Spatial computing as intensional data parallelism

Antoine Spicher, Olivier Michel, Jean-Louis Giavitto

To cite this version:

Antoine Spicher, Olivier Michel, Jean-Louis Giavitto. Spatial computing as intensional data parallelism. 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010), Sep 2010, Budapest, Hungary. pp.196–205, 10.1109/SASOW.2010.73 . hal-00869214

HAL Id: hal-00869214

https://hal.science/hal-00869214

Submitted on 11 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Spatial Computing as Intensional Data Parallelism

Antoine Spicher, Olivier Michel
LACL
Université Paris XII - Paris Est
94010 Créteil cedex, France
Email: {antoine.spicher, olivier.michel}@univ-paris12.fr

Jean-Louis Giavitto
IBISC
CNRS – Université d'Evry
91025 Evry cedex, France
Email: giavitto@ibisc.univ-evry.fr

***Abstract*—In this paper, we show that various concepts and tools developed in the 90's in the field of data-parallelism provide a relevant spatial programming framework. It allows high level spatial computation specifications to be translated into efficient low-level operations on processing units. We provide some short examples to illustrate this statement.**

***Keywords*-spatial computing, collection, data-fields, data flow, declarative definition, intensionnal programming, stream, data parallelism**

I. Spatial Computing

It is customary to make a distinction between operational and semantic properties of a computation. Operational properties affect how a computation is done, but not the result. A semantic property impacts what is computed. Traditional models of computation consider space as an operational property and have abstracted out physical locations in space: implementation predominantly performs computations in time (*i.e.* sequentially) and most of common data structures are spatially agnostic.

In contrast, *spatial computing* [1]–[4] explicitly recognizes computation as intertwined with the physical world: (1) moving information has a non-uniform cost depending on a notion of distance and (2) space appears as a semantic property of the computations (*e.g.* when the functional goal of the system is defined in terms of the system's spatial structure).

The increasing relevance of spatial computing (with the development of new computing media like cyberphysical systems, *ad hoc* mobile and sensor networks, ambient computing, smart dust [5], field-programmable gate arrays, etc.) leads us to ask: " What are the correct relevant abstraction for spatial computing ? "

Various spatial computing tasks have been identified and include: establishing a set of coordinates, generating fields, using gradients to move localized entities, computing Voronoï tesselations, flooding, querying a neighborhood, pattern formation, motion coordination... and more generally specifying a space and the computation which occurs at each location in this space. Ideally, we would like to program these spatial computations at a high level of abstraction so as to lighten the burden exposed to the programmer. We could layer these abstractions on top of an existing, space-agnostic, programming language or we can create new programing languages that naturally capture the spatial aspects and semantics of the spatial programming tasks.

In this paper, we advocate that some concepts and tools developed in the field of data parallelism provide a relevant spatial programming framework allowing high level spatial computation specifications to be efficiently translated into low-level behaviors of elementary processing units. We illustrate this statement using as an example the declarative data parallel programming language $8\frac{1}{2}$. This language was developed in the 90's for the parallel simulation of dynamical systems.

This paper is organized as follows. In the rest of this section, we introduce the notion of data parallelism (Sect. I-A) and the notion of collection (Sect. I-B) which enables the representation of entities with a spatial extension. Handling a collection as a whole allows the intensional definition of spatial computation (Sect. I-C).

The next section is dedicated to the presentation of $8\frac{1}{2}$ from a spatial computing perspective. The handling of data fields in an imperative setting presents some drawbacks that can be solved in a declarative framework. This leads us to introduce a new data structure, the *fabric* (Sect.. II-C), which combines the notion of collection (Sect. II-A) and the notion of stream (Sect. II-B).

Section III illustrates the previous notions on several spatial computations.

The conclusion summarizes our arguments for revisiting the data parallel paradigm in the light of spatial computing, but also points out the shortcomings of this approach.

A. Data Parallelism

There is an obvious link between the notions of parallelism and spatial computing: if two computations are done simultaneously, they must be done in different physical locations. Therefore, the expression of parallelism is tightly coupled with the expression of spatial relationships. Table I extends the Flynn classification of parallel architectures [6] giving a taxonomy of the various expressions of parallelism in programming languages [7]. This classification compares the parallel language features following two criteria: the way they let the programmer express the control and the way they let the programmer manipulate the data.

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

- *Implicit control*: this is the declarative approach. The compiler (static extraction of the parallelism) or the runtime environment (dynamic extraction by an interpreter

TABLE I
A CLASSIFICATION OF LANGUAGES FROM THE PARALLEL FEATURES POINT OF VIEW.

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

or a hardware architecture) has to build a computation order compatible with the data dependencies exhibited in the program.

- *Explicit control* which refines in:
  - *Express what has to be done sequentially*: this is the classical sequential imperative execution model, where control structures build only one thread of computation.
  - *Express what can be done in parallel*: this is the concurrent languages approach. Such languages offer explicit control structures like `PAR, ALT, FORK, JOIN`, etc.

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

- *Collection based languages* allow the programmer to handle sets of data as a whole. Such a set is called a collection [8]. Examples of languages of that kind are: APL, SETL, SQL, *Lisp, C*...
- *Scalar languages* allow also the programmer to manipulate a set of data but only through references to one element. For example, in standard Pascal, the main operation performed on an array is accessing one of its elements.

Historically, the data-parallelism paradigm has been developed from the possibility of introducing parallelism in sequential languages (this is the "starization" of languages: from C to C*, from Lisp to *Lisp...). It relies on sequential control structures (*when...) and collections. A collections is an organized set of data distributed on a space such that they can be managed simultaneously through global operations.

B. Collections and Data Fields

A *collection* is an aggregate of elements handled as a whole: no index manipulation or iteration loop appear in expressions over collections. For example, APL arrays are collections but Pascal arrays are not, because the only available operation on a Pascal array is *the access to one element*. The relationships between the spatial distribution of the collection's elements and the parallel operations are sketched in Fig. 1.

The global view on collections and collections operations corresponds to the coordination of local data structure and local operations done by a processing unit. The spatial properties of the computing medium are then reflected in the aggregation structure of the collection [9], [10]. Collections may have several aggregation structures — *sets* as in SETL [11], *bags* in Gamma [12], *relations* (set of tuples) in SQL. The array structure is one of the most popular: *vectors* (*e.g.* in *LISP), *nested vectors* (in NESL [13], $8\frac{1}{2}$ [7], [14]), and *multidimensional arrays* (HPF [15], MOA [16], Indexical Lucid [17]).

In this paper, we restrict ourself to arrays. However in spatial computing, the aggregation structure are more sophisticated: in [9], [18], [19] we have proposed to organize data structure using the general notion of neighborhood relationship as developed in *cellular complexes* [20], Proto [21]–[23] relies on *manifolds*.

Typical operations on "arrays as collections" are point-wise applied scalar functions, reductions, scans [24] and various permutations or rearranging operations that can be interpreted as communication operations in a data-parallel implementation (Cf. Sect. II-A).

Data fields (the term was introduced in the context of systolic computation [25]) has been proposed by Lisper to enrich the "arrays as collections" parallel data structure. They have been formalized as an extensional partial function from an index set to a value set, with a finite extent (*i.e.* its definition domain is finite) [26]–[28]. As a matter of fact, an array can be abstractly seen as a function from indices to values. This function is not defined intensionally as some "rule of computation" but it is defined extensionally as a set of pairs of index and element. Very often, pairs are listed following a predefined order on the indices, which relieves of storing the indices explicitly (this is the case for the usual implementation of dense arrays). There is an attractive advantage for partial functions: their domain is not restricted to be a n-dimensional box defined by a lower and upper bound in each dimension. Fields may have more complex or dynamic shapes.

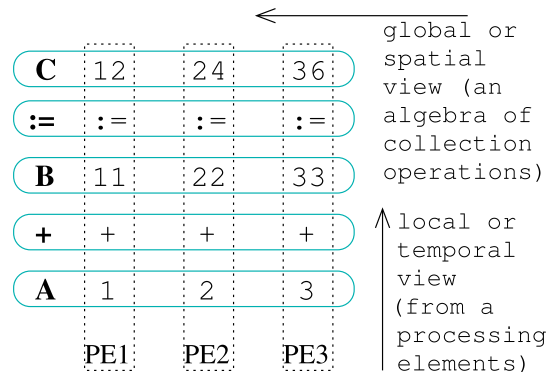

Fig. 1. Local and global view on a collection. The local view considers the data stored and the operations performed on one processing unit. The global view records the simultaneous operations performed on the collection.

The indices of a data field correspond to a physical location. In classical data parallel language, distribution and mapping statements are used to specify this correspondence.

C. Intensional Programming

Collections can be termed as "*intensional* data structures" because elements of a collection are not explicitly referred to [29]. The intensional definition of collections allows the management of the collection as a whole. The spatial distribution of the data elements is reflected in the properties of the collection algebra, especially in the properties of the various rearrangement operations which correspond to communication in a data parallel implementation [30].

Another well known intensional object is the function (like the concept of function found in the λ-calculus): in functional languages [31] functions are first class values that can be composed through a rich algebra of high-order operators without referring to the final function application. A function represents also an aggregate of data: the set of pairs (*argument*, *value*). This has to be compared with an array which represents a set of pairs of index and element. However, arrays *implement* this set *extensionally* by listing all the involved elements (in some predefined order to make implicit the link between indices and elements) whilst functions implement them *intensionally* as a process of going from an argument to a value. We describe the situation by the slogan

*collection (*i.e. *spatial data structure) =*
intensional specification of an extensional data structure.

The intensional approach, based on collection expressions, can be embedded in a variety of host programming languages. For example, HPF extends Fortran with array expressions like $C = A + B$ where A, B and C are arrays. Adding collection expressions in an imperative language presents several drawbacks [32]. Many propositions have been made to embed collection expressions in a declarative framework [26], [28].

In the rest of this paper, we present the approach developed within the $8\frac{1}{2}$ language: collection expressions are embedded in a data flow language.

II. The Fabric Data Structure

$8\frac{1}{2}$ has a single data structure called a *fabric*. A *fabric* is the combination of the concepts of *stream* and *collection*. This section describes these three notions.

A. The Notion of Collection in $8\frac{1}{2}$

A scalar is an indecomposable value. A collection is an aggregate of values. Here, we consider collections that are ordered sets of elements. An element of a collection, also called a *point* in $8\frac{1}{2}$, is accessed through an index. The expression $T.n$ where T is a collection and n an integer, is a collection with one point; the value of this point is the value of the nth point of T (point numbering begins with 0). If necessary, a collection with one point is implicitly coerced into a scalar and vice-versa through a type inference system described in [33].

1) Geometric Operations: Geometric operators change the *geometry* of a collection, *i.e.* its shape or structure. The geometry of a collection of scalars is reduced to its *cardinal* (the number of its points). A collection can also be *nested*: the value of a point is a collection. The geometry of the collection is the hierarchical structure of point values.

The first geometric operation consists in *packing* some collections together:

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

In the previous definition, a and b are collections resulting in a nested collection C.

The *composition* operator # concatenates the values:

$$A = \{a, b\}; \quad B = \{c, d\};$$
$$A\#B \Longrightarrow \{a, b, c, d\}$$

The last geometric operator is the *selection*: it generalizes the dot operator to build a new collection by selecting some values in a given collection. For example:

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

The notation $Source(target)$ has to be understood in the following way: a collection can be viewed as a function from $[0..n]$ to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is the set of natural numbers, collections can be composed and the following property holds:

$$Source(target).i = Source(target.i)$$

mimicking the function composition definition.

The dot operator and the selection imply communications of values between the processing units of a data parallel implementation. In a data parallel language, any kind of communication is allowed. In a spatial computer, the communications are restricted to the neighbors, which corresponds to restricting the geometry and the value of the $target$ argument.

2) Functions: Table II describes the four kinds of function applications defined in $8\frac{1}{2}$, where X means both scalar or collection, and p is the arity of the functional parameter f.

The first operator is the standard function application.

The second type of function applications produces a collection whose elements are the "point-wise" applications of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. *Extension is implicit* for the basic operators ($+$, $*$, the conditional, etc.) but is explicit for user-defined functions to avoid ambiguities between application and extension (consider the application of the $reverse$ function to a nested collection).

The third type of function application is the *reduction*. Reduction of a collection using the binary scalar addition results in the summation of all the elements of the collection. Any associative binary operation can be used, *e.g.* a reduction with the *min* function gives the minimal element

TABLE II
FUNCTION APPLICATIONS ON COLLECTIONS.

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

of a collection. The scan application mode is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\backslash\{1, 1, 1\} \Longrightarrow \{1, 2, 3\}$. See [24] for a complete algorithmics based on scan.

B. The Notion of Stream in 8½

1) Dealing with Infinite Sequence of Values: Streams are infinite sequences of values. Such data structure is implemented for instance in Haskell by lazy infinite lists [34]. In this case, a program only enumerates a bounded prefix of these infinite lists. Streams can also be implemented by processes as in Lustre [35] or Signal [36]. These two languages are originated in the field of real-time programming and a Lustre or a Signal program does not terminate: it processes "forever" the stream of inputs to produce a stream of outputs. The notion of stream makes one of its first appearance in the language Lucid [37] where they are defined by equations (*i.e.* recursive definitions).

8½ streams are very different from those of Lucid. They are tightly linked with the idea of observing a remanent state along time. If you observe a measuring apparatus during an experiment run, you can record the results of the successive measure operations on this apparatus, together with their dates. The timed sequence of data is a 8½ stream. At the very beginning, before the start of the experiment, the initial value of any observable is an undefined value. Then we record the initial value (at time 0 for some observable, later for some others). This value can be read and used to compute other values recorded elsewhere, as long as another observation has not yet been made.

The time used to label the observation is not the computer physical time, it is the logical time linked to the semantics of the program. The situation is exactly the same between the logical time of a *discrete-events simulation* and the physical time of the computer that runs the simulation. Therefore, the time which we refer to is a countable set of "events". An event is something meaningful for the computation, like a change in a point.

2) The Pace of a Stream (Ticks, Tocks and Clocks): 8½ is a declarative language which operates by making descriptive statements about data and relationships between data, rather than by describing how to produce them.

For instance, the definition $C = A + B$ means the value recorded by stream C is always equal to the sum of the values recorded by stream A and B. We assume that the changes of the values are propagated instantaneously. When A (or B) changes, so does C at the same logical instant. Note that C is uninitialized as long as A *or* B are uninitialized.

Table III gives some examples of 8½ streams operations. The first line gives the instants of the logical clock which counts the events in the program. The instants of this clock are called a **tick** (a tick is a column in the table). The dates of the recording of a new observation for a particular observable are called the **tock** of this stream. Tocks represent the set of events meaningful for that stream. A tock is a non-empty cell in the table.

You can always observe your measuring apparatus, which gives the result of the last measurement, until a new measure is made. Consequently, at a tick t, the value of a stream is: the last value recorded at the previous tock $t' \leq t$ if t' exists, the undefined value otherwise.

3) Stream Operations: A *scalar constant stream* is a stream with only one "measurement" operation, at the beginning of the time, to compute the constant value of the stream. A constant n in a 8½ program denotes a scalar constant stream.

Constructs like $Clock\, n$ denote another kind of constant streams: they are predefined sequences of $true$ values with an infinite number of tocks. The set of tocks depends on the parameter n. They represent some clocks used to give the beat of some other observations.

Scalar operations are extended to denote element-wise application of the operation on the values of the streams.

The delay operator $ shifts the entire stream to give access, at the current time, to the previous stream's value. This operator is the only operator that does not act in a point-wise fashion. The tocks of the delayed stream are the tocks of the arguments except for the first one. For example, the value of $\$C$ at tick *0* is undefined whilst its value at tick *4* is 3.

The last kind of stream operators is the set of sampling operators. The most general one is the *trigger*. It corresponds to the temporal version of the conditional. The values of "T when B" are those of T sampled at the tocks where B takes a $true$ value (see table IV). A tick t is a tock of "A when B" if A and B are both defined *and* t is a tock of B *and* the current value of B is $true$. The operator "until" can be computed in terms of "when": "A until B" is a stream which is like A until the first occurrence of B to $true$ and constant after this event. The "after" operator is the dual of "until": "A after B" is undefined until the first occurrence of $true$ in B and then it is like A.

8½ streams present several advantages:

- 8½ streams are manipulated as a whole, using filters, transducers… [38].
- A stream is the ideal implementation for the trajectory of a dynamical system: a temporal sequence of values

TABLE III
EXAMPLES OF CONSTANT STREAMS AND STREAM EXPRESSIONS.

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

is represented by a temporal succession of computations and therefore can be infinite.

- The tocks of a stream represent the logical instants where some computation must occur to maintain the relationships stated in the program.
- The $8\frac{1}{2}$ stream algebra verifies the *causality assumption*: the value of a stream at any tick t may only depend on values computed for previous tick $t' \leq t$. This is definitively not the case for Lucid (Lucid includes the inverse of $, an "uncausal" operator).
- The $8\frac{1}{2}$ stream algebra verifies the *finite memory assumption*: there exists a finite bound such that the number of past values that are necessary to produce the current values remains smaller than the bound (this is not the case for infinite lazy lists in Haskell).

Note that the implementation of $8\frac{1}{2}$ streams enables a static execution model: the successive values making a stream are the successive values of a single memory location and we do not have to rely on a garbage collector to free the unreachable past values (as in Haskell lazy lists for instance). In addition, we do not have to compute the value of a stream at each tick, but only at the tocks.

C. Combining Streams and Collections into Fabrics

A *fabric* is a *stream of collections* or a *collection of streams*. In fact, we distinguish between two *kinds* of fabrics: *static* and *dynamic*. A static fabric is a collection of streams where every element has the same clock (the clock of a stream is the set of its tocks). In an equivalent manner, a static fabric is a stream of collections where every collection has the same geometry. Fabrics that are not static are called dynamic. The $8\frac{1}{2}$ compiler is able to detect the kind of the fabric and compiles only the static ones. Programs involving dynamic fabrics are interpreted.

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

$8\frac{1}{2}$ is a declarative language: a program is a system representing a set of fabric definitions. A fabric definition takes a form similar to:

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

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

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

D. Recursive Definitions

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

1) Temporal Recursion: Temporal recursion allows the definition of the current value of a fabric using its past values. For example, the definition

$$\begin{aligned} T@0 &= 1 \\ T &= \$T + 1 \text{ when } Clock\,1 \end{aligned}$$

specifies a counter which starts at 1 and counts at the speed of the tocks of $Clock\,1$. The $@0$ is a temporal guard that

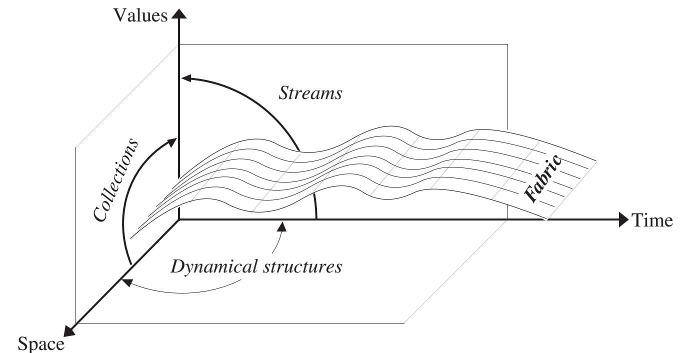

Fig. 2. A fabric specified by a $8\frac{1}{2}$ equation is an object in the ⟨time, space, value⟩ reference axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure.

TABLE IV
EXAMPLE OF A SAMPLING EXPRESSION.

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

quantifies the first equation and means "for the first tock only". In fact, T counts the tocks of $Clock\,1$.

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

2) Spatial Recursion: Spatial recursion is used to define the current value of a point using current values of other points of the same fabric. For example,

$$iota = 0\#(1 + iota : [4]) \tag{2}$$

is a fabric with 5 elements such that $iota.i$ is equal to i. The take operator $:[n]$ truncates a collection to n elements so we can infer from the definition that $iota$ has 5 elements (0 is implicitly coerced into a one-point collection and 1 into a collection with four elements). Let $\{iota_0, iota_1, iota_2, iota_3, iota_4\}$ be the value of the collection $iota$. The definition states that:

$$\{iota_0, iota_1, iota_2, iota_3, iota_4\} = \{0\}\#$$
$$(\{1,1,1,1\} + \{iota_0, iota_1, iota_2, iota_3\})$$

which can be rewritten as:

$$\begin{cases} iota_0 & = & 0 \\ iota_1 & = & 1 + iota_0 \\ iota_2 & = & 1 + iota_1 \\ iota_3 & = & 1 + iota_2 \\ iota_4 & = & 1 + iota_3 \end{cases}$$

which proves our previous assertion. See also Fig. 3.

We have developed the notions that are necessary to check if a recursive collection definition has a well-defined solution and to solve it efficiently. The solution can always be defined as the least solution of some fixpoint equation. However, an equation like "$x = \{x\}$" does not define a well formed array (the number of dimensions is not finite). We insist that all elements of the array solution must be defined and can be computed by a statically predefined enumeration of the elements [39].

III. EXAMPLES IN $8\frac{1}{2}$

A. Three computations of the Factorial Function

We present three ways to compute, for a given value n, $n!$ in $8\frac{1}{2}$. The first one uses the temporal recursion to enumerate the successive values of $n!$:

$$\begin{array}{rcl} i@0 & = & 1 \\ i & = & \$i + 1 \text{ when } Clock\,1 \\ fact@0 & = & 1 \\ fact & = & i * \$fact \end{array}$$

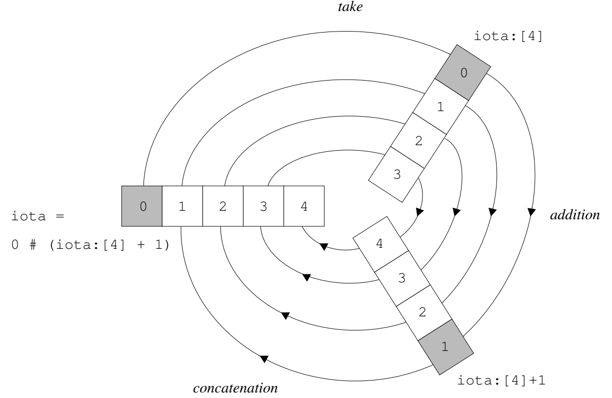

Fig. 3. Resolution of the iota equation.

The value of $fact$ at tock n is $n!$: i is a counter and $fact$ computes the products.

Instead of enumerating the values of $n!$ in time, we can compute it in space, using the scan operation:

$$fact \quad = \quad *\backslash\backslash(+\backslash\backslash 1 : [n])$$

The expression $1 : [n]$ computes a vector of n elements filled with 1. Thus, expression $(+\backslash\backslash 1 : [n])$ is a vector which enumerates the integer from 1 to n. The scan of this vector with the multiplication gives the desired results.

The spatial computation can also be done using spatial recursion (the scan operator hides an implicit but simple form of spatial recursion):

$$\begin{array}{rcl} i & = & 1 + (+\backslash\backslash(1 : [n])) \\ fact & = & (1\#(fact * i)) : [n] \end{array}$$

Collection i enumerates the integer from 2 to $n + 1$ and the definition of $fact$ uses the spatial recursion in a manner similar to $iota$.

B. From Vector to Arrays

We have presented $8\frac{1}{2}$ collection as (nested) vectors, however $8\frac{1}{2}$ handles several kinds of collections: *multidimensional arrays*, *data fields* [40], *GBF* [41], [42] (partial arrays whose elements are indexed by an element in a group) and amalgams [43].

We stick here with the notion of collections as vectors and shows how to emulate a 2D grid with a NEWS neighborhood using nested vectors: a 2D grid is a collection of columns. Two auxiliary functions *left* and *right* are used to shift a vector to the left or to the right:

$$\begin{array}{rcl} \text{function } left(x, c) & = & (x\#c)('x + 1) \\ \text{function } right(x, c) & = & (c\#x) : [x] \end{array}$$

where the argument c gives the value on the boundary. For example, $\mathit{left}(\{0,1,2\},33)$ returns $\{1,2,33\}$ and $\mathit{right}(\{0,1,2\},33)$ returns $\{33,0,1\}$.

In the definition of right, expression $A:[x]$ restricts the collection A to the geometry of collection x. In the definition of left, expression $'x$ computes a collection with the same geometry as x that enumerates the integers. The expression $'x$ is an abbreviation for:

$$'x = +\backslash\backslash(one\hat{\ }x)$$

where one is the constant function that returns 1.

With these functions, it is easy to define the functions giving access to the NEWS neighbors:

$$\begin{aligned} N(x,c) &= \mathit{right}\hat{\ }(x, c:[x]) \\ E(x,c) &= \mathit{left}(x, c:[x]) \\ W(x,c) &= \mathit{right}(x, c:[x]) \\ S(x,c) &= \mathit{left}\hat{\ }(x, c:[x]) \end{aligned}$$

C. The Eden Growth Model

The previous functions can be used to program a growth process similar to the Eden growth model [44]. In this model, a space is partitioned in empty or occupied cells. At each step, empty cells may be invaded by an occupied neighbor:

$$\begin{aligned} \mathit{start} &= \text{initial condition...} \\ \mathit{eden}@0 &= \mathit{start} \\ \mathit{Deden}@0 &= \mathit{false} \\ \mathit{Deden} &= \$\mathit{eden} \\ \mathit{neighbors} &= N(\mathit{Deden},\mathit{false}) \mid S(\mathit{Deden},\mathit{false}) \\ &\quad \mid E(\mathit{Deden},\mathit{false}) \mid W(\mathit{Deden},\mathit{false}) \\ \mathit{ok} &= \text{if } (\mathit{neighbors}\ \&\ \mathit{not}(\mathit{Deden})) \\ &\quad\ \text{then } \mathit{random}() \text{ else } \mathit{false} \\ \mathit{eden} &= (\text{if } \mathit{ok} \text{ then } \mathit{true} \text{ else } \mathit{Deden}) \text{ when } \mathit{Clock}\,1 \end{aligned}$$

Operators $\mid$ and $\&$, corresponding to the usual disjunctive and conjunctive boolean predicates, are naturally extended to collections. The function $\mathit{random}()$ returns a random boolean value. At each tock, for each element in the grid which is not occupied, if a neighbor is occupied (*i.e.*, holds a true value), the element randomly chooses to stay empty or to become occupied.

D. Computing the Connected Components in an Image

We want to determine, in a spatial way, the connected components of an image I. For the sake of simplicity, we suppose that I is a black and white image. A connected component is a maximal set of black points. The idea is to label all the points belonging to a connected component by an integer: all points in a connected component share the same integer and two different components are identified by different integers. A point which does not belongs to a connected component is labeled by -1.

The algorithm consists in assigning a different integer value to each point and then to propagate this value to the neighbors belonging to the same component. Between two labels, a point chooses the label with a maximal value. We iterate the procedure until a fixpoint is reached.

The first step is to generate a different label for each point. This can be done using the auxiliary function

$$\text{function } \mathit{findex}(\mathit{col}) = {}'\mathit{col} + \mathit{col} * |\mathit{col}|$$

Let I be the initial B&W image, then

$$\mathit{index} = \mathit{findex}\hat{\ }('I)$$

computes a collection index with the same geometry as I and where each point has a different value. For example, if I is an array of 256×256 pixels, then $\mathit{index} = \{\{1, 2, \ldots, 256\}, \{257, \ldots\}, \ldots\}$. Let c the fabric that will compute the labeling. Then, the definition of c is given by the following equations:

$$\begin{aligned} c@0 &= \text{if } I \text{ then } \mathit{index} \text{ else } -1 \\ c &= \text{if } I \text{ then } \max(c1, c2, c3, c4) \text{ else } \$c \\ &\quad\ \text{until } \mathit{fixpoint} \\ dc@0 &= -1 \\ dc &= \$c \text{ when } \mathit{Clock}\,1 \\ \mathit{fixpoint} &= \&\backslash(\$dc == dc) \\ c1 &= \mathit{max}(dc, N(dc,-1)) \\ c2 &= \mathit{max}(dc, E(dc,-1)) \\ c3 &= \mathit{max}(dc, W(dc,-1)) \\ c4 &= \mathit{max}(dc, S(dc,-1)) \end{aligned}$$

Computing the number of connected components ncc is immediate:

$$\begin{aligned} x &= \text{if } (\mathit{index} == c \text{ after } \mathit{fixpoint}) \text{ then } 1 \text{ else } 0 \\ ncc &= +\backslash x \end{aligned}$$

E. Spatial Handling of Combinatorial Computations

The example of the computation of $n!$ has shown that 8½ expresses very naturally simple algorithms where each event in time is used to compute new values using previously computed ones. This is the "stream of collections" point of view on fabrics. We extend this point of view here by showing the computation of collections which geometry changes in time. Following the terminology introduced in Sect. II-C, we deal here with *dynamic fabrics*.

1) Stirling's Numbers: Our first example is motivated by the work of the Chinese mathematician Li Shanlan (1811-1882) who has defined a method allowing to spatially compute sequences of integers: the sequence is represented as a pyramid where each element is the result of a computation involving its immediate neighbors and previously computed values. For

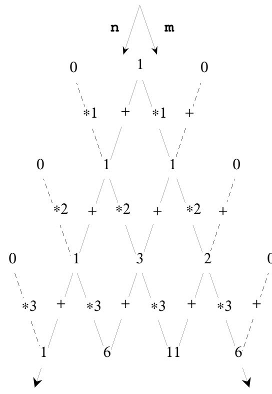

Fig. 4. The first four steps of the spatial computation of the Stirling's numbers.

example, Stirling's numbers of the first kind defined as

$$\begin{aligned} S(0,0) &= 1 & & (3)\\ S(n,0) &= 0 & n \neq 0 & \\ S(0,m) &= 1 & m \neq 0 & \quad (4)\\ S(n,m) &= (n-1)\,S(n-1,m) & n,m \neq 0 & \quad (5)\\ &\quad + S(n-1,m-1) & & \end{aligned}$$

can be computed by considering n in time and by associating a vector with the computation of m. Figure 4 represents the first values of S. The $8\frac{1}{2}$ program is straightforward

$$\begin{aligned} s@0 &= 0 & \text{/* from (3) */}\\ s@1 &= 1 \text{ when } Clock\,1 & \text{/* from (3) */}\\ s &= (|\$f| * \$f) + (\$s \# 0) & \text{/* from (5) */}\\ f@0 &= 1 & \text{/* from (4) */}\\ f &= 1\#s & \end{aligned}$$

Operator $|x|$ computes the *rank* of the fabric x, that is a vector where the i$^{\text{th}}$ value is equal to the size of the i$^{\text{th}}$ dimension of x. Consequently, expression $\$f$ is a counter corresponding to the $(n-1)$ coefficient used to multiply each previous computed line to obtain the current line. The five first lines of the Stirling's triangle are:

```
Tock: 0 : { 1 } : int[1]
Tock: 1 : { 1, 1 } : int[2]
Tock: 2 : { 1, 3, 2 } : int[3]
Tock: 3 : { 1, 6, 11, 6 } : int[4]
Tock: 4 : { 1, 10, 35, 50, 24 } : int[5]
```

2) Pascal's Triangle: It is folk's knowledge that the computational scheme of the binomial coefficients C_n^p maps a triangle, namely Pascal's triangle. In $8\frac{1}{2}$, the triangle can be implemented by mapping columns in a collection and having the computation on each line as an element of the stream.

If we focus on the parity of the numbers found in Pascal's triangle we obtain a triangle of zeros and ones: Pascal's triangle *modulo* 2. This (recursive) triangle can be spatially computed following the scheme described in Fig. 5: if T is such a triangle then the new triangle obtained by first concatenating T aligned with the upper left corner of T with the lower left corner and then by concatenating T again to the previous result by aligning the the upper left corner with the lower right corner, is also a Pascal's triangle. The following $8\frac{1}{2}$ program handles the rectangular shape that embeds the triangle (since we restricted ourselves to only rectangular regions). The program is quite straightforward and expresses faithfully the building process:

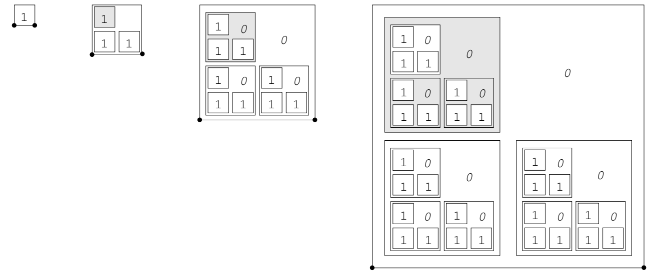

Fig. 5. Recursive building of Pascal's triangle modulo 2. The dark dots correspond to the vertices where the triangle is duplicated. The initial triangle of the construction appears in gray.

$$\begin{aligned} p &= ((\$p \# 0 : |\$p|) \#\hat{}\ (\$p \# \$p)) \text{ when } Clock\,1\\ p@0 &= 1 : [1,1] \end{aligned}$$

We use here a specialized version of the concatenation operator $\#\hat{}$ to allow the concatenation of the second argument *below* the first one. The 4 first tocks of the program are

```
Tock: 0 : { { 1 } } : int[1, 1]

Tock: 1 : { { 1, 0 }, { 1, 1 } } : int[2, 2]

Tock: 2 : { { 1, 0, 0, 0 },
            { 1, 1, 0, 0 },
            { 1, 0, 1, 0 },
            { 1, 1, 1, 1 } } : int[4, 4]

Tock: 3 : { { 1, 0, 0, 0, 0, 0, 0, 0 },
            { 1, 1, 0, 0, 0, 0, 0, 0 },
            { 1, 0, 1, 0, 0, 0, 0, 0 },
            { 1, 1, 1, 1, 0, 0, 0, 0 },
            { 1, 0, 0, 0, 1, 0, 0, 0 },
            { 1, 1, 0, 0, 1, 1, 0, 0 },
            { 1, 0, 1, 0, 1, 0, 1, 0 },
            { 1, 1, 1, 1, 1, 1, 1, 1 } } : int[8, 8]
```

which are spatially rendered in Fig. 6.

IV. Conclusions

The $8\frac{1}{2}$ language has been implemented in several ways: an interpreter in Ocaml [7], a compiler (to C) for the static subset of the language [45], a tool for the mapping and the scheduling of (static) collection operations on parallel computers [46] and a distributed implementation of collection computations on a grid [47].

The examples of section III are toy examples but give a good idea of the possibilities brought by the intensional manipulation of fabrics. Others examples, including advanced cellular automata similar to those used in blob computing [48] have been developed.

The purpose of this article is to show that the notion of collections (parallel data structure) is tightly coupled with a spatial abstraction of the values distributed on a network

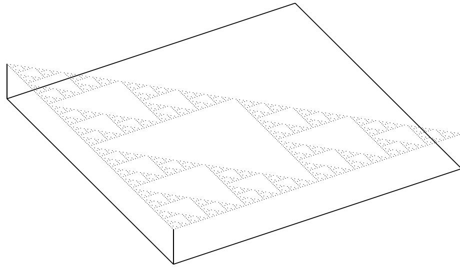

Fig. 6. Seven first steps of the building of Pascal's triangle (producing the famous Sierpiński's gasket).

of processing elements. Combining collection with streams enables the scheduling of sets of local computations. Note that the idea to merge the notions of collections and streams is also exploited in Proto.

One advantage of the equational definition of fabrics allowed in $8\frac{1}{2}$ is that equational reasoning can be used to refine or implement a program. This line of research has been investigated for example by Skillicorn and coworkers [49].

Originated in the data parallel community in the 90's, $8\frac{1}{2}$ lacks completely some of the features investigated for example in the amorphous computing community: amorphous medium, asynchrony, fault tolerance, self-healing, *etc*. They have not been tackled in the framework of data parallelism and are the subject of future works.

Spatial computing also includes *space computation* where space is fundamental to the problem and is a result of a computation. In this area also, $8\frac{1}{2}$ suffers some drawbacks. One of the main shortcomings is the difficulty to compose streams and *dynamic* collections. As a matter of fact, stream equations are defined *a priori* which constraints *a priori* all the collections that appear in a program. Some solution have been investigated with the notion of *amalgams* [50]. An amalgam represents a partially specified computation that awaits to be completed before to be run. However this approach is not flexible enough.

These lessons learned from the $8\frac{1}{2}$ project have lead to the development of the MGS project [9]. MGS investigates the flexible management of complex spatial abstractions using local transformation rules.

Acknowledgment

The authors would like to thank J.-P. Sansonnet, A. Mahiout and D. De Vito that have actively collaborated on the $8\frac{1}{2}$ project.

References

[1] A. De Hon, J.-L. Giavitto, and F. Gruau, Eds., *Computing Media and Languages for Space-Oriented Computation*, ser. Dagsthul Seminar Proceedings, no. 06361. Dagsthul, http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=2006361, 3-8 sptember 2006.

[2] H. Berry, J.-L. Giavitto, F. Gruau, and O. Michel, "From amorphous to spatial computing - workshop. paris." July 2008, http://amorphous.ibisc.univ-evry.fr.

[3] J. Bachrach, J. Beal, O. Michel, J. Werfel, and D. Yamins, "Spatial computing workshop," October 2008, http://projects.csail.mit.edu/scw08/index.html.

[4] F. Zambonelli and M. Mamei, "Spatial computing: a recipe for self-organization in distributed computing scenarios," 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.106.5917

[5] K. S. J. Pister, "Smart dust - hardware limits to wireless sensor networks," in *ICDCS*. IEEE Computer Society, 2003, p. 2. [Online]. Available: http://csdl.computer.org/comp/proceedings/icdcs/2003/1920/00/19200002abs.htm

[6] M. J. Flynn, "Some computers organizations and their effectiveness," *IEEE Trans. on Computers*, vol. C-21, pp. 948–960, 1972.

[7] O. Michel, "Design and implementation of 81/2, a declarative data-parallel language," *Computer Languages*, vol. 22, no. 2/3, pp. 165–179, 1996, special issue on Parallel Logic Programming. [Online]. Available: ftp://ftp.lri.fr/LRI/articles/michel/elsevier96.ps.gz

[8] J. M. Sipelstein and G. Blelloch, "Collection-oriented languages," *Proceedings of the IEEE*, vol. 79, no. 4, pp. 504–523, Apr. 1991. [Online]. Available: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-90-127.ps.Z

[9] J.-L. Giavitto and O. Michel, "Data structure as topological spaces," in *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, vol. 2509, Himeji, Japan, Oct. 2002, pp. 137–150, lecture Notes in Computer Science.

[10] J.-L. Giavitto, O. Michel, and A. Spicher, *Software-Intensive Systems and New Computing Paradigms*, ser. LNCS. Springer, november 2008, vol. 5380, ch. Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems, pp. 235–254. [Online]. Available: http://www.springerlink.com/content/g1357n85j8301078/?p=a5c6f79393724a9d88f508d110a8bfe2&pi=6

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

[12] J.-P. Banatre, A. Coutant, and D. L. Metayer, "A parallel machine for multiset transformation and its programming style," *Future Generation Computer Systems*, vol. 4, pp. 133–144, 1988.

[13] G. Blelloch, "NESL: A nested data-parallel language (version 2.6)," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-93-129, April 1993.

[14] J.-L. Giavitto, "A synchronous data-flow language for massively parallel computer," in *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, D. J. Evans, G. R. Joubert, and H. Liddell, Eds., London, 3–6 Sep. 1991, pp. 391–397.

[15] *High Performance Fortran Language Specification*, 1st ed., Rice University, Houston, Texas, May 93. [Online]. Available: http://www.erc.msstate.edu/hpff/hpf-report-ps/hpf-v11.ps

[16] G. Hains and L. M. R. Mullin, "An algebra of multidimensional arrays," Université de Montréal, Tech. Rep. 782, 1991.

[17] E. A. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge, *Multidimensional Programming*. Oxford University Press, February 1995, iSBN 0-19-507597-8.

[18] J.-L. Giavitto, "Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems," in *Rewriting Technics and Applications (RTA'03)*, ser. LNCS, vol. LNCS 2706. Valencia: Springer, Jun. 2003, pp. 208 – 233.

[19] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," in *Unconventional Programming Paradigms (UPP'04)*, ser. LNCS, vol. 3566. Le Mont Saint-Michel: Spinger, Sep. 2005, pp. 137–152.

[20] J. Munkres, *Elements of Algebraic Topology*. Addison-Wesley, 1984.

[21] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, pp. 10–19, March/April 2006.

[22] ——, "Programming manifolds," in *Computing Media and Languages for Space-Oriented Computation*, ser. Dagstuhl Seminar Proceedings, A. DeHon, J.-L. Giavitto, and F. Gruau, Eds., no. 06361. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informa tik (IBFI), Schloss Dagstuhl, Germany, 2007.

[23] J. Bachrach, J. Beal, and T. Fujiwara, “Continuous space-time semantics allow adaptive program execution,” in *IEEE SASO 2007*, July 2007.

[24] G. Blelloch, “Scans as primitive parallel operations,” *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.

[25] J. A. Yang and Y.-i. Choo, “Data fields as parallel programs,” in *Proceedings of the Second International Workshop on Array Structure*, Montreal, Canada, June/July 1992.

[26] B. Lisper, “On the relation between functional and data-parallel programming languages,” in *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*, ACM. ACM Press, Jun. 1993.

[27] B. Lisper and J.-F. Collard, “Extent analysis of data fields,” Royal Institute of Technology, Sweden, Tech. Rep. TRITA-IT R 94:03, January 1994.

[28] B. Lisper, “Data parallelism and functional programming,” in *Proc. ParaDigme Spring School on Data Parallelism*. Springer-Verlag, Mar. 1996, les Ménuires, France.

[29] M. A. Orgun and E. A. Ashcroft, Eds., *Intensional Programming I*. Macquarie University, Sydney Australia: World Scientific, May 1995.

[30] D. B. Skillicorn, “Architecture-independent parallel computation,” *IEEE Computer*, vol. 23, no. 12, pp. 38–49, Dec. 1990.

[31] J. Backus, “Can programming be liberated from the von neumann style ? A functional style and its algebra of programs,” *Com. ACM*, vol. 21, pp. 613–641, Aug. 1978.

[32] L. Bougé, “The data parallel programming model: A semantic perspective,” in *The Data Parallel Programming Model*, ser. Lecture Notes in Computer Science, vol. 1132. Springer, 1996, pp. 4–26.

[33] J.-L. Giavitto, “Typing geometries of homogeneous collection,” in *2nd Int. workshop on array manipulation, (ATABLE)*, Montral, 1992.

[34] P. Hudak *et al.*, *Report on the programming language HASKELL a non-strict, purely functional language, version 1.3*, Yale University, CS Dept., May 1996. [Online]. Available: ftp://haskell.systemsz.cs.yale.edu/pub/haskell/report

[35] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, “LUSTRE: A declarative language for programming synchronous systems,” in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN. Munich, West Germany: ACM Press, Jan. 21–23, 1987, pp. 178–188.

[36] A. Benveniste, P. L. Guernic, and C. Jacquemot, “Synchronous programming with events and relations: the SIGNAL language and its semantics,” *Science of Computer Programming*, vol. 16, pp. 103–149, 1991.

[37] W. W. Wadge and E. A. Ashcroft, “Lucid - A formal system for writing and proving programs,” *SIAM Journal on Computing*, vol. 3, pp. 336–354, Sep. 1976.

[38] Arvind and J. D. Brock, “Streams and managers,” in *Proceedings of the 14th IBM Computer Science Symposium*, 1983.

[39] J.-L. Giavitto, “A framework for the recursive definition of data structures,” in *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*. Montral: ACM-press, Sep. 2000, pp. 45–55.

[40] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet, “A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$,” in *EuroPar'98 Parallel Processing*, ser. Lecture Notes in Computer Science, Sep. 1998.

[41] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet, “Group based fields,” in *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, ser. Lecture Notes in Computer Science, I. Takayasu, R. H. J. Halstead, and C. Queinnec, Eds., vol. 1068. Beaune (France): Springer-Verlag, 2-4 October 1995, pp. 209–215. [Online]. Available: ftp://ftp.lri.fr/LRI/articles/michel/psls95.ps.gz

[42] J.-L. Giavitto and O. Michel, “Declarative definition of group indexed data structures and approximation of their domains.” in *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, Sep. 2001.

[43] O. Michel, “Introducing dynamicity in the data-parallel language 81/2,” in *EuroPar'96 Parallel Processing*, ser. Lecture Notes in Computer Science, L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., vol. 1123. Springer-Verlag, Aug. 1996, pp. 678–686. [Online]. Available: ftp://ftp.lri.fr/LRI/articles/michel/europar21-96.ps.gz

[44] M. Eden, “A two-dimensional growth process,” in *Proceedings of Fourth Berkeley Symposium on Mathematics, Statistics, and Probability*, vol. 4. University of California Press, Berkeley, 1961, pp. 223–239.

[45] J.-L. Giavitto, D. De Vito, and O. Michel, “Semantics and compilation of recursive sequential streams in 81/2,” in *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, ser. Lecture Notes in Computer Science, H. Glaser and H. Kuchen, Eds., vol. 1292. Southampton: Springer-Verlag, 3–5 Sep. 1997, pp. 207–223. [Online]. Available: ftp://ftp.lri.fr/LRI/articles/michel/plilp97.ps.gz

[46] A. Mahiout and J.-L. Giavitto, “Data-parallelism and Data-flow: automatic mapping and scheduling for implicit parallelism,” in *Franco-British meeting on Data-parallel Languages and Compilers for portable parallel computing*, Villeneuve d'Ascq, 20 avril, 1994.

[47] D. De Vito and O. Michel, “Effective SIMD code generation for the high-level declarative data-parallel language 81/2,” in *EuroMicro'96*. IEEE Computer Society, 2–5 Sep. 1996, pp. 114–119. [Online]. Available: ftp://ftp.lri.fr/LRI/articles/devito/euromicro96.ps.gz

[48] F. Gruau, Y. Lhuillier, P. Reitz, and O. Temam, “Blob computing,” in *Proceedings of the 1st conference on Computing frontiers*. ACM New York, NY, USA, 2004, pp. 125–139.

[49] D. B. Skillicorn, *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[50] O. Michel and J.-L. Giavitto, “Amalgams: Names and name capture in a declarative framework,” LaMI – Universit d'vry Val d'Essonne, Tech. Rep. 32, Jan. 1998, also avalaible as LRI Research-Report RR-1159. [Online]. Available: \url{ftp://ftp.lami.univ-evry.fr/pub/publications/reports/1998/lami_32.ps.gz}