HAL
open science

The Data-Parallel Programming Model: a Semantic Perspective (Final Version)

Luc Bougé

To cite this version:

Luc Bougé. The Data-Parallel Programming Model: a Semantic Perspective (Final Version). [Research Report] RR-3044, INRIA. 1996. inria-00073648

HAL Id: inria-00073648
https://inria.hal.science/inria-00073648v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Data-Parallel Programming Model: a Semantic Perspective (Final Version)

Luc Bougé

N° 3044

Novembre 1996

THÈME 1

Rapport de recherche

ISSN 0249-6399

The Data-Parallel Programming Model: a Semantic Perspective (Final Version) *

Luc Bougé†

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n° 3044 — Novembre 1996 — 16 pages

Abstract: We provide a short introduction to the data-parallel programming model. We argue that parallel computing often makes little distinction between the execution model and the programming model. This results in poor programming and low portability. Using the "GOTO considered harmful" analogy, we show that data parallelism can be seen as a way out of this difficulty. We show that important aspects of the data-parallel model were already present in earlier approaches to parallel programming, and demonstrate that the data-parallel programming model can be characterized by a small number of concepts with simple semantics.

Key-words: Concurrent Programming, Specifying and Verifying and Reasoning about Programs, Semantics of Programming Languages, Data Parallelism, Task Parallelism.

(Résumé : tsvp)

This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism C^3/PRS and Department of Defense DRET Contract 91/1180.

* An early version of this paper has been published in French in *Technique et science informatiques (TSI)*, Special issue on Data-Parallel Languages, Vol. 12, No. 5, 1993, p. 541–562. This paper has been published in "The Data Parallel Programming Model", G.-R. Perrin, A. Darte (Eds.), Lecture Notes in Computer Science Tutorial Series, 1996, p. 4–26. Please mention this reference in citations.

† Author's address: LIP, URA CNRS 1398, ENS Lyon, 46 allée d'Italie, F-69364 Lyon Cedex 07, France. E-mail: Luc.Bouge@lip.ens-lyon.fr.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 76 61 52 00 – Télécopie : (33) 76 61 52 52

Le modèle de programmation à parallélisme de données : une perspective sémantique

Résumé : Nous présentons une courte introduction au modèle de programmation à parallélisme de données. Nous montrons que la distinction entre modèle d'exécution et modèle de programmation a souvent été mal comprise dans le calcul parallèle. Ceci conduit à une programmation de faible qualité et peu portable. À partir de l'analogie "Les GOTO sont-ils nuisibles ?", nous montrons que le parallélisme de données peut être une solution à cette difficulté. Il apparaît que des aspects importants de ce modèle étaient déjà présents dans la programmation parallèle classique. Nous montrons que le modèle de programmation à parallélisme de données peut être caractérisé par un petit nombre de concepts sémantiques simples.

Mots-clé : Programmation parallèle, spécification et validation de programmes, sémantique des langages de programmation, parallélisme de données, parallélisme de tâches.

1 Introduction: Execution Model vs. programming Model

Any abstract approach towards computing has to deal with two different kinds of objects: *computers*, which compute, and *programs*, which are the medium used to explain to the computer what to compute. Computers differ according to their architectures, which can be classified according to the *execution model* they implement. Similarly, programs differ according to the language in which they are expressed. Languages can be classified according to the *programming model* they reflect. In terms of equations,

$$Computing = Computer + Program$$

and

$$Abstract\ Computing\ Model = Execution\ Model + Programming\ Model$$

Sequential Programming

In the world of sequential computing, much work has been done to understand and take advantage of this dichotomy. The usual *execution model* is the Von Neumann model: contents of memory locations are repeatedly loaded into registers, combined and stored back by the processor. On the other hand, many *programming models* have been proposed. Early programming languages such as Fortran derive from a model very close to the Von Neumann execution model. The Pascal language introduced a more abstract "structured programming" model. Programming models derived from non-operational approaches have definitely proven fruitful: logic programming (e.g., Prolog), functional programming (e.g., Lisp, FP, Scheme, ML, Haskell), object-oriented programming (e.g., C++ and Eiffel).

In all cases, a *compiler* (sometimes an interpreter) has to map the statements of the programming model into the execution model while preserving the semantics and maximizing efficiency. Observe by the way that the distinction between programming and execution models is relative to a design level: for instance, the designers of RISC processor microcode would probably consider the Von Neumann model as the programming model (load/store assembly language) and the internal superscalar pipelined machinery as the execution model. Looking back over the 30 last years, one can observe a continuous trend towards more and more abstract programming models. As the gap between programming and execution models becomes larger, compiler technology becomes a more and more central concern for designers.

Parallel Programming

The situation in the world of parallel computing is less advanced. Much effort has been expended over more than 30 years to build more and more powerful computers, and many different *execution models* have been explored and compared. Flynn [14] proposed to classify these models into four categories. They differ in the number, *S*ingle or *M*ultiple, of *I*nstruction streams and *D*ata streams which are (conceptually) active in each point of the computation: *SISD* (PCs, workstations, etc.), *MISD* (vector processors, pipeline architecture, systolic arrays to some extent, etc.), *SIMD* (array processors, so-called massively parallel computers, etc.) and *MIMD* (multicomputers, multiprocessors, networks of processors, Transputer-based architectures, etc.)

In comparison, little effort has been devoted to developing the right part of the above equation, at least in the area of numerically intensive, scientific parallel computing. In fact, looking at commercial announcements of recent years, it seems that (so-called!) massively parallel machines have developed so fast that little time was left to design suitable languages. The general tendency has thus been to design languages by "expanding" the architecture, that is, to define one type of statement to control each hardware feature. This tendency has resulted in programming models that form a subset of the the execution model. They are isomorphically derived from the execution model or, more usually, a part of it.

$$Programming\ Model \subseteq Execution\ Model$$

A striking example of this process is the Transputer architecture and the Occam programming language. Running two Transputers in parallel is expressed by the `PAR` Occam construct. The synchronous communication protocol over hardware Transputer links is expressed by the `chan!` and `chan?` Occam communication constructs. Language restrictions on communication guards (no output guards) are precisely tuned to reflect the behavior of the communication protocol.

Another example is the Connection Machine CM-2 architecture [32] and the (new) C* programming language [30] . The SIMD control of the execution model and the virtual processor facility of the CM-2 sequencer are reflected, respectively, in the semantics of the `where` construct and the `shape` definition. These concepts were even more crucial for the C/ParIS and the *Lisp languages, which were so closely connected to the CM-2 architecture that they could not survive the advent of the CM-5 [33]!

Towards Abstract Programming Models

Many researchers have already noted this imbalance and its bad effects. How to design more abstract models for parallel programming? One possible solution is to import abstract programming models from the sequential world. One can extend them with additional parallel features as needed and map these models onto as many parallel execution models as possible. This approach has been explored for high-level sequential programming models such as logic programming (e.g. Parallel Prolog, etc.), functional programming (e.g. the many varieties of parallel and/or concurrent Lisp, ML, Haskell and others), object-oriented programming (e.g., parallel and distributed extensions of object-oriented languages such as CC++, pC++, etc.). Due to the high level of abstraction, designing efficient implementations for these models is a very difficult challenge, especially when performance is the main goal as in numerically intensive parallel computing. A notable exception is object-oriented languages (see [16] for instance), where the concept of *class* provides a natural way of smoothly introducing and manipulating parallel objects. Yet, it remains a promising research direction for the future, especially because only high-level languages can support powerful and reliable software engineering design methods.

A similar approach can also be applied for less abstract sequential programming models such as Pascal or Dijkstra's Guarded Command language. In his Communicating Sequential Processes (CSP) programming model [20], Hoare simply composes ordinary Guarded Commands programs with an *explicit* parallelism control operator. This programming model is thus a direct abstraction of the MIMD execution model. This approach has proven fruitful, as it has been the basis of many parallel languages such as Occam, the task model of Ada and communication routine libraries used on commercially available MIMD parallel machines (PVM, MPI, etc.). Yet, it turns out that programming with such an explicit parallel control is difficult and error-prone, especially when a large number of interacting processes come into play. Furthermore, this programming model is poorly portable to non-MIMD parallel architectures (SIMD and vector processing architectures, for instance).

With the recent definition of Fortran 90, which extends the sequential Fortran 77 language with parallel arrays, and the successful effort towards a unified High Performance Fortran [15], a promising intermediate solution is now emerging. It is often referred to as the *data-parallel* programming model, to emphasize that control remains sequential whereas data access is done in parallel.

The goal of this presentation is to analyze the semantic nature of this model and to discuss its relationship with the CSP model. We show that the *data-parallel* model of HPF or C* can be seen as a structured version of the *control-parallel* model of CSP. Moreover, we show that the data-parallel model can be characterized by a small number of semantic concepts.

2 From Control Parallelism to Data Parallelism

It is useful at this point to look back at what happened in the 60's in sequential computing. At that time, machine architectures were designed according to the Von Neumann *execution model*: a processor repeatedly fetches an instruction at the memory address stored in the program counter (PC) register, executes it, and finally adjusts the contents of the PC register for the next cycle. In this execution model, the crucial concept is the ability for the processor to store an arbitrary value into the PC register, that is, to `JUMP`. The *programming model* used to control such architectures was directly derived from this execution model: the `JUMP` hardware facility is simply reflected by the `GOTO` instruction in the Fortran language.

The famous question of Dijkstra

"`GOTO` considered harmful?"

stated in [9] was the starting point of a long and fruitful effort to make the distinction between the programming model and the execution model clearer. The design of the Pascal language originated in this movement. It proposed an alternative programming model based on the "structured" `while` construct instead of the low-level "unstructured" `GOTO`. Observe that this new programming construct does not change the execution model in any way.

It is interesting to assess this important step with respect to control management. In the Fortran programming model, the flow of control is *arbitrary* and *unbounded*. You can `GOTO` from anywhere to anywhere. Only *semantic* analysis can allow one to discover structures in the control flow, or determine the bounds of a `GOTO`. In contrast, the control flow of the Pascal programming model is *structured* and *bounded* (more precisely, nested). Semantic information is made explicit in the *syntax*, at the price of some restriction in expressiveness.

Let us illustrate these issues with the following example. It is easy to identify an iteration controlled by a test in Pascal, as it appears *explicitly* in the program text:

$$\texttt{while}\ x < 0\ \texttt{do}\ x := x - 1$$

In contrast, a more careful analysis is required to determine that the following Fortran program

```
1     IF (x .GE. 0) GOTO 99
      x = x - 1
      GOTO 1
99    CONTINUE
```

is such an iteration, whereas

```
1     IF (x .GE. 0) GOTO 99
      x = x - 1
      GOTO 99
99    CONTINUE
```

boils down to a mere alternative `if` $x < 0$ `then` $x := x - 1$, and

```
1     IF (x .GE. 0) GOTO 1
      x = x - 1
      GOTO 1
99    CONTINUE
```

is an infinite loop. In this respect, Pascal can be qualified as "more informative" than Fortran.

On the other hand, it is easy to translate a structured construct of Pascal, such as a `while`, `repeat` or `for` loop, into a complex assembly of `GOTO`'s. Besides the marginal problem of label generation, this translation involves *local* manipulations only (it is called *compositional*). In contrast, it is much more difficult to translate a program written with `GOTO`'s into a program written with `while` loops only. (It is a classical and pleasant exercise to prove that any program can be transformed into a program made of only *one* `while` loop, indeed, and we leave it to the interested reader.) A *global* manipulation of the program text is needed. In this respect, Fortran can be qualified as "more expressive" than Pascal.

This discussion is summed up in Figure 1.

Programming model	`GOTO`	`while`
Language	Fortran	Pascal
Control flow	arbitrary unbounded	structured bounded (nested)
Structure information	semantic	syntactic
Expressiveness	+	−
Informativeness	−	+

Figure 1: `GOTO` considered harmful?

Our thesis is that a similar phenomenon is now occurring in the area of parallel computing. In the CSP programming model as seen in the Occam language, parallel composition applies to arbitrary programs. The asynchrony between them is *arbitrary* and *unbounded*. The resulting behavior is *non-deterministic* in general. Any information concerning synchronization, boundedness of scopes and determinism has to be derived from a *semantic* study of the program. This analysis is, in general, very complex.

Following the Fortran/Pascal seminal analogy, the challenge is thus to design a *programming model* in which parallel composition is *structured*, *bounded* in scope and *deterministic*, and such that all this information is made available in the *syntax*. Paraphrasing Dijkstra, we dare to ask the question

"`PAR` considered harmful?"

Consider an Occam program of the form

$$(P_1; P_2) \parallel (Q_1; Q_2)$$

where processes P_i and Q_j may be very complex. Because of the unbounded asynchrony, it may well happen that the left-hand process already executes P_2 while the right-hand one still completes Q_1. In some situation, this may be a bad thing: for instance, if Q_1 produces a value required by Q_2, and P_2 modifies this value; or if some message sent by P_2 is caught by Q_1 instead of being delivered to Q_2 as intended.

Of course, some semantic information based on the necessary synchronizations between P_1 and Q_1 may help us to rule out this case. We may for instance prove that the left-hand process cannot enter P_2 before the right-hand one has left Q_1. The challenge is here to make this information *explicit* in the syntax. Ultimately, this amounts to writing the program

$$(P_1 \parallel Q_1); (P_2 \parallel Q_2)$$

Such a syntax is more restrictive than the original one, but it is more informative. It is the compiler's job to detect whether a program written in this restricted but informative syntax can be optimized into a program written in the unrestricted syntax. We sum up this discussion in Figure 2.

Programming model	`PAR`	?
Language	Occam	?
Control flow	arbitrary unbounded non-deterministic	structured bounded deterministic
Paradigm	$(;) \parallel (;)$ `PAR` of `SEQ`	$(\parallel); (\parallel)$ `SEQ` of `PAR`
Structure information	semantic	syntactic
Expressiveness	+	−
Informativeness	−	+

Figure 2: `PAR` considered harmful?

In the left-hand model, a program is described as a `PAR`allel composition of `SEQ`uential actions. Note that the name of Hoare's CSP language [20] stands precisely for *C*ommunicating *S*equential *P*rocesses. We can therefore call this model *control-parallel*. To illustrate the application of this model, consider three arrays $A[1..N]$, $B[1..N]$, $C[1..N]$. A typical control-parallel program to compute the component-wise sum would be

$$(A[1] := B[1] + C[1]) \parallel \ldots \parallel (A[N] := B[N] + C[N])$$

In the right-hand model, a program is viewed as a `SEQ`uential composition of actions on `PAR`allel data structures. We can therefore call this model *data-parallel*. A typical program for the same problem would be the following, using the parallel extent notation ':' of Perrot's Actus language [28]:

$$A[1:N] := B[1:N] + C[1:N]$$

or, more concisely,

$$A := B + C$$

The pros and cons of the control-parallel model versus the data-parallel model can be discussed along the same lines as for the `GOTO` versus the `while` programming models. Aspects of program design, validation, transformation, optimization and maintenance can be considered in turn. The common baseline of the discussion eventually reduces to whether to choose to emphasize flexibility or structure, expressivity or safety, short term decisions or long term decisions, etc.

Moreover, as the control-parallel model is a straightforward abstraction of the execution model, it is useful to consider the relationships between the two models in term of compilation and optimization. It is much easier and safer for the *programmer* to work in the structured, bounded and deterministic data-parallel programming model. On the other hand, it is much easier and safer for the *compiler* to work with a program expressed in the control-parallel model. The discussion is thus closely dependent on the balance between the work handled by the compiler and that left to the programmer. These issues are illustrated in Figure 3, reprinted from [33, p. 28]

	Structured Programming	Data Parallel Programming
Primitive Computation	Assignment statements	local computations

Basic Patterns	`BEGIN...END`
	`IF...THEN...ELSE...`
	`WHILE...DO...`
		replication
		reduction
		permutation
		parallel prefix

Common Compound Patterns	`DO` loops `CASE` statements `REPEAT ...UNTIL ...`	regular grids, stencils fetch-with-add sorting, fast transforms
Low-Level Mechanisms	`GOTO`	message passing

Figure 3: Structuring programs, reprinted from the CM 5 Technical Summary [33]

3 Data Parallelism as a Programming Model

Data parallelism can thus be thought as a "Paradigm Revolution" with respect to control parallelism. The essential aspect of this revolution is that the "`PAR` of `SEQ`" execution model is expressed by a "`SEQ` of `PAR`" programming model! This clear distinction between programming and execution models is the source of an enriched vision of programs. In fact, this radical breakthrough can already be seen in several areas of parallel programming. Below, we shortly describe two interesting examples where data parallelism arises as the *natural* programming model associated with a control-parallel execution model.

3.1 Macroscopic and Microscopic Viewpoints

The data-parallel programming model provides two alternative viewpoints for a single program [29]. The existence of these two complementary viewpoints is probably one of the reasons for the growing interest in the data-parallel model: the intuitions and the techniques rooted in each viewpoint can be combined to design, manipulate, transform, validate and optimize data-parallel programs, providing the programmer with a variety of powerful tools.

3.1.1 The Macroscopic Viewpoint

The *macroscopic* viewpoint corresponds to the high-level vision of the user. A program is here viewed as a sequence of complex operations applied on data structures with parallel access ("`SEQ` of `PAR`"). Basically, this is thus a centralized, sequential model very close to the Von Neumann model, in which the usual scalar data are replaced with vectors, that is, arrays with parallel access or *collections*. Seen from the processor, a memory location does not store a scalar value any more, but rather a parallel array of values. This viewpoint can thus be called *Processor of arrays*.

In such an approach, the notion of a general communication amounts to a global rearrangement of the elements of an array. This rearrangement is specified by an auxiliary array of coordinates. In the case of a rearrangement controlled by the emitters (a "scatter", usually called a `send` communication in data-parallel languages), each component of this auxiliary array stores the coordinates of the destination component. Observe that it is then necessary to specify the intended behavior in the case of conflicts on the destination components. In the case of a rearrangement controlled by the receivers (a "gather", usually called a `get` communication in data-parallel languages), each component of this auxiliary array stores the coordinates of the component whose value is requested. Observe that no conflict is possible here.

A subset of the many possible rearrangements can be specified in a more concise way. These *regular* communication schemes often support optimized implementations, depending on the specific features of the target architecture (especially of its communication network). Regular communications include *shifts* along a direction, toroidal *rotations* and prefix operations usually called *scans*. Two special cases of scans are of dramatic importance: *reduction*, which combines several values into one variable (a scatter, `send`-like operation) and its dual operation *broadcast*, which replicates one value into many variables (a gather, `get`-like operation).

Many classical algorithms can in fact be expressed as a combination of these regular communications, even in apparently irregular areas. Striking examples of this approach applied to linked lists can be found in a famous

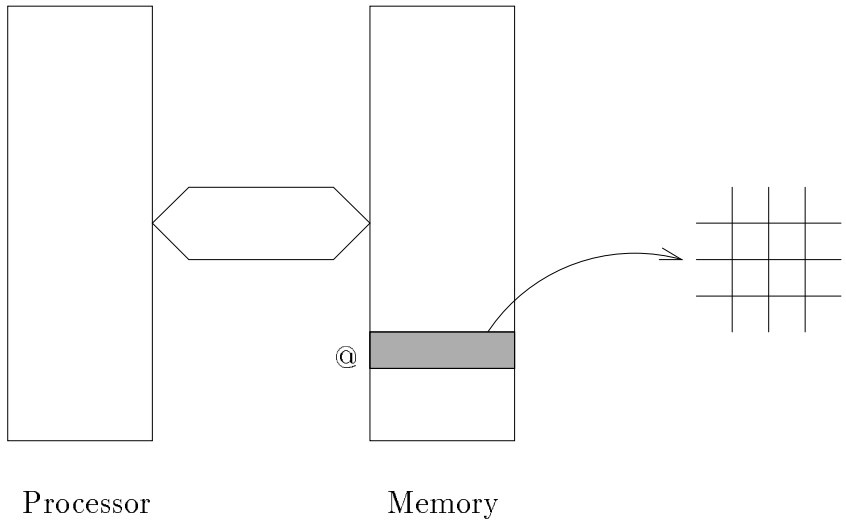

Figure 4: The macroscopic viewpoint: a processor of arrays

paper of Steele and Hillis [19]. In fact, a complete methodology of programming can be developed with these only communication primitives [3].

The macroscopic viewpoint is the basis for data-parallel array manipulation languages. We must of course cite their (glorious!) ancestors APL [22] and Actus [28]. The macroscopic intuition is strongly emphasized in the definition of the Fortran 90 language and of HPF. In particular, these languages define specific communication instructions for each of the regular communication schemes above.

The next step forward in the macroscopic direction is to view an array with parallel access as an abstract function from some domain of indices to some domain of values. This was already implicit in the work Steele and Hillis about CMLisp [29]. The reader is referred to the contribution of Bjørn Lisper [24] in this volume for an extensive presentation of data-parallel programming seen as a special case of functional programming.

3.1.2 The Microscopic Viewpoint

The *microscopic* viewpoint corresponds to the low-level vision of the compiler. The program is seen as the parallel composition of actions on local data ("`PAR` of `SEQ`"), with an additional synchronization mechanism to keep these independent threads of actions synchronous. This approach can thus be called *Array of processors*.

On SIMD architectures, this synchronization mechanism is implemented at the hardware level. It consists of a global sequencer, distinct from the processing elements (PE). Its function is to broadcast the current common instruction to be executed by the PE on their local data at each cycle.

On MIMD architectures, this synchronization mechanism is implemented at the software level. It turns out that data-parallel programs are often self-synchronizing: nothing needs to be done in this case, and the synchronization mechanism is void (see [1] for a detailed study of such a property). If the program does not satisfy this property, or if the compiler is not clever enough to detect it, then it has to insert, between each computation phase, a call to a synchronization routine. Such a routine implements a global synchronization protocol based on a general exchange of messages, and it may be very inefficient: for example, this requires of the order of one millisecond on an iPSC 2 with 128 processors. Therefore, several recent MIMD architectures explicitly designed to support data-parallel languages (HPF and C*, usually) have been equipped with specific hardware mechanism to optimize this function. It is in particular the case with the CM-5 [33] of Thinking Machine and the Paragon [11, 21] of Intel. MIMD architectures with an optimized global synchronization hardware facility are sometimes (somewhat improperly, see below) called SPMD (*Single Program Multiple Data*), as they are specifically designed to execute data-parallel languages efficiently. A better nickname would probably be "C*-machines" or "HPF-machines", very much like the "Lisp-machines" of the 80's.

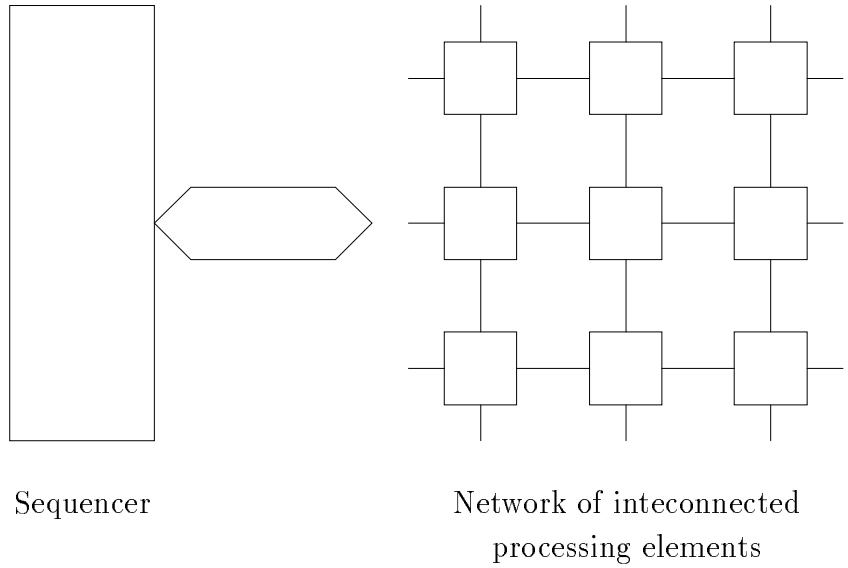

Figure 5: The microscopic viewpoint: an array of processors

In contrast to the general MIMD model, all communication phases are synchronous in data-parallel programs. On entering a communication phase, all processors are ready to send, and then ready to receive all available messages. The communication phase ends only when all sent messages have been received. There is thus no deadlock to threaten as in the CSP point-to-point communication model. In the microscopic viewpoint of data-parallel communication, the basic mode is the all-to-all message exchange. In the emitter-controlled communication flavor (scatter, `send`), each processor specifies the destination address for its message. In the receiver-controlled communication flavor (gather, `get`), each processor specifies the source address for its message. These addresses are specified with an auxiliary parallel variable. As was the case for the macroscopic viewpoint, a subset of *regular* communication schemes can be distinguished and given an optimized implementation.

The microscopic viewpoint is specially promoted by languages such as C*, MPL and HyperC. Usually, these languages do not provide any specific syntactic construct to identify regular communication schemes. An exception is the MPL `xnet` construct, which is very closely connected to the hardware features of the underlying interconnection network. Yet, this information can in certain cases be *recognized* by the compiler from the program text, and then compiled to calls to specific optimized procedures. Compilers that can perform such recognition are often called *communication compilers*. Regular communication operations can also be specified by the programmer through directives or explicit calls to specialized communication libraries. The lack of any explicit syntactic constructs to express regular communications is probably a major weakness of these languages.

3.2 Automatic Parallelization

Several methods have been proposed for automatic parallelization of Fortran scientific programs on MIMD parallel machines with distributed memory. A number of authors have already noticed that data parallelism provides an adequate framework for this purpose, as data parallelism is, in some sense, *hidden* behind control parallelism in the methodology. We follow here the general lines of the presentation of Feautrier [12]. The reader is referred to Feautrier's contribution in this volume for further details [13].

Many studies on Fortran scientific programs have shown that the core of such programs consists of sequences and nestings of `DO`-loops with simple (e.g., affine) index dependencies. For example,

$$\begin{array}{l} \texttt{DO}\ i = 1, M \\ \quad \texttt{DO}\ j = 1, N(i) \\ \qquad A[i,j] = \ldots A[f(i,j), g(i,j)] \ldots \end{array}$$

where $N(i)$, $f(i,j)$ and $g(i,j)$ are simple functions. When parallelizing these programs, the first step is to analyze dependencies, that is to compute (symbolically) a graph made of edges $(i,j) \to (f(i,j), g(i,j))$. Then, each occurrence (i,j) is allocated on a processor $P_{u(i,j)}$. Finally, the parallelized program is generated as a **PAR**allel composition of **SEQ**uential processes, according to the intended execution model.

The parallelized program is usually of the following form:

$$P = [P_1 \parallel P_2 \parallel \ldots]$$

where each process P_u is in charge of computing variables $A[i,j]$ allocated to it. This strategy is called the *Owner Computes Rule* (OCR). Each process P_u has the following form:

$$\begin{array}{ll} P_u :: & t_u := 0 \\ & \texttt{while}\ ok_u\ \texttt{do} \\ & \quad \textit{Communication}_u \\ & \quad \textit{Local-computation}_u \\ & \quad t_u := t_u + 1 \\ & \quad \textit{Synchronization}_u \\ & \texttt{end} \end{array}$$

The local variable t_u stores the iteration number. The local variable ok_u is set to false when the computation is locally terminated: the parallelization method guarantees that this occurs at the same iteration number for all processes. In the program part $\textit{Communication}_u$, processes exchange the data required to the current iteration. New data are computed and assigned locally in $\textit{Local-computation}_u$. The $\textit{Synchronization}_u$ part implements a global synchronization protocol in order to keep everybody in phase. As we explain below, this synchronization can often be eliminated in an optimization step.

Looking once again at the general shape of the program, it is clear that the *Synchronization* part can be more simply expressed by commuting the **PAR** and the **SEQ**. Also, local variables t_u and ok_u can be merged respectively as single parallel variables t and ok. These remarks lead to the following form for the whole program. (We write C_u for $\textit{Communication}_u$ and L_u for $\textit{Local-computation}_u$.)

$$\begin{array}{l} t := 0 \\ \texttt{while}\ ok\ \texttt{do} \\ \quad (\ldots \parallel (C_u; L_u) \parallel \ldots \parallel (C_v; L_v) \parallel \ldots); \\ \quad t := t + 1 \\ \texttt{end} \end{array}$$

Observe now that L_u contains local computations only. Therefore, we can safely introduce an additional global synchronization between the C's and L's. The inner part can thus be written

$$\begin{array}{l} (\ldots \parallel C_u \parallel \ldots \parallel C_v \parallel \ldots); \\ (\ldots \parallel L_u \parallel \ldots \parallel L_v \parallel \ldots) \end{array}$$

Now, processes can test their own identity. Say it is stored in some special parallel variable (it was called *this* in the old CM-2 C*, now called Data-Parallel C [18]). There is thus no need to assume that programs for P_u and P_v have distinct syntactic forms, and we can write

$$\begin{array}{l} t := 0 \\ \texttt{while}\ ok\ \texttt{do} \\ \quad (\ldots \parallel C \parallel \ldots \parallel C \parallel \ldots); \\ \quad (\ldots \parallel L \parallel \ldots \parallel L \parallel \ldots); \\ \quad t := t + 1 \\ \texttt{end} \end{array}$$

or, more concisely,

$$t := 0; \texttt{while}\ ok\ \texttt{do}\ (C; L; t := t + 1)\ \texttt{end}$$

that is, as a typical data-parallel program!

The data-parallel "**SEQ** of **PAR**" model appears thus here as the natural framework in which this systematic parallelization can be carried out and studied. In fact, using this level of abstraction allows one to concentrate on extracting data dependencies in the programming model, without any reference to process synchronization problems introduced by the underlying execution model. The problem of how much synchronization is needed between processes to respect the semantic correctness of the program is then seen as an optimization question, which is raised at a level lower than that of the intrinsic correctness of the method.

3.3 CSP Communication-Closed Layers

A dozen years ago, Elrad and Francez [10] performed an extensive semantic study on the correctness of certain design methodologies for programs written in Hoare's CSP language [20]. This work applies to Occam programs as well. In essence, their work addressed the following problem. Consider a CSP `PAR`allel program of the form

$$(P_1; P_2) \parallel (Q_1; Q_2)$$

Under which condition is it semantically equivalent to the following `SEQ`uential program?

$$(P_1 \parallel Q_1); (P_2 \parallel Q_2)$$

In the terminology of this paper, under which condition can a control-parallel program be recast into a data-parallel form?

To get a feeling for the subtlety of the question, consider the following CSP programs:

$$P :: [c \texttt{ ? } x \rightarrow \texttt{skip} \,\square\, \texttt{true} \rightarrow \texttt{skip}]$$

$$Q :: [c \texttt{ ! } 0 \rightarrow \texttt{skip} \,\square\, \texttt{true} \rightarrow \texttt{skip}]$$

and take $P_1 = P_2 = P$, $Q_1 = Q_2 = Q$.

Intuitively, P says: "I am ready to receive a message from Q or to terminate" and Q says: "I am ready to send a message to P or to terminate" and the underlying communication protocol ensures that Q can send the message if and only if P receives it (*synchronous handshake*). It is clear that $P \parallel Q$ always terminates.

But $(P_1; P_1) \parallel (Q_1; Q_2)$ is *not* semantically equivalent to $(P_1 \parallel Q_1); (P_2 \parallel Q_2)$, as a communication between P_2 and Q_1 may occur in the former and *not* in the latter according to the the semantics of CSP!

A program $(P_1 \parallel Q_1)$ such that $(P_1; P_2) \parallel (Q_1; Q_2)$ is semantically equivalent to $(P_1 \parallel Q_1); (P_2 \parallel Q_2)$ for any program $(P_2 \parallel Q_2)$ is called by Elrad and Francez a *communication-closed layer*. As observed by the authors, designing a CSP parallel program as a sequence of communication-closed layers allows one to validate each layer in isolation, and yet to compile the entire program without any additional global synchronization. This is therefore a highly desirable property, often referred as *modularity* or *compositionality* . Observe also that adding a synchronization barrier at the end of a CSP program makes it communication-closed at the cost of an additional global communication.

4 Basic Semantic Concepts for Data Parallelism

As shown above, the data-parallel programming model can be characterized by the "`SEQ` of `PAR`" paradigm, that is, a sequential composition of actions on objects with parallel access. In this section, we shortly describe five fundamental semantic concepts related to this paradigm. All of them can be found more or less explicitly in programming languages such as C* [30] for the CM-2 [32] and CM-5 [33], MPL [25] for the MasPar MP-1 [2], HyperC [27], a language originally designed for the POMP massively parallel SIMD machine [23], Fortran 90 and its derived companions CM-Fortran [31] for the CM-2 and CM-5, MP-Fortran for the MasPar MP-1 [26] and the recent High Performance Fortran HPF [15]. The reader interested in a more detailed presentation together with a formal semantics is referred to [5] and the other contribution in this volume [4].

Conceptually, a *parallel variable* X is made of a fixed number of *components*. A parallel access to a parallel variable yields a *parallel value*. We are not concerned here with the spatial structure of these variables (layout, shape, geometry, etc.) or with their physical implementation (virtual processors, etc.). We rather concentrate on the various ways in which parallel variables are handled.

4.1 Data-Parallel Assignment

A data-parallel assignment $X := E$ synchronously assigns the components of a parallel value to the respective components of parallel variable X. Values are computed from a parallel expression E, where the semantics of usual operators is extended component-wise. *No inter-component effect occurs.* Also, observe that expression E can be arbitrarily complex, and involve tests on a parallel variable, say, *this*, whose value at each component is precisely the index of this component. One can thus define a single expression whose behavior is arbitrarily

different at each component:

```
switch (this){
  case 0 : do something;
  case 1 : do something else;
  ...}
```

The data-parallel assignment concept is thus sufficiently general to account for what is often called *SPMD* programming (*S*ingle *P*rogram *M*ultiple *D*ata). Observe this terminology is a somewhat improper generalization of the SIMD term. In the SPMD term, the word *Program* refers to syntax: this simply expresses that the program text is identical for all processes, without any restriction for their respective dynamic behaviors. This is thus more related to software engineering than with semantics. In contrast, in the SIMD term, the word *Instruction* refers primarily to an identical dynamic behavior of all processes. This is thus much stronger.

4.2 General Data-Parallel Communication

At the conceptual level of this presentation, a general communication is deeply different from a component-wise computation. It consists in *rearranging* the components of a parallel variable according to a given addressing. It is thus related to the distribution of the components among the nodes of the network, without regard to their values. In contrast, a local computation is related to the respective values of the components, without regard to their location. One can therefore say that these two concepts are *dual*. Communication, or more precisely rearrangement, should thus be seen as a first-class concept in the model. It should be considered as an instruction distinct from the synchronous parallel assignment, though it is presented in most data-parallel languages as a special case of expression (the `router` and `xnet` notation of MPL, the left-hand bracketing of parallel variables of C*, etc.).

This aspect deserves careful discussion, as it has dramatic consequences for the design of data-parallel languages. The structure of current parallel machines is an assembly of powerful general-purpose processors (Sparc, i860, Alpha, etc.) interconnected by a network. This network is often comparatively slow. This description includes both *real* parallel machines and *virtual* ones (more or less integrated networks of workstations). A communication operation can then be more than 1000 times more expensive than a local arithmetic operation. It is thus desirable that programmers specify communications as explicitely as possible, so that they are encouraged to avoid them (or at least become aware of the reasons for poor performance).

The recent history of parallel programming languages provides a striking illustration of the improrrtance of making communication operations explicit. In the framework of the CM-2 project, *Thinking Machines* designed a language called C*, a data-parallel extension of C. (This language is now called Data-Parallel C, DPC [18].) In this innovative design, the concepts of rearrangement and local computations were unified by using (pseudo-)pointers, in the C++ spirit. A component of a variable value is accessed through an indirection: Pseudo-pointer *this* gives access to the local component, $this + 1$ to the component of the neighbor, etc. Thus, the statement

$$this \rightarrow x = (this \rightarrow x) + 1$$

performs a local incrementation of each component of parallel variable x, whereas the statement

$$this \rightarrow x = (this + 1) \rightarrow x$$

stores into the local component of parallel variable x the value of the neighbor one shift distant.

A difficulty with this notation is that the latter operation may be 100 or 1000 times more expensive than the former! It is therefore very difficult to compile such a language, as it is impossible to obtain statically any reasonable prediction concerning the value of the pseudo-pointers, and thus to decide whether an access is local or remote. It is also very difficult to estimate program performance by inspecting its text. In order to get good performances, the programmer is led to enrich the program text with comments and/or macro definitions, so as to tag the potentially expensive parts, which adds an extra level of difficulty to parallel programming.

The language designers have therefore produced a new version of the language, which is the one currently known under the C* name. In this new version, communications are made *explicit* in the syntax. Accessing a non-local component of a parallel variable is denoted by left-hand bracketing, $[. + 1]x$. The C language allows (and even encourages!) assignment *expressions* like $y = (x = 2) + 1$. A natural extension of this feature is to introduce rearrangement as expressions, too: $y = [. + 1]x + 1$. Here again, we can witness a trade-off between *expressiveness* and *informativeness* in language design. The first version of C* was quite expressive, but not informative enough: the language was very difficult to use in practice for large software development. The new version of C* enhances informativeness at the price of a slightly more complex syntax.

Another important aspect of data-parallel language design is the conceptual nature of the rearrangement specification. At the level of the *execution models*, one often finds two distinct communication primitives: `send` and `get`, or, in the Fortran terminology for vector architectures, `scatter` and `gather`. Their precise relationship is strongly dependent on the specific features of the underlying architecture.

- On a CM-2, a `get` operation is twice as expensive as a `send` operation. A `get` is in fact implemented as two waves of `send` operations: the first conveys the requests of the receiving processors to the processors owning the values; the second brings the values back to the requesting processors.
- In contrast, on the MP-1 architecture, these two operations have similar costs. In both cases, a circuit has to be set up between the requester and the owner. Observe that several successive circuits may have to be set up in case of access conflicts: the cost of the communication strongly depends on the communication pattern.

Today, many parallel architectures use interconnection networks in which each message is independently routed. This is, in particular, the case for virtual parallel architectures built on packet-switched interconnection networks (TCP/IP). In this case, `send` operations are much cheaper than `get` operations, and very complex methods have been proposed to compile sequences of high-level `get` instructions into low-level `send` operations (see for instance [18]). The situation may nevertheless evolve dramatically within the coming years with the advent of new network technologies: it is very likely that the *circuit-switched* ATM networks will require specific optimizations different from the current *packet-switched* TCP/IP networks.

In contrast, at the level of *programming models*, the balance seems to be reversed. In effect, the `get` primitive enjoys a much simpler behavior than `send`.

- The `get` primitive does not generate any *write* conflict: each component receives at most one value. In the case of a `send`, a single component may be selected as the target by many components. It is thus necessary for the programmer to specify the merging protocol explicitly: combining the value through some associative, commutative operator with a neutral element ($+$, $\times$, min, max, logical conjunction and disjunction, etc.).
- The `get` primitive allows each component to be aware of its own state. If a component does not require any value, then its internal state is not modified by the instruction. In contrast, the `send` operation does not respect this basic intuition: a component may be selected as the target of a rearrangement independently of its own behavior.

In a formal semantic framework, these issues can be expressed by stating that the `get` primitive respects a fundamental *invariant*: a component is modified only if it so requests. This invariant is not respected by the `send` primitive. In this respect, it make sense to consider `get`-based data-parallel programming methodologies as safer than `send`-based ones, even though the former turns out to yield less efficient performances on most currently available architectures.

4.3 Synchronous Parallel Control

As far as the programming model is concerned, the flow of control $P;Q$ is completely synchronous. Each assignment is processed globally in turn. Of course, it is likely that this is *not* the case in the execution model (think of a network of asynchronous MIMD processors!). It is up to the compiler to guarantee that the asynchronous implementation of a data-parallel program is correct with respect to the synchronous semantics.

The parallel iteration `while` B `do` P `end` raises a problem. It is clear that it exits as soon as *all relevant* components of the boolean parallel expression B evaluate to false. But, does the iteration exit locally at a component where B evaluates to false? Because of the `send` semantics, it can happen that the value of B changes at one component *even though no assignment has been done on this component*! The answer to this subtle question varies from language to language. For instance, the answer is 'yes' for MPL, and it is 'no' for HyperC. The interested reader will find in [6] a detailed study of the semantic consequences of such design choices.

4.4 Bounded Asynchrony

All of the concepts detailed above induce a fully synchronous behavior on parallel components. It is thus important to add on top of them an orthogonal concept to cater for a bounded asynchrony. This concept is

usually called `where`. In a `where` B `do` P `end` program, all components for which the parallel boolean expression B evaluates to true execute program P synchronously, whereas the remaining ones skip it. Again, observe that this construct does not say anything about how the execution model will implement such a semantic specification. In a SIMD execution model, the components where B evaluates to false will remain idle waiting for the other components to proceed through P. In contrast, in a MIMD execution model, the compiler will seek to transform the code so that the idle components can safely initiate the rest of the program as soon as possible. This amounts to checking that the implicit synchronization barrier at the end of the `where` block can be removed without modifying the overall semantics of the program.

It is interesting to note that the `where` construct respects the fundamental invariant mentioned above: once a component has turned idle on entering a `where` block, its internal state is not modified until the `where` block is exited. Indeed, nesting `where` blocks may only restrict activity. Yet, this is *not* the case in presence of some unconditional activation constructs, such as `all` in MPL and `everywhere` in C*. In this case, an idle component may be temporarily re-activated. This makes reasoning about programs more difficult, very much as the escape constructs `break` and `continue` found in the C language make it more difficult to reason about loops than in Pascal. In this respect, it makes sense to state that using unconditional `all` or `everywhere` constructs within `where` blocks is not good programming practice and should be avoided as much as possible.

5 Conclusion

This paper describes a preliminary attempt towards an abstract presentation of data parallelism. We have mostly discussed the aspects related to *control*, that is, logical correctness of programs. But many other aspects are of dramatic importance for the efficient use of such languages. A similar study should be performed concerning data distribution and communication compiling. The interested reader will find in [7] an early presentation of these problems. A recent synthesis can be found in [16].

We could dare to say that parallel computing is still in its infancy. This judgment has to be understood in the sense that there is not yet a clear distinction between the *execution model* and the *programming model*. This distinction took many years to clarify in the area of scalar, sequential computing, and we can see now that there is no fixed connection between architectures and languages in the sequential context. It is considered as a desirable feature that all languages be available on all (most?) architectures. This effort has paved the way for the emergence of programming models with no immediate operational counterparts: functional programming, logic programming, some aspects of object-oriented programming, etc.

A similar awareness is now slowly emerging in parallel computing [19]. In some sense, the emergence of PVM and now MPI is a first step in this direction: at least, a standard low-level programming model for MIMD Distributed Memory (MIMD-DM) architectures is now widely accepted. But this programming model is still very close to the execution model, very much as the Fortran programming model is close to the Von Neumann execution model. There is still a long way to go before we achieve a complete and clear separation between parallel architectures and parallel languages, so that each can freely develop according to its own philosophy. We have tried to demonstrate that data parallelism is a first step in this direction. The emergence of "abstract" parallel languages may probably be the next step: the Linda language, among many others, suggests an interesting direction [8].

Another remark is that such a separation is desirable only if compilers are intelligent enough to fill the gap between the programming model and the execution model. We would even say, intelligent enough to use this gap to provide users with an implementation better than what they themselves could have coded by hand. A crucial milestone on the way is thus parallel compiler technology. This is already a noticeable trend. It is not so difficult today to build a Teraflops machine within a couple of years. Many companies and academic research centers have already announced such architectures. The *real* Grand Challenge is to build *compilers* to use their power [18]. Today, nobody would accept to program a workstation in assembly language, because it is too complex and wasteful: programming in C is much easier, and often leads to better overall performance. The *real* power comes from the compiler!

The data-parallel programming model appears as a fundamental model for massively parallel programming, mainly oriented towards regular applications. It allows programmers to reason about their program according to the usual sequential model, without struggling with the combinatorial explosion inherent in the potential interactions between thousands of asynchronous processes. Moreover, the sequential nature of the programming model enables one to import many software engineering methods originally designed for scalar languages such as Fortran, Pascal, C, etc.: programming environments for program design, validation, compilation, optimization, monitoring, debugging, etc. These methods simply have to be extended to handle arrays as first-class citizen.

One of the main reasons for the relative lack of success of control-parallel languages such as Occam was the lack of any programming environment: after a first phase of enthusiasm and low-level hacking, a language cannot survive on a large scale without such tools! In contrast, Thinking Machines and MasPar could announce within one year the availability of powerful programming environments (Prism for TMC, MPPE for MasPar). They are in fact based mainly on similar tools for the C language (`dbxtool`, C profiler, etc.). Of course, the counterpart is the difficulty of designing an efficient compiler for these languages on MIMD architectures.

Finally, it is important to stress that data parallelism and control parallelism are not mutually exclusive. The two models can be usefully combined. They are rather two complementary programming models. The former is a high-level model and the latter is a low-level model. Just as it can be useful to mix C and an assembly language to get *really* high performances in a sequential language, one can use a data-parallel procedure within a control-parallel harness, or a control-parallel program within a data-parallel harness. The C* programming environment explicitly supports the latter approach: the programmer can make "holes" in a data-parallel program, where the low-level message-passing library can be directly activated. The HyperC and the HPF programming models are also specifically extended in this direction. The interested reader is referred to the contribution of Ian Foster in this volume [17] for a detailed discussion on mixing the two models in high-performance languages.

At this time, very little verification can be done by the compiler about the actual correctness of the interaction between the two models. This is at the programmer's own risk, just as a C compiler does not verify the low-level manipulations specified by an `asm` instruction. Also, little guideline exists to assist the programmer in mixing the two models: the only criterion is still to use the right model to do the right thing... Mastering such heterogeneous parallel programming models will probably become a major research challenge in the coming years.

Acknowledgments

This work is the result of many discussions with the members of the "ParaDigme" project. It has also been substantially improved by the remarks of Ian Foster and Sanjay Rajopadhye. Their contribution is gratefully acknowledged.

References

[1] C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Correctness of automated distribution of sequential programs. In *Parallel Architectures and Languages Europe (PARLE'93)*, volume 694 of *Lecture Notes in Computer Science*, pages 517–528, 1993.

[2] T. Blank. The MasPar MP-1 architecture. In *Proc. of the 35th IEEE Computer Society Int. Conf., Spring COMPCON'90*, pages 20–24, San Francisco, 1990.

[3] G.E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, 1990.

[4] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science, Tutorial*, chapter Formal validation of data-parallel programs: introducing the assertional approach, pages 252–281. Springer Verlag, 1996.

[5] L. Bougé. On the semantics of languages for massively parallel SIMD architectures (extended abstract). In *Proc. Parallel Arch. and Lang. Europe Conf., PARLE'91, vol. 2*, number 506 in Lect. Notes Comp. Science, pages 166–183, Eindhoven, 1991. Springer.

[6] L. Bougé and J.-L. Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems*, 8(3/4):363–378, 1992.

[7] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[8] N. Carreiro and D. Gelertner. Linda in context. *Communications of the ACM*, 32(4):323–357, 1989.

[9] E.W.G. Dijkstra. GO TO statement considered harmful. *Communications of the ACM*, 11:147–148, 1968.

[10] Tz. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2:155–173, 1982.

[11] R. Esser and R. Knecht. Intel Paragon XP/S – architecture and software environment. Technical Report KFA-ZAM-IB-9305, Central Institute for Mathemantics, Research Center Juelich, 1993.

[12] P. Feautrier. Asymptotically efficient algorithms for parallel architectures. In *Decentralized Systems*, pages 273–284. IFIP Working Group 10.3, North-Holland, 1989.

[13] Paul Feautrier. *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science, Tutorial*, chapter Automatic Parallelization in the Polytope Model, pages 79–103. Springer Verlag, 1996.

[14] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computers*, C-21(9):948–960, 1972.

[15] High Performance Fortran Forum. *High Performance Fortran language specification (draft version).* CITI/CRPC, Rice Univ., Houston, January 1993. Version 1.0 Draft.

[16] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[17] Ian Foster. *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science, Tutorial*, chapter Task Parallelism and High-Performance Languages, pages 179–196. Springer Verlag, 1996.

[18] Ph.J. Hatcher and M. Quinn. *DataParallel C: Compiling DataParallel Languages on MIMD architectures.* MIT Press, 1992.

[19] W.D. Hillis and G.L. Steele, Jr. Data-parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[20] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[21] Intel Corporation, Beaverton, OR. *Paragon XP/S Product Overview*, 1991.

[22] K.E. Iverson. *A programming langage.* Wiley, New-York, 1962.

[23] R. Keryell, Ph. Matherat, and N. Paris. POMP, or how to design a massively parallel machine with small developments. In *Proc. Conf. on Parallel Arch. and Lang. Europe, PARLE'91, vol. 1*, number 505 in Lect. Notes Comp. Science, Eindhoven, 1991. Springer.

[24] Björn Lisper. *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science, Tutorial*, chapter Data Parallelism and Functional Programming, pages 220–251. Springer Verlag, 1996.

[25] MasPar Computer Corporation, Sunnyvale, CA. *Maspar Parallel Application Language Reference Manual*, 1990.

[26] MasPar Computer Corporation, Sunnyvale, CA. *MP-Fortran Reference Manual*, 1990.

[27] N. Paris. Définition de POMPC (version 1.99). Technical Report 92-5, LIENS, Paris, 1992. English version in preparation.

[28] R.H. Perrot. A language for array and vector processors. *ACM Trans. on Progr. Lang. and Syst.*, 1(2):177–195, 1979.

[29] G.L. Steele, Jr. and W.D. Hillis. Connection machine lisp : Fine-grain parallel symbolic processing. In *Proc. 1986 ACM Conf. on Lisp and Funct. Progr.*, pages 279–297, Cambridge, Mass., 1986.

[30] Thinking Machine Corporation, Cambridge MA. *C* Programming Guide*, 1990.

[31] Thinking Machine Corporation, Cambridge, MA. *CM-FORTRAN Programming Guide*, 1990.

[32] Thinking Machine Corporation, Cambridge, MA. *CM-2 Technical Summary*, 1991.

[33] Thinking Machine Corporation, Cambridge, MA. *CM-5 Technical Summary*, 1991.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399