81/2: data-parallelism and data-flow*

Olivier Michel†, Dominique De Vito and Jean-Paul Sansonnet

LRI u.r.a. 410 du CNRS
Bâtiment 490, Université de Paris-Sud
F-91405 Orsay Cedex, France.
Tel: 33.1.69.41.76.01
Fax: 33.1.69.41.65.86

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 rationale 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 is based upon a static execution model, enabling the efficient execution of programs is introduced. The concepts of fabric, collection and stream are described.


Topics: data-flow computation, intensional concepts, parallel programming, languages such as Lucid and GLU.

1 The parallelism and its expression through data

1.1 A taxonomy of parallelism in programming languages

The many existing architectures and languages used to program those architectures lead to an overlap of concepts making the design of a taxonomy a hard task. This is the reason why we propose in table 1 a framework used to specify the concepts presented in this article. 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>
<thead>
<tr>
<th></th>
<th>Declarative Languages</th>
<th>Sequential Languages</th>
<th>Concurrent Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-instruction counter</td>
<td>1-instruction counter</td>
<td>n-instruction counters</td>
</tr>
<tr>
<td><strong>Scalar Languages</strong></td>
<td>Sisal, Id, LAU, Actor languages</td>
<td>Fortran, C, Pascal</td>
<td>Ada, Occam</td>
</tr>
<tr>
<td><strong>Collection Languages</strong></td>
<td>Gamma, Apl, 81/2</td>
<td>*LISP, HPF, CMFortran</td>
<td>CMFortran + multi-threads</td>
</tr>
</tbody>
</table>

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

*This research is partially supported by the operation "Programmation parallèle et distribuée" of the GDR de programmation.
†email: michel101ri.fr
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 possible expression of sequencement by the programmer. It is the task of the compiler (static extraction of the parallelism), or the execution support (dynamical extraction through a software interpret or an hardware architecture) to construct a sequence of computation 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 ALT** 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** allow also the programmer to manipulate sets of data but only through a point-wise access of its elements. For example, in standard Pascal, the main operation on an array is the access of one of its items.

Data-parallelism comes from the introduction of parallelism onto sequential languages (this is the "starisation" 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 two alternatives and we will see in the next section that an embedding of data-parallelism in a concurrent framework is very attractive. Therefore, the "data-parallelism + data-flow" is, from the clarity point of view, the most interesting solution and may yield to efficient exploitation.

### 1.2 The problems of data-parallelism in 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 (corresponding to the sequencement of the scalar part and the parallel part of the program). For example, in the following Pomp-C program:

```plaintext
    collection[512, 512] Pixel;
    Pixel int picture;
    int a;
    ...
    picture = FALSE;
    where(picture){
        a = 5;
    }
```

- **picture** is a collection of integers.
- **a** is a scalar integer.
- All the points of **picture** are set to false.
- So, no processor should be active within the **where**.
- But this scalar assignment is performed because it is...
everywhere{
    printf("hello");
    picture = TRUE ;}
}

does not depend on the where.
reactivates locally all the processors.
since this is a scalar instruction, only one hello
will be printed.
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, a scalar instruction (a=5) or a parallel instruction in the immediate scope
of an everywhere (picture=TRUE).

More seriously, a where cannot therefore trigger a function call located into its scope. 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[6]):

collection generic int fact(generic int n)
{} if (nx=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. A special mechanism has to be introduced to get a correct recursive expression
of factorial: the recursivity is stopped when all the processors are idle. But this is semantically
incorrect.

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 (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 very naturally.

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 PT AH [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, MISIMD,
MIMD with firm synchronization . . . We generally believe that these architectures will reach the
TeraFlop required by the numerical applications of the “Grand Challenge”. It is therefore natural
to implement data-parallel languages on these new systems. Indeed, data-parallel languages are
well fitted to intensive numerical computation 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 [10]). The main problem is probably the bad
utilisation of computation resources due to the usage of a strictly sequential flow control, as en-
countered 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 \texttt{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-parallelism 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.

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 exhibiting 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 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 hierarchic 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 different strategies. For example, as far as functional languages\(^1\) are concerned, a \textit{confluence theorem} is used. In the case of languages derived from imperative languages (LAU [11], Sisal [12], Val [13], \ldots), a programming constraint has to be added: this is the \textit{single assignment} principle which allows the variable 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), 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 scheduling of the instructions is implicit. The programmer only describe the dependencies between the 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:

\[
\begin{align*}
\text{sum} &= a + b + c + d \\
\text{This program is a system of equations where the} \\
\text{order of the equations does not matter.}
\end{align*}
\]

\[
\begin{align*}
a &= \ldots \\
b &= \ldots \\
c &= \ldots
\end{align*}
\]

\(^1\)Pure functional languages (like FP) are data-flow languages: no side-effect, no explicit sequencement, no concept of memory nor assignment. But most of the functional languages in use (like ML) have imperative features (mutable structures for example) allowing side-effects
and let a tool automatically extract the parallelism, rather than to explicitly write the parallelization (in Occam):

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

In the Occam version of the 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

---

1. Data-flow graph of the program

```
T1: *
T2: *
T3: *
T4: -
```

2. Declarative style: \( \text{d} = b*b - 4*(a*c) \)

In a declarative program, the concept of variable corresponds to the mathematical notion: a variable has a "constant" value and the symbol := is a definition, not an assignment.

3. Functional style: \( ((\text{fork} ; *) || (* ; *)) ; - \)

In a pure functional program, there is no variables. The program is written using the primitive functions and the combinators to connect them. The combinators used here are:

- `fork` to duplicate a value
- `;` allowing the serial composition
- `||` allowing the parallel composition

4. Data-flow execution

![Data-flow execution diagram]

Figure 1: Data-flow representation in three different manners: graphical, declarative and functional of the Pascal program \( \text{d} := b*b - 4*a*c \). 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 coming from the simultaneous activation of the nodes on different processors.
computed leading to a maximal expression of the parallelism. This scheduling can be statically inferred (by a compiler that can take into account the peculiarities of the target architecture), or dynamically (by the execution support: an interpreter or a data-flow architecture [14, 15]).

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: there has been long discussions to decide if the choice of a guarded command in an ALT has to be theoretically fair or not and two actions PAR can occur in an undefined order (and, therefore, eventually in parallel). To get the desired deterministic result, the programmer has to express the sequencement, through the use of semaphores, guards, etc. On one hand, if too much sequencement is expressed, some parallelism is lost; on the other one, if too less sequencement 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 sequencement 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 and every reference to a variable can be replaced by its definition. So, it is easier to check and to achieve formal manipulations on the programs. The implementation or optimization of such programs can be done automatically [16, 17]. Finally, considering a program as a set of equations ensure that the program will compute a result, if this result is formally derivable from the set of initial equations: this is the declarative completeness property [18].

1.3.3 The drawbacks of the data-flow languages

In a famous article [19], Gajski 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 the I-structures in Id [20], 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 approach. 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 [21], real-time programming [22, 23], the design of systolic circuits and algorithms [24] and automatic parallelization [25]. It is now possible to develop statical 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’s not 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” passing values in time.

It is possible to extend this notion of a variable representing a sequence of values to all the variables of a program and not only those involved in loops. This is done in languages like Lucid\(^2\) [26, 27] 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 sequencement, 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 later 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 [28] or as *stream* in Lucid. This data-structure is used in various different applications domains: real-time programming with Lustre and Signal, PLD programming with Palaam [29], the VLSI design with Daisy [30] and Stream [31], the specification of parallel programs with Unity [32], the design of systolic circuits and algorithms through the notion of recurrent equations with Crystal [24] and Alpha [33].

- The stream is a communication paradigm: either in the pipe and its generalization to the sockets and streams in Unix. The communication scheme involved by a stream is static: the sender and receiver are known at compile-time.

- In a single assignment language, the stream is, quoting [27], a “mathematically respectable” structure and even [17] 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 and data-flow in the expression of temporal constraints. With these two concepts, it is possible to express all the parallelism present in an application. Expressing parallelism in excess (w.r.t. a target architecture) is very important because it lets the compiler: a) hide the communication delays by independent computations and b) chose parallel activities with a minimal management cost.

Furthermore, the expression of time and space relationship 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 program 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 widespread more and more (see section 1.3) seems to fit well with numerical problems (as seen in section 3).

\(^2\)we do not speak here of the new Lucid appeared recently [2]
But, in order to have 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 81/2.

2 The experimental language 81/2

We are studying the combination of the data-flow and data-parallel concepts and their implementation in a static execution model through an experimental language called 81/2 [34]. 81/2 introduces a new data structure, the type fabric corresponding to a temporal sequence (i.e. a stream) of collections, or dual point of view, a collection of streams [Cf. figure 2].

\[ \text{Fabric} = \text{collection of streams} = \text{stream of collections} \]

Figure 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;\ldots<.

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

\[ \{<1;1;1;\ldots<,1;1;1;\ldots<,1;1;1;\ldots<\} \]

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

2.1 A 81/2 program is a system of equations

A 81/2 program is a set of equations where each equation defines a fabric. Each equation has the following shape:

\[ z = f(x,y,\ldots,z) \]

where \(x, y, \ldots, z\) are fabric names and \(f\) a function over fabrics. A 81/2 program expresses a relation between the input fabrics (fabrics with no definition and correspond 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

\(^3\)The data structure was primarily called a “web”, but due to the extension of the WWW, we choose to call it now a fabric.
(this is similar to the “serendipity principle” of Lucid [2]). For example, the data-parallel addition + is implicitly extended so that:

\[
< \{1, 1, 1\} ; \{2, 2, 2\} ; \ldots < \\
+ < \{4, 5, 6\} ; \{6, 7, 8\} ; \ldots < \\
\Rightarrow < \{5, 6, 7\} ; \{8, 9, 10\} ; \ldots <
\]

(the sign \(\Rightarrow\) means “evaluates to”). In the second class, there are four different kind of function application (Cf. [1] for a presentation of these operations): the function application, the explicit \(\alpha\)-extension, the \(\beta\)-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 \(81/2\), 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 compilation-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 hold 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 \(\text{when}\). 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 \(\text{when}\) operator in \(81/2\) should not be mixed up with the \(\text{where}\) or \(\ast\text{when}\) operators of collection-based languages):

\[
\begin{align*}
T & = < 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 \ldots < \\
B & = < F ; F ; F ; T ; F ; T ; T \ldots < \\
\text{\$T} & \Rightarrow < 1 ; 2 ; 3 ; 4 ; 5 ; 6 \ldots < \\
T\ \text{when}\ B & \Rightarrow < 4 ; 5 ; 7 \ldots <
\end{align*}
\]

Table 2: Examples of delays and sampling on streams. The operators \$ and \(\text{when}\) introduce “holes” in the progression of the streams (Cf. section 3.2).

### 2.3 Examples of \(81/2\) 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 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:

\[
A = B \\
B = A
\]

defined \(A\) (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 \(81/2\) program computes the smallest solution in the sense of a particular
order relation, defined by the language semantic [36]. A naive 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 \(\text{cpt}\) defined below is a counter which counts at the rhythm of the true values of another fabric, \(\text{event}\), defined elsewhere:

\[
\text{cpt}(0) = 0 \\
\text{cpt} = (\#\text{cpt} + 1) \text{ when event}
\]

Equation (3) defines the initial value of the counter, 0. The next values are defined by the equation (4). The construction “\(\#\)” defines a temporal quantification for an equation defining a fabric: (3) is used to define the first value of \(\text{cpt}\) (value indexed by 0 in the stream) and equation (4) is used to compute all the 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,

\[
\text{iota}[10] = 0 \# (\text{iota} : [9] + 1)
\]

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 \(\text{iota}\) with 10 elements. The value of the first of these 10 elements is 0. If we denote \(\text{iota}_j\) the \(j^{th}\) element of \(\text{iota}\), and if we adopt the “fabric = collection of streams” point of view, the equation (5) can be rewritten in:

\[
\text{iota} = \{<0<\} \# (\text{iota}_0, \ldots, \text{iota}_9) : [9] + \{<1<, \ldots, <1<\}
\]

\[
\text{iota} = \{<0<, <1<, \text{iota}_0+<1<, \ldots, \text{iota}_8+<1<\}
\]

since \(\text{iota}_0 = 0\), then:

\[
\text{iota} = \{<0<, <1<, \text{iota}_0+<1<, \ldots, \text{iota}_8+<1<\}
\]

which shows that \(\text{iota}_1 = 1\), and therefore: \(\text{iota}_2 = 2\), etc. Finally:

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

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 at a given instant.

This is clearly not the most evident way to define an array where the \(n^{th}\) elements has for value \(n\). A simple yet efficient way would be: \(\text{iota}[10] = +1\) 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 \(81/2\) 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 (Cf. [36]).

2.3.2 An example: the resolution of partial differential equation by a finite difference method

The solution of the parabolic equation:

\[
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}
\]

gives the temperature \(u(x, t)\) at a distance \(x\) from one end of the rod after time \(t\). An explicit method of solution uses 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 [37]:
\[
\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2}
\]
which can be rewritten as
\[
U_{i,j+1} = rU_{i-1,j} + (1 - 2r)U_{i,j} + rU_{i+1,j}
\]
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 data of the problem), so are the temperature at $i = 0$ and $i = x/h$ (boundary conditions).

![Figure 3: Resolution by a finite difference method of a parabolic equation. The 81/2 equations corresponding to the initial conditions at the boundaries are in bold face.](image)

The corresponding 81/2 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):

\[
\begin{align*}
\text{Begin}[1] &= \text{conditions at one side of the rod} ; \\
\text{End}[1] &= \text{conditions at the other side of the rod} ; \\
\text{U}@0 &= \text{distribution of the temperatures (at } t = 0) \text{ along the rod} \\
U &= (\text{Begin} \# \text{V} \# \text{End}) \text{ when } Clock
\end{align*}
\]

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

\[
\begin{align*}
r &= k/(h \times h) \\
V[98] &= r \times left(SU) + (1 - 2 \times r) \times middle(SU) + r \times right(SU)
\end{align*}
\]

$V$, the inside of the rod, is a fabric of 98 elements. $\text{left()}$ (resp. $\text{right()}$) is a function shifting to the left (resp. to the right) a collection (like the $\text{EOSHIFT}$ intrinsic of Fortran90) and the function $\text{middle()}$ the segmentation of a collection. These operations are not primitives in 81/2, but are expressed in terms of communication operations of 81/2 (Cf. [4] for an extensive description of the language).
3 The compilation of a data-flow data-parallel language

The experimental language 81/2 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 81/2, based upon a formal semantic of the language in the style of [38]. In this section, we will only refer to the concepts developed in the 81/2 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 type systems are used in 81/2 to check that, at compile-time, the execution of a program has a known behaviour. 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, of the values of some parameters of its execution (the required memory for example). For example, a program of the following shape:

\[
T \otimes 0 = \{0\} \\
T = (\$T \neq \$T) \text{ when Clock}
\]

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

\[
T \Rightarrow \prec \{0\}; \{0,0\}; \{0,0,0,0\}; \ldots \prec
\]

each collection of the stream has twice more elements that 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 dynamic 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 support of the fabric. The value of a fabric at a given instant is a collection (of scalar values) called the instantaneous value of the fabric. A 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 and comparing two logical instants corresponds to the production of the \( n^{th} \) value of a stream.

The clock of a fabric \( T \) is a boolean stream that has, at the logical time \( t \), the \text{True} value if the fabric \( T \) produces an instantaneous value at this instant. An instant where the clock of \( T \) has the \text{True} value is called a tock; all the other instants are called ticks\footnote{A clock is thought to make tick-tock.}. 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, 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 \( T^N \), computed in the following way: by convention, the value at a tock of \( T \) is the value at this tock of \( T \); at a tick, the value of \( T \) 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 \( T \) corresponding to the instants where the clock of \( T \) has a \text{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
\[ \varepsilon = \text{if } \text{clock}(x) \text{ then } f(y) \] (13)

where \( \text{clock}(x) \) is an expression defining the clock of the fabric \( \varepsilon \). This expression is synthesized by induction on the structure of the definition of \( x \).\(^5\) For example,
\[ \text{clock}(A \text{ when } B) = B \land \text{clock}(B). \]

Applying this transformation to a Si/2 program produces its normal form. Roughly, the compiler will generate, for each expression of this program a task that will implement the equation (13). But the dependencies between the tasks are still to be computed to get their proper activation instants.

3.3 The tasks scheduling

The DFG associated to a Si/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 every elements of \( B \):

\[ A = +/\!\!\!B \]

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 only. Indeed, this DFG is cyclic (see figure 4) which will forbid 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 constructed because it contains as many nodes as there are points in the fabrics of the program (for example, matrixes are often of size 1000\( \times \)1000 in numerical computation and this would produce dependency graphs of over one million nodes).

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

- an expression \( e \) depends on the fabric \( x \) if \( x \) appears syntactically in \( e \). Nevertheless, we remove the dependencies corresponding to variables appearing in the scope of a delay: these 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 \( t \) if a point \( i \) of \( e \) depends on the value of all the points of \( x \) (total dependency).

- the dependency is annotated \( + \) if the value of point \( i \) depends on the value of the points \( j \) of \( x \) with \( j < i \).

\(^5\)The clock calculus in \( \text{Si}/2 \) 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 [34].
In the sequencement 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 expression which they depend on, have been computed.

Whereas in recursive definition of function, 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 elements of defined values. This problem is similar to the problem of the determination of the strictness of a function in static analysis. Another near problem is the detection of deadlocks in a data-flow graph [39] or the effectiveness of the recursive definition of a lazy list [40].

Actually, the effective processing of the whole sequencement graph is a little more complicated. Indeed, we previously assumed the hypothesis that the computations of the instantaneous values of $x$ do not depend on the instantaneous values of $z$, but the clock of $x$ depends on the clock of $z$ (it is the same one, but the first tock). Furthermore there can be instantaneous cycles between boolean expressions representing clock expressions in the sequencement 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 semantic 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 sequention 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 sequention 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 sequention graph with a total cycle whereas the semantic of the language gives the obvious solution \( \{0, 0\} \). In order to not reject such programs, more precise approximations of the scheduling graph are possible, but their implementation is more computation power consuming [41].

### 3.4 Static mapping and scheduling of the tasks

After the sequention 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 sequention graph. To solve this problem, we’ll 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.

![The sequency graph to fold](image)

Figure 5: Mapping and scheduling a sequencing graph using a two dimensional bin-packing method.

To produce the pattern, a rectangle in the time x 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 (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 [42].

We are currently testing a greedy strategy [43, 44] consisting in the early mapping of the possible part of a ready task. A task becomes ready at the end of the tasks which it depends on. 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 lately). 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 [45] 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 different target architectures.

4.1 The code generation scheme

The code generation scheme adopted 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 value of 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 \((dT, hT)\) 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 moment, the value of a fabric is computed only if those predicates hold.

- **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 \( d_i(t) \) then compute \( h_i(t) \)
if \( h_i(t) \) then compute the current data-parallel value of \( T \)
Save the current values accessed in 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 by a loop [46]. For vectorial or SIMD architectures, an ad hoc library can be used (like in CVL [47] or in the Paris Interface [48]).

### 4.2 A brief evaluation of the generated code

To evaluate the adopted generation scheme, we have performed a 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>
<thead>
<tr>
<th>Number of iterations ( \rightarrow )</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the rod ( \downarrow )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4.11</td>
<td>3.77</td>
<td>3.96</td>
<td>3.93</td>
<td>3.93</td>
</tr>
<tr>
<td>50</td>
<td>1.98</td>
<td>1.85</td>
<td>1.85</td>
<td>1.83</td>
<td>1.83</td>
</tr>
<tr>
<td>100</td>
<td>1.62</td>
<td>1.63</td>
<td>1.58</td>
<td>1.58</td>
<td>1.59</td>
</tr>
<tr>
<td>500</td>
<td>1.43</td>
<td>1.40</td>
<td>1.40</td>
<td>1.39</td>
<td>1.40</td>
</tr>
<tr>
<td>1000</td>
<td>1.43</td>
<td>1.42</td>
<td>1.41</td>
<td>1.42</td>
<td>1.42</td>
</tr>
<tr>
<td>5000</td>
<td>1.20</td>
<td>1.20</td>
<td>1.20</td>
<td>1.20</td>
<td>1.20</td>
</tr>
<tr>
<td>10000</td>
<td>1.36</td>
<td>1.35</td>
<td>1.35</td>
<td>1.36</td>
<td>1.35</td>
</tr>
</tbody>
</table>

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

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.

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. Fairly good results are obtained and validate the approach adopted.

### 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 the most present parallelism in an application and efficiently exploit it on the 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 which lead us to focus on the 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 evaluations of code generation. The algorithms used here are derived from the formal semantic of the language which ensure its correctness. 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 (stackless 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 [44]. 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 [49]. We are also working on the efficient treatment of dynamically shaped fabrics and their relations to symbolic computation.

Acknowledgments. The author wishes to thank the member of the 81/2 team: Jean-Louis Giavitto and Abderrahmane Mahiout for many stimulating discussions.

References


