal models for data parallelism have been proposed that include the notion of data distribution: the language $\mathcal{L}$ [3], concrete data structures [12], and PEI [22]. However they do not include the notion of directive.

4.2 Directives

While directives for parallelization are introduced as an ad hoc solution when immediate benefits are looked for, as with the HPF and OpenMP initiatives, they have benefited from more fundamental studies in the context of functional programming. For example Tringer et al. present the elaborate concept of programming strategies for evaluation of the program [19]: the idea, as for directives, is that the evaluation strategy should be distinguished from the algorithm itself. However transposition to imperative contexts like Fortran—which is the main kind of input for parallelization of legacy code—is not obvious. Also, these works on functional languages do not consider non-mandatory directives or strategies.

4.3 Preference

For modeling preference, we were inspired by the semantics of *deontic logic* [13]. Deontic logic is about *obligations* (in the legal acception, where the principal characteristic of an obligation is that it will surely be violated). As a modal semantics, its semantics naturally uses the notion of "possible words": and preference is modeled as an order relation on these worlds.

There exists a domain where preference on the execution of programs has already been considered: *preference logic programming* (PLP) [8]. Here, preference is also modeled as an order relation on possible worlds. However the coding is not the same as ours: in PLP, a possible world w satisfies or not a predicate which express the preference, and the order relation is defined by the constraints given by these predicates. In our modelling, by contrast, a model does not satisfy a preference: the order relation is defined externally—even if it refers to characteristics of the models.

The advantage is that it is possible then to consider several order relations on a same set of models, and to combine then, as we did above.

5 Conclusion

Using directives allows for a fair compromise between high-level programming and control of the execution model. In this article we aimed at showing that directives might indeed be a good idea. We have not sought to reveal special programming possibilities with directives; but we have looked at them from the important point of view of their "non-mandatory" status, and we have given a formalization of this status. We have argued that non-mandatory directives are interesting in that they can be "just a bit ahead of the current compilers technology", and the compiler can be capable of handling them later; and also when some directives become less relevant—i.e., the compiler becomes capable of doing without them, they are not an hindrance for the application (an example of this is given by the C "`register`" directive).

We have also emphasized two basic principles for designing a directive language:

1. it should be easy to add directives into legacy code for migrating it—that is, for adapting it to a new execution model. HPF is a good example for this, for migrating a sequential code onto a parallel machine.
2. Symmetrically, it should be easy to strip directives off for having the initial legacy code back, for writing new directives (for a new migration).

Beyond parallel programming for parallel machines, we believe directives can be very useful when addressing the challenges given by two emerging disciplines:

- programming for real-time and embedded systems: those execution environnement provide hard constraints for the programmer. Real-time constraints are naturally expressed with directives. In the context of multi-processor platforms, directives for distributing data and/or threads can be handy. Symmetrically, directives can also serve to express those constraints from the platform which are taken into account in the programming task. Thus directives can be a medium for doing codesign. We have begun work in this direction.
- Programming for the computational grid. For example, migrating MPI code from a parallel machine to the GRID might be done with the help of directives.

References

[1] *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999. Edited by Ian Foster and Carl Kesselman.

[2] *OpenMP Fortran Application Program Interface*, November 2000. Version 2.0.

[3] Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: Semantics and implementation. *Future Generation Computer Systems*, 8:363–378, 1992.

[4] Fabien Coelho, Cécile Germain, and Jean-Louis Pazat. *State of the Art in Compiling HPF*, volume LNCS 1132, pages 104–133. Springer-Verlag, May 1996.

[5] HPF Forum. *High Performance Fortran Language Specification*, 2.0 edition, January 1997.

[6] Ph. Gerner and E. Violard. A theoretical framework of data parallelism and its operational semantics. In Arndt Bode, Thomas Ludwig, and Roland Wismüller, editors, *Proceedings of Euro-Par 2000 – Parallel Processing,* LNCS 1900, pages 668–677, Munich, September 2000. Springer Verlag.

[7] Sergei Gorlatch. Programming with divide-and-conquer skeletons: An application to FFT. *J. Supercomputing*, 12(1).

[8] Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Preference logic programming. In *International Conference on Logic Programming*, pages 731–745, 1995.

[9] William Groppa, Ewing Lusk, and Anthony Skjellum. *Using MPI.* Scientific and Engineering Computation. MIT Press, 2nd edition, November 1999.

[10] C. A. Gunter and D.S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. Elsevier Science Publishers, 1990.

[11] G. Hains and F. Loulergue. *Functional Bulk Synchronous Parallel Programming using the BSMLlib Library*, chapter 11, pages 165–178. Advances in Computation: Theory and Practice. Nova Science Books and Journals, 2002.

[12] G. Hains, F. Loulergue, and J. Mullins. Concrete data structures and functional parallel programming. *Theoretical Computer Science*, 258(1–2):233–267, April 2001.

[13] Sven Ove Hansson. *The Structure of Values and Norms.* Cambridge University Press, 2001.

[14] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. Technical Report 1997/8, PRiSM, Versailles University, France, March 1997.

[15] Björn Lisper. *Data Parallelism and Functional Programming*, volume LNCS 1132, pages 220–251. Springer-Verlag, May 1996.

[16] Björn Lisper and Jean-François Collard. Extent analysis of data fields. In *Static Analysis Symposium SAS'94*, pages 208–222, September 1994.

[17] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, January 1994.

[18] Thinking Machines Corp. *C* Programming Guide*, November 1990.

[19] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithms + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[20] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[21] Eric Violard. Typechecking of PEI programs. In *LNCS 1300, Proceedings of EUROPAR'97*, pages 521–529. Springer-Verlag, August 1997.

[22] Eric Violard and Guy-René Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.

[23] Doran K. Wilde. The alpha language. Technical Report 827, IRISA, Rennes University, France, May 1994.