Pergamon

S0096-0551(96)00012-4

Comput. Lang. Vol. 22, No. 2/3, pp. 165–179, 1996

0096-0551/96 $15.00 + 0.00

DESIGN AND IMPLEMENTATION OF $8_{1/2}$: A DECLARATIVE DATA-PARALLEL LANGUAGE

OLIVIER MICHEL

LRI u.r.a. 410 du CNRS, Bâtiment 490, Université de Paris-Sud, 91405 Orsay Cedex, France

(Received 18 March 1996; revision received 17 April 1996)

Abstract—In this article we advocate a declarative approach to data-parallelism to provide both parallelism expressiveness and efficient execution of data intensive applications. $8_{1/2}$, an experimental language combining features of collection and stream oriented languages in a declarative framework, is presented. A new structure, the web, allows the programmer to write programmes as mathematical expressions and to implicitly express data and control parallelism. The first part of this paper proposes a classification of the various expressions of parallelism in programming languages. We show that hybrid execution models combining both data and control parallelism are possible and necessary to get an effective speedup. We sketch the advantage of the declarative style with respect to parallelism expression (application side) and exploitation (compiler side). In the second part we describe the $8_{1/2}$ language and the concepts of collection, stream and web. A web is a multi-dimensional object that represents the successive values of a structured set of variables. Some $8_{1/2}$ programmes are given to show the relevance of the web data structure for simulation applications (a resolution of O.D.P.E. and a simulation in artificial life). Examples of $8_{1/2}$ programmes, involving the dynamic creation and destruction of webs, are also given. Such programmes are necessary for simulations of growing systems. In the third part, the implementation of a compiler restricted to the static part of the language is described. We focus on the process of web equations compilation towards a virtual SIMD machine. We also present the clock calculus, the scheduling inference and the distribution of the computations among the processing elements of a parallel computer.

data-parallelism declarative languages collection-oriented languages synchronous data-flow recursive collection data-distribution and scheduling

1. INTRODUCTION

1.1. A proposal for a taxonomy of parallelism expressions

Table 1 proposes a classification of the various expressions of parallelism in programming languages. Such a framework is required for the analysis of existing languages and the development of a new one. We propose to mimic the Flynn classification of parallel architectures [1] and to compare parallel languages constructs following two criteria: the way they let the programmer express the control and the way they let him manipulate the data. The programmer has three choices to express the flow of computations:

- *Implicit control*: this is the declarative approach. The compiler (static extraction of the parallelism) or the runtime environment (dynamic extraction by an interpreter or a hardware architecture) has to build a computation order compatible with the data dependencies exhibited in the programme.
- *Explicit control* which refines in:
  —*Express what has to be done sequentially*: this is the classical sequential imperative execution model, where control structures build only one thread of computation.
  —*Express what can be done in parallel*: this is the concurrent languages approach. Such languages offer explicit control structures like PAR, ALT, FORK, JOIN, etc.

For the data handling, we will consider two major classes of languages:

- *Collection based languages* allow the programmer to handle sets of data as a whole. Such a set is called a collection [2]. Examples of languages of this kind are: APL, SETL, SQL, *Lisp, C* ...

- *Scalar languages* allow also the programmer to manipulate a set of data but only through references to one element. For example, in standard Pascal, the main operation performed on an array is accessing one of its elements.

Historically, the data-parallelism has been developed from the possibility of introducing parallelism in sequential languages (this is the "starization" of languages: from C to C*, from Lisp to *Lisp ...). It relies on sequential control structures (*when ...) and parallel data. However, Table 1 shows that the concept of collection can be freely mixed with other expressions of control. As a consequence, collection based languages can be mixed with concurrent languages (multiple SIMD model or MSIMD) and declarative languages (Gamma [3] or $8_{1/2}$ [4]).

1.2. Declarative structure and massive parallelism

Now a short overview of the advantage of the declarative style with respect to the parallelism expression and exploitation is going to be presented. Nowadays new architectures appear [5–8] to efficiently support an SPMD or MSIMD execution model. This motivates the development of new programming paradigms able to express more than one kind of parallelism. However, to quote [9]: "simplicity and efficiency of the SIMD approach" must be preserved while acquiring the "processor utilisation and the flexibility of control structure afforded by the MIMD approach".

The development of a declarative framework supporting both data and control parallelism relies on the construction of an adequate data structure and its subsequent algebra. As a matter of fact, stream algebra is well fitted to control-parallelism [10] while collection algebra supports implicit data-parallelism [11]. Consequently, this leads to merge streams and collections into a unique data structure. The $8_{1/2}$ language is based on *webs* which is such a combination. From the parallelism point of view, managing streams and collections in a declarative framework exhibits several advantages:

- There is no explicit construct for parallelism in the language, in accordance with the "parallelism as an implementation property" point of view (i.e. parallelism is in the scope of implementation, and is irrelevant at the semantic level).
- The declarative form of the language makes it easy to perform dependence analysis between tasks and the subsequent exploitation of control parallelism.
- Collections are a natural support of the data-parallelism and collection operations between webs naturally lead to a data-parallel implementation.
- Collections introduce a natural support for the distribution of data.
- Introducing collections corrects some of the drawbacks sustained against the stream oriented data-flow model [12], mainly by adding some specific handling of arrays with a consistent concept of time.
- Transparential references allow a formal treatment of programmes, and programme optimization using programme transformations are possible (cf. for example [13, 14]).

Furthermore, embedding collection in a synchronous data-flow model combines the advantages of the synchronous and asynchronous parallel styles [9]. Consider for example the *actor model*: it proposes a minimal kernel to deal with control parallelism but handling of homogeneous sets of data, like arrays, is definitively inefficient [15]. From another point of view, the handling of communications in sequential data-parallel oriented languages, like *LISP, forbids overlapping of communications and computations because there is only one thread of control.

Table 1. A classification of languages from the parallel constructs point of view

	Declarative languages 0 *instruction counter*	Sequential languages 1 *instruction counter*	Concurrent languages *n instructions counters*
Scalar languages	Sisal, Id, LAU, Actors	Fortran, C, Pascal	Ada, Occam
Collection languages	Gamma, $8_{1/2}$	*LISP, HPF, CM Fortran	CM Fortran + multi-threads

These two examples show the advantage of combining data and control parallelism. Using *implicit* data- and control-parallelism enables:

- the maximal expression of the parallelism inherent to an application (this does not imply the maximal exploitation of parallelism);
- the use of the effective parallelism which implies cheaper implementation overheads (with respect to the target architecture); and
- the hiding of communication costs by overlapping computations of independent activities.

The rest of the paper describes the language $8_{1/2}$ and its compilation. It is an embedding of data-parallelism in a declarative framework. $8_{1/2}$ does not support all styles of parallel programming, but we argue that it combines advantages of the two approaches for a large class of applications. A stream is a direct representation of a trajectory of a dynamical system (i.e. the sequence of the successive states of the system), a collection corresponds to the value of a multidimensional state or to the discretization of a continuous parameter. In addition, the declarative form of the language fits well with the functional description of a dynamical system. Thus we advocate the use of $8_{1/2}$ for the parallel simulation of dynamical systems (e.g. deterministic discrete events systems [16]).

2. THE DECLARATIVE DATA-PARALLEL LANGUAGE $8_{1/2}$

$8_{1/2}$ has a single data structure called a *web*. A *web* is the combination of the concept of *stream* and *collection*. This section describes these three notions.

2.1. The collection in $8_{1/2}$

A *collection* is a data structure that represents a set of elements *as a whole* [17]. Several kinds of aggregation structure exist: *set* in SETL [18] or in [19], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [20] or even *finite discrete space* in Cellular Automata [21]. Data-parallelism is naturally expressed in terms of collections [2, 22]. From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

Here we consider collections that are *ordered* sets of elements. An element of a collection, also called a *point* in $8_{1/2}$, is accessed through an index. The expression $T.n$ where T is a collection and n an integer, is a collection with one point; the value of this point is the value of the nth point of T (point numbering begins with 0). If necessary, we implicitly coerce a collection with one point into a scalar and vice-versa through a type inference system described in [23]. More generally, the system is able to coerce a scalar into an array containing only the value of the scalar.

Geometric operators change the *geometry* of a collection, i.e. its structure. The geometry of a collection of scalar is reduced to its *cardinal* (the number of its points). A collection can also be *nested*: the value of a point is a collection. Collection nesting allows multiple levels of parallelism and exists, for example, in ParalationLisp [24] and NESL [25]. The geometry of the collection is the hierarchical structure of point values. The first geometric operation consists of *packing* some webs together:

$$T = \{a,b\}$$

In the previous definition, a and b are collections resulting in a nested collection T. Elements of a collection may also be named and the result is a *system*. Assuming

$$car = \{velocity = 5, consumption = 10\}$$

the points of this collection can be reached through the dot construct using uniformly their label, e.g. *car.velocity*, or their index: *car*.0. The *composition* operator # concatenates the values and merges the systems:

$$A = \{a,b\};\ B = \{c,d\};\ A \# B \Rightarrow \{a,b,c,d\}$$
$$ferrari = car \# \{color = red\} \Rightarrow \{velocity = 5, consumption = 10, color = red\}$$

The last geometric operator we will present here is the *selection*: it allows selection of some point values to build another collection. For example:

$$\begin{aligned} Source &= \{a,b,c,d,e\} \\ target &= \{1,3,\{0,4\}\} \\ Source(target) &\Rightarrow \{b,d,\{a,e\}\} \end{aligned}$$

The notation *Source*(*target*) must be understood in the following way: a collection can be viewed as a function from [0..n] to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is a set of natural numbers, collections can be composed and the following property holds: $Source(target).i = Source(target.i)$, mimicking the function composition definition. From the parallel implementation point of view, selection corresponds to a gather operation and is implemented using communication primitives on a distributed memory architecture.

Four kinds of function application can be defined:

Operator	Signature	Syntax
application:	$(collection^p \rightarrow X) \times collection^p \rightarrow X$	$:f(c_1, \ldots, c_p)$
extension^:	$(scalar^p \rightarrow scalar) \times collection^p \rightarrow collection$	$:f\hat{}(c_1, \ldots, c_p)$
reduction\:	$(scalar^2 \rightarrow scalar) \times collection \rightarrow scalar$	$:f\backslash c$
scan\\:	$(scalar^2 \rightarrow scalar) \times collection \rightarrow collection$	$:f\backslash\backslash c$

X means both scalar or collection; p is the arity of the functional parameter f.

The first operator is the standard function application. The second type of function application produces a collection whose elements are the "pointwise" application of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. Extension is implicit for the basic operators (+, *, ...) but is explicit for user-defined functions to avoid ambiguities between application and extention (consider the application of the *reverse* function to a nested collection). The third type of function application is the *reduction*. Reduction of a collection using the binary scalar addition results in the summation of all the elements of the collection. Any associative binary operation can be used, e.g. a reduction with the *min* function gives the minimal element of a collection. The scan application mode is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\backslash\{1,1,1\} \Rightarrow \{1,2,3\}$. See [26] for a programming style based on scan. Reductions and scans can be performed in $O(\log_2(n))$ steps on SIMD architecture, where n is the number of elements in the collection, if there are enough PEs.

2.2. The stream in $8_{1/2}$

LUCID [27] is one of the first programming languages defining equations between infinite sequences of values. Although $8_{1/2}$ streams are also defined through equations between infinite sequences of values, $8_{1/2}$ streams are very different from those of LUCID.

A metaphor to explain $8_{1/2}$ streams is the sequence of values of a register. If you observe a register of a computer during a programme run, you can record the successive store operations on this register, together with their dates. The (timed) sequence of stores is an $8_{1/2}$ stream. At the beginning, the content of the register is uninitialized (a kind of undefined value). Then it receives an initial value. This value can be read and used to compute other values stored elsewhere, as long as the register is not the destination of another store operation.

The time used to label the changes of values of a register is not the computer physical time, it is the logical time linked to the semantics of the programme. The situation is exactly the same between the logical time of a *discrete-events simulation* and the physical time of the computer which runs the simulation. Therefore, the time to which we refer is a countable set of "events" meaningful for the programme.

$8_{1/2}$ is a declarative language which operates by making descriptive statements about data and relations between data rather than describing how to produce them. For instance, the definition $C = A + B$ means the value in register C is always equal to the sum of the values in register A and B. We assume that the changes of the values are propagated instantaneously. When A (or B)

Table 2. Examples of streams

	0	1	2	3	4	5	6	7	8
1	1								
1 + 2	3								
Clock 2	*true*		*true*		*true*		*true*		*true*
Assuming A	1		2	3		4	5	6	
Assuming B		1		2			1		1
$C = A + B$		2	3	5		6	6	7	7
$\$C$			2	3		5	6	6	7

changes, so do C at the same logical instant. Note that C is uninitialized as long as A or B are uninitialized.

Table 2 gives some examples of $8_{1/2}$ streams. The first row gives the instants of the logical clock which counts the events in the programme. The instants of this clock are called a **tick** (a tick is a column in the table). The date of the "store" operations of a particular stream are called the **tock** of this stream (because a clock is thought to make "tick-tock"): they represent the set of events meaningful for that stream (a tock is a non-empty cell in the table). At a tick t, the value of a stream is: the last value stored at tock $t' \leq t$ if t' exists, the uninitialized value otherwise. For example, the value of $\$C$ at tick 0 is undefined, whilst its value at tick 4 is 3.

A *scalar constant stream* is a stream with only one "store" operation, at the beginning of time, to compute the constant value of the stream. A constant n really denotes a scalar constant stream. Constructs like *Clock n* denote another kind of constant streams: they are predefined sequences of *true* values with an infinite number of tocks. Scalar operations are extended to denote elementwise application of the operation on the values of the streams. The delay operator \$ shifts the entire stream to give access, at the current time, to the previous stream value. This operator is the only operator that does not act in a pointwise fashion. The tocks of the delayed stream are the tocks of the arguments with the exception of the first one.

The last kind of stream operator is the sampling operator. The most general one is the "trigger", which is very close to the T-gate in data-flow languages [28]. It corresponds to the temporal version of the conditional. The values of T when B are those of T sampled at the tocks where B takes a *true* value (see Table 3). A tick t is a tock of A when B if A and B are both defined *and* t is a tock of B *and* the current value of B is *true*.

$8_{1/2}$ streams present several advantages:

- $8_{1/2}$ streams are manipulated as a whole, using filters, transducers ... [29].
- Like other declarative streams, this approach represents imperative iterations in a "mathematically respectable way" [30] and to quote [13]: "... series expressions are to loops as structured control constructs are to gotos."
- The tocks of a stream really represent the logical instants where some computation must occur to maintain the relationships stated in the programme.
- The $8_{1/2}$ stream algebra verifies the *causality assumption*: the value of a stream at any tick t may only depend upon values computed for previous tick $t' \leq t$. This is definitively not the case for LUCID (LUCID includes the inverse of \$, an "uncausal" operator).
- The $8_{1/2}$ stream algebra verifies the *finite memory assumption*: it exists as a finite bound such that, the number of past values that are necessary to produce the current values remains smaller than that bound.

The last two assumptions have been investigated in two real-time programming languages derived from LUCID: LUSTRE [31] and SIGNAL [32]. Such streams enable a static execution model: the successive values making a stream are the successive values of a single memory location and we do not have to rely on a garbage collector to free the unreachable past values (as in Haskell [33]

Table 3. Example of a sampling expression

A	1	2	3	4	5	6	7	8	9
B	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*false*	*true*
A when B				4		6	7		9

for example). In addition, we do not have to compute the value of a stream for each tick, but only for the tocks.

2.3. *Combining streams and collections into webs*

A *web* is a *stream of collections* or a *collection of streams*. In fact, we distinguish between two *kinds* of webs: *static* and *dynamic*. A static web is a collection of streams where every element has the same clock (the clock of a stream is the set of its tocks). In an equivalent manner, a static web is a stream of collections where every collection has the same geometry. Webs that are not static are called dynamic. The compiler is able to detect the kind of the web and compiles only the static ones. Programmes involving dynamic webs are interpreted.

Collection operations and stream operations are easily extended to operate on static webs considering that the web is a collection (of streams) or a stream (of collections).

$8_{1/2}$ is a declarative language: a programme is a system representing a set of web definitions. A web definition takes a form similar to:

$$T = A + B \tag{1}$$

Equation (1) is an $8_{1/2}$ expression that defines the web T from the web A and B (A and B are the parameters of T). This expression can be read as a *definition* (the naming of the expression $A + B$ by the identifier T) as well as a *relationship*, satisfied at each moment and for each collection element of T, A and B. Figure 1 gives a three-dimensional representation of the concept of web.

Running an $8_{1/2}$ programme consists of solving the web equations. Solving a web equation means "enumerating the values constituting the web". This set of values is structured by the stream and collection aspects of the web: let a web be a stream of collections; in accordance with the time interpretation of stream, the values constituting the web are enumerated in the stream's ascending order. So, running a $8_{1/2}$ programme means enumerating, in sequential order, the values of the collections making the stream. The enumeration of the collection values is not subject to some predefined order and may be done in parallel.

2.4. *Declarative definition of recursive collections*

A definition is recursive when the identifier on the left-hand side appears also directly or indirectly on the right-hand side. Two kinds of recursive definitions are possible.

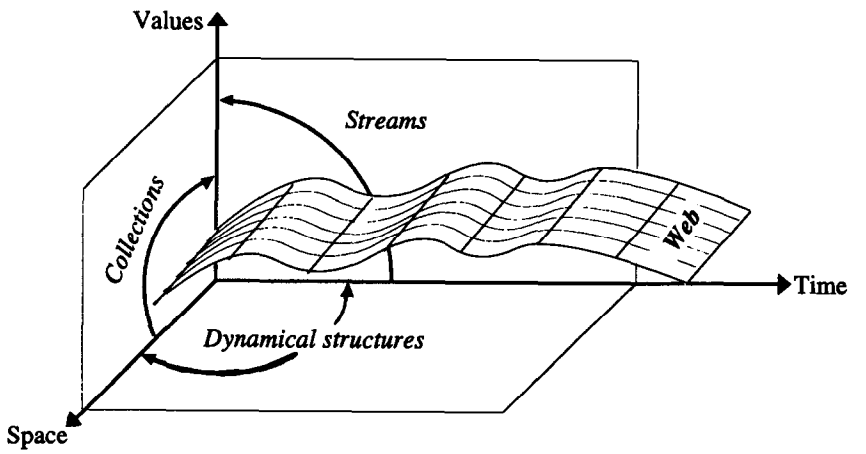

Fig. 1. A web specified by an $8_{1/2}$ equation is an object in the ⟨time, space, value⟩ axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (cf. Section 2.6).

2.4.1. Temporal recursion. Temporal recursion allows the definition of the current value of a web using past values of it. For example, the definition

$$T@0 = 1$$
$$T = \$T + 1 \text{ when } Clock\ 1$$

specifies a counter which starts at 1 and counts at the speed of the tocks of *Clock* 1. The @0 is a temporal guard that quantifies the first equation and means "for the first tock only". In fact, T counts the tocks of *Clock* 1.

The order of equations in the previous programme does not matter: the unquantified equation applies only when no quantified equations apply. The language for expressing guards is restricted to @n with the meaning "for the nth tock only".

2.4.2. Spatial recursion. Spatial recursion is used to define the current value of a point using current values of other points of the same web. For example,

$$iota = 0\#(1 + iota{:}[2])$$

is a web with three elements such that *iota.i* is equal to i. The operator: [n] truncates a collection to n elements so we can infer from the definition that *iota* has 3 elements (0 is implicitly coerced into a one-point collection). Let $\{iota_1,iota_2,iota_3\}$ be the value of the collection iota. The definition states that

$$\{iota_1,iota_2,iota_3\} = \{0\}\#(\{1,1\} + \{iota_1,iota_2\})$$

which can be rewritten as:

$$\begin{cases} iota_1 = 0 \\ iota_2 = 1 + iota_1 \\ iota_3 = 1 + iota_2 \end{cases}$$

which proves our previous assertion.

2.5. Examples of webs with static structure

2.5.1. Numerical resolution of a parabolic partial differential equation. We want to simulate the diffusion of heat in a thin uniform rod. Both extremities of the rod are held to 0°C. The solution of the parabolic equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{2}$$

gives the temperature $U(x,t)$ at a distance x from one end of the rod after time t. An explicit method of solution uses finite-difference approximation of equation (2) on a mesh $(X_i = ih, T_j = jk)$ which discretizes the space of variables [34]. One finite-difference approximation to equation (2) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2} \tag{3}$$

which can be rewritten as

$$U_{i,j+1} = rU_{i-1,j} + (1 - 2r)U_{i,j} + rU_{i+1,j} \tag{4}$$

where $r = k/h^2$. It gives a formula for the unknown temperature $U_{i,j+1}$ at the $(i,j+1)$th mesh point in term of known temperatures along the jth time-row. Hence, we can calculate the unknown pivotal values of U along the first time-row $T = k$, in terms of known boundary and initial values along $T = 0$, then the unknown pivotal values along the second time-row in terms of the first calculated values, and so on.

The corresponding $8_{1/2}$ programme is very easy to derive and simply corresponds to the description of initial values, boundary conditions and the specification of the relation (4). The

stream aspect of a web corresponds to the time axis, while the collection aspect represents the rod discretization.

$$
\begin{aligned}
start &= \textbf{some initial temperature distribution};\\
LeftBorder &= 0;\\
RightBorder &= 0;\\
U@0 &= start;\\
U &= LeftBorder \# inside \# RightBorder;\\
float\ inside &= 0.4{*}pU(left) + 0.2{*}pU(middle) + 0.4{*}pU(right);\\
pU &= \$U \text{ when } Clock;\\
left &= '6;\\
right &= left + 2;\\
middle &= left + 1;
\end{aligned}
$$

The second argument of the *when* operator is *Clock* which represents the time discretization (cf. Fig. 2). The expression *'n* generates a vector of *n* elements where the *i*th has a value *i*.

2.5.2. The simulation of a reactive system. Here is an example of a hybrid dynamical system, a "wlumf" which is a "creature" whose behaviour (eating) is triggered by the level of some internal state (see [35] for such model in ethological simulation).

More precisely, a wlumf is *hungry* when its *glycaemia* is under 3. It can eat when there is some food in its environment. Its metabolism is such that when it eats, the glycaemia goes up to 10 and then decreases to zero at a rate of one unit per time step. All these variables are scalar. Essentially, the wlumf is made of counters and flip-flop triggered and reset at different rates.

$$
\begin{aligned}
&boolean\ FoodInNeighbourhood = Random;\\
&System\ wlumf = \{Hungry@0 = false;\\
&\qquad\qquad Hungry \quad\ = (Glycaemia < 3);\\
&\qquad\qquad Glycaemia@0 = 6;\\
&\qquad\qquad Glycaemia \ = \text{if } Eating \text{ then } 10 \text{ else max } (0, \$Glycaemia - 1) \text{ when } Clock \text{ fi};\\
&\qquad\qquad Eating \quad\ = \$Hungry \ \&\&\ FoodInNeighbourhood;\}
\end{aligned}
$$

The result of an execution is given in Fig. 3.

2.6. *Examples of web with dynamic structure*

Webs with a static structure cannot describe phenomena that grow in space (like plants). To describe these structures, we need dynamically structured webs. The rest of this section gives some

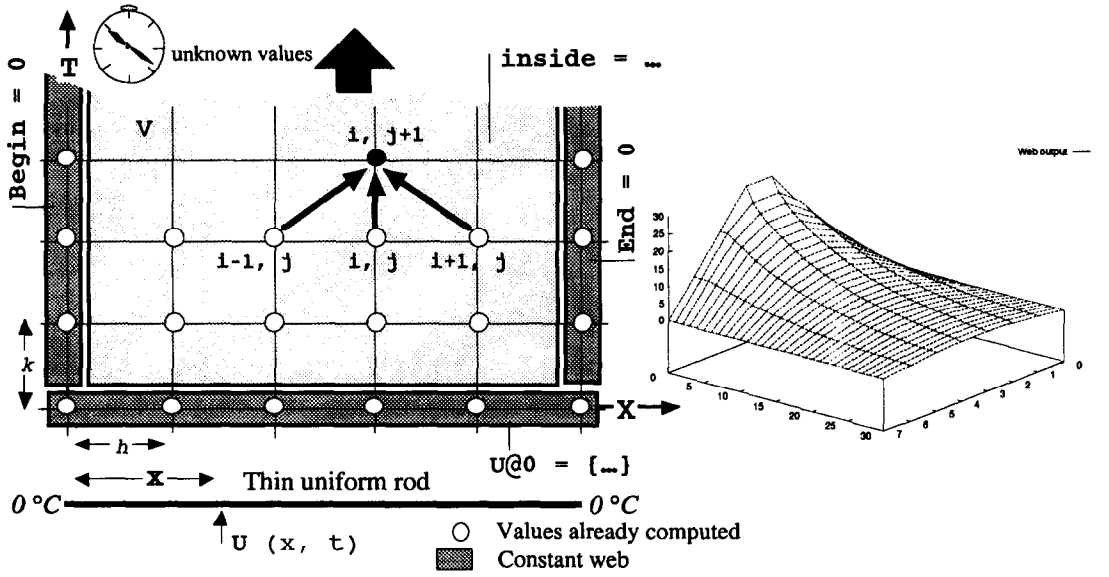

Fig. 2. Diffusion of heat in a thin uniform rod.

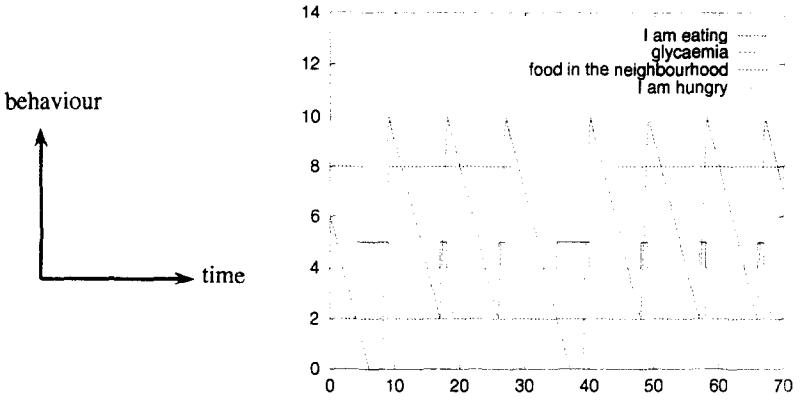

Fig. 3. Behaviour of an hybrid dynamical system.

examples of this kind of web. Note that we do not need to introduce new operators; actual web definitions already enable the construction of dynamically shaped webs.

2.6.1. Pascal's triangle. The numbers in Pascal's triangle give the binomial coefficients. The value of the point (*line*, *col*) in the triangle is the sum of the point value (*line* − 1, *col*) and point value (*line* − 1, *col* − 1). We decide to map the rows in time, thus the web representation of Pascal's triangle is a stream of growing collections. This web is dynamic because the number of elements in the collection varies in time.

We can identify that the row l ($l > 0$) is the sum of row ($l - 1$) concatenated with 0 and 0 concatenated with row ($l - 1$). The $8_{1/2}$ programme is straightforward.

$$t = (\$t \# 0) + (0 \# \$t) \text{ when } Clock;$$
$$t@0 = 1;$$

The first five values of Pascal's triangle are:

Top:0 : {1}:*int*[1]
Top:1 : {1,1}:*int*[2]
Top:2 : {1,2,1}:*int*[3]
Top:3 : {1,3,3,1}:*int*[4]
Top:4 : {1,4,6,4,1}:*int*[5]

2.6.2. Eratosthenes's sieve. We present a modified version of the famous Eratosthenes's sieve to compute prime numbers. It consists of a generator producing increasing integers and a list of known prime numbers (starts with a single element, 2). Each time we generate a new number, we try to divide it with all known prime numbers. A number that is not divided by a prime number is a prime number itself and is added to the list of prime numbers.

Generator is a web that produces a new integer at each tock. *Extend* is the number generated with the same size as the web of already known prime numbers. *Modulo* is the web where each element is the modulo of the produced number and the prime number in the same column. *Zero* is the web containing boolean values that are true every time that the number generated is divided by a prime number. Finally, *reduced* is a reduction with an *or* operation, that is, the result is *true* if one of the prime numbers divides the generated number. The $x{:}|y|$ operator shrinks the web x to the rank specified by y. The rank of a collection is a vector where the ith element represents the number of element of x in the ith dimension.

$$generator@0 = 2;$$
$$generator = \$generator + 1 \text{ when } Clock;$$
$$extend = generator{:}|\$crible|;$$
$$modulo = extend \ \% \ \$crible;$$
$$zero = (modulo = = (0{:}|modulo|));$$
$$reduced = or\backslash zero;$$

crible = $*crible* # *generator* when (not *reduced*);
crible@0 = *generator*;

The first five steps of the execution give for *crible*:

Top:0 : {2}:*int*[1]
Top:1 : {2,3}:*int*[2]
Top:2 : {2,3}:*int*[2]
Top:3 : {2,3,5}:*int*[3]
Top:4 : {2,3,5}:*int*[3]

3. IMPLEMENTATION OF THE $8_{1/2}$ COMPILER

The compiler described hereafter is restricted to programmes defining webs with a static structure. A high-level block diagram of the compiler is shown in Fig. 4. The output can either be a sequential C code or a code for a virtual SIMD machine (similar to CVL [36]).

3.1. The structure of the compiler

We describe briefly the various phases of the compiler written in a dialect of ML [37]:

Parsing: parses the input file and creates the programme graph representation used in the remaining modules of the compiler. This is a conventional two-pass parser implemented using the ML version of *lex* and *yacc*.

Binding: the compiler enforces static scoping of all variables. This phase is also responsible for inline expansion of functions, removal of unused definitions and the detection of undefined variables.

Geometry inference: the geometry of a web is inferred at compile time by the "geometric type system" (see [23]). Programmes involving dynamic webs are detected by the geometry inference and rejected. For example, the following programme: $T@0 = 0$; $T = (\$T \# \$T)$ when *Clock* defines a web T with a number of elements growing exponentially in time:

$$T \Rightarrow \langle \{0\};\{0,0\};\{0,0,0,0\}; \ldots \rangle$$

every collection of the stream has twice as many elements as the previous one. This kind of programme implies dynamic memory allocation and dynamic load balancing and is rejected by the compiler (but such programmes can be interpreted).

Scheduling inference: to solve the $8_{1/2}$ equations between webs, we have to extract the sequencing

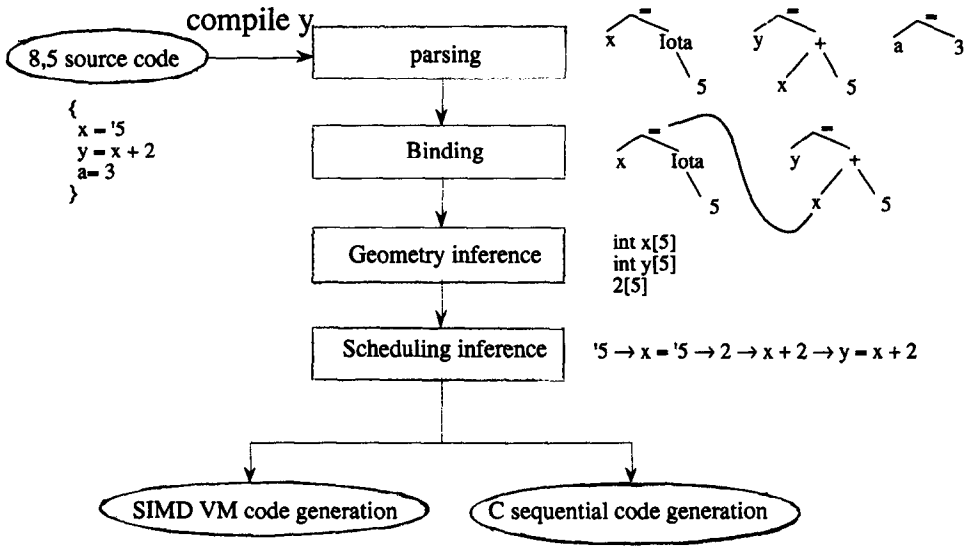

Fig. 4. Block diagram of the compiler. Ellipses indicate source or target code, and rectangles are processing modules.

of the computations of the various right-hand sides from the data flow graph. Once the scheduling of the instructions is done, the compiler computes the memory storage required by a programme execution.

Code generation: the compiler generates a standalone sequential **C** code running on work-stations or a code to be executed by the SIMD virtual machine. However, all the compiler phases assume a full MIMD execution model and we are working on the MIMD code generation. The sequential **C** code is stackless and does not use **malloc** or any other dynamic runtime features.

3.2. The clock calculus

The clock calculus of a web is needed to decide whether the computation of a collection has to take place at some tick or not (a static web is viewed as a stream of collections for the implementation). The *clock* of a web X is a boolean stream holding the value *true* at tick t if t is a tock of X. Let x be the value of X at a tick t, and $clock(x)$ the value of the clock associated with X at the same tick. Every definition

$$X = f(Y)$$

in the initial programme is translated into the assignment:

$$x: = \text{if } clock(X) \text{ then } f(y) \tag{5}$$

This statement is synthesized by induction on the structure of the definition of X. For example:

$$clock(A \text{ when } B) = b \wedge clock(B)$$
$$clock(clock(X)) = True$$

This transformation produces a normal form from the original web definition. Roughly, the compiler will generate for any expression of the programme, a task performing the assignment shown in equation (5). It is still necessary to compute the dependencies between the tasks to determine their relative order of activation.

3.3. The scheduling inference

The data-flow graph associated with an $8_{1/2}$ programme is directly extracted from the programme in normal form. Unfortunately, this graph cannot be directly used to generate the task scheduling. In the case of a scalar data flow programme, the data-flow graph is the same as the dependencies graph. It is no longer true with collections. For example, in the following programme:

$$A = B$$

every point of A (i.e. every element of the collection of the web A) depends on the corresponding point of B. On the other hand, the following programme that sums all elements of B:

$$A = +\backslash B$$

produces a web A of only one point, depending on all the points of B. Nevertheless, both programmes give the same data flow graph where the nodes A and B are connected.

The data flow graph can be viewed as an approximation of the real dependencies graph. This approximation is too rough; for example, on this basis, we cannot compile spatial recursive programmes. The work of the compiler is to annotate the data-flow graph to get a finer approximation of the dependencies graph. The true graph of the dependencies cannot be explicitly built because it has as many nodes as points in the web of the programme (for example, in numerical computation, matrices of size 1000×1000 are usual and would give dependency graphs of over 10^6 nodes).

We call *task sequencing graph* the approximation of the dependencies graph annotated in the following way (Fig. 5):

- An expression e depends on the web X if X appears syntactically in e. However, we remove the dependencies of variables appearing in the scope of a delay: those dependencies correspond to a past value and the compiler is scheduling the computation of the present iteration only.

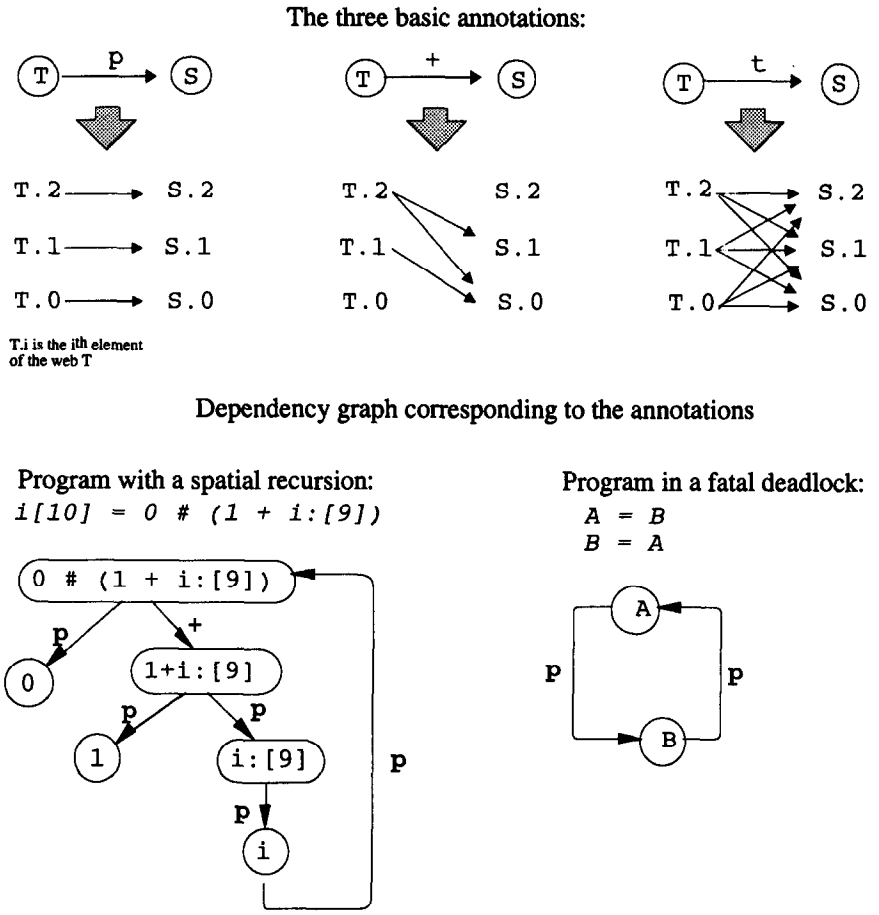

Fig. 5. Representation of the three possible annotations used to build the sequencing graph. Two examples are given. i is a vector such that the jth element of i has value j. A and B correspond to empty streams which can be interpreted as a fatal deadlock.

- The (instantaneous) dependency between an expression and a variable is labelled p if the value of point i of e depends only on the value of point i of X (point-to-point dependency).
- The dependency is labelled t if a point i from e depends on the value of all points of X (total dependency).
- The dependency is labelled $+$ if the value of point i depends on the values of point j of X with $j < i$.

In the sequencing graph, the cycle with an edge of type t or no edge of type $+$ are dead cycles. The webs defined in those cycles have always undefined values. The remaining cycles (with edges $+$ and no edge t) correspond to spatial recursive expression requiring a sequential implementation. An expression not appearing in a cycle is a data-parallel expression. It can be computed as soon as its ancestors have been computed. Here, we are dealing with recursive definitions of collections but see [38] for a similar approach which handles recursive streams and [39] for recursive lists.

In fact, the complete processing of the sequencing graph is a bit more complicated. We made the assumption that the calculus of the instantaneous value of $\$X$ does not depend on the instantaneous value of X, but the clock of $\$X$ depends on the clock of X (it is the same one, but the first tock). So, the sequencing graph might have instantaneous cycles between boolean expression representing clock expressions. The computation of this value is based on a finite fixed point computation in the lattice of clocks. One of the benefits of this approach, besides being fully static, is that it allows us to detect the expression that will remain constant (we can therefore optimize the generated code), or that will never produce any computation and generates tasks in dead-lock (that might be a programming error).

Using the sequencing graph of the tasks as an approximation of the true dependencies graph,

we might detect as incorrect some programmes with an effective value. With some refinements of the method, it is possible to handle additional programmes. Anyway, the sequencing graph method effectively schedules any collections defined as the first n values of a primitive recursive function, which represents a large class of arrays.

In fact, this corresponds to the use of a prefix-ordered domain on vectors, instead of a more general Scott domain. The use of a Scott order on vectors (which identifies *de facto* vectors with functions from $[0,n]$ to some domain) allows more general recursive definition. This is at the expense of efficiency. For example, in the following 8_{12} programme computing the n first Fibonnaci numbers:

$$
\begin{aligned}
fib[n] = \text{if } iota &= = 0\\
\text{then } &1\\
\text{else } &\text{if } iota = = 2\\
&\text{then } 1\\
&\text{else } (1 \# fib{:}[n-1]) + (\{1,1\} \# fib{:}[n-2])
\end{aligned}
$$

the time-complexity of the evaluation process remains linear with n because we know that we can compute the element value in a strict ascending order (in comparison, the time-complexity of the *functional* evaluation of *fib* is exponential, but can be simulated in polynomial time by memoization).

In the current compiler, the sequencing graph method is used to determine if the evaluation of the vector element can be done in parallel, in a strict ascending order, or in a strict descending order.

3.4. *The data-flow distribution and scheduling*

After the scheduling inference, the compiler is able to distribute the tasks on to the PEs of a target architecture and to choose for every PE a scheduling compatible with the sequencing graph. To solve this problem, we limit ourselves to *cyclic scheduling*. In our case, such a scheduling is the repetition by the PEs of some code named *pattern*. The pattern corresponds to the computation of the values of a web for one tick. The last operation of the compiler is therefore to generate such a pattern from the scheduling constraints.

To generate a pattern, the compiler associates to every task a rectangular area in a Gantt chart (a *time* × *space*). The width of the rectangle corresponds to the execution time of the task and its height to the number of PE ideally required for a fully parallel execution of the task (cf. Fig. 6). For example, if the task corresponds to the data-parallel addition of two arrays of 100 elements, the height of the associated rectangle will be 100.

With the representation, the problem of the optimal distribution and the minimal scheduling of the tasks is to find a distribution of the rectangles that will minimize the makespan and that is

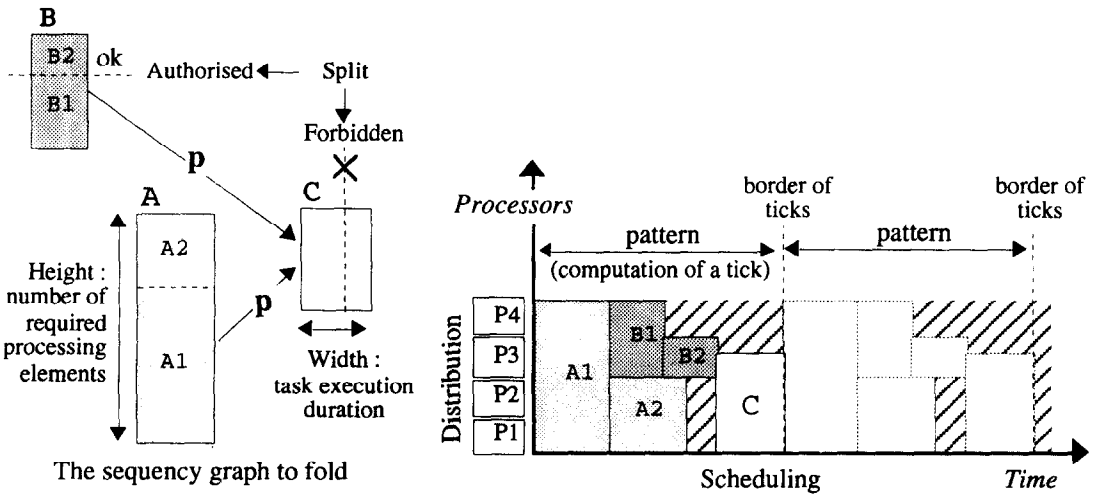

Fig. 6. Scheduling and distribution of a sequencing graph using a two-dimensional bin-packing method.

bound in height by the number of PEs in the architecture. Some very efficient heuristics exist for this problem known under the name "bin-packing" in two dimensions (which is NP-complete in the general case [40]).

At the moment, we are testing a greedy strategy [41, 42] consisting of placing as soon as possible the largest ready task on the critical path. A task becomes ready at the time when all the tasks from which it depends are done, time plus the communication time needed to transfer the data between PEs. If more than one task is available at the same time, an additional criterion is given to choose which one has to be taken first (for example, a task being on the critical path).

If the width of the chosen task is bigger than the number of available PE, we "split" the task in two pieces. The first one is scheduled and the other one is put back in the pool of available tasks (to be scheduled and distributed later). We only admit the split in the horizontal direction (cf. Fig. 6). In fact, that is possible because a data-parallel task requiring n PEs corresponds to n independent scalar tasks. Vertical split corresponds to pre-emptive scheduling.

A well-known result in [43] can be used to bound the worst case performance of this strategy. It guarantees the good quality of the heuristic used here.

4. CONCLUSIONS

The current compiler is written in C and in an ML dialect. It generates a code for a virtual SIMD machine implemented on a UNIX workstation. However, all the compiler phases assume a full MIMD execution model and we are working on the MIMD code generation. Evaluation of webs with dynamic structure is done through a sequential interpreter.

It is interesting to evaluate the quality of the sequential **C** code to estimate the overhead induced by the high-level form of the language. This comparison was done on the example of heat diffusion (cf. Section 2.5.1) against a hand-coded **C** programme (the parameters are the size of the rod which varies from 10 to 10^5 and the number of iterations from 100 to 10^7). The ratio between the two programmes is less than 2 in favour of the **C** programme for any parameters. However, the code generated from the $8_{1/2}$ programme is not optimized and especially the concatenation involves copying instead of sharing and communications are not translated into vector shifts. Optimizing by hand the communications lowers the ratio to 1.3 which proves the efficiency of our compilation scheme (more results are given in [44]).

As a matter of fact, our concept of collection relies on nested vectors. Nested vectors differ in many ways from the multidimensional arrays generally used in space–time simulations. For example, assuming a row–column representation of a two-dimensional array by a two-nested vector, it is not possible to define an evaluation process propagating along the diagonal. This is because of the prefix or suffix ordering of vector-domains. More generally, the problem is to define the neighbourhood of a collection element and to enable arbitrary moves from neighbour to neighbour. A possible answer relies on the extension of collection on a rich structure based on groups [45].

Acknowledgements—The author wishes to thank Jean-Louis Giavitto, Jean-Paul Sansonnet, Dominique De Vito, Abderrahmane Mahiout, Dan Truong, Laurence Cathala and the anonymous reviewers for their constructive comments.

REFERENCES

1. Flynn, M. J. Some computers organizations and their effectiveness. *IEEE Trans. on Computers* **C21**: 948–960; 1972.
2. Sipelstein, J. and Blelloch, G. E. Collection-oriented languages. *Proc. of the IEEE* **79**(4): 504–523.
3. Bânatre, J.-P., Coutant, A. and Le Metayer, D. A. Parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems* **4**: 133–144; 1988.
4. Giavitto, J.-L. A synchronous data-flow language for massively parallel computer. In *Proc. of Int. Conf. on Parallel Computing* (*ParCo*'91) (Edited by Evans, D. J., Joubert, G. R. and Liddell, H.), pp. 391–397, London 3–6 September 1991. Amsterdam: North-Holland; 1991.
5. Siegel, H., Siegel, L., Kemmerer, F., Mueller, P. Jr, Smalley, H. Jr and Smith, D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers* **C-30**(12): 934–947; 1981.
6. Cornu-Emieux, R., Mazaré, G. and Objois, P. A VLSI asynchronous cellular array to accelerate logical simulations. In *Proc. of the 30th Midwest International Symposium on Circuit and Systems*; 1987.
7. Koren, I. and Mendelson, B. A data-driven VLSI array for arbitrary algorithms. *IEEE Computers*, 30–43; October 1988.
8. Cappello, F., Béchennec, J.-L., Delaplace, F., Germain, C., Giavitto, J.-L., Neri, V. and Etiemble, D. Balanced

distributed memory parallel computers. In *Int. Conf. on Parallel Processing, St Charles, Ill.*, pp. 72–76. Boca Raton, FL: CRC Press; 1993.
9. Steele, G. Making asynchronous parallelism safe for the world. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pp. 218–231. San Francisco, January 1990. San Francisco: ACM Press; 1990.
10. Davis, A. L. and Keller, R. M. Data-flow graphs. *Computer*, pp. 26–41; February 1982.
11. Skillicorn, D. Architecture-independent parallel computation. *Computers*, 38–49; December 1990.
12. Gajski, D. D., Padua, D. A., Kuck, D. J. and Kuhn, R. H. A second opinion on data flow machines and languages. *IEEE Computer*, 489–500; February 1982.
13. Waters, R. C. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems* **13**(1): 52–98; January 1991.
14. Leiserson, C. and Saxe, J. Optimizing synchronous systems. *Journal of VLSI and Computer Systems* **1**(1): 41–67; 1983.
15. Giavitto, J.-L., Germain, C. and Fowler, J. OAL: an implementation of an actor language on a massively parallel message-passing architecture. In *2nd European Distributed Memory Computing Conf. (EDMCC2)*, volume 492 of *LNCS*, Münich, 22–24 April 1991. Berlin: Springer-Verlag; 1991.
16. Michel, O., Giavitto, J.-L. and Sansonnet, J.-P. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21–23 September, 1994. Moscow: Office of Naval Research USA & Russian Basic Research Foundation; 1994.
17. Blelloch, G. E. and Sabot, G. W. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing* **8**: 119–134; 1990.
18. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. and Schonberg, E. *Programming with Sets: and Introduction to SETL*. Berlin: Springer-Verlag; 1986.
19. Jayaraman, B. Implementation of subset-equational program. *Journal of Logic Programming* **12**: 299–324; April 1992.
20. Thinking Machines Corporation, Cambridge, MA. *The Essential *Lisp Manual*; 1986.
21. Tofooli, T. and Margolus, N. *Cellular Automata Machine*. Cambridge, MA: MIT Press; 1987.
22. Hillis, W. D. and Steele, G. L. Data parallel algorithms. *Communication of the ACM* **29**(12): 1170–1183; December 1986.
23. Giavitto, J.-L. Typing geometries of homogeneous collection. In *2nd Int. Workshop on Array Manipulation (ATABLE)*. Montreal; 1992.
24. Sabot, G. W. *The Paralation Model: Architecture, Independent Parallel Programming*. Cambridge, MA: MIT Press; 1988.
25. Blelloch, G. E. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University; April 1993.
26. Blelloch, G. E. Scans primitive parallel operation. *IEEE Trans. on Computers* **38**(11): 1526–1538; November 1989.
27. Wadge, W. W. and Ashcroft, E. A. LUCID—a formal system for writing and proving programs. *SIAM Journal on Computing* **3**: 336–354; September 1976.
28. Denis, J. B. First version of a data flow procedure language. In *Proceedings of the Programming Symposium*, April 9–11 1974. Berlin: Springer-Verlag; 1974.
29. Arvind and Brock, J. D. Streams and managers. In *Proceedings of the 14th IBM Computer Science Symposium*. Berlin: Springer-Verlag; 1983.
30. Wadge, W. W. and Ashcroft, E. A. *Lucid, the Data Flow Programming Language*. London: Academic Press; 1985.
31. Caspi, P., Pilaud, D., Halbwachs, N. and Plaice, J. Lustre: a declarative language for programming synchronous systems. In *Fourteenth annual symposium on Principles of Programming languages*. Munich, Germany: ACM Press; January 1987.
32. Le Guernic, P., Benveniste, A., Bournai, P. and Gautier, T. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP* **34**(2): 362–374; 1986.
33. Hudak, P. and Wadler, P. Report on the programming language haskell version 1.1. *SIGPLAN Notices* **27**(5); April 1992.
34. Smith, D. A basis algorithm for finitely generated abelian groups. *Math. Algorithms* **1**(1): 13–26; January 1966.
35. Maes, P. A bottom-up mechanism for behavior selection in an artificial creature. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior* (Edited by Bradford). Cambridge, MA: MIT Press; 1991.
36. Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Reid-Miller, M., Sipelstein, J. and Zagha, M. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University; February 1993.
37. Leroy, X. *The Caml Light System Release* 0.6. INRIA; September 1993.
38. Wadge, W. W. An extensional treatment of dataflow deadlock. *Theoretical Computer Science* **13**(1): 3–15; 1981.
39. Sijtsma, B. A. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems* **11**(4): 633–649; October 1989.
40. Garey, M. R., Graham, R. L. and Johnson, D. S. Performance guarantees for scheduling algorithms. *Operational Research* **26**(1): 3–20; January–February 1978.
41. Mahiout, A., Giavitto, J.-L. and Sansonnet, J.-P. Distribution and scheduling data-parallel dataflow programs on massively parallel architectures. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.
42. Mahiout, A. Integrating the automatic mapping and scheduling for data-parallel dataflow applications on MIMD parallel architectures. In *Parallel Computing: Trends and Applications*, 19–22 September, Gent. Belgium, 1995. Amsterdam: Elsevier; 1995.
43. Hawang, J.-J., Chow, Y.-C., Angers, F. and Lee, C.-Y. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comp.* **18**(2): 244–257; April 1989.
44. De Vito, D. Compilation portable d'un langage déclaratif à flot de données synchrones, Juin 1994. Rapport de stage du DEA Informatique le l'Université de Paris-Sud; 1994.
45. Giavitto, J.-L., Michel, O. and Sansonnet, J.-P. Group based fields. In *Proceedings of the Parallel Symbolic Languages and Systems (PSLS'95)* (Edited by Halstead, R. H., Takayasu, I. and Quinnec, C.), volume 1068 of LNCS, p. 204–215. Beaune (France), 2–4 October 1995. Springer-Verlag.

About the Author—OLIVIER MICHEL, born in 1969, received his Masters degree from the University Paris VI, Pierre et Marie Curie in 1992. He is currently a Ph.D. student at the University Paris XI in L.R.I. Since 1992, he worked on the extension of $8_{1.2}$, a declarative data-parallel language. His research interests include the design and implementation of new data structures for simulations with a special interest in the representation of growing processes.