Extended Abstract Submitted to ICLP'94 Post-Conference workshop on Parallel and Data-Parallel Execution of Declarative Languages

Design and implementation of a declarative data-parallel language

Olivier Michel, Jean-Louis Giavitto.

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

Key-words: data-parallelism, declarative languages, collection-oriented languages, synchronous data-flow, data-distribution and scheduling.

I. Introduction

Figure 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 and to compare parallel languages constructs following of two criteria: the way they let the programmer express the control and the way they let him manipulate the data.

		Control		
		Declarative Languages *0 instruction counter*	**Sequential Languages** *1 instruction counter*	**Concurent Languages** *n instructions counters*
Data	**Scalar Languages**	Sisal, Id, Lau Actors FP	Fortran, C Pascal	Ada, Occam
	Collection Based Languages	$8\frac{1}{2}$ (APL, Gamma)	*LISP, HPF, CMFortran	CMFortran multi-threads

Figure 1: A classification of languages from the parallel construct point of view

The programmer has three choices to express the flow of computation:

- Not to express the 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 dependency exhibited in the programme.
- To express what has to be done sequentially: this is the classical sequential imperative execution model, where control structures build only one thread of computation.
- To 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 [SIP 91]. Examples of languages of that 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 only valid operation performed on an array is accessing one of its element.

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, …). However figure 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 (MSIMD model) and declarative languages (Gamma [BAN 88] or 8 1/2 [GIA 91]).

This paper describes the language 8 1/2, an embedding of data-parallelism in a declarative framework.

II. 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 those three notions.

II.a. The concept of collection

A *collection* is a data structure that represents a set of elements *as a whole* [BLE 90a]. Several kinds of aggregation structure exist: *set* in the SETL [SDB 86], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [TMC 86] or even *finite discrete space* in Cellular Automata [TOF 87]. Data-parallelism is naturally expressed in terms of collections [HIL 86], [SIP 91]. 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 : $(\text{collection}^p \to X) \times \text{collection}^p \to X$: $f(c_1, \ldots, c_p)$

alpha **^**: $(\text{scalar}^p \to \text{scalar}) \times \text{collection}^p \to \text{collection}$: $f\hat{}(c_1, \ldots, c_p)$

beta ****: $(\text{scalar}^2 \to \text{scalar}) \times \text{collection} \to \text{scalar}$: $f\backslash c$

scan ****: $(\text{scalar}^2 \to \text{scalar}) \times \text{collection} \to \text{collection}$: $f\backslash\backslash c$

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 [BLE 89] 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 8 1/2, 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 n^{th} 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 [SAB 88] and NESL [BLE 93a]. 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.

II.b. The concept of stream

LUCID [WAD 76] is one of the first programming languages defining equations between infinite sequences of values. This approach has the advantage of representing iterations in a "mathematically respectable way" [WAD 85] and to quote [WAT 91]: "… series expressions are to loops as structured control constructs are to gotos".

Streams can be manipulated as a whole, using filters, transducers… [ARV 83], and so can be visualised as collections. Nevertheless we carefully make a distinction between stream and collection: accessing elements of a stream is done in a sequential way and there is possibly an infinite number of stream elements. These characteristics make a clear distinction between stream's and collection's management. Moreover, unlike LUCID, we add two constraints that restrict the basic algebra over streams:

- *causality assumption*: stream elements have to be computed in the strict increasing order;
- *finite memory assumption*: a stream element has to be computed as a function of only a finite number of previous element.

These assumptions are also used in two real-time programming languages derived from LUCID: LUSTRE [CPH 87] and SIGNAL [LGB 86]. The purpose of these restrictions is to enable a static execution model. As a consequence, we consider only three classes of operations over streams: arithmetic, delay and sampling.

Arithmetic operations act on stream in an elementwise fashion, similarly to the *alpha-denotation* on collection (Cf. table 1). 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. However it maintains the two constraints: only past values are used to compute a new stream value and references to past values are relative to each element. So only the last w values of a stream are stored where w is a constant computable at compile time.

T	1	2	3	3	4	5	5	6	9
U	0	2	7	9	3	6	4	4	1
T + U	1	4	10	12	7	11	9	10	10
$U		0	2	7	9	3	6	4	4

Table 1: Examples of the addition of two streams and of a delayed stream;.

The last kind of stream operators is the sampling operator. The most general one is the trigger, which is very close to the *T*-gate in data-flow languages [DEN 74]. 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 (Cf. table 2).

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

Table 2: Example of a sampling expression.

Streams have to be thought as a *temporal succession* of values. Consequently, element of the same order in the stream may occur at different date, if the streams do not share the same "rhythm". In addition, the delay operator postpones the beginning of a stream and the sampling may induce "holes" in the stream succession. This makes the behaviour of streams very different of the behaviour of lists: they are not a linear storage of data and stream values vanish and leave room to the next values. To reason out the progression of streams, it is convenient to introduce an abstract global clock that measures the progression of time in the programme. The instants of this clock are called a **tick**. An instant where a stream value occurs, is called a **tock** (a clock is thought to make « tick-tock »). A tick is a column in the table 1 and 2; a tock is a non empty column. The tocks of the expression « `T+U` » are the ticks where the value of the expression may change, that is, a tick `t` is a tock if `t` is a tock of `T` or a tock of `U` *as soon as* both `T` and U are defined. 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. The value of a stream at some tick is the value of the stream if the tick is also a tock, or the value computed at the previous tock if it exists, else the special value nil. So, we can imagine a stream as the successive values appearing at some memory address, a tock corresponding to a write, and the memory address being able to be read at each tick.

II.c. Combining streams and collections into webs

A *web* is a *stream of collections* or in an equivalent manner, a *collection of streams*. This data structure is a multi-dimensional object that represents the successive values of a structured set of variables. More precisely, a web is made of an ordered, constant and finite set of *points*. The number of points is constant over the time. Each point carries its own value. Point values evolve synchronously; the rhythm of the changes is given by the clock of the web. The tocks of a web can be visualised as a *sub-clock* of a global clock. This global clock gives *all* events of interest for the simulated system.

Collection operations easily extend to operate on webs: the collection operation applies on the successive values of the web along the stream axis. In a same manner, stream operations apply on each value attached to the web points.

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   (1)

is a 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 "T") as well as a *relationship*, satisfied at each moment and for each point of T, A and B. Equation (1) holds only when A and B are valid, or in other words, T is defined only when A *and* B are defined. When T is defined, the values of T change as often as necessary to maintain the relationship, that is, when the values of A *or* the values of B change but no more. In addition, equation (1) implies that webs T, A and B have the same number of points. Figure 2 gives a three-dimensional representation of the equation (1) for webs with only one point. Running a 8 1/2 programme results in "weaving" the webs, that is, in enumerating the successive webs' values.

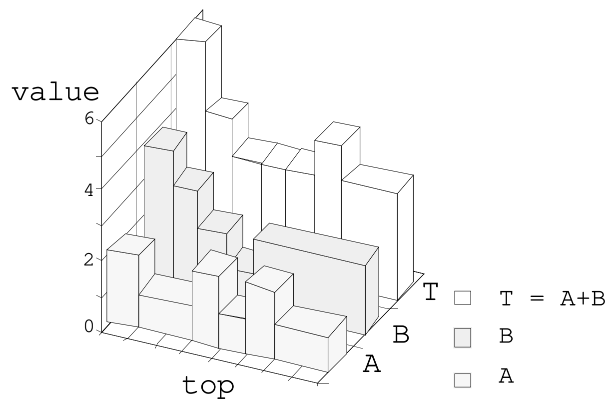

Figure 2: Webs draw in a <time, value, point> space. Webs A and B have only one point. Consequently the web T defined by T = A+B has only one point. The tock of T are the A and B ones.

A definition is recursive when the identifier in the left hand side appears also directly or indirectly in the right hand side. Two kinds of recursive definitions are possible. *Temporal recursion* allows the definition of the current value of a web using past values of it. For example, the definitions

    T@0 = 1, T = $T + 1 when Clock

specify a counter which starts at 1 and counts at the speed of the tocks of Clock. The « @0 » is a temporal predicate that quantifies the first equation and means "for the first tock only".

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:[9])

is a web with 10 elements such that iota.*i* is equal to *i*. The operator : [n] truncates a collection to n elements. The 8 1/2 compiler infers the correct size of iota and determines a correct execution order to compute the point values.

III.a. A more comprehensive example

Let H(x, t) represents the height at time t of a propagating wave along at the x axis. An explicit method of solution uses finite-difference approximation of the partial differential equations modelling the propagation on a mesh $(X_i = ih,\ T_j = jk)$ which discretizes the space of variables. The value of the unknown height $H_{i,\,j+2}$ at the $(i,\ j+2)^{th}$

mesh point can be computed using the known heights along the `(j+1)` th and `j`th time row (Cf. figure 3). Hence we can calculate the unknown pivotal values of U along the second time-row, in terms of known boundaries and initial values along T=0 and `T=1`, then the unknown pivotal values along the third time-row in terms of the calculated values, and so on.

The corresponding 8 1/2 programme is very easy to derives and corresponds simply to the description of initial values, boundary conditions and the specification of the computation. The stream aspect of a web coincides to the time axis while the collection point of view represents the discretization along the `x` axis.

```
PropagatingWawe = {
      start = sin('50 * (2*pi/49)),        /* or any other initial condition */
      begin = 0, end = 0,                  /* or any others boundary conditions */
      H@0 = start,
      H@1 = (0 # (start(left)+start(right) - 2*start(middle)) # 0) when Clock,
      H   = begin # inside # end,
      inside = …
}
```

The operator « `'` » is similar to the *iota* operator in APL, e.g. « `'50` » generates the web `{0, 1, …, 49}`. `begin` and `end` represent boundaries and « `@0` » and « `@1` » are initial conditions. The web `inside` is the interior of the space×time discretization domain. It is defined by:

```
inside = −0.5*($v(left) + $v(right)) + 0.55*$v(middle) + 0.45*$$v(middle)
```

`left()`, `middle()` and `right()` are functions enabling the shifting and the segmentation of a collection: they are not basic operations but build upon the selection primitive. We have only presented some of the primitive operators; a complete presentation can be found in [MIC 94b].

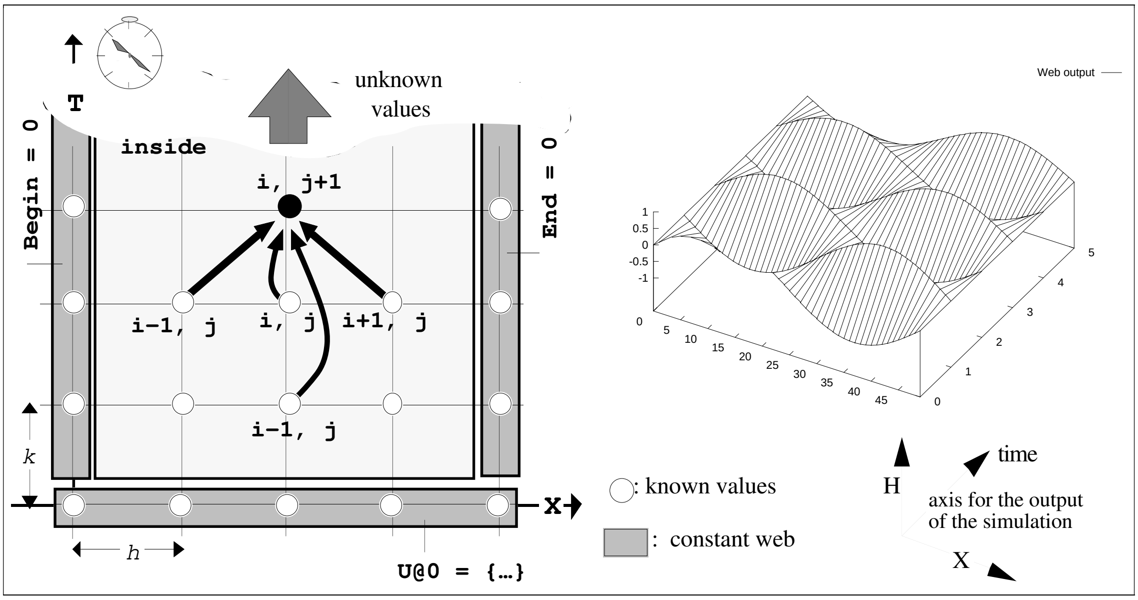

Figure 3: example of an equation defining a propagating wave

III. Webs and massive parallelism

III.b. The parallelism in 8 1/2

There is no explicit construct for parallelism in 8 1/2. Automatic extraction of parallel activities is simple because:

- The declarative form of the language makes easy to perform the dependence analysis between tasks and the subsequent exploitation of control parallelism.
- Web corrects some of the drawbacks sustained against the data-flow model [GAJ 82], mainly by adding some specific handling of arrays with a consistent concept of time.
- Web is a natural support of the data-parallelism and collection operations between webs naturally lead to a data-parallel implementation.
- Web introduces a natural support for the distribution of data.
- Transparential references allow a formal treatment of programmes and programmes optimisation by programme transformations are possible (Cf. for example [WAT 91], [LEI 83]).

So, the 8½ approach combines the advantage of the SIMD and MIMD execution model through the embedding of collections in a synchronous and static data-flow framework resulting in an hybrid SPMD or MSIMD execution model.

Embedding collection in a synchronous data-flow model combines the advantages of the synchronous and asynchronous parallel style. Consider for example the *actor model*: it proposes a minimal kernel to deal with control parallelism but handling of homogeneous set of data (like arrays) is definitively inefficient [GGF 91]. From another point of view, the handling of communications in a sequential data-parallel oriented languages, like *LISP, forbid overlapping communications and computations because there is only one thread of control.

Theses two examples show the interest of combining data and control parallelism. 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.

Nowadays new architectures appear [PAS 81] [CMO 87] [KB 88] [CBD 93] supporting a SPMD or a MSIMD execution model. This motivates the development of new programming paradigms able to express more than one kind of parallelism. Thus, to quote [STE 90]: "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". 8½ does not support all styles of parallel programming, but we argue that it combines advantages of the two approaches for a large class of interesting algorithms. A stream is a direct representation of a variable trajectory, a collection corresponds to multidimensional variable or to the discretization of a continuous parameter, and the declarative form of the language fits well with the functional description of a dynamical system. Thus we advocate the use of 8½ for the parallel simulation of dynamical system (e.g. deterministic discrete event systems [MIC 94]).

IV. Implementation of the 8½ compiler

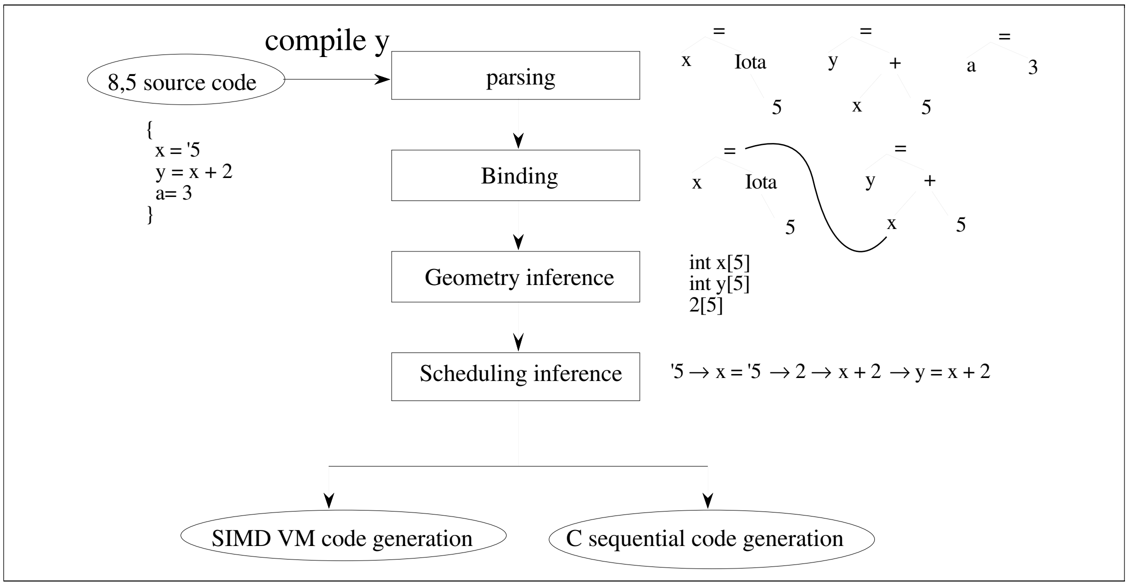

Figure 4: Block diagram of the compiler. Ellipses indicate code and rectangles, processing modules.

IV.a. The structure of the compiler

A high-level block diagram of the compiler is shown in Figure 4. The output can either be sequential C code or code for a virtual SIMD machine (in the line of CVL [BLE 93b]).

We describe briefly the various phases of the compiler written in a dialect of ML [LER 93]:

1. **Data-flow graph generation**: 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 ML version of `lex` and `yacc`.
2. **Binding**: the compilers binds lexically all variables. This phase is also responsible of expansion of functions, removal of unused definitions and the detection of undefined variables.
3. **Geometry inference**: the geometry of a web is inferred at compile time by the "geometric type system" (see [GIA 92]). Some incorrect programmes 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` ⇒ `<{0}; {0, 0}; {0, 0, 0, 0}; ...>` », every collection of the stream has two times more elements than the previous stream. This kind of programme implies dynamic memory allocation and dynamic load balancing and is rejected in the current version of the language.

4 **Scheduling inference**: to solve the 8 1/2 equations between webs, we have to extract the sequencing of the computations of the various right hand-sides, from the data flow graph. Once the scheduling of the instructions done, the compiler computes the memory storage required by a programme execution.

5 **Code generation**: the compiler generates standalone sequential C code running on work-stations or 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 some other dynamic runtime features.

IV.b. 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 tic 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 tic 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)                         (2)
```

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 (2). It is still necessary to compute the dependency between the tasks to determine their relative order of activation.

IV.c. The scheduling inference

The data-flow graph associated to a 81/2 programme is directly extracted from the programme in normal form. Unfortunately, this graph cannot be directly used to generate the task scheduling. Although the data-flow graph is the same as the dependency graph in the case of scalar data flow programme, this 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 from the corresponding point of `B`. On the other hand, the following programme that sums all elements of `B`:

```
A = +\ B
```

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

The data flow graph can be viewed as an approximation of the real dependency graph. This approximation is too rough; for example we cannot, on this basis, compile spatial recursive programmes. The work of the compiler is to annotate the data-flow graph to get a finer approximation of the dependency graph. The true graph of the dependencies cannot be explicitly build because it has as many nodes as points in the web of the programme (for example, in numerical computation matrix 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 dependency graph annotated in the following way (Cf. figure 5):

- An expression `e` depends from the web `X` if `x` appears syntactically in `e`. However, we remove the dependency of variables appearing in the scope of a delay: those dependencies correspond to a value in the past and not to an instantaneous value of a web.
- the (instantaneous) dependency between an expression and a variable is labelled `p` if the value of point `i` of `e` depends only of the value of point `i` of `X` (point to point dependency).
- The dependency is labelled `t` if a point `i` from `e` depends of the value of all points of `X` (total dependency).
- The dependency is labelled + if the value of point `i` depends of the values of point `j` of `X` with `j<i`.

In the sequencing graph, the cycle with an edge of type `t` or no edges 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 deal with recursive definitions of collections but see for a similar approache [WAD 81] which handles recursive streams and [SIJ 89] which handles recursive lists.

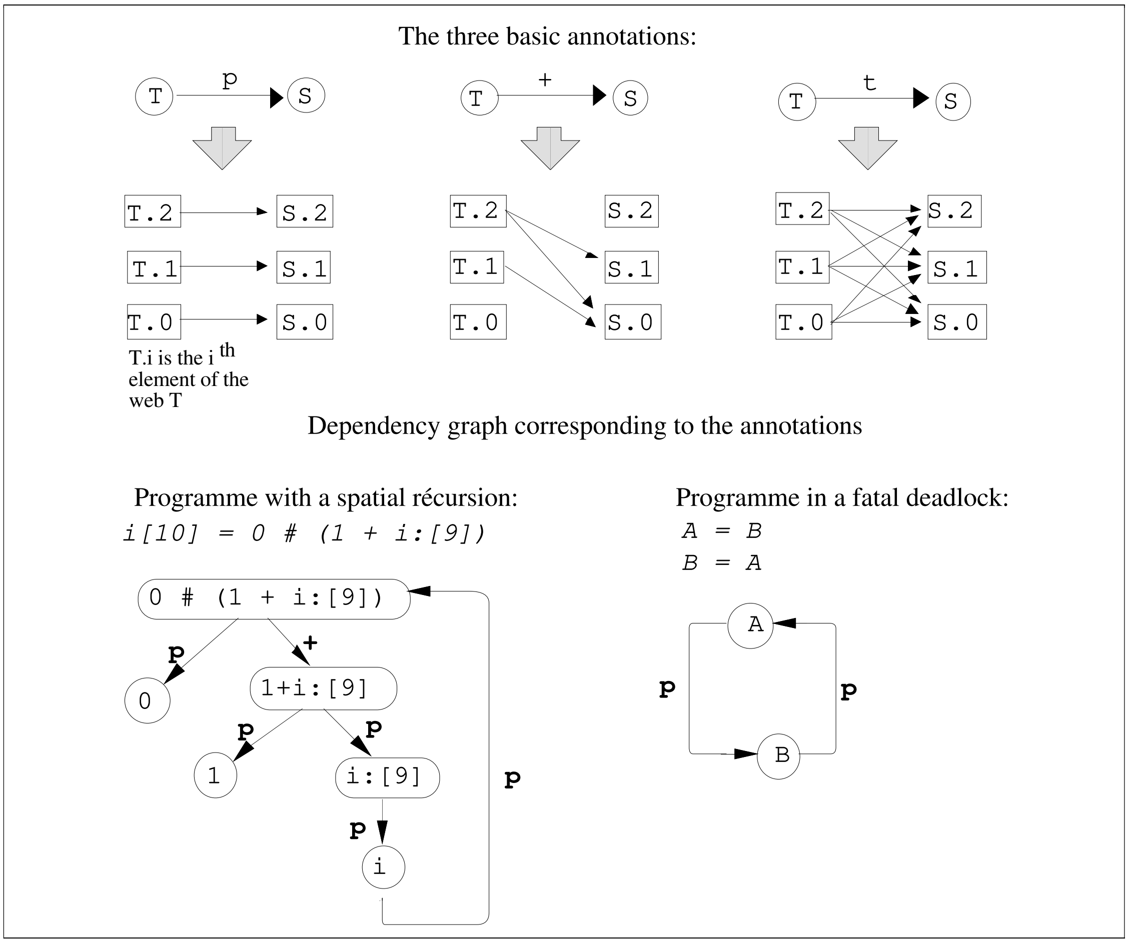

Figure 5: Representation of the three possible annotations used to build the sequencing graph. Two examples are given. `i` is a vector such that the j th element of `i` has value j. A and B corresponds to empty streams which can be interpreted as a fatal deadlock.

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 `$X` does not depend from the instantaneous value of `X`, but the clock of `$X` depends from 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 expression that will remain constant (we can therefore optimise the generated code), or that will never produce any computation and generates tasks in dead-lock (that might be a programming error).

The sequencing graph of the tasks being an approximation of the true dependency graph, we might detect as incorrect some programmes with an effective value. It is possible, with some refinementsof the method, 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.

IV.d. The data-flow distribution and scheduling

After the scheduling inference, the compiler is able to distribute the tasks onto the PEs of a target architecture and to choose on 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 PE 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×PE space). The width of the rectangle corresponds to the execution time of the task and its height to the number of the PE ideally required for a fully parallel execution of the task (Cf. Figure 6). For example, if the task corresponds to the data-parallel addition of two 100 elements array, the height of the associated rectangle will be 100.

With this 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 bound in height by the number of PEs in the architecture. Some efficient heuristics exist for this problem known under the name “bin-packing” in two dimensions (which is NP-complete in the general case [GAR 78]).

At the moment we are testing a greedy strategy [MAH 92] consisting in 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 PE. If more than one task is available at the same time an additional criterion is given to choose which one to take first (for example, a task being on the critical path).

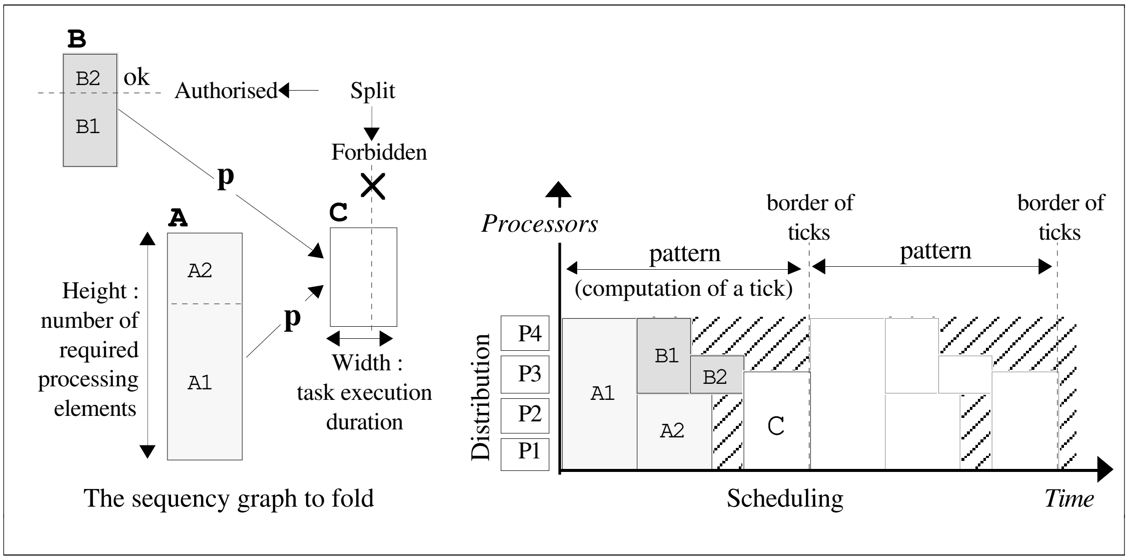

Figure 6: scheduling and distribution of a sequencing graph using a two dimensional bin-packing method

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 being scheduled and the other one being put back in the pool of available tasks (to be scheduled and distributed later). We only admit the split in the horizontal direction (Cf. Figure 6). In fact, it 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 [HAW 89] can be used to give a low boundary the performance of this strategy. This result bounds the worst case and guarantees the good quality of the heuristic used here.

V. Conclusions

The current compiler is written in C and in a 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.

We are also working on the implementation of dynamic geometry to allow the definition of non constant-size webs in time and space. Such features are needed to model growing systems (e.g. L-system), Penrose tiling, etc.

VI. References

[ARV 83] Arvind, J.D. Brock, Streams and Managers, Proceedings of the 14th IBM Computer Science Symposium, 1983.

[BAN 88] J.-P. Banâtre, A. Coutant, D. Le Métayer, *A parallel machine for multiset transformation and its programming style*, Future Generation Computers Systems, N° 4, 1988.

[BLE 89] G. E. Blelloch, *Scans primitive Parallel Operations*, IEEE Transactions on Computers, Vol. 38, N° 11, November 1989.

[BLE90a] 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)

[BLE90b] G. E. Blelloch, S. Chatterjee, *VCODE: A data-parallel intermediate language*, Proceedings Frontiers of Massively Parallel Computation, pages 471-480, October 1990.

[BLE93a] 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.

[BLE93b] 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, February 1993.

[CBD 93] F. Cappello, J. L. Béchennec, F. Delaplace, C. Germain, J. L. Giavitto, V. Néri, D. Etiemble, *Balanced Distributed Memory Parallel Computers*, International Conf. on Parallel Processing 93, St Charles, Illinois, Aout 93

[CMO 87] 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.
[CPH 87] 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.
[DEN 74] J. B. Denis, *First Version of a Data Flow Procedure Language*, proc. of the Programming Symposium, Paris, April 9-11, 1974, LNCS 19, Springer.
[GAJ 82] D.D. Gajski, D.A. Padua and D.J. Kuck, *A second opinion on data flow machines and languages*, IEEE Computer, february 1982.
[GAR 78] M.R. Garey, R.L. Graham, D.S. Jonson, *Performance Guarantees for Scheduling Algorithms*, Operation research, Vol. 26, N° 1, January-February 1978.
[GGF 91] J.-L. Giavitto, C. Germain, J. Fowler, *OAL: an Implementation of an Actor Language on a Massively Parallel Message-Passing Arcitecture*, proc. of the Second European Distributed Memory Computing Conference, 22-24 april 1991, Münich, LNCS 492
[GIA 91] J.-L. Giavitto, *A synchronous data-flow language for massively parallel computers*, Proc. of Parallel Computing'91, 3-6 September 1991, London.
[GIA 92] J.-L. Giavitto, *Typing geometry of Homogeneous Collection*, Proc. of the 2nd Int. Workshop on array, ATABLE-92, Montréal, Jun 1992.
[HAW 89] J.-J. Hawang, Y.-C. Chow, F. Angers, C.-Y. Lee, *Scheduling precedence graphs in systems with interprocessor communication times*, SIAM J. Comp., Vol. 18, N°2, pp 244-257, April 1989.
[HIL 86] W. Daniel Hillis and Guy L. Steele JR, *Data Parallel Algorithms*, Communication of the ACM, vol. 29 N°12, December 1986.
[KB 88] Israel Koren and Bilha Mendelson, *A data-driven VLSI array for Arbitrary Algorithms*, IEEE COMPUTER, October 1988.
[LEI 83] Charles Leiserson, James Saxe, *Optimizing Synchronous Systems*, Journal of VLSI and Computers systems, Vol 1 N°1, pp41-67, 1983
[LER 93] X. Leroy, *The Caml Light system release 0.6 - Documentation and users manual*, INRIA, Sep 1993.
[LGB 86] P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, *SIGNAL, a dataflow oriented language for signal processing*, IEEE-ASSP, 34(2):362-374, 1986.
[MAH 92] A. Mahiout, *Scheduling and static distribution of big data-flow graphs for MSIMD architectures*, DEA d'Architecture des Machines Informatiques Nouvelles de l'université de Paris-Sud (in french), LRI, Septembre 1992.
[MIC94b] O. Michel, The 8 1/2 reference manual, to be published as LRI Technical Report, Université de Paris-Sud, Orsay, 1994.
[MIC 94] O. Michel, J.L. Giavitto, J.P. Sansonnet, *A data-parallel declarative language for the simulation of large dynamical systems and its compilation*, Software for Multiprocessors and Supercomputers, September 21-23, 1994, Moscow
[PAS 81] H. Siegel, L. Siegel, F. Kemmerer, P. Mueller Jr., H. Smalley Jr. and D. Smith, *PASM: A Partionnable SIMD/MIMD System for Image Processing and Pattern Recognition*, IEEE Transaction on Computers, Vol. C-30, N°12, December 1981.
[SAB 88] G. W. Sabot, *The Paralation Model: Architecture*, Independent Parallel Programming. MIT Press, Cambridge MA, 1988
[SDB 86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, Programming with sets: An introduction to SETL, Springer-Verlag 1986.
[SIJ 89] B. A. Sijtsma, *On the productivity of recursive list definitions*, ACM Trans. on Programming Languages and Systems, Vol. 11, N° 4, Octonber 1989.
[SIP 91] J. M. Sipelstein, G. E. Blelloch, *Collection-oriented languages*, proc. of the IEEE, Vol. 79, N° 4, april 1991, pp. 633-649.
[STE 90] G. Steele, *Making asynchronous parallelism safe for the world*, Seventeenth annual Symposium on Principles of programming languages, San-Franscisco, 17 January 1990, pp 218-231.
[TMC 86] Thinking Machine Corporation, *The Essential *Lisp Manual*, Cambridge MA, 1986
[TOF 87] Tommaso Tofooli and Norman Margolus, *Cellular Automata Machine*, The MIT Press, Cambridge MA, 1987
[WAD 76] W. W. Wadge and E.A. Ashcroft, *Lucid - A formal system for writing and proving programs*, SIAM Journal on Computing (3):336-354, Sept., 1976.
[WAD 81] W. W. Wadge, *An extensional treatment of dataflow deqdlock*, Theoretical Computer Science, Vol. 13, N° 1, 1981, pp 3-15.
[WAD 85] W. W. Wadge and E.A. Ashcroft, *Lucid, the Data flow Programming Language*, Academic Press U.K., 1985.
[WAT 91] R. 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.

∴