$8_{1/2}$: Data-Parallelism and Data-Flow*

OLIVIER MICHEL, DOMINIQUE DE VITO and JEAN-PAUL SANSONNET
LRI u.r.a. 410 du CNRS, Bâtiment 490, Université Paris-Sud
F-91405 Orsay Cedex, France
E-mail: michel@lri.fr

ABSTRACT

We advocate a data-flow approach to data-parallelism to ensure both parallelism expressiveness and efficient exploitation of data-parallel applications on new massively parallel architectures. The rational of this approach is introduced in the first part of the paper. Then we develop an experimental language following these lines and sketch the techniques used to compile such a data-flow data-parallel language. Its compilation, based upon a static execution model enabling an efficient execution of programs, is introduced.

1. The Parallelism and its Expression Through Data

1.1. A Short Taxonomy of Parallelism Expression in Programming Languages

The many existing models and languages for parallel programming lead to an overlap of concepts making the design of a taxonomy a hard task. However, we propose in table 1 a framework used to specify the concepts presented in this paper. Two criteria have been selected to classify the languages: the way they let the programmer express control and the way they let him manipulate data.

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-instruction counters*
Scalar Languages	Sisal, Id, LAU, Actor languages	Fortran, C, Pascal	Ada, Occam
Collection Languages	Gamma, APL $8_{1/2}$, Lucid	*LISP, HPF, CMFortran	CMFortran + multi-threads

The programmer has the choice, as far as control expression is concerned, between three different strategies:

- *Not to express it:* this is the data-flow execution model. In this model, there is no feature allowing the expression of sequencing by the programmer. It is the task of the compiler (static extraction of the parallelism), or of the execution support (dynamical extraction through a software interpret or an hardware

*This research is partially supported by the operation "Programmation parallèle et distribuée" of the GDR de programmation.

architecture) to construct a sequence of computations compatible with the functional dependencies between data.

- *Express what has to be done sequentially:* this is the classical imperative sequential execution model.
- *Express what would not be done sequentially:* this is the concurrent languages approach. This approach exhibits explicit control structures like `PAR` in `Occam`, `FORK` and `JOIN`, etc.

For the handling of data, we can state two major classes of languages:

- *Collection oriented languages* give the programmer the ability to manipulate a set of data as a whole. Such a set is called a *collection*[1]. Languages like `APL`, `SETL`, `*Lisp`, intensional languages[2] etc. are in this line.
- *Scalar languages* also allow the programmer to manipulate sets of data but only through a point-wise access of their elements. For example, in standard Pascal, the main operation upon an array is the access to one element.

Data-parallelism comes from the introduction of parallelism into sequential languages (this is the "starization" of classical languages: from `C` to `C*`, `Lisp` to `*Lisp`, etc.). But table 1 states that the collection feature is orthogonal w.r.t. the control expression. As a consequence, collection oriented languages can be freely mixed with concurrent (multi-threading) and data-flow languages (like `Gamma`[3] or 81/2 [4]).

In the following section, we show that semantic and efficiency problems arise when data-parallelism appears in a sequential framework. This lead us to explore the other alternatives and we will see in the next section that an embedding of data-parallelism within a concurrent framework is very attractive. Therefore, the "data-parallelism + data-flow" approach is, from the soundness of the approach point of view, the most interesting solution and may yield to efficient exploitation.

1.2. The Problems of Data-Parallelism within a Sequential Framework

Data-parallelism in a sequential framework induces semantics as well as efficiency problems

1.2.1. Semantic Problems

Data-parallel languages like `*Lisp` or `Pomp-C`[5] introduce data-parallelism through the use of control structures allowing a synchronous control of the parallel activity of the processors. These control structures can lead to serious semantic difficulties. They arise from the interaction of *1)* the concept of collection and *2)* the management of the two kinds of control flow encountered in a program, the sequencing of the scalar part and the parallel part of the program. For example, in the following Pomp-C program:

`collection[512, 512] Pixel;`	
`Pixel int picture;`	`picture` is a collection of integers.
`int a;`	`a` is a scalar integer.
`...`	
`picture = FALSE;`	all the points of `picture` are set to `false`
`where(picture){`	so, no processor should be active within
`    a = 5;`	the `where`, but this scalar assignment is performed because it does not depend on the `where`.
`    everywhere{`	reactivates locally all the processors.
`        printf("hello");`	since this is a scalar instruction, only one `hello` will be printed.
`        picture = TRUE;}}`	here, all the points of `picture` are modified.

Even if all processors are inactive, some instructions can still be executed in the the scope of a `where`: for example, the scalar instruction (`a=5`) or a parallel instruction in the immediate scope of an `everywhere{picture=TRUE}`.

More seriously, a processor made iddle by a `where` cannot skip the triggering of a functio call. Even if all the processors are idle, the *Lisp and Pomp-C compilers still generate the function-call because there might be some `everywhere` or scalar instructions. Then the following factorial function (example taken from the Pomp-C manual[5]):

```
collection generic int fact(generic int n)
{ if (n<=1) return 1; else return n*fact(n-1); }
```

is wrong because the recursive call is always performed: the recursivity is not bounded and the program never ends. If the recursivity is stopped when all the processors are idle semantically incorrect behavior may occur.

In the following example (from *Lisp), problems arise with the creation of the collection A:

`(*when ( ... )`	in this `*when`, some processors are supposed to be idle
`    (*let ((A (!! 1)))`	a new collection, `A`, is created, local to the `*let`, and initialized to `1` the elements of `A` on active processors.
`        (*all (*sum A))))`	because of `*all`, the sum of **all** elements of `A` is performed.

Actually, the collection A has an extension on all the processors but it has only been partially initialised in the `*let`. The sum of all the "active" elements of A will also perform the addition of the undetermined values located on the idle processors.

The conclusion is that a sequential data-parallel language does not always allow the programmer to manipulate collections in a natural way.

1.2.2. Implementation Problems of the Sequential Data-Parallel Languages on the New Architectures with Hybrid Control

Recently, a new class of massively parallel architectures has emerged, like the CM5[6], T3D[7] or PTAH[8]. These architectures are able to exploit several sources of parallelism at the same time while preserving the simplicity and the efficiency of SIMD architectures. They put aside the synchronization constraints thus acquiring the flexibility and productivity of the processors characterizing MIMD architectures. These architectures have hybrid control[9]: SPMD, MSIMD, MIMD with firm synchronization ... We generally believe that these architectures will reach the TeraFlop required by the numerical applications of the "Grand Challenge"[10]. It is therefore natural to implement data-parallel languages on these new systems. Indeed, data-parallel languages are well fitted to intensive numerical computations because they give an easy way to handle the objects of numerical computation: vectors, matrices, ... ; more generally, data-parallel languages fit well with massively parallel computations (data-bases, image processing, etc.) because of their fine-grain parallelism.

Nevertheless, the sequential data-parallel model faces serious problems on hybrid control computers, and more generally on MIMD systems (see[11]). The main problem is probably the bad utilization of computation resources due to the usage of a *strictly sequential* flow control, as encountered on strict SIMD implementations: the very well known drawbacks of the SIMD model are brought into the MIMD model. For example, while scalar values are computed, the processors in charge of parallel computations remain idle; also, nested instructions like `where` may exponentially reduce the number of active processors.

But the main restriction is probably the incapacity to overlap communication cycles with the execution of independent computations. This restriction, which cannot be canceled by the SPMD implementation leads to severe shortcomings because the communication network is, in most cases, the less efficient part of an architecture. The efficiency of a sequential data-parallel execution is therefore bounded by the weakest part of the architecture. This is a direct consequence of the strict sequential framework set by sequential flow control structures.

1.3. Advantages and Shortcomings of the Data-Flow Approach

The conclusions of the previous section lead us to take the data-parallel model and its data structure, the collection, out of the strictly sequential framework. Another framework has to be designed to embed the parallel data-structures.

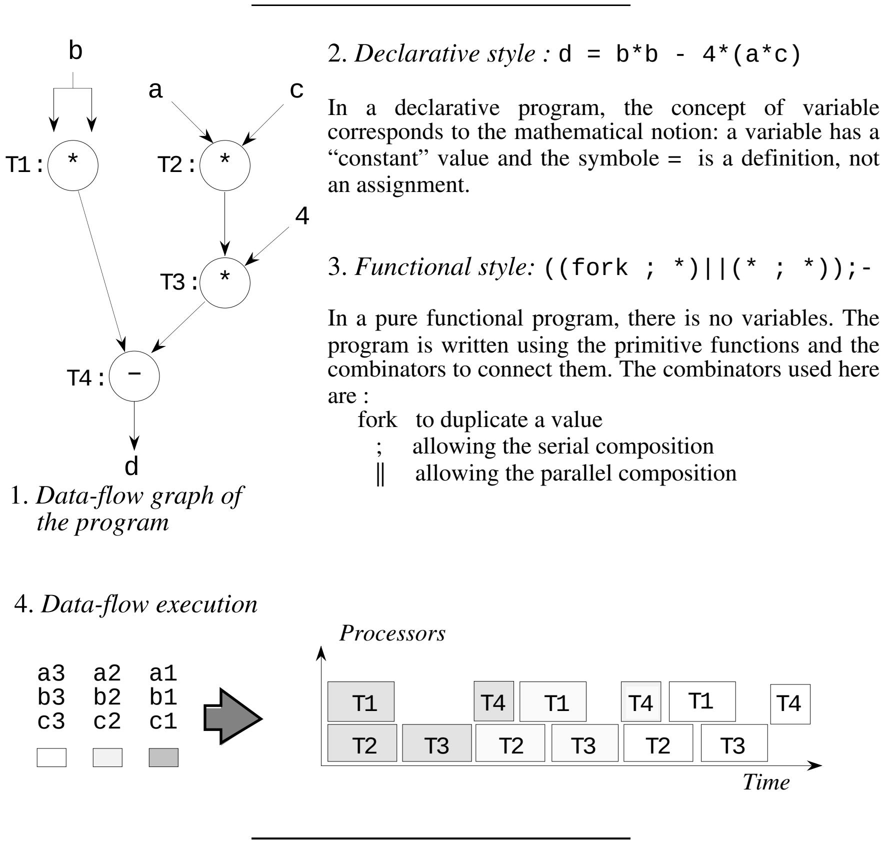

Fig. 1. Data-flow representation of the Pascal program `d:= b*b - 4*a*c` in three different manners: graphical, declarative and functional. The figure 1.4 represents a possible scheduling of the activation of each node of the data-flow graph represented in figure 1.1. Each node T_i is associated to a processor. A sequence of three parameters is present at the input which leads to a pipe-line evaluation. Parallelism is also provided by the simultaneous activation of the nodes on different processors.

1.3.1. The Data-Flow Choice

The mix of the data-parallel approach and the concurrent approach leads to languages with attractive properties: first, these languages reflect the hardware structure of the new architectures with hybrid control; then, they are easy to understand by the programmer because they correspond to the juxtaposition of the notion of task and the notion of collection; finally, they take benefit of the tools and formalisms already developed for the concurrent systems. Nevertheless, they require the explicit expression of the control parallelism, minimizing the parallelism to be exploited (Cf. section 1.3.2). They also impose an artificial hierarchical design of the applications in two levels: MIMD coarse-grain tasks manipulating arrays used in SIMD mode. Moreover, the natural framework for these execution models is an explicit asynchronous framework with all the related problems (e.g. undeterminism).

This lead us to consider the data-flow approach as the natural framework for the expression of data-parallelism. An important property of the data-flow model is that the order induced by the data-dependencies is sufficient to determine the result of a computation. To ensure this property, various data-flow languages use multiple strategies. For example, as far as functional languages[a] are concerned, a *confluence theorem* is used. In the case of languages derived from imperative languages (`LAU`[12], `Sisal`[13], `Val`[14], ...), a programming constraint has to be added: this is the *single assignment* principle which allows the variables of the program to be set only once.

This property makes possible the representation of the programs by a graph, the data-flow graph (DFG in short, see figure 1.1), where the nodes correspond to the expressions and the edges to dependencies between expressions. The evaluation of a DFG is possible when its inputs receive data. There is no use of instruction counters: an expression can be evaluated as soon as all its arguments have been computed (see figure 1.4).

1.3.2. The Advantages of the Data-Flow Execution Model

The first advantage is ergonomic. In a data-flow language, the instructions scheduling is implicit. The programmer only describe the dependencies between data. Therefore, the programmer does not describe the operations that have to be done in parallel. For example, it is better to naturally write the program that performs the sum of four values:

```
sum = a+b+c+d
a = ...
b = ...
c = ...
d = ...
```

This program is a system of equations where the order of the equations does not matter.

and let a tool automatically extract the parallelism, rather than to explicitly write the parallelization (e.g. in `Occam`):

```
PAR
    PAR
        a := ...
        b := ...
        c := ...
        d := ...
    t1 := a + b
    t2 := c + d
sum := t1 + t2
```

[a] Pure functional languages (like FP) are data-flow languages: no side-effect, no explicit sequencing, no concept of memory nor assignment. Actually, most of the functional languages in use (like ML) have imperative features (mutable structures for example) allowing side-effects.

In the `Occam` version of this program, the programmer has to use two instructions, two auxiliary variables and has to describe the proper mapping of the tasks on the processors. This example is caricatural but shows that the implicit expression of parallelism is more natural for the programmer (on the reverse, there are some examples, with lots of sequences involved, that are hard to write in a declarative style).

The second advantage of the data-flow model comes from the implicit scheduling in a program: this leads to an optimal exploitation of the parallelism. A minimal scheduling is automatically computed leading to a maximal expression of the parallelism. This scheduling can either be statically infered (by a compiler that can take into account the peculiarities of the target architecture), or dynamically (by the execution support: an interpret or a data-flow architecture[15,16]).

A third advantage of the data-flow model is the *determinism of the result* of a computation. In an asynchronous language like `Occam`, the expression of parallelism takes place through the non-determinism of the execution: actions in `PAR` can occur in an undefined order, and therefore, possibly in parallel. To get the desired deterministic result, the programmer has to express the sequencing, through the use of semaphores, guards, etc. On one hand, if too much sequencing is expressed, some parallelism is lost, on the other one, if too less sequencing is expressed, the program could compute different results, depending on the execution path. A data-flow language does not suffer this problem: the expression of sequencing is implicit and corresponds to "only what is necessary", to ensure a deterministic result.

Another advantage of the data-flow approach is the referential transparency: a data-flow program can be seen as a set of mathematical equations where every reference to a variable can be replaced by its definition. So, it is easier to check and to achieve formal manipulations on programs. Some optimizations of such programs can be done automatically[17,18]. Finally, considering a program as a set of equations ensure that the program will compute a result, if this result if formally derivable from the set of initial equations: this is the *declarative completeness* property[19].

1.3.3. The Drawbacks of the Data-Flow Languages

In a famous article[20], Gajski & al. criticized the data-flow approach in comparison with the automatic parallelization of sequential programs. If the functional semantic and the lack of side-effects of the data-flow program makes possible their analysis by a compiler, the cost of the single memory assignment is prohibitive. For example, it is necessary to have a *garbage collector*; the manipulation of arrays is also very inefficient: the whole array has to be copied each time one of its element is changed.

To answer these criticisms, data-flow languages designers have introduced new mechanisms to deal with arrays: *mutable* data-structures like I-structures in `Id`[21], explicitly parallel expressions like `forall` and `expand` in `LAU`, `Val` or `Sisal`. It is possible to perform, through the use of such structures, multiple accesses to arrays.

We see, that a possible answer to the criticisms of Gajski & al. can be found in the introduction of data-parallel operators to manipulate arrays *as a whole*.

But the main argument of Gajski & al. against the data-flow approach is based upon the dynamical execution model, that is, on the computation, at run-time, of the scheduling of the instructions and on the dynamic memory management. Delaying the computation phase until the execution of the program should theoretically lead to an optimal evaluation of the program. For example, it is possible to compute only the required value needed to get the final result: this is the lazy evaluation strategy. But this management, according to Gajski & al., lead to a prohibitive cost of the execution support. This is why they prone the automatic parallelization of conventional sequential languages because a parallelizing compiler will statically infer the elements required to the execution and therefore minimize the cost of the execution support.

Nevertheless, the criticism is no more valid: because of efficiency problems due to the dynamical extraction, the compilation of data-flow networks has seen a gain of interest. This is particularly true in the field of signal processing[22], real-time programming[23,24], the design of systolic circuits and algorithms[25] and automatic parallelization[26]. It is now possible to develop static execution schemes for the data-flow model and in particular for the iterative data-flow model.

1.3.4. The Iterative Data-Flow Model: a Static Execution Model for the Data-Flow

To compile a data-flow program, it is necessary to know statically its graph. Recursive function-calls are therefore forbidden: they correspond to the (dynamical) creation of a (sub-)data-flow graph. It isn't possible to get statically the complete graph by unfolding the functional calls when recursive functions are involved. On the other side, if recursive functions are forbidden, iteration loops must be allowed. To follow the single assignment rule, we have to consider that each variable involved represents a sequence of values. As an analogy with a data-flow graph, we can say that a vertex, in a cycle, "sees" values passing in time.

It is possible to extend the notion of a variable representing a sequence of values to all the variables of a program and not only to those involved in loops. This is done in languages like `Lucid`[27,28] (we do not speak here of the *new* `Lucid` appeared recently[2]) where the main data-structure is the sequence of values. Then, all the operators of the language act on such sequences.

We call *declarative data-flow model* a model of language with implicit sequencing, based upon the notion of sequences of values. From this point onwards, we will refer to sequence of values as *streams*. Indeed, these sequences are potentially infinite, and, as we will detail lately in the paper, a temporal interpretation is bound to the step from an element to the next. The concept of stream is essential for three reasons:

- The stream is a sequence of values. The sequence of values is a versatile and important data-structure found in many programming languages: under the name of *sequence* in `Common-Lisp` and in `FP`[29] or as *stream* in `Lucid`. This data-structure is used in various different applications domains: real-time programming with `Lustre`[23] and `Signal`[24], PLD programming with `Palasm`[30], VLSI design with `Daisy`[31] and `Stream`[32], specification of parallel programs with `Unity`[33], design of systolic circuits and algorithms through the notion of recurrent equations with `Crystal`[25] and `Alpha`[34].

- The stream is a communication paradigm: either in the *pipe* and its generalization to *sockets* and *streams* in Unix.
- In a single assignment language, the stream is, quoting Wadge and Ashcroft[28], a "mathematically respectable" structure and Waters[18] says : "(...) series expressions are to loops as structured control constructs are to gotos".

1.4. Conclusion: the Benefits of a Data-Parallel + Data-Flow Combination

Some drawbacks of the data-flow models are solved by the introduction of data-parallel data structures. This is done with a specific management for arrays. Furthermore, the collection is a naturally distributed data structure which is usually not a focussed issue in data-flow. Besides, the data-flow execution model is an alternative to the sequential SIMD execution model of the first data-parallel languages. This new alternative may lead to efficient implementations on the new massively parallel computers with hybrid control.

Data-parallelism and data-flow are orthogonal notions that are able to maximize the expression of parallelism: data-parallelism excels in expressing space relationships and data-flow in the expression of temporal constraints. With these two concepts, it is possible to express *all* the parallelism inherent to an application. Expressing parallelism in excess (w.r.t. a target architecture) is very important because it lets the compiler: *a)* to hide the communication delays by independent computations and *b)* to choose parallel activities with a minimal management cost.

Furthermore, the expression of time and space relationships is done in an implicit manner. It is the compiler's task to "fold" the computations of a program, with their respective spatio-temporal structure on a given architecture. The programmer does not care of the target architecture and it does not appear at the programming level: a data-parallel data-flow language is a *high-level parallel language* able to design *portable* parallel programs. The abstract formalism of the language, based upon equations, does not look to us as an obstacle. Indeed, the large diffusion of equation based languages like `Prolog` or `SQL` and this programming style widespreading more and more (see section 1.3) seems to fit well with numerical problems (as seen in section 3).

But, in order to have an efficient exploitation of parallelism in a data-flow execution model, theoretical and software tools have to be designed. This is the subject of the third part of this paper, after the description of the experimental language $8_{1/2}$.

2. The Experimental Language $8_{1/2}$

We are currently studying the combination of the data-flow and data-parallel concepts and their implementation into a static execution model through an experimental language called[35] **$8_{1/2}$** . $8_{1/2}$ introduces a new data structure, the type *fabric*[b] corresponding to a temporal sequence (i.e. a stream) of collections, or as a dual point of view, a collection of streams (Cf. figure 2).

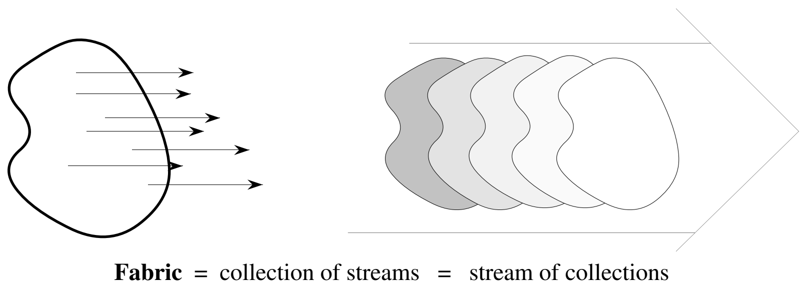

Fig. 2. Two points of view for a fabric.

The following notation will be used to specify the values of the `fabric` type:

- collection values are enumerated between braces $\{\ldots,\ \ldots\}$;
- stream values are specified between less signs $< \ldots\ ;\ \cdots <$.

The comma sign is a separator expressing the spatial neighborhood and the semicolon the temporal sequence. For example, the constant fabric with a value of three 1 can be represented by:

$$< \ \{1,1,1\}\ ;\ \{1,1,1\}\ ;\ \ldots\ <$$

or

$$\{< \, < 1;1;1;\ \cdots < \,,\ < 1;1;1\ \cdots < \,,\ < 1;1;1;\ \cdots < \,\}$$

depending on the point of view chosen: the "stream of collections" or the "collection of streams" one.

2.1. A $8_{1/2}$ Program is a System of Equations

A $8_{1/2}$ program is a set of equations where each equation defines a fabric. Each

[b]This data structure was primarily called a "web", but due to the extension of the WWW, we choose now to call it a *fabric*.

equation has the following shape:

$$x = f(x, y, \ldots, z)$$

where $x, y, \ldots, z$ are fabric names and f a function over fabrics. A 8½ program expresses a relation between the input fabrics (fabrics with no definition thus corresponding to constants or to parameters of the program) and the output fabrics (fabrics defined by an equation).

There are two classes of functions f overs fabrics. In the first class are "geometrical" functions used to manipulate the collection aspect of a fabric. These functions correspond to the classical data-parallel operators but are extended to perform the operation on each element of the stream (this is similar to the "serendipity principle" of Lucid[2]). For example, the data-parallel addition $+$ is implicitly extended so that:

$$\begin{array}{ccccccc} & < & \{1,\ 1,\ 1\} & ; & \{2,\ 2,\ 2\} & ;\ \ldots & < \\ + & < & \{4,\ 5,\ 6\} & ; & \{6,\ 7,\ 8\} & ;\ \ldots & < \\ \Rightarrow & < & \{5,\ 6,\ 7\} & ; & \{8,\ 9,\ 10\} & ;\ \ldots & < \end{array}$$

(the sign $\Rightarrow$ means "evaluates to"). In the second class, there are four different kinds of function applications[1]: the function application, the explicit α-extension, the β-reduction and the scan (the terminology is derived from APL).

2.2. Operations on Streams

Functions acting on the temporal aspect of fabrics belong to the second class of functions. In 8½, functions on streams have the following property: the current value of a stream can only depend on the n previous values, this number n being computed at compile-time. This constraint can be found in real-time languages like Lustre and Signal which are far relatives of Lucid (but this property is not held in Lucid). From the implementation point of view, a sequential evaluation order of the stream values is ensured and it allows a static allocation of the memory used by the computations.

The delay operator, \$, is used to shift an entire stream. It allows at a given instant to reference the value at the previous instant (see table 2). This is the only operator allowing an access to the temporal axis. The other operator that we will detail here is the sampling operator *when*. This operator is similar to the T-gate of the data-flow languages[36] and is the *temporal* version of the conditional: the values of T *when* B are those of T when the values of B have the *true* value (Cf. table 2; the *when* operator in 8½ should not be mixed up with the `where` or `*when` operators of collection-based languages):

Table 2. Examples of delays and samplings on streams. The operators \$ and *when* introduce "holes" in the *progression* of the streams (Cf. section 3.2).

T	$=$	$<$	1 ;	2 ;	3 ;	4 ;	5 ;	6 ;	7	...	$<$
B	$=$	$<$	F ;	F ;	F ;	T ;	F ;	T ;	T	...	$<$
\$T	$\Rightarrow$	$<$		1 ;	2 ;	3 ;	4 ;	5 ;	6	...	$<$
T when B	$\Rightarrow$	$<$				4		6 ;	7	...	$<$

2.3. Examples of 8½ Programs

2.3.1. Recursive Definitions

It is possible, in the definition of a fabric to reference itself: this is a recursive definition. The recursion might be of direct type (the identifier on the left hand-side appears in the right hand-side of a definition) or indirect type (in this case, one or more variables are involved). If this mechanism is not carefully used, it might not define anything useful. For example, the program:

$$A = B \tag{1}$$
$$B = A \tag{2}$$

defines A (and B) by an indirect recursive definition and the values of A and B will remain forever undefined. To understand the mechanism, the equations (1) and (2) must be seen as a system of equations between series of collections. These equations must be valid for all elements of the series. The system of equations (1) and (2) have an infinite number of solutions: all the pair of fabrics of the shape (T, T) where T is any fabric. In a general manner, when many different solutions are possible a 8½ program computes the *smallest* solution in the sense of a particular order relation, defined by the language semantic[37]. A rough definition of the smallest solution could be "the stream with the less possible elements".

Recursivity can be used to define non-trivial fabrics. For example, the fabric *cpt* defined below is a counter which counts at the rhythm of the *true* values of another fabric, *event*, defined elsewhere:

$$cpt@0 = 0 \tag{3}$$
$$cpt = (\$cpt + 1)\,\text{when}\,event \tag{4}$$

Equation (3) defines the initial value of the counter, 0. The next values are defined by the equation (4). The construction "@0" defines a temporal quantification for an equation defining a fabric: (3) is used to define the first value of *cpt* (value indexed by 0 in the stream) and equation (4) is used to compute all other values.

The previous recursion corresponds to a *temporal recursion* because it is performed through a delay operator. It is also possible to define a *spatial recursion*. For example,

$$iota[10] = 0 \,\#\, (iota : [9] + 1) \tag{5}$$

defines the fabric $< \{0,1,2,3,4,5,6,7,8,9\} <$. In equation (5), two different types of constant are used: constant of *scalar* type (like 9 or 10) and constants of fabric type like 0 or 1; in fact, 0 stands for $< \{0\} <$. The operator # performs the concatenation of collections and : $[n]$ takes the n first elements of the collection. Equation (5) defines a fabric called *iota* with 10 elements. The value of the first of these 10 elements is 0. If we denote $iota_j$ the j^{th} element of *iota*, and if we adopt the "fabric = collection of streams" point of view, the equation (5) can be rewritten in:

$$\begin{aligned} iota &= \{< 0 <\} \,\#\, \{iota_0,\ \ldots,\ iota_9\} : [9] + \{< 1 <,\ \ldots, < 1 <\} \\ &= \{< 0 <\} \,\#\, \{iota_0 + < 1 <,\ \ldots,\ iota_8 + < 1 <\} \\ &= \{< 0 <,\ iota_0 + < 1 <,\ \ldots, iota_8 + < 1 <\} \end{aligned}$$

since $iota_0 = 0$, then:

$$iota = \{< 0 <,\ < 1 <,\ iota_0 + < 1 <,\ \ldots, iota_8 + < 1 <\}$$

which shows that $iota_1 = 1$, and therefore: $iota_2 = 2$, etc. Finally:

$$iota = \{< 0 <, < 1 <, < 2 <, < 3 <, < 4 <, < 5 <, < 6 <, < 7 <, < 8 <, < 9 <\}$$

or $< \{0,1,2,3,4,5,6,7,8,9\} <$. This last example illustrates the formal manipulations allowed in a declarative language where a variable can be replaced by its definition. This programs correspond to a spatial recursion because it uses the value of other points to compute the value of a point.

This is clearly not the most evident way to define an array where the n^{th} element has for value n. A simple yet efficient way would be: $iota[10] = +\backslash\backslash 1$ where \\ is the scan operator and the constant 1 is a fabric with the shape $\{1,\ \ldots,\ 1\}$, the shape computation being left to the 8$_{1/2}$ compiler. The systematic overload of constants simplifies the expressions; the counterpart is that the compiler must be able to infer the exact type for each constant[37].

2.3.2. An Example: the Resolution of Partial Differential Equation by a Finite Difference Method

The solution of the parabolic equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{6}$$

gives the temperature $u(x,t)$ at a distance x from one end of the rod after time t. An explicit method of solution[38] uses a finite-difference scheme of equation (6) on a mesh $(X_i = ih, T_j = jk)$ which discretizes the space of the variables x and t. One finite-difference approximation to equation (6) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2} \tag{7}$$

which can be rewritten as

$$U_{i,j+1} = rU_{i-1,j} + (1-2r)U_{i,j} + rU_{i+1,j} \tag{8}$$

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 terms of known temperatures along the j^{th} time-row (Cf. figure 3). The temperatures at the initial instant ($j = 0$) are given by the initial distribution of the temperatures in the rod (this is a parameter of the problem), so are the temperature at $i = 0$ and $i = x/h$ (boundary conditions).

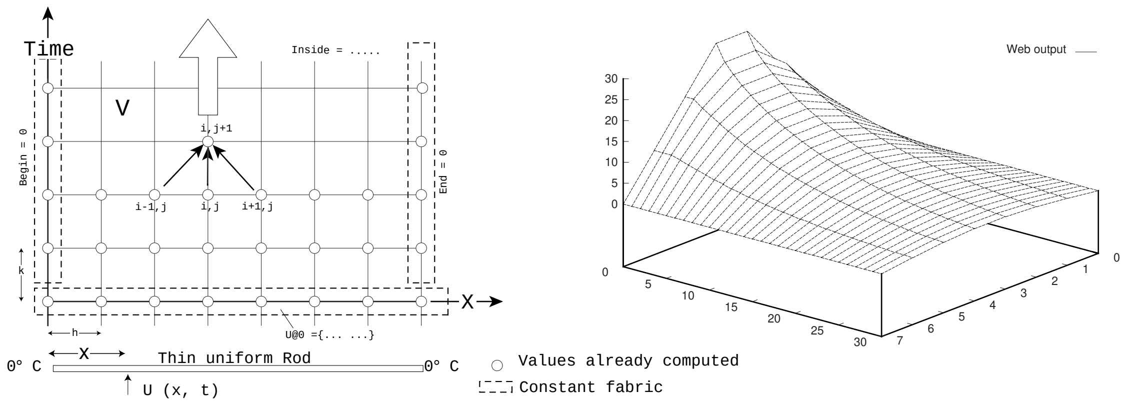

Fig. 3. Resolution by a finite difference method of a parabolic equation. The 8½ equations corresponding to the initial conditions at the boundaries are in bold face. A graphical output of the resolution of the equation is given.

The corresponding 8½ program is very easy to derive and simply corresponds to the description of the initial values, the boundary conditions and the specification of the relation (8). The stream aspect of a fabric corresponds to the time axis while the collection aspect represents the discretization of the rod (Cf. figure 3):

$$\begin{array}{lcll} Begin[1] & = & \text{initial conditions at one side of the rod ;} & (9) \\ End[1] & = & \text{initial conditions at the other side of the rod;} & (10) \\ U@0 & = & \text{distribution of the temperatures (at } t=0\text{) along the rod} & (11) \\ U & = & (Begin \,\#\, Inside \,\#\, End)\, \text{when}\, Clock & (12) \end{array}$$

Equations (9)-(12) describe the temperature of the rod through the aggregation of the initial conditions with the boundary conditions. The fabric *Inside*, which represents the evolution of the inside of the rod and $Clock$ is a predefined fabric with an infinite number of $True$ values. $Clock$ represents the passing time: $Clock = < \{True\};\ \{True\};\ \cdots <$. The fabric $Inside$ is defined by:

$$
\begin{array}{l}
r = k/(h * h) \\
Inside[98] = r * left(\$U) + (1 - 2 * r) * middle(\$U) + r * right(\$U)
\end{array}
$$

$Inside$, the inside of the rod, is a fabric of 98 elements. The function $left()$ (resp. $right()$) is a 8½ function shifting to the left (resp. to the right) a collection (like the `EOSHIFT` intrinsic of Fortran90) and the function $middle()$ the segmentation of a collection. These operations are not primitives in 8½, but are expressed in terms of communication operations[4] of 8½.

3. The Compilation of a Data-Flow Data-Parallel Language

The experimental language 8½ has been designed to be compiled both on sequential computers and parallel computers, with a completely static data-flow execution scheme. Several tools have been designed in the framework of 8½, based upon a formal semantics of the language in the style of denotational semantics[39]. In this section, we will only refer to the concepts developed in the 8½ compiler. It is roughly made of four separate phases: the synthesis and checking of the expression typing, the clock calculus of the expressions, the dependency calculus of tasks associated with expressions and the static mapping and scheduling of these tasks.

3.1. Type Checking of an Expression

Many typing systems are used in 8½ to check that, at compile-time, the execution of a program has a known behavior. These types are automatically synthesized by the compiler, without any intervention of the programmer. If the program is ill-typed it is therefore not possible to decide, at compile-time, the values of some parameters of its execution (the required memory for example). For example, the program:

$$
\begin{array}{lcl}
T@0 & = & \{0\} \\
T & = & ((\$T)\,\#\,(\$T))\ \mathrm{when}\ Clock
\end{array}
$$

defines a fabric T with an exponentially increasing number of elements in time:

$$T \ \Rightarrow < \{0\};\ \{0,0\};\ \{0,0,0,0\};\ \ldots\ <$$

each collection of the stream T has twice more elements than the previous one. This kind of program is detected (and rejected) by the typing system of the geometry of collections (but this kind of program, with dynamical structure can still be interpreted).

3.2. Clock Calculus of an Expression

In the execution of a 81/2 program, the computation of a fabric corresponds to the computation of the successive collections that are values of the stream underlying the fabric. The value of a fabric at a given instant is a collection (of scalar values) called the *instantaneous value* of the fabric. The study of the table 2 shows that some operators (like $ and *when*) introduce "holes" in the relative progression of a stream. To detect these holes, it is necessary to introduce a global logical time. The evolution of this logical time corresponds to the succession of the stream values.

The *clock* of a fabric T is a boolean stream that has, at the logical time t, the $True$ value if the fabric T produces an instantaneous value at this instant. An instant where the clock of T has the $True$ value is called a *tock*; all the other instants are called *ticks*.[c] It is the clock calculus of a fabric that determines if, at a given logical time, the computation of an instantaneous value has to take place or not. It is necessary, in order to perform the clock calculus, to reason on *normalized fabrics*. A normalized fabric is a fabric with an instantaneous value at every logical instant. We associate to each fabric T its normal form $\overline{T}$, computed in the following way: by convention, the value at a tock of $\overline{T}$ is the value at this tock of T; at a tick, the value of $\overline{T}$ is the value of T at its last tock, if it exists, else it is the special value nil. It is easy to derive, for a fabric T, its normal form, if its clock is known: it is the sub-series of $\overline{T}$ corresponding to the instants where the clock of T has a $True$ value (a clock is a normalized fabric).

The computation of an expression clock is simply done through a transformation of the initial program. Each definition:

$$x = f(y)$$

is translated into

$$\overline{x} = \textbf{ if } clock(x) \textbf{ then } f(\overline{y}) \tag{13}$$

where $clock(x)$ is an expression defining the clock of the fabric $\overline{x}$. This expression is synthesized by induction on the structure of the definition of x. For example,[d]

$$clock(A \,\text{when}\, B) = B \wedge clock(B).$$

Applying this transformation to a 81/2 program produces its normal form. Roughly, the compiler will generate, for each expression of this program a task that will im-

[c] A clock is thought to make *tick-tock*.

[d] We warn the reader that the clock calculus in 81/2 corresponds to a particular notion of time which is not like the notion of time proposed in the real-time synchronous languages such as Lustre or Signal[35].

plement the equation (13). But the dependencies between these tasks are still to be computed to get their proper activation instants.

3.3. The Tasks Scheduling

The DFG associated to a 81/2 program is immediately synthesized in the normal form. Unfortunately, this graph cannot be directly used in this form by the compiler to produce the task scheduling. Indeed, even if in the case of scalar data-flow programs the DFG corresponds to the dependency graph, this is no longer true when collections are concerned. A collection is a set of scalars. For example, in the following program:

$$A = B$$

each point of A (that is, each element of the collection support of the fabric A) depends on the only corresponding point of B. On the contrary, the following program, which is the sum of all the elements of B:

$$A = +\backslash B$$

produces a fabric A with only one point, depending on all the points of B. Nevertheless, these two programs are translated in the same DFG where the nodes associated to A and B are connected.

We can see the data-flow graph as an *approximation* of the real dependency graph. This approximation is too rough; for example, we cannot implement the program (5) with this graph alone. Indeed, this DFG is cyclic (see figure 4) thus forbidding its implementation.

The task of the compiler is to annotate the DFG to obtain a finer approximation of the dependency graph. The real dependency graph cannot be explicitly build because it contains as many nodes as there are points in the fabrics of the program (for example, matrices are often of size 1000×1000 in numerical computation and this would produce dependency graphs of over one million nodes).

We call *task sequencing graph* the approximation of the dependency graph by applying the following annotations:

- an expression e depends on the fabric x if x appears syntaxically in e. Nevertheless, we remove the dependencies corresponding to variables appearing in the scope of a delay: theses dependencies correspond to past values and not to instantaneous values of a fabric.
- the (instantaneous) dependency between an expression and a variable is annotated p if the value at a point i of e depends on the only value at i of x (point-to-point dependency).
- the dependency is annotated $+$ if the value of point i depends on the value of the points j of x with $j < i$.

- the dependency is annotated t if a point i of e depends on the value of all the points of x (total dependency) and more generally, if the dependencies scheme does not fit the two previous category.

In the sequencing graph, the cycles with an edge of type t or with no edges of type $+$ are dead branches. These cycles are called *instantaneous cycles*. Fabrics defined in instantaneous cycles have always undefined values. The remaining cycles (i.e. with $+$ edges and no t edges) correspond to spatial recursive expressions requiring a sequential implementation. Expressions that do not appear into a cycle are data-parallel expressions. They can be computed as soon as the expressions on which they depend, have been computed.

Whereas in recursive definition of functions, definition of partial functions are allowed, we want to avoid definitions of partial collections: the recursive definition of a collection must lead to the computation of a fixed size collection with all its elements with defined values. This problem is similar to the problem of the determination of

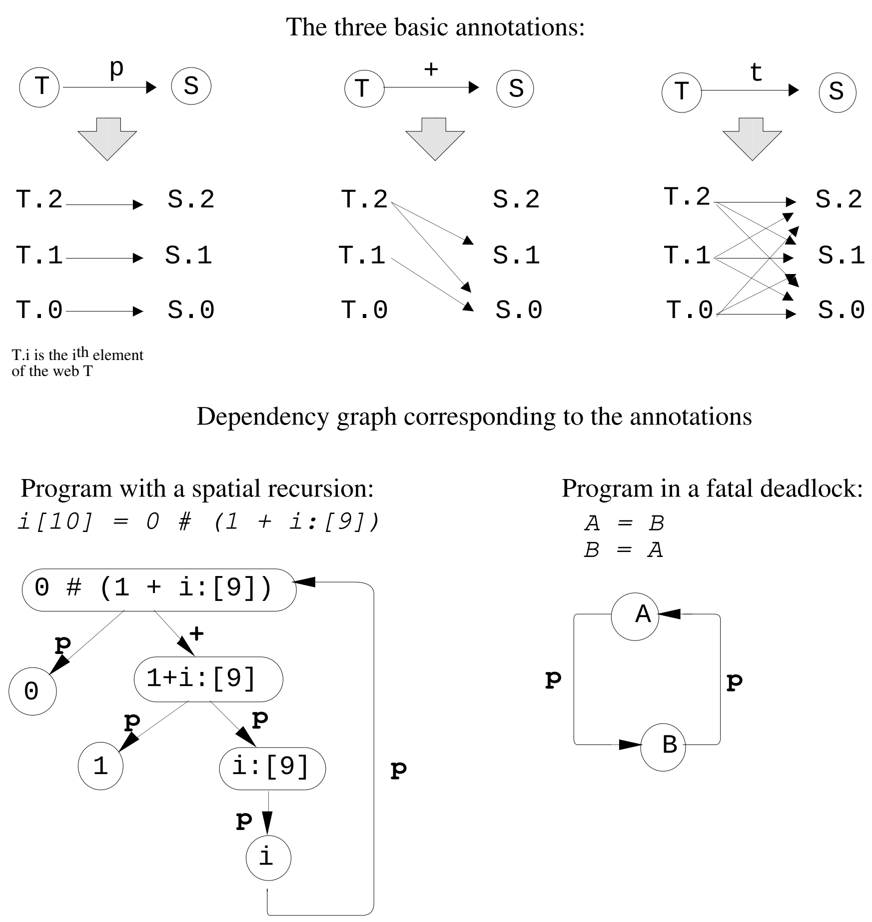

Fig. 4. Representation of the three types of annotations used to build the scheduling graph and application to two examples.

the strictness of a function in static analysis. Another similar problem is the detection of deadlocks in a data-flow graph[40] or the effectiveness of the recursive definition of a lazy list[41].

Actually, the effective processing of the whole sequencing graph is a little more complicated. Indeed, previously we have assumed as an hypothesis that the computations of the instantaneous values of $\$x$ do not depend on the instantaneous values of x, but the clock of $\$x$ depends on the clock of x (it is the same one, but for the first tock). Furthermore there can be instantaneous cycles between boolean expressions representing clock expressions in the sequencing graph. But these expressions have a defined value. The computation of this value is based upon the abstract evaluation of the clock expression from the denotational semantics of the language. This method has at least three advantages:

- it is completely static,
- expressions that will remain constant are detected and the generated code can therefore be optimized,
- expressions that will not produce any value and correspond to deadlock tasks are detected (this is probably a programmer error).

The task sequencing graph being an approximation of the real dependency graph, it is possible that we might detect as incorrect, programs that have a defined value. For example:

$$T[2] = \{T..`1', 0\}$$

($T..`1'$ refers to the second element of $T$). The sequencing of $T$ totally depends on itself because so does $T..`i'$ (i can be a value computed at run-time). This program results in a sequencing graph with a total cycle whereas the semantics of the language gives the obvious solution $\{0, 0\}$. In order not to reject such programs, more precise approximations of the scheduling graph are possible, but their implementation is more computation consuming[42].

3.4. Static Mapping and Scheduling of the Tasks

After the sequencing computation phase is achieved, the compiler is ready to map the tasks onto the processors of a target architecture with a scheduling compatible with the sequencing graph. To solve this problem, we will restrict ourselves to a particular scheduling: *the cyclic scheduling.* In our case, such a scheduling corresponds to the (eventually infinite) repetition by the processors of the execution of a piece of code. We call this piece of code a *pattern.* The pattern corresponds to the computation of the fabric values at any tick. The last task of the compiler will be to produce this pattern.

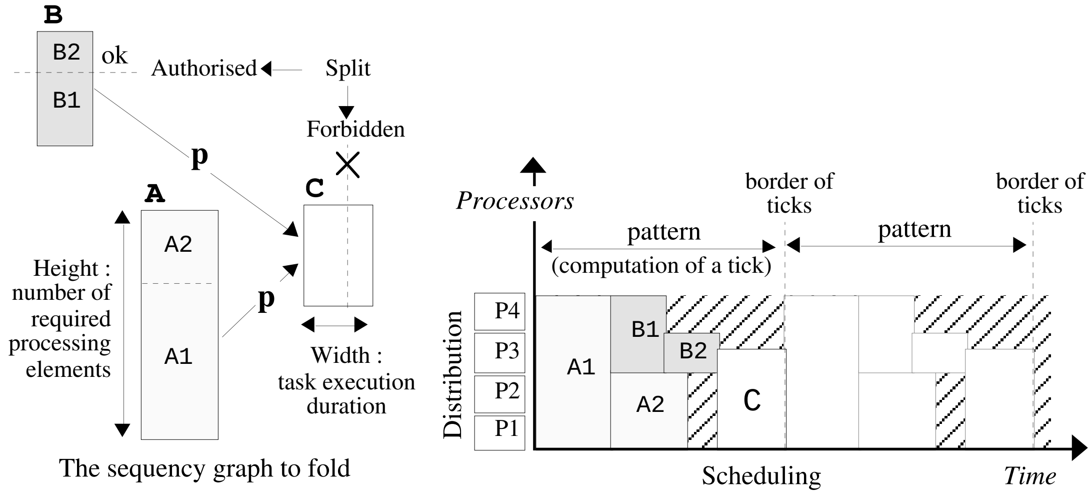

Fig. 5. Mapping and scheduling of a sequencing graph using a two dimensional bin-packing method.

To produce the pattern, a rectangle in the time$\times$processor space is associated to each task by the compiler. The width of the rectangle is the task execution length while its height is the number of Processors Elements (PEs in short) ideally required for a fully parallel execution of the task (Cf. figure 5). For example, the task associated to the data-parallel addition of two 100 element arrays is represented by a rectangle with a height of 100.

In this framework, the optimal solution of the mapping and scheduling of a task is to find a place for the rectangle minimizing the length of the packing while being bounded in height with the number of PEs of the target architecture. Efficient heuristics have been designed for this NP-complete (in the general case) problem of rectangles packing that is known as "bin-packing" in two dimensions[43].

We are currently testing a greedy strategy[44,45] consisting in the early mapping of the possible part of a ready task. A task becomes ready at the end of the tasks on which it depends. We take into account, at the end of each task execution, the communication delays between the PEs. If more than one task is ready at the same time, an additional choice criteria is used (like always choosing the task being on the critical path). If the length of the task exceeds the number of available PEs, we "split" the task into two parts, the first being scheduled immediately and the second being put in the set of ready tasks (so that it could be mapped and scheduled later). We suppose and admit that it is always possible to split a task into pieces, in the horizontal dimension (Cf. figure 5). Actually, this is possible because data-parallel tasks requiring n PEs are n independent scalar tasks.

A known result[46] can be used to lower bound the performances of this strategy. In the worst case, this strategy is lower-bounded and ensure the high efficiency of the heuristic used here.

4. Code Generation of 81/2 Programs

A compiler for the 81/2 language has been designed in ML. We briefly present in this section the code generation phase and an evaluation of the code. After the inference of the resources needed to implement the data defined in a program (Cf. section 3), the compiler is ready to generate a code towards several differents target architectures.

4.1. The Code Generation Scheme

The choosen code generation scheme corresponds to an evaluation that is:

- *sequential:* A topological sort of the dependency graph leads to the sequence of computation needed to solve the equations of the program. The parallelism involved here is therefore reduced to the inherent data-parallelism present in the operations, as in the SIMD execution model (`C*`, `*LISP`, `Fortran 90`, ...).
- *cyclic:* the fabric values are computed in the time-ordered instants and the computation of the values at the next instant only takes place after the whole computation of the current instant has been completed. Thus, the execution is linear in time with the simulation steps.

A program generated by the compiler is made of three parts:

- *the variable declaration:* the static execution model of 81/2 leads to no allocation of dynamical memory.
- *the computation:* a couple of predicate (d_T, h_T) is associated to each fabric T by the clock calculus phase (Cf. section 3.2). These predicates define the *definition domain* and the *clock* of the fabric for each instant of the simulation. They correspond to the instant set where respectively the value of T is defined and T has a new value. Thus, at a given instant, the value of a fabric is computed only if these predicates hold. (For the sake of simplicity, we have restricted our description of the clock calculus in section 3.2 to the clock predicates, but the full management of 81/2's streams introduces an additional predicate not detailed here.)
- *the delay copy:* any expression referenced through a delay operator has to be saved at each instant to be accessed later in time. This operation is called the *delay copy.*

At first sight, the generated code has the following aspect:

```
Declaration section
for each instant t
    for each fabric T in the sequencing order
        compute dT(t)
        if dT(t) then compute hT(t)
                      if hT(t) then compute the current
                                    data-parallel value of T
    Save the current values accessed at the next instant
```

The content of the `Declaration section` is determined by the type inference phase of the compiler (Cf. section 3.1).

In this code generation scheme, it is possible to generate code for a *sequential*, *vectorial* or *SIMD parallel architecture*. In the first case, any collection-oriented data-parallel operator used in 81/2 can be translated into a loop[47]. For vectorial or SIMD architectures, an ad hoc library can be used (like CVL[48] or the Paris Interface[49]).

4.2. A Brief Evaluation of the Generated Code

To evaluate the adopted generation scheme, we have performed a benchmark test based on the resolution of a partial differential equation by a finite element method (Cf. section 2.3.2). We have chosen to compare the sequential generated `C` code from the 81/2 equations with an ad hoc hand-coded resolution of the problem.

Table 3. Time ratio between a hand-written code against the generated code

Number of iterations → / Size of the rod ↓	100	500	1000	5000	10000
10	4.11	3.77	3.96	3.93	3.93
50	1.98	1.85	1.85	1.83	1.83
100	1.62	1.63	1.58	1.58	1.59
500	1.43	1.40	1.40	1.39	1.40
1000	1.43	1.42	1.41	1.42	1.42
5000	1.20	1.20	1.20	1.20	1.20
10000	1.36	1.35	1.35	1.36	1.35

The results of the execution time are given in table 3. Each element represents the ratio of the hand-written code against the generated one.

The version of code generated by the 81/2 compiler is 750 lines long but it includes a lot of `define`'s and declarations which will not be translated into true assembly code. The size of the two executable codes are about 16Kb and 4Kb respectively. This shows an expansion factor of about 4 in favor of 81/2.

Both program have been compiled using the GNU C compiler with the optimization option set (gcc -O). The evaluation has been performed on a *HP 9000/705 Series* under the *HP-UX 9.01* operating system.

The mean execution time corresponding to compiler generated code is about 1.35 times slower than the hand-written one when collection of significant sizes are used. These performances are obtained after some (automatic) optimizations of the generated code: introduction of a specialized shift operation (shifts are done in 81/2 through a gather/scatter operator) and sharing of the delay expressions. This fairly good resul validates the approach adopted[50].

5. Conclusions and Future Work

We have proposed a new approach to data-parallelism based upon the data-flow model. Our motivation is to capture most of the inherent parallelism in an application and efficiently exploit it on various massively parallel architectures. To achieve this goal, we think that the *implicit* expression of parallelism is essential because it maximizes the expression of parallelism and frees the programmer from low-level details. As a counterpart, it is the compiler's task to fold, the most efficiently that can be achieved, the parallel computation on a given architecture. The expression of parallelism is made by operations manipulating space and time. Therefore, the implicit parallelism expression has to use an implicit uniform representation of the time and space. This leads us to focus on data-structures that have a time and space interpretation rather than to focus on the sequential flow control. The natural answer to this problem is the iterative data-flow introducing the notion of stream, data-parallelism with the notion of collection. The mix of these two notions into a single structure, the fabric, is studied through the experimental language 81/2. This high-level data-parallel language is independent from the execution support and takes advantage of the formal basis of the declarative and intensional languages.

In this article, we have sketched the basics of a 81/2 compiler and we have given the very first evaluation of some code generation benchmark. The algorithms used here are derived from the formal semantics of the language which ensure its correctness[51]. From the implementation point of view, a compiler for the static subset of 81/2 has already been implemented. It generates a code for a virtual SIMD machine implemented on a UNIX workstation and is able to generate an "external" static (stack-less and `malloc`-less) `C` code. This code does not use any run-time and is embeddable in any other `C` program. All the compiler phases assume a full MIMD execution model and we are currently working on the MIMD code generation[45]. A sequential interpreter is also available, allowing the computation of dynamically shaped fabrics (fabrics that involve collections of dynamical size). Remark that this interpreter triggers low-level vector operations (currently implemented in `C` as a virtual SIMD machine). So, this interpreter may exploit data-parallelism just by adapting the low-level virtual machine.

The current work on the 81/2 language concerns the extension of the notion of collection towards a group structure[52]. We are also working on the efficient treatment of dynamically shaped fabrics and their relations to symbolic computation.

6. Acknowledgments.

The authors wish to thank the members of the 81/2 team: Jean-Louis Giavitto and Abderrahmane Mahiout for many stimulating discussions.

7. References

1. J. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. of the IEEE*, 79(4), April 1991.
2. E. Ashcroft, A. Faustini, R. Jagannatha, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.
3. J.-P. Bânatre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.
4. O. Michel. Design and implementation of 81/2, a declarative data-parallel language. *special issue on Parallel Logic Programming in Computer Languages*, 1996. (to appear).
5. N. Paris. Définition de POMP-C (version 1.99). Technical Report LIENS95-2, Département de Mathématiques et d'Informatique de l'École Normale Supérieure, mars 1992.
6. Thinking Machine Corporation. *The Connection-Machine CM5 technical Summary*, October 1991.
7. Cray Research, Inc., Eagan, Minnesota. *CRAY T3D System Architecture Overview*.
8. F. Cappello, J.-L. Béchennec, and J.-L. Giavitto. PTAH: Introduction to a new parallel architecture for highly numeric processing. In *Conf. on Parallel Architectures and Languages Europe, Paris, LNCS 605*. Springer-Verlag, 1992.
9. G. Steele. Making asynchronous parallelism safe for the world. In *Seventeenth annual Symposium on Principles of programming languages*, pages 218–231, San Francisco, January 1990. ACM, ACM Press.
10. Grand challenges: High performance computing and communications. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, DC 20550, 1991.
11. P. J. Hatcher and M. J. Quinn. *Data-parallel programming on MIMD computers*. Scientific and engineering computation series. MIT Press, 1991.
12. D. Comte, G. Durrieu, O. Gelly, A. Plas, and J. C. Syre. Parallellism, control and synchronisation expressions in a single assignment language. *Sigplan Notices*, 13(1), 1978.
13. J. R. McGraw, S. K. Kedzielewski, S. Allan, R. Oldehoeft, J. Glauert,

C. Kirkham, W. Noyce, and R. Thomas. SISAL: Stream and iteration in a single assignement language. report Version 1.2 M-146, Lawrence Livermore National Laboratory, Livermore CA USA, March 1985.

14. J. R. McGraw. The VAL language. *ACM Trans. on Programming Languages and Systems*, 4(1), January 1982.
15. A. Plas, D. Comte, O. Gelly, and J. C. Syre. LAU system architecture: a parallel data-driven processor based on single assignment. In *Proceedings of the International Conference on Parallel Processing*, pages 293–302, 1976.
16. J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, Januaray 1985.
17. C. Leiserson and J. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computers Systems*, 1(1):41–67, 1983.
18. R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems*, 13(1):52–98, January 1991.
19. C. M. Hoffman, M. J. O'Donnel, and R. I. Strandh. Implementation of an interpreter for abstract equations. *Software, Practice and Experience*, 15(12):1185–1204, December 1985.
20. D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, February 1982.
21. R. S. Nikhil, K. Pingali, and Arvin. Id nouveau. CSG Memo 256, MIT Laboratory for Computer Science, Cambridge, MA, July 1986.
22. K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding,. *IEEE Trans. on Computers*, 40(2), February 1991.
23. P. Caspi, D. Pilaud, Halbwachs N., and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *Fourteenth annual symposium on principles of programming languages*. ACM, ACM Press, January 1987.
24. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP*, 34(2):362–374, 1986.
25. M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Principles of Programming Languages*, pages 131–139, Florida, 1986.
26. P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
27. W. W. Wadge and E. A. Ashcroft. Lucid - A formal system for writing and proving programs. *SIAM Journal on Computing*, 3:336–354, September 1976.
28. W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language*. Academic Press U. K., 1985.
29. J. Backus. Can programming be liberated from the von neumann style ? A functional style and its algebra of programs. *Com. ACM*, 21:613–641, August 1978.

30. N. Schmitz and J. Greiner. Software aids in PAL circuit design, simulation and verification. *Electronic Design*, 32(11), May 1984.
31. S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* ACM Distinguished Dissertations. ACM Press, 1983.
32. C. Delgado, P. Loos, K.Fritzson, and N. Andersson. Semantics of digital circuits. In *Lecture Notes in Computer Science*, number 285 in Lecture Notes in Computer Sciences. Springer Verlag, 1986.
33. K. Chandy and J. Misra. *Parallel Program Design - a Foundation.* Addison Wesley, 1989.
34. C. Mauras. Definition of Alpha: a language for systolic programmation. Technical Report 482, INRIA, June 1989.
35. J.-L. Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.
36. J. B. Denis. First version of a data flow procedure language. In Springer Verlag, editor, *Proceedings of the Programming Symposium*, April 9-11 1974.
37. J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montréal, 1992.
38. G. D. Smith. *Numerical solution of partial differential equations: finite difference methods.* Oxford Applied Mathematics and Computing series. Oxford University Press, 1985.
39. G. Kahn. The semantics of a simple language for parallel programming. In *proceedings of IFIP Congress'74*, pages 471–475. North-Holland, 1974.
40. W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.
41. B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.
42. J.-L. Giavitto. Otto e Mezzo: un modèle MSIMD pour la simulation massivement parallèle. Thèse de Doctorat de l'Université de Paris-Sud, France, 1991.
43. M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operation research*, 26(1), January-February 1978.
44. A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. 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.
45. A. Mahiout. Integrating the automatic mapping and scheduling for data-parallel dataflow applications on MIMD parallel architectures. In ??, editor, *Parallel Computing: Trends and Applications*, pages ??–??, 19-22 September,

Gent, Belgium, 1995. Elsevier. poster session.
46. J.-J. Hawang, Y.-C. Chow, F. Angers, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comp.*, 18(2):244–257, April 1989.
47. D. De Vito. Compilation portable d'un langage déclaratif à flot de données synchrones, Juin 1994. Rapport de stage du DEA Informatique de l'Université de Paris-Sud.
48. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.
49. Thinking Machines Corporation, Cambridge, Massachusetts. *Connection machine programming in C/Paris*, 1989.
50. D. De Vito and O. Michel. Effective SIMD code generation for the high-level declarative data-parallel language 81/2. In *Euro Micro '96*, 1996. To appear.
51. D. De Vito. Semantics and compilation of sequential streams into a static SIMD code for the declarative data-parallel language 81/2. Technical Report 1044, Laboratoire de Recherche en Informatique, May 1996. 34 pages.
52. J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Sciences*, pages 209–215, Beaune (France), 2-4 October 1995. Springer-Verlag.