A data-parallel declarative language for the simulation of large dynamical systems and its compilation

Olivier Michel, Jean-Louis Giavitto, Jean-Paul Sansonnet.

LRI - U.R.A. 410 du CNRS.
Bât 490 - Université Paris Sud
F-91405 Orsay cedex - France
email: `michel@lri.lri.fr`

Abstract: 81/2 is a declarative data-parallel language designed for the simulation of large dynamical systems. Such simulations are of growing importance and they requires more and more computing power. In consequence, 81/2 introduces a new entity, the web, that combines features of collection-oriented and data-flow language to express data-, stream- and control-parallelism. In this paper, we present the language 81/2, some examples of dynamical systems programmed in 81/2 and we describe the compilation process of a 81/2 programme.

Key-words: data-parallelism, collection-oriented languages, declarative languages, synchronous data-flow languages, simulation of dynamical systems.

I. Introduction

Nowadays, simulation of large dynamical systems represents the majority of supercomputers applications. In this article, a dynamical system refers as any application that modelizes space-time phenomena. Three usual examples are:

- numerical resolution of *partial differential equations* [1] describing continuously evolving systems;
- *discrete event simulation* where the system transitions occur when discrete events happen [2];
- *hybrid systems* simulation which are dynamical systems whose phase space involves continuous and discrete components [3] [4].

A language for the modelisation and simulation of such systems requires the following features:

- being *high-level:* most of the simulation processes are developed using low-level languages leading the designer to spend a lot of time in debugging and tuning its programs [5]. To let the designer concentrate on the modelling aspects, we advocate the use of a high-level language, where the entities expressed are close to the concepts used in the application problem [6] and hiding the low-level implementation details.
- being *parallel:* simulation of large dynamical systems (global atmospheric circulation, molecular dynamics,… [7]) as described in the *Grand Challenge Project* [8] requires a huge amount of computations. Typical requirements can only be satisfied using *massively parallel computers*.

Consequently, we propose 81/2, an experimental high-level parallel language devoted to the simulation of large dynamical systems.

II. The parallel language 81/2

The simulation of a system consists in computing the trajectories of the variables describing the system. By *variable trajectory*, we mean the successive values in time of a variable. This notion fits well with the concept of *stream* [9]. The 81/2 language focuses on models described by a set of functional relationships between stream variables, i.e. an 81/2 program takes the following form:

$$
(1) \qquad \begin{cases} x = f_1\,(x, y, \ldots) \\ y = f_2\,(x, y, \ldots) \\ \ldots \end{cases}
$$

where `x, y, …` denotes streams. Because relations like (1) often apply between large homogeneous sets of variables, (e.g. for multi-dimensional valued variables or when continuous variables are discretized), the value of a variable at a given time is often an array. These sets of values must be managed as a whole. Moreover the elements of these arrays are distributed among the processors according to the data-parallel paradigm [10], therefore we speak of *collection*. A *stream* of *collections* is named a *web* in 81/2.

II.1. The concept of synchronous stream

[11] have considered, to simplify the formal treatment of a program, that a variable denotes an infinite sequence of values rather than a single value. This approach takes advantage of representing iterations in a « mathematically respectable way » [9] and to quote [12]: « series expressions are to loops as structured control constructs are to gotos ». Such infinite sequences are called *streams* and are manipulated as a whole, using filters, transductors, etc.

Consider the following nested `C` loops:

```
for (i = 0; i < 3; i++)              i ⇒   < 0;       1;       2 <
  for (j = 0; j < 2; j++)            j ⇒   < 0;  1;  0;  1;  0;  1 <
          k = i+j;                   k ⇒   < 0;  1;  1;  2;  2;  3 <
```

We observe the successive values of the variables `i`, `j` and `k` to build the corresponding streams These streams do not have the same number of elements, although they are issued from the same time interval of observation, because they are related to the change of a memory element (Cf. fig 1). If we consider the element-wise addition of stream `i` and `j` (as in Lucid for example), the result is not the stream `k`.

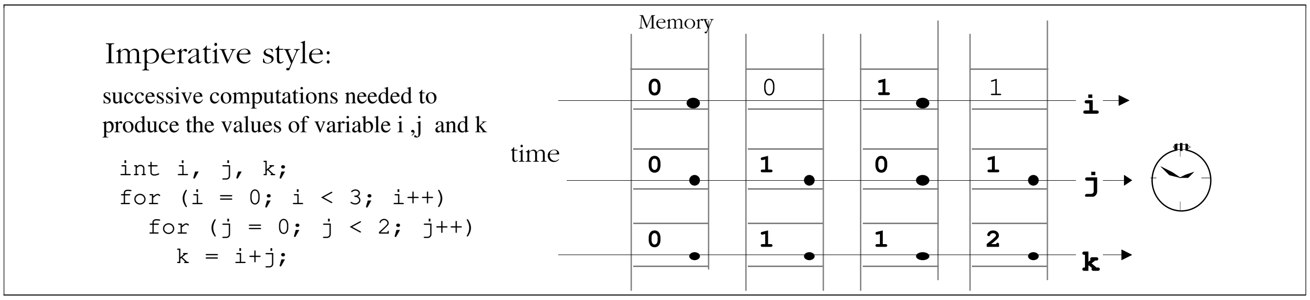

Figure 1: Successive values written in memory for two nested loops indexes

In consequence, we need a more sophisticated data-structure to take into account the relative "tempo" between streams of successive values. A *synchronous stream* introduces a global clock to handle the temporal succession. An instant of this clock is named a *tick*. The instants where the stream possibly changes its value are called the *tocks* of the stream. Synchronous streams are very different from a linear storage of data and the values of a synchronous stream vanish and leave room to the next values. At some tick that is not a tock, the stream value can be used and its value is the value of the previous tock, if it exists, else it is undefined. Here we consider 4 classes of operators to build a stream (Cf. the table 1):

- *Stream constant.* Scalar constants are promoted to stream constant with only one tock at tick 0. In addition, we use the constant `Clock n`, which always holds a true value with tocks randomly distributed with a density of `1/n` over the ticks.
- *Extension of a scalar operator.* Let ⊕ a scalar binary operator (i.e. an operator defined on the values of a stream), we define the stream `x⊕y` from the streams `x` and `y`. The expression `x⊕y` has a tock (i.e. is susceptible to change its value) if `x` has a tock, *or* `y` has a tock, *as soon as* both `x` and `y` are defined. The value of `x⊕y` at some tock is the result of the combination by ⊕ of the values of `x` and `y` at the corresponding tick (that is, ⊕ acts in an element-wise fashion on the stream elements).
- *Delay.* The operator `$` is used to shift an entire stream giving access to the previous value. The tock of `$x` are the same of the tock of `x`, *except* for the first one. The value of `$x` at some tock is the value of `x` at the previous tock.

- *Sampling operator*. We present only one sampling operator: `x when y`. The values of `x when y` are those of `x` sampled at the tock where `x` takes a true value, as soon as `x` and `y` are defined.

Table 1: Examples of stream expressions. A column in this table represents a tick. A non empty column is a tock.

input	`i`	0		1		2	
streams	`j`	0	1	0	1	0	1
	`i+j`	0	1	1	2	2	3
output	`$i`			0		1	
streams	`1`	1					
	`i==1`	false		true		false	
	`j when (i==1)`			1			

A declarative synchronous stream programme is a set of equations: « *`variable = stream expression`* ». In addition, stream equation may be quantified by some temporal predicates stating its validity domain. For example,

```
{ cpt@0 = 33, cpt = $cpt + 1 when Clock 1 }
```

is a set of two equations defining the stream `cpt`. The equation quantified by « `@0` » defines the first *tock* of `cpt` (tocks are numbered from 0). The other equation, without quantification, specifies the default definition of `cpt`. `Cpt` is a counter starting at `33`, and increasing by `1` at the rate of `Clock 1`.

II.2. The concept of collection

A *collection* is a data structure that represents a set of elements *as a whole* [13]. Several kinds of aggregation structure exist: *set* in the SETL [14], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [15] or even *finite discrete space* in Cellular Automata [16]. Data-parallelism is naturally expressed in terms of collections [17], [10]. From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

We consider here collections that are *ordered* sets of elements. Four kinds of function application can be defined on such data:

apply : *f(c1, …, cp)* : (collection^p → X) × collection^p → X

alpha **^**: *f^(c1, …, cp)* : (scalar^p → scalar) × collection^p → collection

beta ****: *f\c* : (scalar^2 → scalar) × collection → scalar

scan ****: *f\\c* : (scalar^2 → scalar) × collection → collection

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

The first function application mechanism is the standard one: collections are considered as a whole and function application acts as usual. 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. The process of promoting a scalar to a collection function is known as an *alpha-denotation* in APL. The third type of function application is the *beta-reduction*. Beta-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 beta-reduction with the *min* function gives the minimal element of a collection. The scan application mode is similar to the beta-reduction but returns the collection of all partial results. See [18] for a programming style based on scan. Beta-reduction and scan can be performed in $O(\log_2(n))$ steps on SIMD architecture, where `n` is the number of elements in the collection, if there is enough PEs.

Some additional operators are defined. An element of a collection, also called a *point* in 8½, is accessed through an index. The expression `T.n` where `n` is 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). The *if* operator extends the conditional construct to collection:

```
Q = if B then T else F fi
```

must be taken pointwise in space, that is, for each point n: `Q.n = if B.n then T.n else F.n fi` .

Geometric operators change the *geometry* of a collection, i.e. its structure. The geometry of a scalar collection 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 [19] and NESL [20]. The geometry of the collection is the hierarchical structure of point values. The first geometric operation consists in *packing* some webs together. The result is a *system*:

```
T = {a, b}
```

In the previous definition, `a` and `b` are collections resulting in a nested collection `T`. Elements of a system may also be named, achieving the idea of environment (a binding between names and values). 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 environment. The following example concatenates the two collections `A` and `B`:

```
A = {a, b};  B = {c, d};  A # B ⇒ {a, b, c, d}
ferarri = car # {color = red} ⇒{velocity = 5, consumption = 10, color = red}
```

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

```
Source = {a, b, c, d, e}
target ={1, 3, {0, 4}}
Source(target) ⇒ {b, c, {a, e}}
```

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 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.

III. Examples of 81/2 programmes

III.a. 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 equation of the diffusion takes the following form:

$$\frac{\partial u}{\partial t} = k \cdot \frac{\partial^2 u}{\partial x^2}$$

which is discretized as [1]:

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

The corresponding 81/2 programme is:

```
start  = some initial temperature distribution;
begin = 0;
end   = 0;

U@0 = start;
U   = begin # inside # end;

float inside = 0.4*left(pU) + 0.2*middle(pU) + 0.4*right(pU);
pU           = $U when Clock;
```

The `$` (resp. `left`, `middle` and `right`) operator represents a shift in the past (resp. in space). The `#` operator represents the concatenation; `when` is a sampling operator and `clock` the time discretization (Cf. Figure 2).

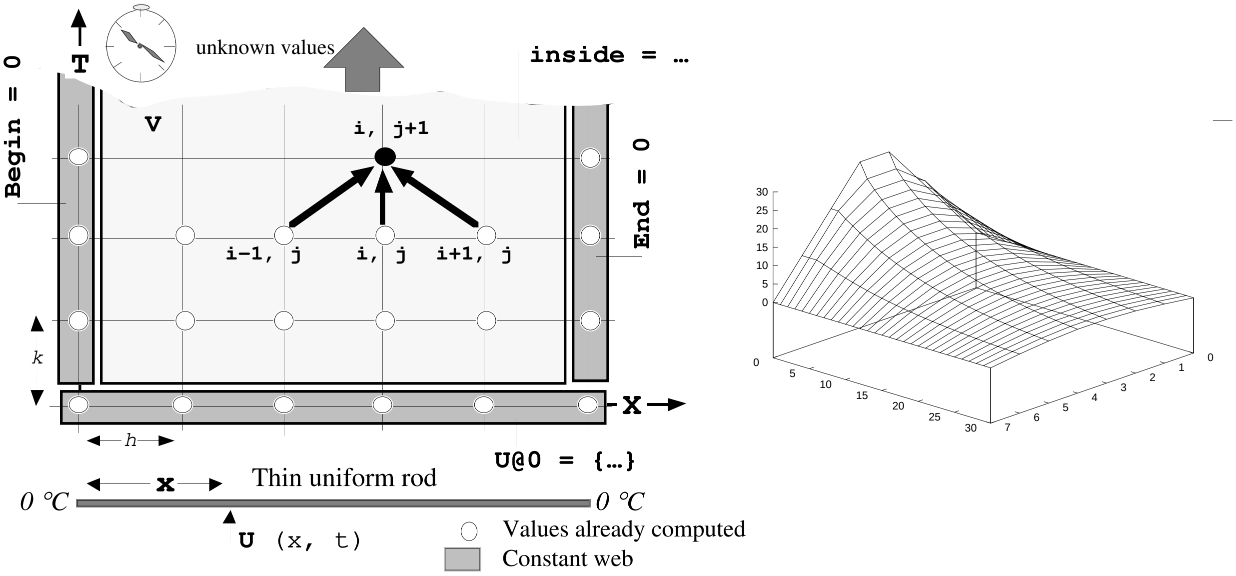

Figure 2: diffusion of heat in a thin uniform rod

III.b. The simulation of a reactive system

Here is an example of an hybrid dynamical system, a "wlumf" which is a "creature" whose behaviour (eating) is triggered by the level of some continuous internal state (Cf. [21] for such model in ethological simulation):

```
System environment = { boolean FOOD = Random };;
System wlumf = {  i_am_hungry@0 = false;
                  i_am_hungry   = (glucose < 3);

                  glycaemia@0 = 6;
                  glycaemia   =  if i_am_eating
                                    then 10
                                    else max(0, $glucose - 1) when Clock
                                 fi;

                  i_am_eating@0 = false;
                  i_am_eating   = $i_am_hungry && environment.FOOD };;`
```

The result of an execution is given in Figure 3.

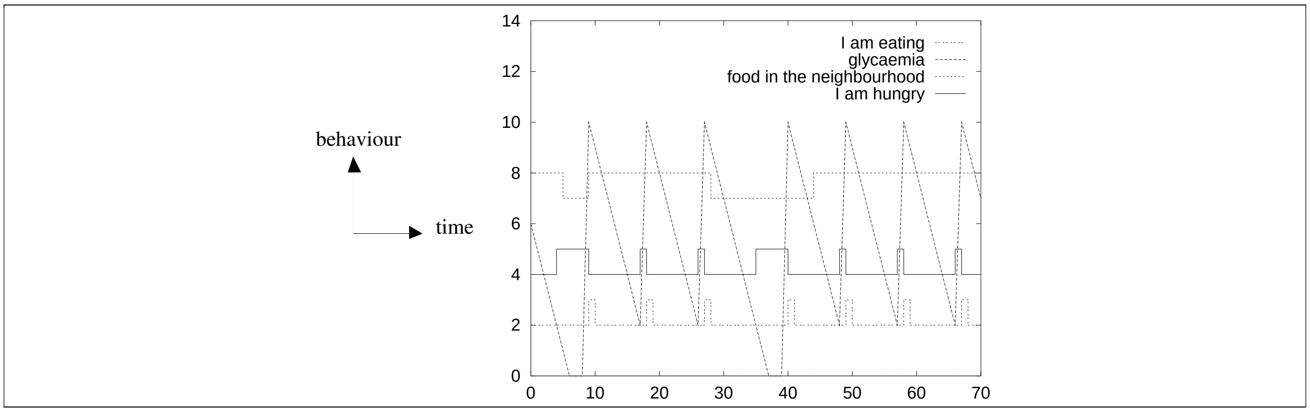

Figure 3: behaviour of an hybrid dynamical system.

III.c. Iterations of the logistic map

The logistic equation is: `f (x) = k*x*(1-x)`. The following 81/2 programme computes a `map` composed by the concatenation of iterations of the logistic map for parameters `k` starting from 1.2 and step 0.1, and for a discretization `x` of the initial interval [0, 1]:

```
start = 0.05 * '20;
k@0   = 1.2;
k     = $k + 0.1 when Clock;
separate[5] = 0;
map = ((start # separate) # k*map*(1-map)):[125];
```

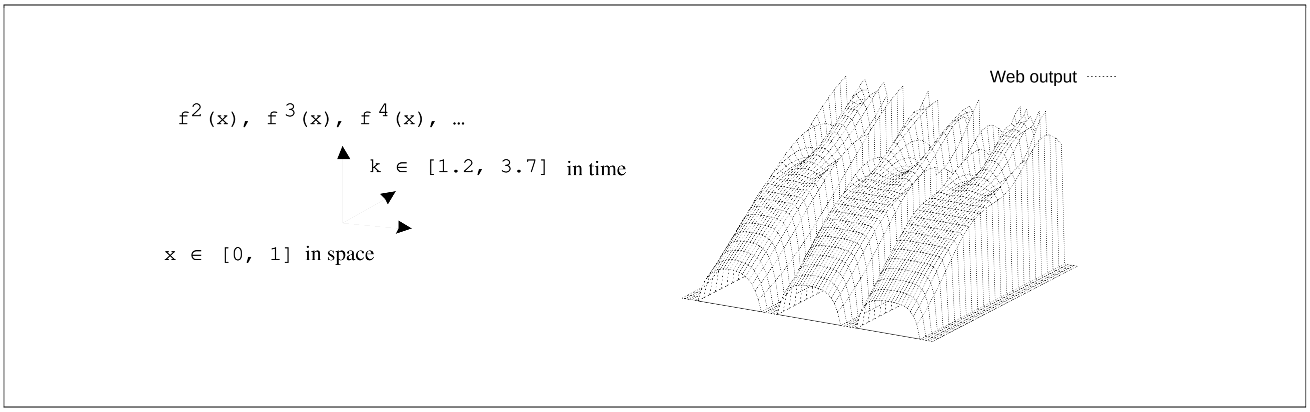

Figure 4: iteration of the logistic equation

IV. Compilation

The compilation process is composed of four phases :

- The *type inference* and *analysis* computes the shape of the collections involved in a programme [22].
- The *dependency analysis* builds a partial evaluation order between the expressions. This step is trivial for scalar data-flow programmes but not obvious when dealing with array. For example, the programme `a = 0 # (a + 1): [10]` is a recursive definition of an array whose *i*th element has the value *i*. The expressions involved by this definition cannot be evaluated in parallel and must be sequentially processed.
- The *clock computation:* 81/2 advocates a static execution model. That is, the triggering of parallel activities is inferred at compile-time rather than at run-time. This is done by synthesising for each expression, through an abstract interpretation, a predicate *Clock(t)* which indicates at time *t*, whether the evaluation has to be done or not. The static evaluation model answers the drawbacks sustained by Gajsky [23] against the standard dynamic data- or demand- driven execution models.
- The *mapping* and *scheduling:* the mapping and scheduling phase of the compilation consists in distributing the evaluation task over the parallel processors. The clock predicates ensure an iterative scheduling of the tasks [24].

IV.1. The clock calculus

The computation of a web value in a 81/2 programme corresponds to the computation of every successive collections that are solution. The value of a stream at a given time is a collection (of scalar values) named *instantaneous value* of the web.

The *clock* of a web `X` is a boolean stream holding the value `true` if tick is a tock for the web X, else holding a `false` value. Each tick is a tock for a clock. The clock calculus of a web is needed to decide whether the computation of an instantaneous value has to take place at some tick or not. Let `x` denotes the instantaneous value of `X`, and `clock(X)` the instantaneous value of the clock associated to `X`. Every definition

```
X = f(Y)
```

in the initial programme, is translated in:

```
x = if clock(X) then f(y)                                        (1)
```

This expression is synthetised by induction on the structure of the definition of X. For example:

```
clock (A when B) = b ∧ clock(B)
```

This transformation produces a normal form from the original web definition. Roughly, the compiler will generate for any expression of the programme, a task executing the process shown in (1). It is still necessary to compute the dependency between the tasks to determine their relative order of activation.

IV.2. The code generation

Three different kinds of code generation have been thought of and are to be generated:

- code for a virtual SIMD machine written in C very close to CVL [25] and adaptated to the execution on a SIMD or vectorial architecture;
- a sequential standalone C code using no dynamical memory allocation nor function-call stack;
- a full MIMD code.

At the moment, the compiler written in C [26] and in an ML dialect [27], generates a code for a virtual SIMD machine implemented on a UNIX workstation and the C code. However, all the compiler phases assume a full MIMD target architecture and we are working on the MIMD code generation.

V. Related works and conclusions

V.1. Related works

The transformation of stream expressions into loops is extensively studied in [12]. The expressions considered do not allow recursive definitions of streams. Our proposition handles this important extension as well as “off-line cycle” expressions. We share however the preorder restriction, i.e. the succession of stream elements must be processed in time ascending order (this is not the case in a language like LUCID). We focus also on unbounded stream and therefore we do not consider operations like *catenating* (Cf. [12]).

[24] consider the static scheduling of a class of data-flow graphs used in digital signal processing. The translation of a (recursive) stream definition into a (cyclic) data-flow graph is straightforward. Their propositions apply but are limited to the subset of “on-line” program [28]. This restriction excludes the sampling operator presented above (an operator similar to the T-gate of [29]) and requires the presence of at least one delay on each cycle of the data-flow graph.

Compiled execution model for data-flow are also investigated in the domain of real-time programming LUSTRE [30], SIGNAL [31] and in the domain of signal processing [32]. The reactive languages LUSTRE and SIGNAL include the full set of stream operators we use and allow recursive definitions. However in real-time programming, the interests are focussed on the synchronization constraints: the handling of time is quite different and data-parallelism is out of considerations. They consider only the compilation of “strongly synchronous” programs [33] which correspond to the combination of streams “progressing at the same rate” or “with the same clock”: a property called “isochrony”. In accordance with the specificity of their applications domain, real-time programming, LUSTRE and SIGNAL do not allow “heterochronous” expressions and sophisticated techniques [34] have been developed to check and reject heterochronous programs. Similar approaches exists in the field of systolic programming: Crystal [35], ALPHA [36] or for asynchronous cellular array [37]. But this approaches lack the concept of collection (element accesses are explicit).

V.2. Conclusions

The 81/2 language is an experimental high-level parallel language combining the advantage of the SIMD and MIMD execution model through the embedding of collections in a synchronous and static data-flow scheme resulting in an hybrid SPMD or MSIMD execution model. (see a companion paper [38] in this conference for the mapping and scheduling of data-parallel dataflow tasks on MIMD architectures) Using data- and implicit control- parallelism enables:

- the maximal expression of the parallelism inherent to an application;
- the effective use of the parallelism which implies the less implementation overhead (with respect to the target architecture);
- the hiding of communication costs by the computation of (parallel) independent activities.

All the examples stated in this paper have been processed by the existing compiler and the data visualisation done through Gnuplot [39]. We are currently working on the generation of MIMD code and the extension of the language to handle dynamically shaped webs.

VI. References

[1] G. D. Smith, Numerical solution of partial differential equations: finite difference methods, third edition, 1985, Oxford Applied Mathematics and Computing science series, Oxford University Press.

[2] R.Righter, J.Walrand, *Distributed simulation of discrete event systems*, Proc. of the IEEE, vol.77, n°1, Jan.1989.

[3] A. Benveniste, M. Le Borgne, P. Le Guernic, *Hybrid Systems: the SIGNAL approach*, LNCS 736, Springer-Verlag, 1993.

[4] A. Back, J. Gluckenheimer, M. Myers, *A Dynamical Simulation Facility for Hybrid Systems*, LNCS 736, Springer-Verlag, 1993.

[5] P. Fritzson, N. Andersson, *Generating Parallel Code from the Equations in the ObjectMath Programming Environments*, LNCS 734, Springer-Verlag, 1993.

[6] E. V. Zima, *Recurrent Relations and Speed-up of Computations using Computer Algebra Systems*, LNCS 721, Springer-Verlag 1992.

[7] G. S. Almasi, A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, 1989.

[8] *Grand Challenges - HPCC*, U.S. Off. of Sci. and Tech. Policy's Committee on Phy., Math., and Eng. Sci., 1991.

[9] W. W. Wadge and E.A. Ashcroft, *Lucid, the Data flow Programming Language*, Academic Press U.K., 1985.

[10] J. M. Sipelstein, G. E. Blelloch, *Collection-oriented languages*, proc. of the IEEE, Vol. 79, N° 4, april 1991.

[11] L.G. Tesler, H.J. Enea, *A language design for concurrent processes*, AFIPS Conference Proceedings, vol. 32, pp 403-408, 1968

[12] Richard C. Waters, *Automatic Transformation of Series Expressions into Loops*, ACM Trans. on programming Languages and Systems, Vol. 13, N°1, Jannuary 1991, pp 52-98.

[13] G. E. Blelloch and G. W. Sabot, *Compiling Collection-oriented Languages onto Massively Parallel Computers*, Journal of Parallel and Distributed Computing, 8, 119-134 (1990)

[14] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, Programming with sets: An introduction to SETL, Springer-Verlag 1986.

[15] Thinking Machine Corporation, *The Essential *Lisp Manual*, Cambridge MA, 1986

[16] Tommaso Tofooli and Norman Margolus, *Cellular Automata Machine*, The MIT Press, Cambridge MA, 1987

[17] W. Daniel Hillis and Guy L. Steele JR, *Data Parallel Algorithms*, Communication of the ACM, vol. 29 N°12, December 1986.

[18] G. E. Blelloch, *Scans primitive Parallel Operations*, IEEE Transactions on Computers, Vol. 38, N° 11, November 1989.

[19] G. W. Sabot, *The Paralation Model: Architecture*, Independent Parallel Programming. MIT Press, Cambridge MA, 1988

[20] G. E. Blelloch, *NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129*, School of Computer Science, Carnegie Mellon University, April 1993.

[21] P. Maes, *A bottom-up mechanism for behavior selection in an artificial creature*, proc. of the first int. conf. on simulation of adaptative behavior, Bradford Book, MIT Press, 1991.

[22] J.-L. Giavitto, *Typing geometry of Homogeneous Collection*, Proc. of the 2nd Int. Workshop on array, ATABLE-92, Montréal, Jun 1992.

[23] D.D. Gajski, D.A. Padua and D.J. Kuck, R. H. Kuhn, *A second opinion on data flow machines and languages*, IEEE Computer, feb 1982.

[24] K. K. Parhi, D. G. Messerschmitt, *Static rate-optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding*, IEEE Trans. on Comp., Vol 40, N°2, February 1991.

[25] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, M. Zagha, *CVL: A C Vector libray. Technical Report CMU-CS-93-114*, School of Computer Science, Carnegie Mellon University, 1993.

[26] B. W. Kernighan, D. M. Ritchie, *The C Programming language*, Englewood Cliffs, NJ, Prentice Hall 1978.

[27] X. Leroy, *The Caml Light system release 0.6 - Documentation and users manual*, INRIA, Sep 1993.

[28] A. Aho, J. Hopcroft, J. Ullman, The design and analysis of computer algorithms, Addison-Wesley, 1974.

[29] Jack B. Dennis, *First Version of a Data Flow Procedure Language*, proc. of the Programming Symposium, Paris, April 9-11, 1974, LNCS 19, Springer.

[30] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, *Lustre: a declarative language for programming synchronous systems*, in 14th ACM Symposium on Principles of Programming Languages, january 1987.

[31] P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, *SIGNAL, a dataflow oriented language for signal processing*, IEEE-ASSP, 34(2):362-374, 1986.

[32] Edward A. Lee, David G. Messerschmitt, *Static Scheduling of Synchronous Dataflow programs for Digital Signal Processing*, IEEE Trans. on Comp., Vol C-36, N°1, January 1987.

[33] G. Berry, A. Benveniste, *The synchronous approach to reactive and real-time systems*, Proc. of the IEEE, vol. 79, n° 9, September 1991.

[34] A. Benveniste, P. Le Guernic, *Hybrid dynamical systems theory and the SIGNAL language*, IEEE transactions on Automatic Control, vol. 35, n° 5, May 1990.

[35] M. C. Chen, *A Parallel Language and its compilation to multiprocessor machines or VLSI*, POPL'86, Florida, pp 131-139.

[36] C. Mauras, *Definition of Alpha: a language for Systolic Programmation*, INRIA research report N° 482, Jun 1989.

[37] R. Cornu-Emieux, G. Mazaré, P. Objois, *A VLSI asynchronous cellular array to accelerate logical simulations*, proc. of the 30th. Midwest Internationnal Symposium on Circuit and Systems, 1987.

[38] A. Mahiout, J.L. Giavitto, J.P. Sansonnet, *Distribution and scheduling data-parallel dataflow programs on massively parallel architectures*, Software for Multiprocessors and Supercomputers, September 21-23, 1994, Moscow

[39] T. Williams, C. Kelley, (C) GNUPLOT An Interactive Plotting Program, reference manual, Free Software Foundation, 1993