The Foundation of Self-developing Blob Machines for Spatial Computing

Frédéric Gruau, Christine Eisenbeis, Luidnel Maignan

The Foundation of Self-developing Blob Machines for Spatial Computing

Frédéric Gruau [a,b,c,d] Christine Eisenbeis [b] Luidnel Maignan [b]

[a] *LRI - Université de Paris-Sud 11, bâtiment 490, 91405 Orsay Cedex, France*
[b] *Inria Futurs Saclay - Parc Orsay Université, 4, rue Jacques Monod, 91893 ORSAY Cedex, France*
[c] *Laboratoire d'informatique, de robotique et de microélectronique de Montpellier, 31 rue Ada, 34000 Montpellier, France*
[d] *University of the West of England, Frenchay Campus, Coldharbour Lane Bristol BS16 1QY, United Kingdom*

Abstract

The current trend in electronics is to integrate more and more transistors on a chip and produce massive hardware resources. As a consequence, traditional computing models, which mainly compute in the temporal domain, do not work well anymore since it becomes increasingly difficult to orchestrate these massive-scale hardware resources in a centralized way. Spatial computing is a unifying term that embodies many unconventional computing models and means computing on a relatively homogeneous physical medium made of hardware components, where the communication time is dependent on the euclidean distance between the components (locality constraint). This constraint makes the programming for high performance significantly more complex compared to classical non-spatial hardware because performance now depends on where computation happens in space (mapping problem). Blob computing is a new approach that addresses this parallel computing challenge in a radically new and unconventional way: it decouples the mapping of computations onto the hardware from the software programming while still elegantly exploiting the space of the underlying hardware. Hardware mapping of computations is done by a physical force-based approach that simulates forces between threads of computation (automata). Attractive forces are used to keep automata that need to communicate with each other closer while repulsive forces are used for load balancing. The advantage of theses primitives is that they are simple enough to be implemented on an arbitrary computing medium. They form the basis of a runtime system (RTS) that transforms an arbitrary computing medium into an easier-to-program virtual machine called the blob machine. The basic objects of the blob machine are those automata, and the instructions let automata create new automata in specific ways so as to maintain a hierarchical organisation (which facilitates both the mapping and the programming). We detail the basic instructions of the blob machine and demonstrate their confluence. Programming a spatial medium to perform a given algorithm then boils down to programming the blob machine, provided the RTS is implemented on it. The advantage of this approach is the hardware independency, meaning that the same program can be used on different media. By means of several examples programmed using a high level langage description, we further show that we can efficiently implement most current parallel computing models, such as Single Instruction Multiple Data (SIMD), data parallelism, "divide and conquer" parallelism and pipelining which demonstrates parallel expressiveness. On sorting and matrix multiplication algorithms, we also show that our approach scales up optimally with the number of basic hardware components.

1. Introduction

1.1. *Motivation: scalability and expressiveness*

An important part of computer science is devoted to designing a context that is conducive to programming the computer and obtaining results efficiently. This covers many areas of research, organized in layers, including hardware, architecture, machine language, and high-level language. The lowest layer deals with the physical properties of physical entities to provide a preliminary set of manageable components and rules, and the highest level attempts to provide an abstract representation of the former so that the programmer can describe the desired task in an expressive way, while offering the best feasible performance.

The scalability problem. At the lowest physical level, technology already produces chips with billions of transistors. As a result, the number of processing elements (PEs) is steadily increasing and research in unconventional computing currently focusses on building hardware on a scale that outreaches today's technology. Given such magnitudes, scalability becomes imperative at the high level: as the number of PEs increases, does performance increase accordingly? This paper advocates that the conventional ways of designing parallel hardware architecture are not appropriate for scaling to arbitrary large size because too many of the physical level properties are lost in ab-

straction. For example, many classical features of parallel architecture, such as shared memories or all-to-all routers, design out the notion of space to establish a single Uniform Memory Architecture (UMA). In shared memories the actual location of data is not an issue. With all-to-all routers, any two PEs are considered as being close together. But the performance of these features cannot be scaled, which is necessary in larger systems where communication time (to access a single memory or communicate with another PE) increases with size. The time required for a signal to travel the length of the wire is not taken into account [1]. The spatial computing framework identifies space as the key physical property and proposes to organize hardware resources as a spatially extended homogeneous computing medium in order to take space into account and achieve greater scalability. Computation and data must be distributed in 2D or 3D space and the particular spatial arrangement must be closely articulated with performance. Section 1.2 outlines state-of-the-art spatial computing as a hierarchy of horizontal layers from hardware to software.

The expressiveness problem. At the highest level, the computing medium must be programmed, i.e. the behavior of each individual processing element (PE) in space must be described. This is a difficult task in comparison to programming sequential machines: there is no centralized control, no overall coherent memory image of the machine configuration, and no global clocking. Because of locality in space, each PE communicates only with its nearby neighbors. Most programs running on spatial computers implement a single purely spatial algorithm where input and output data are located in space and data computation can also take place naturally in space. This is clearly insufficiently expressive when aiming to use spatial computing to solve complex tasks involving different algorithms and data structures. The task is even more difficult when performance is important.

The blob machine concept proposes to solve the programming problem by using a vertical approach to spatial computing, i.e. by proposing two levels that gradually abstract space while never completely ignoring it. Programming is carried out on an intermediate virtual machine called the blob machine, whose primitives are based on physics simulation and are sufficiently simple to be implemented on an arbitrary computing medium. On one hand this allows the user to program traditional parallel algorithms without worrying about the exact spatial location of data and computation. On the other, if the programmed task graph is simple enough, as is the case for a planar graph, then the runtime system can efficiently map it in two-dimensioanl space. Section1.3 informally introduces the blob approach and its main features of interest. The self-mapping property, a significant facet of the blob model, is described in a separate sub-section.

The rest of this article is organized in two sections. Section 2 gives a formal definition of a simplified blob machine, i.e, the binary blob machine and the state of the art of its implementation. The model semantics are described in detail and semantic confluence is demonstrated. An explanation follows describing implementation of the blob machine on a computing medium and defining a complexity model, i.e., "dDcomplexity", used to measure performance in the examples of blob execution presented in Section 3. Those examples have been chosen to cover a wide range of parallel algorithms. The purpose is to establish the feasibility of programming and assess the resulting efficiency. Optimal time and space complexity can be achieved, as long as implementation meets dDcomplexity requirements.

1.2. *Background on spatial computing*

Spatial computing is an umbrella term that groups together different approaches, all based on the observation that future computing platforms — whether VLSI, bio, or nano — will consist of a vast number of Processing Elements (PEs) homogeneously embedded in 2D or 3D space, where the magnitude involved obliges the programmer to incorporate the *locality constraint*, where each PE has a specific location in space, and communication time is a function of Euclidian distance in that space. For example, in the (classic) VLSI complexity model [1], this relationship is linear. Communication costs have always been a major issue in parallel computing. But scaling up to an arbitrary large space is rarely considered as an option, where communication must be optimized by taking into account physical distance. For example, a black-box router can be implemented efficiently in all-to-all communication, but it abstracts away spatial location. Architectures embedded in space, referred to here as *"computing media"*, include not only *regular* classic models, such as cellular automata, systolic arrays and FPGAs [2], but also *irregular* models where the constraints of lattice tiling of space and synchronism in time are relaxed, as exemplified in the amorphous computing model [2]. Spatial computing was the subject of a recent workshop [3]. A complexity model of computing media called "spatial machines" is presented in [4].

Spatial computing calls for a departure from computing in time, which uses a conventional centralized programming approach with a step-by-step modification of a given overall state. Intuitively, to exploit space, computation must be deployed in space by dynamically constructing spatial entities such as circuits. Spatial computing also calls for new architectural designs. The hierarchy of different long-distance connections used in FPGAs is an example of a scalable building block that reduces total network diameter.

[1] Consider the example of a router where spatial location has been abstracted away and the router diameter is the measure of router performance. If the communication time between any pair of PEs does not depend on the communicating PEs, it necessarily depends on this diameter, which represents the worst case.

[2] Field-Programmable Gate Array. An FPGA architecture can be reconfigured on the fly and therefore be adapted to the dynamic features of the program it executes.

Spatial computing is a broad subject covering a large part of unconventional computing. The following classification proposes to include all levels from hardware to software.

(i) Hardware: the development of new technologies that enable spatial computing. In this context, *space is a factor of scalability*. Nanotechnologies include nanotubes [5] [6], DNA computing [7], and chemical reactions [8]. In Goldstein's programmable matter concept [9], processing elements themselves can move.

(ii) Architecture: a structure designed on existing spatial computing platforms, such as FPGAs, in such a way that when the number of hardware resources increases, the same given program can use the added space and thereby improve performance. In this context, *space is a factor of performance*. Dehon [10] proposes a framework called SCORE that deploys pipelined circuits in space during runtime and can adapt to various hardware dimensions by trading time for space. The "poetic" project at EPFL [11] has developed a chip specialized in bio-inspired algorithms including evolution, development, and learning.

(iii) Algorithm: the development of *spatial primitives* that use space and compute information about space. In this case, *space is a factor of functionality*. For example, the MIT amorphous computing group [2] shows how to compute a set of coordinates for each PE of an amorphous medium and Eric Rauch [12] simulates wave propagation, which can support communication on amorphous computers. Using reaction diffusion computers, Adamatzky [13] computes the Voronoï tessellation, which partitions space and also establishes a network using Delaunay triangulation.

(iv) Spatial language: in this context the notion of space is explicitly used as a metaphor of programming and semantics, where *space is a factor of expressiveness*. Giavitto and Michel [14] [15] use the data structure itself as the computation space. Their MGS language shows that reasoning in terms of space leads to compact programs if the task has a spatial formulation. Gamma formalism [16] uses the parallelism inherent in chemical reactions while avoiding any artificial constraints caused by sequential execution; this also leads to very concise programs. The idea of using encapsulated membranes has stimulated two projects based on specific languages: Paun [17] studies the formal language of a model called P-system while Cardelli [18] programs algebraic systems of membranes focused on simulating actual biological cells.

Vertical approaches to spatial computing.

Ideally, the overall goal of spatial computing is to encompass these four levels in a complete vertical model that combines both language and machine: a program in an *expressive* high-level language based on a *functional* library of *spatial primitives*, run on an appropriate *architecture* designed for scalable hardware. Several research projects already address this great challenge: Nagpal [19] proposes a language based on primitives for folding a sheet of paper as in Origami; Coore [20] has developed the Growing Point language, based on primitives which manipulate particles that produce gradients and move along them. Both Nagpal and Coore use amorphous computers and have developed patterns or circuits that are above the amorphous medium. Once the structure is laid out, however, it cannot evolve any more. This limits programming expressiveness. Toffoli [21] proposes a programming framework also called "programmable matter" (like Goldstein), that focuses on efficient compilation for cellular automata with block rules. The language level, however, is not very high. It covers mainly simulations in physics with problems such as noise filtering and polymer simulation. Dehon [10] is also working on both a language and an architecture with a focus on performance that specifically targets stream processing.

In a vertical framework both performance and expressiveness requirements must be met, but are very difficult to achieve simultaneously. In practice, research on spatial languages does not seek efficient use of the computing medium, and performance-oriented research contents itself to use languages whose expressiveness is limited to a specific niche of applications, as long as significant improvements in speed are obtained.

1.3. *The Blob approach*

The blob machine is a vertical framework for spatial computing that aims to achieve both expressiveness and performance by using two types of simple building blocks to manage space: **blobs** and **channels**. With regards to performance, blobs and channels are like physical 2D or 3D objects, such as membranes and filaments, and can thus be parallelized by using discretized physical laws on an arbitrarily large 2D/3D computing medium. This implementation requires a library of spatial algorithms (Level (iii) in the previous classification). In terms of expressiveness, blobs define a virtual machine above the computing medium which is much easier to program (Level (iv)). Blobs and channels can be developed dynamically and therefore are naturally capable of handling dynamic data structures. In general, blobs represent the concept of compartmentalization and therefore allow the programmer to ignore the specific spatial arrangement of the hardware. Blobs alone can create only pure hierarchical structures, where communication occurs between a blob and its sub-blobs. Channels allow communication between arbitrary blobs by providing a dedicated point-to-point communication pathway through the computing medium between a pair of blobs.

Blobs compartmentalize the medium for non-uniform processing. Blobs act functionally like membranes that divide the computing medium into different connected regions. This type of compartmentalization is not required in the case of pure *spatial primitives*, such as providing a gradient to measure the distance to a given point, where the problem can be solved by having all participating PEs ex-

ecute the same simple local rule. But whenever different types of tasks must be run in parallel, by compartmentalizing the computing medium into disjoint connected regions, different parts of the medium can run different programs more efficiently. In the present example, each connected region has its own execution thread and only needs to store and execute the program it is running. The example only considers a simplified case where the program is a finite-state automaton (FSA) that controls the region and can be stored on a single PE. This master FSA can broadcast commands to the other PEs within the region. The PEs are either empty or are holding an elementary data item. The commands tells them which operation to apply to the data. They can be thought of as special blobs referred to as atomic blobs, without sub-blobs.

Blobs embody dynamic development for dynamic computation. Programming with dynamic data structures greatly enhances expressiveness. The memory can be colonized with new data structures created on the fly, which can then be deleted. Using a single recursive function can result in the allocation of arbitarily large space on the stack. To achieve similar expressiveness in spatial computing, the language description must be able to develop and install spatial structures on the computing medium and delete them dynamically. Blobs embody this feature of dynamic development. Each FSA that controls a region can output instructions. This results in a distributed coordinated effort between the PEs in the region and the creation of compartments and the corresponding master FSA. An FSA that outputs instructions is usually called a Mealy machine, but the term FSA has been kept here for simplicity.

A blob machine is a type of virtual machine called a self-developing machine. All the FSAs together with the communication pathways involved in the topology of encapsulated membranes and channels define a graph of FSAs. An entire region, or "blob", is abstracted by a vertex of this graph. FSA instructions add or delete compartments, add or delete vertices and edges, thereby expanding or reducing the FSA graph. In [22] the authors defined a general framework referred to as the "self-developing automata graph" or more simply, a Self-Developing Machine (SDM), based on FSA graphs with output instructions that can add or delete vertices or edges. Certain specific edges, called *"ports"*, are used for parallel input and output and remain fixed during development. The initial configuration is a single vertex called the ancestor, connected to the port edges p, whose FSA is the SDM program. During execution, the graph expands or contracts according to the program and the instructions ordered by the automata. All the automata execute the same FSA but with distinct states stored locally. A blob machine is a particular example of an SDM, but there are others, such as those described in [23], introduced to model self-reproduction.

Self-developing graphs generalize task graphs. The parallel computing community uses task graphs as a practical generic intermediate format to expose parallelism: vertices contain tasks to execute, and edges are used to communicate data between tasks. For certain problems such as "quicksort", however, the number of tasks and the graph structure between tasks is dynamic, i.e. it depends on the data, therefore it cannot be drawn statically for arbitrary data. Quicksort sorts a list of numbers by randomly selecting a pivot in the list, comparing all the items in the list with the pivot, and recursively sorting the two lists of items that are smaller (or greater) than the pivot. Self-developing automata graphs can also be used to handle dynamic cases, thereby allowing self-development to expose greater parallelism. Figure 1 illustrates an example of how the computation used to quicksort four values $n_1, \ldots, n_4$ are mapped on a dynamic graph having up to eight vertices (not counting the top "master vertex"). Each instance $n_1, \ldots, n_4$ is stored on a distinct automaton $a(n_1), \ldots, a(n_4)$, and all the comparisons between a given n_i and the various pivots encountered during execution are mapped to the same automaton $a(n_i)$. This is accomplished by passing the pivots along a dynamically evolving intermediate graph structure consisting of one, then two, then four vertices. The intermediate vertices pass the pivot and store a temporary rank.

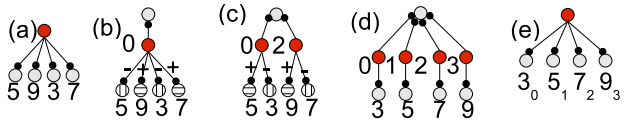


Fig. 1. These snapshots illustrate quicksorting of four integers: 5,9,3,7. The program is given and explained on page 14

.

1.4. *Self-development + hardware freedom + physical forces = self-mapping*

The parallel semantics of SDMs do more than just expose parallelism: communication is always local, i.e., an automaton only communicates with its direct neighbors. Note that this network is virtual, which does not imply that two communicating automata can always be placed on neighboring PEs [3]. What it does imply is that there is no shared memory, or even a global name space. Who communicates with whom is thus clearly represented at any time by the network itself. This allows the runtime system to automatically map the network. The architectures in question, which are computing media, are embedded in a 2D or 3D space. Space is partitioned and each PE is responsible for a portion of it. Each SDM vertex has coordinates that determine which PE is responsible for hosting and updating the SDM. To dynamically map the SDM, the runtime system must carry out two operations:

– First, vertices and edges must be allowed to move freely between neighboring PEs without interference from any

[3] Consider, for example, a 2D grid of PEs. If an automaton is connected to n other automata and each PE stores a single automaton, it takes a sub-grid of at least n PEs to store all the neighbors and the number of hops to cross in order to communicate will be the sub-grid diameter $n^{1/2}$.

underlying computation underway. This is referred to as a "hardware-free" distributed representation of the self-developing graph.
– Second, the runtime system must determine in which direction the vertices are to be moved. This is accomplished by simulating physical forces of attraction between adjacent vertices so as to optimize communication latency, and forces of repulsion between nearby vertices so as to homogenize density and thus optimize load balancing.

In the initial situation, development starts with a single ancestor vertex balanced with the fixed vertices used as ports. Whenever self-development occurs, vertices or edges are added or deleted, thereby upsetting the balance. Each vertex locally computes the force applied to it from its neighbors, and moves according to these forces, possibly migrating to a neighboring PE if its coordinates are no longer in the area managed by its current owner PE. After a few iterations, the situation stabilizes again and computation can carry on.

Comparable techniques based on force simulation have already been developed. 1) The placement and routing problem on VLSI systems [24] uses a technique called force-directed placement where, starting from random initial states, wires linking gates act as springs to move the gates over the VLSI surface. In the present example, the combination with step-by-step development intuitively reduces the plague of local convergence. The adjustment required at each step is hopefully simple enough to allow vertices to be directly attracted to their new optimal position. There are fewer chances to be trapped by local sub-optimal basins of attraction. Computer simulations to support this claim have not yet been developed. It should be interpreted in light of the development examples presented in this article which use planar graphs, simple enough to be convincing. 2) In the MaRS dataflow machine [25], task density has been used for load balancing, which is similar to a repulsive force that distributes density evenly. But this example uses an all-to-all network, resulting in a non-scalable single space where each pair of PEs is at the same distance.

Self-mapping of blob machines. Self-mapping applies to blob machines, which represent a particular type of SDM. But since blob machines use encapsulated membranes to represent adjacency, the different concepts must be re-interpreted, in particular hardware-free compartments which can be adjusted at runtime. Let's assume there are two tasks to be run and the medium is to be divided in two, thus creating two compartments, one for each task. If the tasks perform any dynamic allocation and the compartments subdivide further, it is impossible to predict at compile time the amount of resources that should be assigned to each task, and therefore, how big the compartment should be and where the boundary should be placed between the compartments. If the compartment boundaries can move dynamically once they have been initiated, then the system can adjust the location of the separating membranes and likewise adjust the amount of hardware resources allocated to each task. The system performs a kind of runtime load balancing. Since blobs hold computation as well as data, moving blobs balances not only the CPU load, but also memory occupation. The system also balances communication load by moving channels to obtain a uniform distribution of channel density.

2. The Blob Machine

The virtual blob machine is a particular type of self-developing machine (SDM) whose instruction set is designed to satisfy the opposing requirements of efficiency and expressiveness (see page 3). The instructions are simple enough to run on arbitrary computing media and expressive enough to program non-trivial applications.

Section 2.1 explains how blob machines self-develop and illustrates this with an example. To avoid the complexity of handling integers in the example, only binary digits are considered in a simplified device called the Binary Blob Machine, which demonstrates the essential features while simplifying the presentation. The exact semantics of the instructions are given and confluent behavior is established. Section 2.2 presents the state of the art for this implementation.

2.1. *The formal binary model*

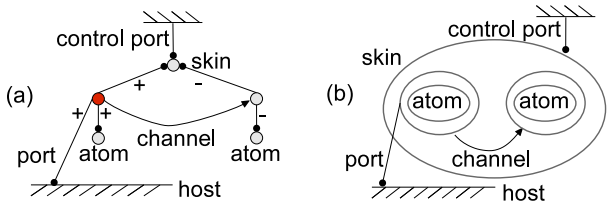


Fig. 2. Two representations of a blob graph: (a) a network representation where arrows with a round (or sharp) head represent a vertical edge (or channel) and their dynamic orientation, while the static orientation of vertical edges is upward; (b) a topological representation that omits vertical edges and automata.

2.1.1. *Overview of the virtual blob machine*

Blob graph: A blob machine is self-developing and is represented by a graph — called *"blob graph"* — of nodes executing a Finite State Automaton (FSA). Actions output by the automaton are the blob management instructions. The FSA is referred to as a Self-Developing Automaton (SDA). If the instruction returns a value (i.e. instruction **setp**), then this value is input to the SDA. "Blob" is just another name for the nodes of the SDA. Throughout blob graph development the tree representing blob hierarchy is always embedded in the graph and is therefore a spanning tree — a connected tree that includes all the nodes in the graph. A natural way to represent this hierarchy is to draw nodes as if they were blob-like membranes, as shown in Figure 2 (b). The blob at the root of the tree contains all the

other blobs and is called the *"skin"*. The empty blobs forming the leaves are called *"atoms"*. An *inner blob* (or *outer blob*) of a blob is one that is immediately encapsulated (or encapsulating).

Topological and graph representations of blobs are like Dr. Jekyll and Mr. Hyde: when looking at one of them, one tends to forget the other, which is hidden. For example, the topological representation (right-hand side of Figure 2) does not show that an automaton is associated with each blob. The network representation is more conveniently used for programming. The topological representation is more "physical", and reflects the actual implementation on a computing medium (see Figure 7). The topological representation recalls the membrane systems, called P-systems, developed by Paun [17]. The difference is that each blob is controlled by a single FSA, whereas in Paun's system, each membrane hosts an unlimited number of molecules. The program is a finite set of rules modeling chemical interactions between molecules. If atomic blobs are used to represent molecules, P-systems can be thought of as a particular "chemical flavored" way to program blob systems capable of exploiting the parallelism inherent in chemical reactions, which is not within the scope of this paper.

There are three kinds of edge in the blob graph: *vertical edges*, *channels*, and *ports*. First, vertical edges encode blob hierarchy. They are oriented so that a blob can distinguish between its unique outer blob and its inner blobs. Vertical edges can represent tree-structured blob graphs. This already makes it possible to program non-trivial algorithms such as sorting algorithms. Certain algorithms, however, require non-hierarchical graphs, such as 2D grids. The second type of edge, the *channel*, is then used to connect arbitrary blobs and is represented as a filament on the computing medium (see Figure 2 (b) and Figure 7). Last, the blobs may need to exchange input/output data with an external *host*. The host is modeled as a distinct node whose behavior is controlled externally and the *port* edges establish a connection between the host and a blob, which can receive inputs or send outputs to the host. The number of ports is fixed during execution and determines the amount of parallelism available for input and output. Like channels, port edges must be represented as filaments on the computing medium.

Blobs are controlled by the same automaton, but with distinct states stored locally on the node. Each blob automaton can run asynchronously in parallel. But at any given phase in time, a blob can run its automaton only if it satisfies two readiness predicates: the parallel readiness predicate and the confluence readiness predicate.

The parallel readiness predicate establishes mutual exclusion between adjacent blobs that share a connecting edge and are about to modify it. This is accomplished using dynamic orientation of the edges, which is different from the up/down static orientation defined for all edges, not just vertical edges. Dynamic means that an instruction called a **flip** can change the orientation. Dynamic orientation defines which of the two blob ends owns the edge, thereby becoming the source blob of the edge. The parallel readiness predicate states that a blob can modify only the edges that it owns. If the next instruction to be performed needs to modify an edge that is not owned by the blob, then the blob is not ready. A precise definition of "modify" will be given later. If it is not ready, the blob must wait for the neighbor that owns the edge to flip it back or delete it. This is the first condition.

The second condition — the confluence readiness predicate — ensures confluence. This notion is more technical and will be explained later in the paper.

Labeling of edges: vertical edges carry a polarity, noted $+$ or $-$, used for communication and differentiation:

- For communication purposes, the polarity bit behaves as a register that is shared between a blob and its elements. The polarity of an edge is set from one end using the **setp** instruction, and is tested from the other using the **testp** instruction. Setting the polarity downwards with the **setp down** $\pm$ instruction is analogous to a broadcast, since there can be an arbitrary number of inner blobs.
- Whenever a blob creates a child blob, they are initially set to the same state specified by the SDA transition function. However, the upper edge of the created blob (or creating blob) is negative (or positive) (see Figure 4 or upper part of Figure 3 after the wrap instruction is performed). Therefore, by testing the polarity of the upper edges, the creating and created blobs can execute their next transition to distinct states.

Vertical edges also carry a static orientation: *up* (or *down*), which refers to the outer (or inner) blob. Channel edges do not carry static orientation or polarization. Port edges are treated like vertical edges: communicating with ports or moving ports is like communicating with inner blobs or moving blobs, respectively. In total, as shown in Figure 4 upper left, there are five labels used to address edges: *up*$+$, *up*$-$, *down*$+$, *down*$-$, *chan*.

The *initial configuration* consists of a single ready blob called the *ancestor blob*, connected to the port edges by a $+$ polarity, owned by the host (upper left of Figure 3). One of the ports, called the control port, is statically oriented upwards to the host so that the ancestor, and later the skin, also has a unique Up vertical edge, like any blob. All the other edges are linked downwards to the host. The ancestor state is the SDA initial state q_0.

Example of SDA. Figure 3 shows how to implement a priority queue using an SDA. The purpose is to present a simple example in order to show the details and basic steps of development. The only self-developing instruction used is the **wrap** instruction that develops the blob graph vertically in a degenerated tree representing a stack. The SDA uses the "push" operation in an ordered stack. Values flow from top down to the appropriate location so that the stack is ordered. In general this stack processes integer values by binary-encoding them as a sequence of $+$ or $-$ polarities. For the sake of simplicity, the example is shown for $0(-)$ and $1(+)$ values. Each blob evolves according to the automaton illustrated in Figure 3. The upper part of the

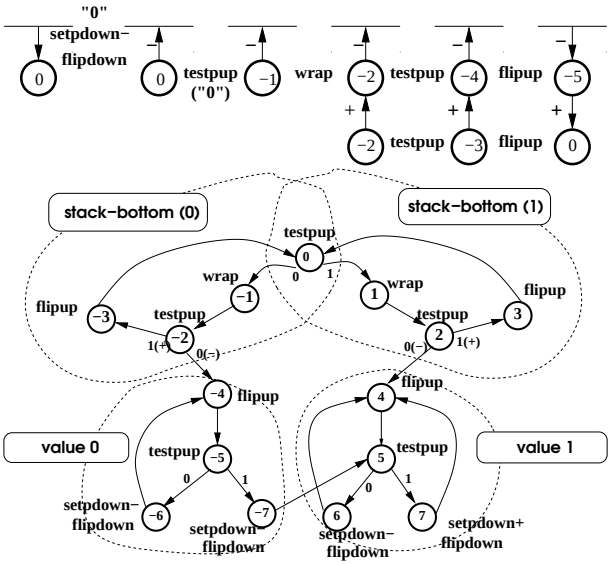


Fig. 3. Example of a basic self-developing automaton (SDA) implementing a priority queue. To simplify, values are only 0 (-) or 1 (+). The automaton has 3 parts: the stack bottom and the stack elements set to 0 and 1.

figure describes the first development steps starting from the ancestor blob b connected to the host via the control port. The initial state is 0, the initial stack bottom state. The host sends the value "0" by setting the polarity of the control port to − and gives port ownership to b using the **flip down** instruction. Then b can receive this value using **testp up** to test polarity followed by a **wrap**. The inner blob becomes the new stack bottom, while the outer blob is a dynamically created stack element set to "0", corresponding to states -4 to -7. The symmetric states 4 to 7 correspond to a stack element set to "1". To push a "1", the host sets polarity to positive instead of negative. When a stack element set to v receives a pushed value v', it propagates $min(v, v')$ downward and stores $max(v, v')$. Values are sorted as they travel down the stack elements, and a new stack element is inserted when the value reaches the stack bottom. The **flip up** and **flip down** instructions synchronize the receivers with the senders. They change the direction of the up/down edges. This results in a pipelined parallelism, since a new value can be pushed every five clock cycles.

2.1.2. *Semantics of the eight binary blob instructions*

The binary blob machine is defined by the eight instructions listed in Table 1, which develop a blob system starting from the initial configuration. Figure 4 gives a complete description of the semantics in the eight instructions in a graphic representation. Each instruction is formally a graph-rewriting rule. This rule is local and does not require knowledge of the entire blob graph. Only adjacent edges of each blob are taken into account to trigger rewriting.

Triggering predicate. The possible contexts on the left-hand side are multisets of oriented labels carried by edges connected to the blob. The triggering predicate applies certain restrictions on the multiplicity of certain labels. The conjunction of both *readiness predicates* imposes the absence of certain labels, plus an optional instruction-specific predicate used in the **merge** and **testp** instructions: 1) For the **merge down** instruction, the predicate $\mid down+ \mid = 1$ states that merging takes place only if there is exactly one positive inner blob. 2) The **testp down** instruction can be seen as two rewrite rules with the predicates $\mid down+ \mid = 0$ and $\mid down+ \mid > 0$. The two rules are conveniently reduced into one by making the SDA input a parameter computed from the context. The $down+$ label is said to be tested by **testp down** and **merge down**. To prevent deadlock, if an instruction is ready but the instruction-specific predicate is not true, then the instruction is considered to be **nop**, i.e. the neutral instruction that has no effect. This only occurs for the **merge down** instruction.

Static			
Op Code	**Semantics**	**modified**	**tested**
Flip dir	Changes orientation of edges dir	dir±	
Setp dir q	Polarizes edge dir	dir $\overline{q}$	
Testp dir	Tests for presence of + edges		dir+
Move	Exchanges parent blob and channel	up±,chan	
Self-developing			
Op code	**Semantics**	**modified**	**tested**
Wrap	Encapsulates a new blob	up±	
Div	Divides blob, duplicates channel	up±	
New chan	Creates a brother, linked by a channel	up±	
Mrg up	Merges blob, deletes channel	up±, chan	
Mrg down	Same as above	up±, chan, down+	down+

Table 1
Blob instruction set: four static instructions to modify edge labels, four developing instructions to add or delete vertices and edges. Operand "dir" takes values $\{up, down\}$; and operand "q" represents + or - polarity.

Rewriting process. The left member of the rewriting rule that models an instruction represents the node n being rewritten in the center of a gray disk, with edges leaving n. There is at most one edge for each of the five possible labels, all represented in the upper left figure. An instruction does not necessarily use all five labels. The right member contains a blob graph with a distinctive node called the *root*, identified as n, which has two implications: first, rewriting n only requires the creation of other nodes if the right member contains nodes other than the root; second, any edges of n with labels that are not contained in the left member implicitly remain connected to the root. An edge labeled l in the left context points to a specific location on the perimeter of the disk called the l-location that represents all the neighbors of b, connected to an edge labeled l called the l-neighbor. The l-location is used in the right member to specify how to establish a connection to the l-neighbor, and the blob graph of the right member. If the right member contains an edge labeled l_1 between an l_0-location and node n (or another l_2-location), then an edge labeled l_1 must be created to connect each l_0-neighbor to n (or to each l_2-neighbor). Rewriting could potentially add an arbitrary number of connections. But a closer look shows that an instruction never adds more than two edges.

1) Only the outgoing channel and the upper edge are duplicated, (by **divide** and **new chan**). But the number of upper edges (or outgoing channels) is always one (or less than one). This is true for the initial blob graph and remains true for each instruction. 2) When direct connections between l-neighbors are established (by **mrg**), one of the two is *up* or *down*+ with multiplicity restricted to 1.

Modified labels, used labels. Figure 4 illustrates the precise meaning of the expression "modified label" used to define the parallel readiness condition: a label l is modified if the l-neighbor context changes after the instruction has been executed. This includes any modification to a label associated with an edge, removing an edge, or a adding new one. The labels represented in the left member are called *used labels* and may include labels that have not been modified. For example, with the **divide** instruction, the polarity of inner blobs is used to determine which node they will be connected to. The *down*+ and *down*− edges are used, but not modified. By convention tested edges are considered to be used, i.e. they must appear in the left context. Table 1 indicates which edges have been modified and tested by each instruction.

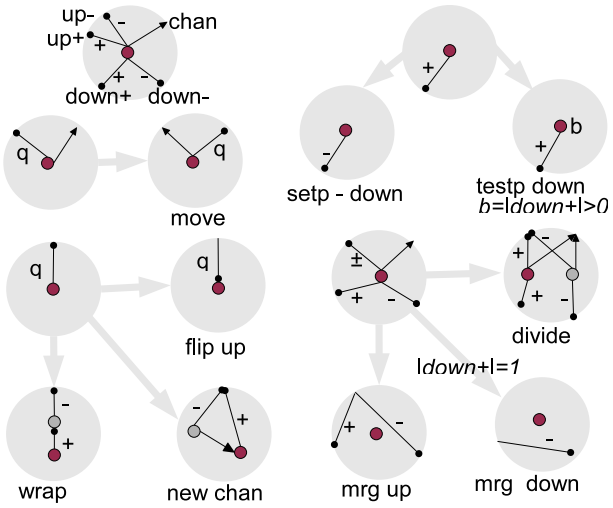


Fig. 4. Complete semantics of the eight binary blob instructions. Notation is the same as in Fig. 2. The root of the right member is represented using a darker gray. The **setp**, **testp**, and **flip** instructions are symmetric with respect to *up* and *down*, so only one direction is represented. **testp** returns an input b to the SDA, which is true if there is at least one positive inner blob.

The static instructions **setp**, **testp** and **flip** have already been described in sufficient detail. Figure 5 shows a topological representation illustrating instructions that create, delete or move blobs, where the semantics are more straightforward.

Creating a blob. **wrap** encapsulates a new blob, which receives all the inner blobs, and develops the blob hierarchy vertically. The **divide** instruction produces a brother blob, moves any negative elements to this blob, and develops the blob hierarchy horizontally. Last, **new_chan** also creates a brother blob that does not contain any elements and is linked to the creating blob via a channel edge. An additional **wrap-** instruction is then used to move the negative elements into the newly created encapsulated blob. It can be thought of as a macro-instruction that combines the **wrap**, **divide** and **mrg up** instructions. In **divide** and **new_chan** the upper edge is duplicated. But since the port edge cannot be duplicated, the skin, which always has an upper edge that is the control port edge, cannot execute **divide** or **new_chan**. In turn, this implies that the skin maintains its property of enclosing all the other blobs.

Deleting a blob. Blobs are deleted when all five labels are used and the root is isolated, any root disconnected from the blob graph being deleted automatically. The **mrg up** instruction merges the blob with its outer blob, which receives any inner blobs, and deletes any channel. This amounts to simply deleting the membrane and the channel. The **mrg down** instruction is not symmetrical to **mrg up**. The executing blob must have a single positive element whose membrane is noted $m+$. It moves all the other negative inner blobs inside $m+$ and then deletes its membrane m and channel. To avoid moving blobs, $m+$ is deleted instead of m, as shown in Figure 5. The automaton controlling m must be replaced by the automaton that was controlling $m+$, which is not shown in the figure. Port edges cannot be deleted, therefore it is not possible to merge through a port edge. As a result, the skin cannot merge up and is preserved.

Communicating blobs through channels. Note that all self-developing instructions preserve the following invariant, referred to as the channel invariant. A given blob b_0 owns at most one channel which, if it exists, points to another blob, b_1, that is not contained in b_0. Because of this invariant, blob b_0 may choose to move in order to become an element of b_1 using the **move** instruction. In the graphic representation, **move** simply exchanges the labels of the single owned channel, if it exists, with the single outer blob. The topological representation in Figure 5 illustrates why this is called a movement, since changing the outer blob actually implies moving into the membrane of the new outer blob. Flipping a channel is not allowed, as this would break the channel invariant rule. This is why communication through channels consists exclusively of moving blobs.

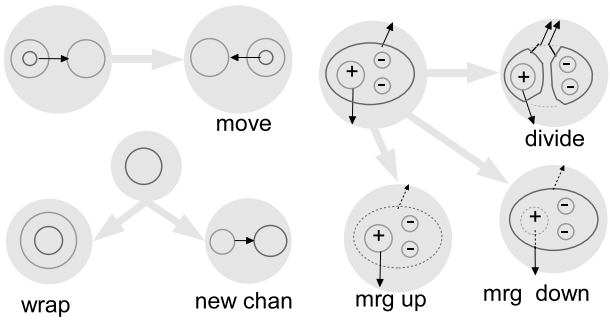


Fig. 5. The developing instructions shown in Figure 4 along with the move instruction in the — more intuitive — topological representation.

2.1.3. *Confluent Asynchronous Parallel execution*

Asynchronism eliminates the need for a global clock, which is important for scalability. In asynchronous execution, only a (randomly selected) subset of all the ready nodes execute their instructions simultaneously. One of the aims of this work is to define a *confluence* property to ensure that the order in which instructions are executed has no effect, the system consistently converging to the same configuration. For example, a data-flow network runs instructions asynchronously, but is confluent. Confluence is analyzed in a systematic way [4] that may be used for other self-developing instruction sets. The first consideration concerns the conditions applied to orientation and edge labels for each instruction, as well as the effect of each instruction on these edges. This is summarized in Figure 6. Let's consider a node n_0 and an edge e_0 connecting n_0 with another node n_1, such that e_0 is not owned by n_0. The node n_1 can thus modify e_0, and create a new edge e_1 that connects it to n_0. Edge e_0 is said to "produce" e_1. Figure 6 represents the production relationship, which is defined from the point of view of n_0. It uses oriented labels, formed by a label and an orientation coded as 0 for incoming, and 1 for outgoing. For each oriented label (l_0, d_0), the relationship indicates the possible oriented labels (l_1, d_1) of new edges that b_0 can get from its neighbor at any time. The confluence readiness predicate states that a blob is ready to execute an instruction i only if its context does not contain oriented labels that can produce labels used by i. To be ready, a node must simply check that there are no labels of this type, which is a local criterion. The confluence predicate implies that all used labels are outgoing, because an outgoing label produces itself. This generalizes the fact that modified labels are outgoing, and is indeed true for the blob instructions.

It can also be shown that the confluence predicate implies commutativity, which is stronger than confluence: commutativity is true if the result of executing any two nodes n_0 and n_1 that are ready simultaneously does not depend on the order in which they are executed. Whether or not n_0 and n_1 are neighbors is unimportant. As before, let's consider an edge e_0 connecting n_0 to another node n_1, such that e_0 is owned by n_1. Let i_0 and i_1 be the instructions executed by n_0 and n_1. Executing i_1 first brings new edges $e_1 \dots e_k$ to n_0. These edges are not used by i_0, otherwise it would contradict the confluence readiness predicate for i_0. Being not used implies that the labels of $e_1 \dots e_k$: (1) are not tested by i_0; (2) do not appear in the left member of i_0. Condition (1) implies that the instruction executed by n_0 after i_1 has been executed is still i_0. Condition (2) implies that when i_0 is executed, $e_1 \dots e_k$ remain connected to n_0, which also occurs if i_0 is executed before i_1, because e_0 is not owned by n_0 and therefore e_0 remains connected to n_0 after i_0 has been executed.

Figure 6 shows two extra readiness confluence predicates in the case of blob instructions: 1) Since $(up\pm, 0)$ generates $(up\pm, 1)$ and $(down\pm, 0)$ generates $(down\pm, 1)$, an instruction that uses *up* (or *down*) should own all the *up* (or *down*) edges. If the instruction is a modification, the resulting condition was already specified in the parallel readiness condition. If the instruction is a test, i.e for the **testp** *dir* instruction, confluence readiness states that **testp** *dir* must own *dir* edges. This ensures that the value will not be changed by the neighbor while it is being read. The automaton in Figure 3 illustrates a direct consequence of this principle: to send bits, the edges must be flipped back and forth by the sender (state $q_{\pm 6}$,$q_{\pm 7}$) and receiver (state $q_{\pm 3}$, $q_{\pm 4}$) for each bit exchanged. 2) Since $(up\pm, 0)$ and $(chan, 0)$ generate $(down\pm, 1)$, an instruction that uses *down* should not have incoming channels, nor any incoming upper edges.

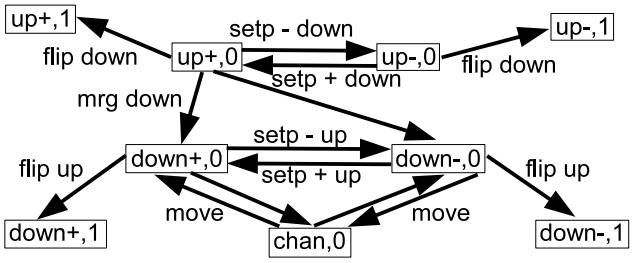


Fig. 6. Production relationship between edge labels. A label l_0 can produce another label l_1 if there is a path leading from l_0 to l_1. The arrows between nodes are labeled by the instruction responsible for production. Certain instructions have not been shown because they do not add new edges to this graph.

2.2. *Implementation of the runtime system*

For the moment this discussion has considered essentially only 2D or 3D Cellular Automata (CA) as target computing medium. The specific features of more realistic hardware structures such as long lines in FPGAs, or memory in coarser grain hardware, can be exploited to improve implementation. When considering computation complexity for a problem of an arbitrary size, however, it is the spatial lo-

[4] It is difficult to apprehend confluence in the blob model without actually trying to program something. With practical experience, it is easier to understand exactly what can be programmed using the eight instructions. To illustrate how the model works, let's examine the classical (non-confluent) example of a non-deterministic merge of two input streams to an output stream, and see why it cannot be implemented in a blob network. A blob b_0 can, for instance, receive inputs from two different inner blob sources, b_1 and b_2. b_1 and b_2 can each send one bit to b_0 using the **Setp up** instruction. First of all, b_0 cannot receive anything before both b_1 and b_2 have flipped back the edge to their outer blob. Second, with the **Testp down** instruction, b_0 does not read the two bits, but a logical OR of the bits. To distinctly receive both bits, b_1 and b_2 must agree to send their bit each one in turn, while the other sends a 1 bit. These two points imply bit-level synchronization, which excludes a non-deterministic merge. Data-flow graphs use non-deterministic merges while remaining confluent in the case where either b_1 or b_2 does not receive any input stream at all. It may be preferable to allow the other node to forward its input stream. In blob computing, additional non-confluent instructions could be added to increase expressiveness and facilitate programming of these gates. Alternately, if only the eight instructions are used, one of the two inner blobs can be informed that it will not receive input streams and that it can send a flow of ones of the appropriate length.

cality constraints that dictate asymptotic behavior. At this stage, the aim is to derive this complexity, rather than gain speed by a constant factor by exploiting technologically dependent features. The 2D CA is a simple framework exhibiting these spatial constraints. The term Processing Element (PE) is used here to refer to the automata of the 2D CA to avoid confusion with the SDA automata. Because PEs have a finite small state, it may be assumed that the state of each node in the blob graph is stored on a distinct PE. In the basic blob machine, each PE runs the same automaton, but in a different state. Alternately, a more universal representation for running different automata uses two levels for PE configuration: a fixed automaton implements another parameterized automaton where parameters are input from the host in a preliminary phase, and can include the netlist of an FPGA-like circuit with flip-flop registers and look-up table, or a more conventional microprogrammed controller. Fine-grain implementations such as CAs are also interesting for their own sake: 1) With a finer grain, the blob membrane has a larger diameter, counted in number of PEs, thereby augmenting the parallelism of blob movement across the hardware (corresponding to dynamic code and data migration). Parallelism increases in two ways: in the pipelined (or data parallel) direction, and in the direction parallel to (or perpendicular to) blob movement. 2) The problem of finding the finest possible granularity is an interesting challenge. What is the simplest hardware building block that can be combined in an *arbitrary* number, and provide a computing medium capable of simulating an *arbitrary* blob machine? The problem is the finite memory of this building block, which forces partitioning of the SDA itself into several elementary SDAs of fixed size whose state fits on a PE. This problem is addressed in [26].

2.2.1. *Mapping a blob graph on a 2D CA*

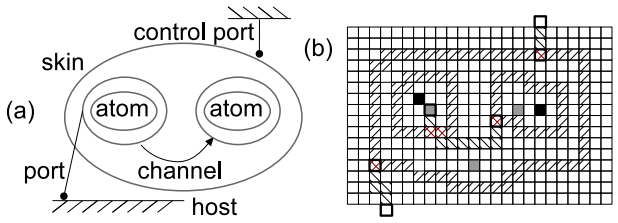


Fig. 7. (a) A blob graph. (b) Fine-grain mapping of this graph. Atomic blobs are black PEs. Gray PEs store the automaton of a non-atomic blob. Membranes are represented by hatched PEs. Channels and port filaments are also hatched in opposite directions. PEs at the end of a filament are shown in bold lines.

The entire blob graph is mapped on a 2D CA by discretizing its topological representation: mapping is simply a "pixelized" version of the topological representation. Consider a given instant t. 1) An atomic blob b is mapped on a single PE $A(b,t)$, storing its state. $A(b,t)$ is called a *particle*. As mentioned previously, in a realistic implementation the PEs can execute distinct parameterized automata. In this case $A(b,t)$ also stores the automaton itself, which must be loaded from the host, and the blobs can execute distinct automata.

2) A non-atomic blob b is represented by a membrane, which is a connected set of PEs, $M(b,t)$ dividing the space into two connected components: the inside and the outside. The inside is called the *hardware medium* of b and noted $H(b,t)$. $H(b,t)$ contains the inner blobs of b and a particle $A(b,t)$ containing b's automaton, that is distinct from other particles representing atoms of b. Let $N(b,t)$ be the number of automata contained in b. This gives $|H(b,t)| > N(b,t)$. 3) A channel c from b_0 to b_1 is represented by a filament, which is a connected set of PEs called $H(c,t)$ adjacent to $A(b_0,t)$ and $H(b_1,t)$. Filaments must be represented on a different layer than automata and membranes in order to cross membranes and move freely over automata and membranes. If the computing medium is 2D, filaments cannot cross each other to represent a non-planar graph. The solution is to provide a third dimension with a minimum thickness of 2.

Consider a blob b connected to n blobs by channels. In 2D all the channels need to connect to the perimeter of $H(b,t)$, which is $\mathcal{O}(n)$. If the blob is round, the minimum compatible bound on $H(b,t)$ is $\mathcal{O}(n^2)$. In contrast, if b is connected to n blobs by vertical edges, a size of $\mathcal{O}(n)$ is sufficient for $H(b,t)$. Saving space in this way is one of the key advantages of membranes, justifying their use. Membranes allow communication along vertical edges, without having to explicitly represent these vertical edges, by using pure flooding within $H(b,t)$. This means that reaching n nested membranes only requires $n^{1/d}$ steps in d dimensions.

Hardware freedom means that changes in the hardware media of blobs, automata, and channels, i.e. the membranes, particles and filaments, must be possible in the course of time. This is why $H(b,t)$, $A(b,t)$ and $H(c,t)$ all depend on time t. These movements must keep the topological properties that provide functionality. As stated above, these properties include, in addition to connectedness, separating inside from outside for membranes, and maintaining adjacency for filaments. In [27] it is shown that separation and adjacency can also be expressed in terms of the connectedness of specific sets of PEs, where connectedness is defined by predicates relevant to membranes or filaments. This work introduces the "*blob rule*", specifying which modifications are allowed and which are not in order to maintain local connectedness and prove that it is sufficient to preserve global connectedness. In other words, modifications leading to movement are accepted if they are allowed by the blob rule. The idea of a connected "spot" that can move explains the origin of the term *"blob"*. The blob rule not only establishes blobs, but also channels. In [27] it is also established that this local rule can be defined on arbitrary architectures, and asynchronously. This opens up the possibility of mapping blob graphs on irregular architectures, embodied in the amorphous scalable architecture model.

2.2.2. *The blob "dDcomplexity" model*

Definition of dDcomplexity. The time required to iterate an SDA node b includes the time it takes to update the automaton state, and the time $T(i, b)$ necessary to execute instruction i. The state is updated locally on a PE, requiring one unit of time. Most blob instructions, however, need to modify the hardware medium at time t, $H(b) = H(b, t)$, which is distributed in space. In general i needs to reach all the PEs in the hardware media under its control. $T(b)$ is the time required for this purpose, with $T(i, b) < T(b)$ and $T(b) = \mathcal{O}(D(H(b)))$, where diameter $D(H(b))$ is defined as the maximum distance between any two PEs of $H(b)$ for the distance induced by $H(b)$, representing the hop count required to go from one PE in $H(b)$ to another while remaining inside $H(b)$. Certain instructions also use channel c and its associated filament at time t, $H(c) = H(c, t)$, in which case $T(b) = \mathcal{O}(D(H(b)) + D(H(c)))$. In this case d is the dimension of the computing medium, $d = 2$ or $d = 3$. The "dDcomplexity" model is a set of three conditions that must be met by the implementation: *(i)* the particle density is homogeneous throughout the skin, equal to α, resulting in $|H(b)| = \alpha^{-1}(N(b))$; *(ii)* $H(b)$ is always close to the disk or sphere: $D(H(b)) = \mathcal{O}(|H(b)|^{1/d})$; *(iii)* the model targets the best possible performance: $T(i) = \mathcal{O}(D(H(b)) + D(H(c)))$. For a blob graph with channels of bounded length, combining *(i)* with *(ii)* and *(iii)* using simple substitution results in *(iv)* $T(i) = \mathcal{O}(N(b)^{1/d})$, which defines dDcomplexity time $T(i)$ of a blob instruction independently of mapping. This makes it possible to calculate the time complexity of SDA execution given only the dimensionality of the computing medium. This principle is applied in Section 3 for a variety of SDAs.

The stalling problem. Consider a second-order blob b containing first-order blobs $b_1, \ldots b_n$, all dividing simultaneously, thus doubling the density within b. Condition *(iii)* takes into account the time required to move particles within b_i in order to divide b_i. It does not consider the time needed to inflate b's membrane and relocate b_i's particles and b_i's membrane inside b in order to re-establish uniform density of α throughout b. This homogenization process involves the entire blob system, because b itself needs to push on neighboring blobs until the skin membrane is reached. Furthermore, if blob b_i needs to repeat duplication several times, the duplication rate will be greater than homogenization speed, and development may stall due to a shortage of empty PEs: a particle that needs to duplicate and has no empty PEs in its neighborhood is stalled. Complexity as defined in *(iv)* does not take into account the waiting time incurred due to stalling. Stalling occurs particularly in the initial stage of development, which starts from a single ancestor blob that creates a multitude of blobs, for example, to store the inputs of the algorithm as they are loaded from the ports, or to establish a circuit.

The allocate instruction. This instruction involves algorithms featuring only one growth phase, followed by a stable phase and a reduction phase. For any blob b created, it is possible to dynamically evaluate the maximal size $N_{max}(b)$ that it will reach during its lifetime. In the cases considered here, a program variable directly indicates how many subblobs will be contained in the created blob. This is a loose condition that usually tests as true and does not imply that the algorithm has a static task graph (such as quicksort).

In this restricted framework, a simple development strategy provides good complexity estimates. When created, a blob that is called on to grow later executes a specific instruction called **allocate** $N_{max}(b)$ that generates a repulsive force emanating from $A(b)$ which inflates the membrane of b so that it becomes large enough to contain the expected future $N_{max}(b)$ particles. The density of b temporarily falls below α so conditions *(i)* and *(iv)* need to be modified by replacing $N(b)$ with the "virtual size" $N_{max}(b)$. This method must be applied first to the ancestor, whose very first executed instruction is **allocate** N, where N is the maximum number of blobs developed during the entire run. This inflates the skin's membrane to its maximum size right from the first step of development. An algorithm systematically using **allocate** becomes *size-preserving*: the size of all blob membranes remains constant after creation. A size-preserving algorithm does not need hardware-free membranes: the initial location of a created membrane can be maintained throughout the blob's existence. The **allocate** instruction is like a pragma in parallel computing: it can enhance parallel performance without modifying the semantics, in an architecture-independent way.

Uniformly dividing development. Since a certain amount of bounded variability in density can be tolerated, blobs whose size is quasi-preserved, i.e. varies between n and $2n$, do not need to allocate space. The following development, referred to as *uniformly dividing development*, is naturally *quasi-size-preserving*, without any use of **allocate**: a first-order blob that initially has n atoms divides iteratively into two blobs that have an equal share of atoms until they contain a single atom. In terms of sets, a set is divided into two subsets iteratively until singletons are obtained. After each division, a different processing operation can be applied to each subset, which is actually a standard method for processing the elements in a set in a *non-uniform* way. This method is used along with uniformly dividing development programming to obtain divide-and-conquer parallelism in Program 2, and to align two arrays coded as sets in Program 3. Uniformly dividing developments are quasi-size-preserving because the outer blobs, where division takes place, are sized between n and $2n$ as is clearly shown in the figure illustrating Program 2. The size of the outer blob grows from n to $2n$ because of the master particles created when dividing the inner blobs. Since the blob size is divided by two at each division step, in dimension d, the diameter $D(H(b))$ is divided by $q = 0.5^{1/d}$. The total time required for uniformly dividing development is $\sum_{1 \leq i \leq n}(D(H(b))/(q^i))$, or $\mathcal{O}(D(H(b)))$. In other words, uniformly dividing development within blob b occurs over time period $\mathcal{O}(T(b))$, i.e. the same time b needs to execute a single instruction. Uniformly dividing development resem-

bles "nested parallelism" used on sequences in the NESL programming language introduced by Blelloch [28], who also proposes a simple framework to evaluate the parallel complexity of the relevant programs.

2.2.3. *State of the art of model implementation*

How can a runtime system that tests the three conditions be implemented on a particular computing medium? Conditions *(i)* and *(ii)* correspond to a set of background rules running continuously through the CA. A set of rules of this type was proposed in [29] consisting of well-known, very simple CA rules for discretizing simple physics, explained in [30]. The CA rule HPP models particles as a gas under pressure. This does not place an exact lower bound on it, but makes the creation of empty spots highly improbable, so that *(i)* is true on an average basis. The membrane is implemented as an elastic bubble on which the gas exerts a pressure. A constant outside pressure exerted on the skin determines the average density α. The CA rule called the "voting" rule (one state bit) ensures *(ii)* by modeling surface tension. Condition *(iii)* must be checked one instruction at a time. Some instructions are implemented naturally: communication is accomplished by flooding the inside membrane or through filaments; **Mrg** is achieved by deleting membranes and filaments, while **Wrap** splits the membrane as if it was a zipper. **Move** for an atomic blob causes a particle to travel through the filament. The difficult instructions are **divide** and certain new instructions, not present in the binary blob machine, that improve communication. Their implementation is presented when blob b execution is first order, i.e. contains only atoms $a_1, \ldots a_n$. The membrane contains $n+1$ particles: the atom's particle and b's master particle $A(b)$. Results in communication can easily be extended to higher-order blobs, whereas division of higher-order blobs needs to move membranes through hardware, which has not yet been solved. The algorithms described in this article only call on division of first-order blobs.

Communication instructions. The following example uses scalar instructions that can call on the 2D topological representation to improve communication performance. First, when sending the instruction upward, **Send** v_i, executed by atom a_i, can communicate scalar value v_i to b instead of just one polarity bit. To receive the n values of $v_1, \ldots v_n$, b must execute an associative and commutative operator such as **sum**, which retrieves $v_1 + \ldots + v_n$. Reduction is performed using a connected spot c, initially equal to $H(b)$, which is allowed to shrink, while maintaining its connectedness using the blob rule described in [27]. **Send** v_i generates a temporary particle p_{v_i} containing v_i. These temporary particles are collected by c, i.e., they stay within c as c shrinks. They will eventually meet on the same PE. When p_{v_i} meets p_{v_j} they merge into $p_{v_i+v_j}$. Because it remains connected, c eventually has only one hardware media PE p_{end}, containing only one temporary particle, with the reduction result $r = v_1 + \ldots + v_n$. Finally, when c goes beyond $A(b)$, c is also allowed to collect $A(b)$ so that $A(b)$ stays within c. $A(b)$ is also hosted by p_{end}. $A(b)$ can test termination and read result r. This shrinking blob has the additional advantage of centering $A(b)$. Shrinking can be applied in parallel, on every other PE of the border of c. Mutual exclusion is required to maintain the connectedness of c. The whole shrinking process, illustrated in Fig. 8, takes an amount of time proportional to the diameter of b, which meets condition *(iii)*. This is also the best time, in general, to reduce PEs on a 2D or 3D grid.

Second, when communicating downward, instruction **Broadcast v** broadcasts a scalar v instead of just one polarity bit. In addition, instead of flooding a signal inside the membrane, **Broadcast v** sends waves separated by constant space intervals. This allows a list of scalar values to be broadcast at a constant throughput and avoids waiting for latency across the entire diameter, which would be implied by flipping back and forth. The receiving instruction is called **Rec**.

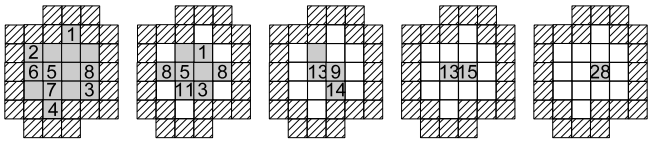


Fig. 8. Four steps in the blob shrinking process using the "sum" reduction operator.

Mathematical proof of division for rectangular blobs with static membrane. Consider a 2D CA and a vertical rectangular membrane. Vertical division requires a mechanism to move the positive particles down and the negative particles up so that they can be separated by a new horizontal wall. This separation can be achieved by stacking positive particles downward (or negative particles upward) into two heaps with a 45-degree slope. Stacking is sufficient if the particle density is less than 1/4. A simple CA stacking rule, shown in Figure 9, has been presented with Tromp [31]. Proof has been given (in 3 pages!) that the stacking time is less than three times the rectangle length, which meets condition *(iii)*. This demonstration does not take into account hardware-free membranes. However, it is sufficient to justify our result on complexity, since the algorithms described only use uniformly dividing development, where hardware-free membranes are not necessary and rectangular membranes are sufficient.

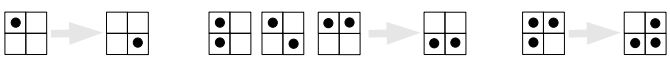


Fig. 9. Tabulation of a 2D CA block rule that stacks particles by moving them downward. It uses one state bit that indicates whether the particle is occupied. The rules not represented are either quiescent, or can be obtained by horizontal symmetry.

Measuring division with round blobs and a hardware-free membrane. In the case where blob b has a hardware-free round membrane, no formal results are available except for implementation results in [29] using gas models for conditions *(i)* and *(ii)*. First $A(b)$ broadcasts a divide signal.

When the other particles receive the divide signal, plus and minus migrate to opposite ends of the blob according to the stacking rule. This automatically creates a retraction in the equatorial zone. During this process, the center-shrinking blob pushes the master particles toward the center of the dividing blob, and collects information on whether or not there are only plus and minus particles on each side. When $A(b)$ touches the two retracting walls, it waits until all the plus particles are on one side and the minus on the other side; then $A(b)$ does the final cut-and-duplicate, producing two blobs (see Figure 10 (d)). Testing covered the division of blobs having up to 1600 atoms. Measurements showed that the latency time for dividing a blob containing n particles is proportional to $n^{1/2}$, which corresponds exactly to dDcomplexity condition *(iv)*.

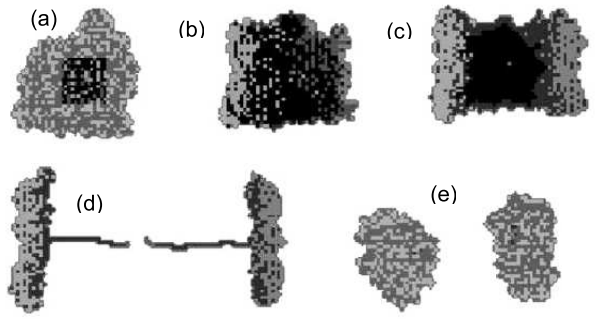


Fig. 10. Snapshot of blob division: (a) start; (b) grouping plus and minus together; (c) wall retraction; (d) final cut; (e) end. The video is available at [32].

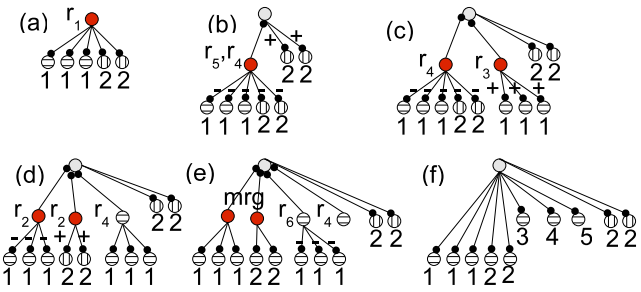
3. Programming efficient blob development

Since there is no order among the blobs inside a given blob, a blob is functionally a multiset of its inner blobs. A multiset, also referred to as a "bag", is a set where elements can be repeated. It is a free monoidal data structure. This property has served to describe parallelism in a concise way. In [33], Tannen defines algebraic high-level parallel operations such as reduction. In Haskell and SETL [34], language constructs allow bags to be defined in comprehension. The NESL programming language introduced by Blelloch [28] uses nested parallelism on sequences that are collections similar to bags. He manipulates sequences to implement divide-and-conquer algorithms in a style very similar to the present authors' technique. The work of Lisper [35] uses sets (data fields) to describe a generic data parallelism in a functional way. In [36] the efficient parallelization of set-based language is also considered. The blob language under development in the present authors' works is also based on bag properties. This paper presents language constructs for describing SDAs at a higher level than just finite state automaton. The description is compact enough to present elaborate blob algorithms with optimal dDcomplexity. It is close enough to blob graphs to provide the programmer with a clear mental model of how it can be compiled to SDA, and can optimize his code to develop small blob graphs. To increase compactness, OCAML [37] syntax is used to enable automatic type inference. The function type is inferred and given in italics, according to common practice. This work is still in progress: the blob language compiler has not yet been implemented and blob language itself has not yet been stabilized.

The finite state constraint. An SDA in a finite state automaton, i.e., a blob, can only host scalar variables, not collections of data such as arrays or lists. Collections of this type must be distributed on different blobs, and must be encoded as bag hierarchies. OCAML arrays and lists are therefore not allowed (although OCAML arrays have a fixed size and could be used in a static framework). Likewise, recursive functions are banned, unless they are terminal, in which case they can be compiled by using a statically bound stack. With parallel languages, it is generally accepted that certain conditions must be met in order to produce a feasible code. The finite state constraint has a simple formulation and can be easily understood by the programmer. He is obliged to use bags extensively. To achieve good performance, the program must generate uniformly dividing development, which corresponds to iterative bag division.

Code size. In a first approach it is assumed that each blob contains the entire program and can therefore access the code to be executed. Since the program is finite, this results in an FSA. By using object-oriented programming, objects can be distributed as well as the code itself so that each blob only carries the code of the method used by the object it represents. But this is beyond the scope of this article.

3.1. *Basic manipulation of bags (SIMD parallelism).*




```
line0: let X = newbag () and Y = newbag ();;
line1: X <- { 1 || 1 ||1 }; Y <- { 2 || 2 };;
line2: X <- union of { (X || Y) || { sum of X || 4 || 5 } };;
 val X: int bag = {1, 1, 1, 2, 2, 3, 4, 5}
line3: ...next...
```

Prog. 1. (a) is the blob graph obtained after executing line 1, (b)-(f) describe the steps for executing line 2. The top node can already begin executing line 2 in (b). The bag id X (or Y) is represented by horizontal (or vertical) hatching. Master nodes are labeled by the compilation rule, which produces the code that the nodes execute in the next step. Slave nodes are labeled by the integer element hosted. The round arrow head indicates dynamic orientation.

Basic manipulation of bags involves a parallelism similar to SIMD: a bag of n integers $S = \{v_1, \ldots, v_n\}$ is stored in n atoms $b_1, \ldots, b_n$. Atom b_i is called an *elt-blob* and

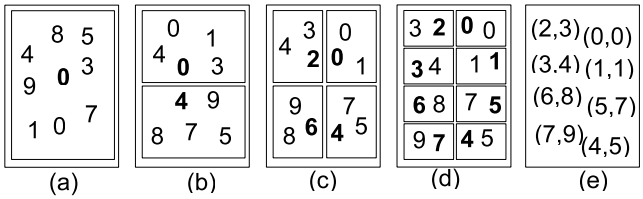
stores v_i in a SDA register called *elt*. The elt-blobs execute commands broadcast by their outer blob, called the master blob. Program 1 illustrates the development obtained when blob b builds a new bag by forming the union of two existing bags X and Y, and a third bag {`sum of X,4,5`}, where the first element is the sum of X's elements. Union is implemented as a reduction operator. Elt-blobs have a second register called *id* containing the "*bag address*" S. The bag address is allocated by the `newbag` primitive. In the example, $X = \{1, 1, 1\}$ and $Y = \{2, 2\}$, thus b contains three atoms with $id = X$ and $elt = 1$, and two atoms with $id = Y$ and $elt = 2$. Blob b sends commands to its elt-blobs by broadcasting down a list of scalars $[S, f, p_1, .., p_k, end]$ including bag address S, a function name f, and scalar parameters $p_1, .., p_k$; it also **flips down** to give edge ownership to elt-blobs. The instruction **broadcast**$([l_1, \ldots, l_k])$ is a shortcut for **broadcast** l_1, ..., **broadcast** l_k,**broadcast** *end*. The elt-blobs execute a fixed SDA implementing the following slave loop: they receive a command $[S, f, p_1, .., p_k]$, they test if their id matches S, and if so, they execute the function call $f(p_1, .., p_k)$. In any case, they finish by **flipping up** to return edge ownership. The catalog of predefined remote function calls includes (1) crude blob instructions: **setp**, **divide**, **merge**, (2) two functions using the blob-elt registers `let send_elt () = send !elt;;` and `let set_id x = id:=x;;` [5] which send the element up to b and set the identifier, and (3) two functions for routing blobs: `let duplicate () = broadcast([*,duplicate]); divide;; let delete () = broadcast([*,duplicate]); merge;;` which duplicate and delete a blob hierarchy, by calling themselves recursively on all inner blobs, using the symbol '*' to match all ids. The SDA program part executed by b is compiled using the six rules described in figure 11 [6] .

```
r1: code(S<-exp1;exp2) = route(exp1,exp2); wrap-;
    if testp up then code2(exp1,S) else code(exp2)
r2: code2(S2,S)=broadcast([S2,setid,S]);flip down;merge;
r3: code2( {exp},S)= id:= S; elt:=code(exp);flip up
r4: code2(exp1||exp2,S)=route(exp1,exp2); divide;
    if testp up then code2(exp1,S) else code2(exp2,S)
r5: code2(union of{exp},S)=code2(exp,S)
r6: code(SUM of S)= broadcast([S,send_elt]);flip down;sum;
```

Fig. 11. Rules for compiling blob programs into blob primitives

The rules in r_1 compile the bag assignment `S<-exp1;exp2`. The code produced by `route(exp1,exp2)` routes inner blobs by analyzing liveness of bag variables present in exp_1 and exp_2. It commands a negative polarization (or positive polarization, duplication, deletion) to inner blobs representing bags which are alive only in exp_1 (or only in exp_2, in exp_1 and exp_2, neither in exp_1 nor in exp_2). In the example, the code produced by `route` to go from (a) to (b) is `broadcast [Y, duplicate]; broadcast [X,setp -];flip down;` Y is duplicated because it is assumed to remain alive in the upcoming instruction $exp_2 = next$, while X is not because it is being redefined. As a result of this preliminary routing, the instruction **wrap-** encapsulates all the elements needed to evaluate exp_1 into blob b_1, which can evaluate the bag expression exp_1 in parallel with b, which in turn continues evaluating exp_2. b_1 is called the *bag-ancestor*. The bag expression exp_1 is not compiled using `code`, but `code2`, which has an additional second argument S carrying the address of the bag being computed. The parameter S is used in rule r_2 and r_3 to initialize the *id* of the elements produced by the bag ancestor. Rule r_4 compiles the double bar `||` that makes the bag ancestor divide. Each of the two children computes a separate sub-bag of elements in parallel. As for bag assignment, this parallelism implies a preliminary routing, so that each child gets the inner blobs it needs. Rule r_5 is a simple compilation optimization. Rule r_6 performs reduction. When a bag variable is no longer alive, the inner blob representing it must be deleted. This garbage collection function is not completely described in the presented rules. For example, in Figure (e) of program 1, the blob executing r_6 must delete its inner blob, as shown in Figure (f). Bag hierarchy can be defined as `S<-{(1, {(2, {(3, {})})})}`, of the type `blob-list = (int*blob-list) bag`. Circular structures, however, cannot be used because they would break the blob hierarchy. An assignment such as `S<-{(1,{ (2,{(3, S)})})}` defines the new value of S by making a copy of the old value.

3.2. *Remote function calls in inner blobs (DVC parallelism)*




```
let card var X = sum of for x in X do return 1 done
val card: var 'a bag -> int
let qsort(X, nX, i) = let pivot = sum of X / nX and Y = newbag () in
    if n = 1 then { (i, pivot) }
    else Y <- for x in X do if !x < pivot then return (consume x) done;
         let nY = card Y in union of
         { qsort(Y, nY, i) || qsort(X, nX-nY, i+nY) }
val quicksort: int bag * int * int -> (int * int) bag
```

Prog. 2. Quicksort program. A development for this algorithm is shown in Fig. 1 in network representation. This shows the topological development for the call `qsort( { 0 || 1 || 3 || 4 || 5 || 8 || 9 }, 8, 0 )`. It is uniformly dividing. Particles are represented by the content of their *elt* register: the master particle by the i index in bold font, and the slave by their element being sorted.

The for-in construct. Basic manipulation of bags only provides a fixed catalog of functions that elt-blobs can perform. The for-in construct `for s in S do exp done` allows master blob b to remotely trigger an arbitrary computation

[5] In our OCAML notation, `elt` is a reference, so `!elt` returns the referenced value.

[6] The compiler that produces low-level blob pseudo-code is formalized, but not yet implemented. A low-level blob pseudo-code simulator has been implemented to check the quicksort algorithm.

on its inner elt-blobs $b_1 \ldots b_n$ whose id is S. S is called the range, s the range index and exp the body. The master b first broadcasts down information contained in exp that the elt-blobs $b_1 \ldots b_n$ do not have locally. Thereafter, $b_1 \ldots b_n$ can evaluate exp in parallel on the elements that they host, which is referred to in exp by the `range index` s. The body may contain an instruction `return v`, in which case the elt-blobs return the value v to b. The values that are returned can be reduced by b, as is the case for the `card` function in program 2. Reduction is restricted to some predefined set of available operators. The returned values can also be regrouped in a new bag Y, as performed by the for-in of function `qsort` in program 2. From a theoretical point of view, there is no difference between applying a reduction such as a sum or forming a new bag. In the first case, one uses addition to reduce the values, in the second case, one uses a kind of merge function to aggreate the values. These are two instances of bag homomorphism. However, a bag is a particular value, a collection that can take any large space, and therefore needs to be returned as a set of blobs. It is built directly using blob operations. When returning a bag, there is no need to create a bag ancestor, since the new elt-blobs produced are directly computed by the elt-blob of b.

Lastly, it is also possible that nothing is returned. For-in action produces two kinds of side effects: they can modify the elements, as is the case of the for-in of the bmerge function (next subsection); or generate communication as in function `input` of program 4. Usually, accepting side effects in functions called asynchronously and remotely leads to a form of non-determinism that makes it very difficult to achieve confluence. But this is true only when side effects apply to certain globally shared variables. The two kinds of possible side effects created by elt-blobs are local to the elt-blob's internal memory and connecting channels.

Each for-in body is compiled into a function f whose code is appended to the SDA of the elt-blob of S. The for-in itself is replaced by remote call f in the elt-blob of S. For example, the `card` function is compiled into: `let SDA_card X = broadcast[X,f1];flip down;reduce SUM;;` where `let f1 () = send 1;;`. The for-in of `qsort` changes the id of the elements of X which are smaller than a pivot. It is compiled into `broadcast[X,f2,Y,pivot]` where `let f2 S v = if !elt < v then id := S;;`.

Blob arrays. The parameter of a quicksort call `A <- qsort(X,n,i )` includes the cardinal n of the bag X of values to be sorted (to avoid recomputing it twice), and the starting index i to give to the smallest element. It returns array A, called a *blob array*, encoded as a bag of pairs, where the first element of the pair is the array index, and the second element is the array value. Thus if X is a singleton, `qsort({x},1,i)` returns $A = \{(i, x)\}$. In the Von Neumann style, array indices are not explicitly stored in memory because array elements are implicitly stored in contiguous memory cells. In contrast, implementing an array as a blob array requires that each element be stored with its explicit index. While using more memory, this gives hardware freedom: the actual location of each element with respect to the other ones does not matter. The following three functions illustrate simple computations on blob arrays that are used in the next section: computing the minimum index, dividing an array into two sub-arrays of equal size (up to one element if size is an odd number), and inverting the indices.

```
let Amin var X = min of for x in X do let (i,v)=!x in return i done;;
val Amin: var (int*int) bag -> int
let Adivide var X = let c = card X and m = Amin X in for x in X do;;
    let (i,v)=!x in if i< m + c/2 then return(consume x) done;;
val Adivide: var (int*int) bag -> (int*int) bag
let Ainvert X = let c = card X and m = Amin X in
    for x in X do let (i,v)=!x in return (c+2*m-i,v) done;;
val Ainvert: (int*int) bag -> (int*int) bag
```

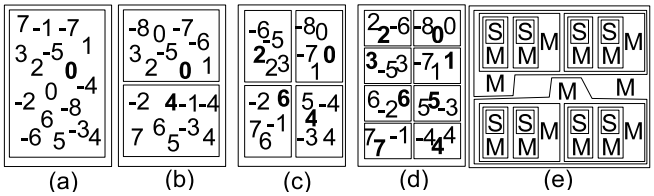
The quicksort algorithm. The `qsort` function uses a DiVide-and-Conquer (DVC) method. A pivot is computed as the average value of X, the for-in renames as Y any elements of X smaller than the pivot. The result is defined as the union of bags produced by a recursive call performed on Y and X, with new parameters for the starting index and cardinals. Recursive bag division continues until singletons are obtained, which can be directly sorted by assigning the starting index as the array index. In the sequential version quicksort cannot be terminally recursive: there are two recursive calls (one on each sub-array) and this does not allow any of the calls to be terminally recursive. The blob framework allows a generalized concept of terminality: the quicksort algorithm shown here is terminal because the bag elements "compute themselves" starting from the bag ancestor, and "return themselves" directly. When the two recursive calls terminate, there is no need for any additional processing to concatenate two sub-arrays, for example. Being terminal, recursion can be compiled with a bounded stack (independently from the data). This secures the finite state constraint. The blob stack that carries out recursion does not even store the element to be sorted, but only the three quicksort parameters X, nX, i that are scalars.

dDcomplexity. If all goes well, the pivot always splits X into two bags of equal size. This results in uniformly dividing development, optimal time complexity in $\mathcal{O}(n^{1/d})$ and optimal space complexity in $\mathcal{O}(n)$. The work in [26] presents a blob machine emulation using dDcomplexity. The average time and space needed on a 2D grid was measured, where elements to be sorted are chosen randomly, and results gave the same time complexity, i.e., $\mathcal{O}(n^{1/2})$.

Technical notes on bag functions: 1) Bag X can be passed as a variable to function f using the keyword `var`. If X is modified by f, the modification will remain when f terminates, and this creates a side effect. For example, the side effect of the `Adivide` function is to consume the lower half of X, whereas `qsort` has no side effect on the bag being sorted. Passing a bag as a variable is also useful for avoiding bag duplication, which is why it is used in the `card` function. 2) Setting the id of bag parameters. Function f can also return a bag. In the assignment `X<-qsort(...)`,

the caller must generate the bag ancestor b of X. The id X must be passed to f as a supplementary parameter, so that b can directly set the *id* of X's elt-blobs.

3.3. *Computation that combines bags (data parallelism)*




```
let bmerge X = if card X = 1 then X else let Y = newbag () in
    Y <- Adivide X; for x in X dot y in Y do
      let (i,vy) = !y and (j,vx) = !x in
      if vx < vy then begin y:= (i,vx); x:= (j,vy) end
    done; union of { bmerge Y || bmerge X }
val bmerge: (int*int) bag -> (int*int) bag
let bsort X = if card X = 1 then X else let Y = newbag () in
    Y <- Adivide X; bmerge union of { bsort Y || Ainvert (bsort X) }
val bsort:(int*int) bag -> (int*int) bag
```

Prog. 3. Bitonic sort program. (a)-(d) illustrate uniformly dividing development caused by the for-in with dot products of the `bmerge` function. The master particle is omitted and the slaves are represented by the index of their array elements. These indices range from -8 to 7. In (d) the element of index $i, 0 \leq i \leq 7$, is paired with the element of index $i-8$. (e) shows how the tree of recursive calls for sorting eight integers is laid out. The letter S stands for bsort, and M for bmerge.

*For-in using **dot** product of **ranges***

Consider a blob b containing two blob arrays, X and Y, with n elements $(x_i, i), (y_i, i), i = 1 \ldots n$. The purpose of data parallellism is to combine values from different arrays in parallel. For example, vector sum or scalar product computation (x_i, i) must be combined with $(y_i, i), i = 1 \ldots n$. This is usually specified using sequential loops to access the values of X and Y. In a grid with d dimensions, the diameter of a blob array with n elements is $n^{1/d}$, hence the time required to read a value is also $n^{1/d}$. Such an expensive read precludes the use of sequential loops. The alternative solution is to use iterative division similar to that used for quicksort through the following process: 1) Create an initial blob b_1 called the *for-in ancestor*, containing both X and Y. 2) Let b_1 divide the blob arrays X and Y, using the `Adivide` function. 3) Let b_1 divide itself: the first (or second) offspring contains the first (or second) half of X and Y. 4) Let both offspring recursively divide in this way until the cardinal of the remaining local parts of X becomes one. 5) This will generate n offspring, $b_1 \ldots b_n$, where b_i contains (x_i, i) and (y_i, i) and can combine them. The iterative division is uniformly dividing and has a time complexity of $\mathcal{O}(n^{1/d})$. It is compiled from a language construct inspired by Sisal that uses a for-in with a dot product of range: `for x in X dot y in Y do exp done` where the body *exp* is evaluated in parallel by each of the b_i. The range index x (or y) refers to (x_i, i) (or (y_i, i)). The "vector sum" and scalar product are programmed as follows:

```
let vectorsum X Y = for x in X dot y in Y do
    let (vx,i)=!x and (vy,j) =!y in return (vx+vy,i) done;;
val vectorsum: (int*int) bag * (int*int) bag -> (int*int) bag
let scalarprod X Y = sum of for x in X dot y in Y do
    let (vx,i)=!x and (vy,j)=!y in return (vx*vy) done;;
val scalarprod: (int*int) bag * (int*int) bag -> int
```

As for for-in with a single range, modifying the range index x or y results in modifying X (or Y), which is what the for-in of `bitonic merge` in program 3 does: it exchanges x_i with y_i if $x_i < y_i$. The first range X is called the master range, i.e. it is the range on which the stop condition is applied. It is necessary to define what happens if X and Y do not have the same number of elements. 1) If X has less elements than Y, then at the end of recursive division, each of the b_i contains one element of X called x, but still has several elements of Y. The solution is simply to keep this set with the same name Y and to skip the range index y. The for-in must be written `for x in X dot Y do exp done` where Y is called a non-terminal range. 2) If X has more elements than Y, then y takes a predefined value called $\perp$, meaning "undefined". By convention $x > \perp$ is always true, and $\{\perp\} = \{\}$. These conventions work nicely in the case of `bitonic merge`: they allow the sorting of arrays of arbitrary size, not just sizes which are exact powers of two.

The bitonic sort algorithm. Program 3 implements the bitonic sort. A bitonic sequence is composed of two subsequences, one ascending and the other descending. A bitonic sequence $[0, 2n)$ has the following property: it can be divided into two halves, $[0, n)$ and $[n, 2n)$, such that 1) each half is a bitonic sequence, and 2) each element in half $[0, n)$ is less than or equal to each element in $[n, 2n)$. The elements in the corresponding positions are simply compared in both halves and exchanged if they are out of order, where the sequential loop is: `for (i=0;i<n;i++) { if (get(i)>get(i+n)) exchange(i,i+n); }`. The `bitonic merge` for-in processes the sequence in exactly this manner. The `bitonic sort` works by using a double DVC mechanism. 1) It sorts the lower half in ascending order and the upper half in descending order. This gives a bitonic sequence. 2) It performs a bitonic merge of the sequence, resulting in a bitonic sequence in each half, with all the larger elements in the upper half. 3) It performs recursive bitonic merges on each half until all the elements have been sorted. A bitonic merge is a terminal recursive function, but a bitonic sort is not. As shown in Figure (e) of program 3, however, the two recursive calls are remotely called on two inner blobs, hence the stack's size remains constant and the finite state constraint is true.

dDcomplexity Unlike quicksort, bitonic sort and bitonic merge of n elements always divide the problem size exactly in two. Time complexity in both instances is $t(n) = \mathcal{O}(n^{1/d})$ and space complexity is $\mathcal{O}(n)$. A demonstration of this result is given in [38]. $\mathcal{O}(n^{1/d})$ is an optimal VLSI complexity as proven in [39] for 2D grids and [40] for 3D grids. A sorting algorithm on a 2D grid reaching optimal complexity has already been given in [40], but it is much more complex than the one presented here: for example, it requires

tiling a 2D grid of PEs, with a tile length of $n^{1/4}$. Another algorithm, optimal for 3D grids, is also shown in [40]. The elegance of the present model lies in the fact that the same simple algorithm is ideal both for 2D and 3D grids.

3.4. *Developing non-hierarchical networks using channels (pipelined parallelism)*

The channel master. An arbitrary graph is not a pure hierarchical structure and cannot be coded using only blobs. For example, a graph link is represented by using a channel c. The channel is owned by a unique blob b_c called the channel master of c. The outer blob of b_c is called the source end and the blob at the other end of channel c the target end. In this way, the channel master can be commanded by a straight `move` that transfers the master b_c from its outer blob (source of c) to the target blob. The target blob consequently becomes the source end of c. The functions `put` and `get` are added to the catalog of predefined functions for channel masters. Communication across the link from source to target is performed using the elt-register of the b_c master channel: `let put x = elt:= x; move ;; let get () = send !elt; move ;;`. When a value is written from the source end, it can be read from the target end and is then available for another write from the source. The predefined `duplicate` function is redefined for channel masters: `let duplicate () = move; flip up; if rec()=duplicate then begin divide; flip up; move end else merge;;`. It performs two `move` instructions, back and forth. Division for duplication is only performed if the duplicate command is given from both ends. This synchronizes the two ends of a link when they want to duplicate the link.

Xblobs polarize ends. The channel master does not have an id, but both the source and target ends of the channel (called Xblobs) have one. Each link is therefore represented by three blobs: a channel master, a source and a target Xblob. Xblobs are used to polarize the link ends separately. This is required when polarization for routing is commanded from the outer blob. For the rest, when Xblobs receive a command $[S, f, p_1, .., p_k]$ where f is `move`, `put`, `get` or `duplicate`, they check if their id matches S as usual, and if so, forward the raw command to their channel master using the instruction `broadcast [f,p_1,..,p_k]; flip down`.

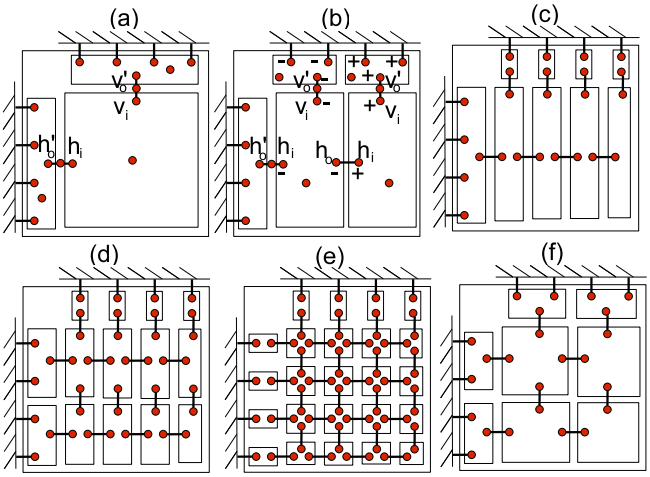
Xpairs handle a target and source coherently. A bag of Xblobs belongs to the special type "Xbag". It is practical to manipulate Xbags using a pair of singletons $(\{i\}, \{o\})$ with `type Xpair = Xbag*Xbag`, where i (or o) is a target (or source) end of an incoming link (or an outgoing link), and can receive (or send) data from (to) the blob at the other end of the link. First, when a link is created by the predefined `new_link` function, it returns an Xpair, the pair of ends of the newly created link. Second, communication and link creation are easily programmed using two functions on Xpairs: `Xshift(i,o)` receives a value v from i, forwards it to o, and returns v, and `Xdivide(i,o)`, which creates a new Xpair and inserts it between i and o.

```
let Xshift var xx = let(i,o) = xx in let v=get i in put o v; v ;;
val Xshift: var Xpair -> float
let Xdivide(var xx)= let(i,o)=xx and (i',o')=new_Link() in
    let r=(i,o')in i<-i';r
 val Xdivide: var Xpair -> Xpair
```

Ports. Xblobs are also used to control ports. Each port edge is assigned a dedicated Xblob. The ports are indexed by arranging their Xblobs into a blob array with `type ports= (int*Xbag) bag`. The `put` (or `get`) function is redefined for those Xblobs that control ports so that they can output (input) data through the port. `let put x = broadcast x ;; let get () = sum ;;`. The `sum` reduction function is coherent because port edges are handled like vertical edges. It sums a single value since the Xblob has a single port. The function `input(var P:ports,x:Xbag,n)` of program 4 gives an example of port usage: a for-in specifies a function that inputs n values, one by one, on each of the port edges, and forwards them through another Xblob x also passed as a parameter. The body of this for-in uses blob x. This deserves explanation because until now, this discussion has only shown for-in bodies using scalars. Unlike scalars, Xblob x cannot be packed into a message and broadcast to inner blobs. So how is this for-in compiled? Using iterative division of for-in ancestor b, in exactly the same way a for-in with a dot product is compiled. The range must therefore be a master range. A copy of x is placed in b, and before each division, b commands x to duplicate so that each offspring gets its own copy of x. A distinct copy of x is then available for each parallel evaluation of the for-in body. More generally, when compiling a for-in, a liveness analysis must determine whether certain blobs $b_1, \ldots b_k$ are used in the body. If this is the case, a copy of $b_1, \ldots b_k$ is placed in a for-in ancestor that repeatedly duplicates $b_1, \ldots b_k$ while dividing.

Using integer intervals and Xpairs as range. For-in can be generalized to make dot products on other types of range than just blob arrays. For a variable v to be eligible as a range, a divide function must be defined on v. 1) For blob arrays, the `Adivide` function, introduced previously, divides an array. 2) For integer intervals, division just splits the interval in the middle. 3) For Xpairs, the `Xdivide` function is used. This case is a bit unusual, since `Xdivide` separates the input from the output, but also adds a new input and a new output, and the size of the Xpair does not decrease. Therefore an Xpair cannot be used as a master range, i.e. the first range of the for-in on which the stopping condition is applied. In addition, Xpair h is always used as a non-terminal range, so the range index must be skipped as stated earlier. A for-in `for i in 1..n dot h` develops a 1D grid: it generates n blobs $b_1 \ldots b_n$ where the Xpair $h = (i, o)$ available in blob b_i is such that i can receive values from b_{i-1}, and o can send values to b_{i+1}. A nested for-in using two Xpairs, h and v, `for i in 1..n dot h for j in 1 .. n dot v` develops a 2D grid. In the outer (or inner) for-in on i (or j), h (v) is divided, while v (h) is duplicated. The Xpair

h (or v) implements communication on the horizontal (or vertical) axis.




```
let input (var P,x,n) = for ip in P do let (_,p)= !ip in
    let xx = (p,x) in for i=1 to n do shift xx done done;;
val input: var ports*Xbag*int -> unit
let prodmat (left,up,n)=
    let h = (newbag (), newbag ()) and v = (newbag (), newbag()) in
    let (_,h')= Xdivide h and (_,v') = Xdivide v in union of {
        for i in 1..n dot h do for j in 1..n dot v do let C = ref 0 in
            for k = 1 to n do C := C + prod of { shift h||shift v } done;
            return ((i,j),C)
                   || input(left,h',n)
                   || input(up,v',n)
        done done }
val prodmat: ports * ports ->((int*int)*float)bag
```

Prog. 4. Program for matrix multiplication. (a) to (e) illustrate the development induced by the nested for-in. (f) replaces (c) when horizontal and vertical development are interleaved. To save PEs, Xblobs do not have membranes: they are mapped as particles at the ends of the filament representing the channel, and the channel master is mapped as a particle inside the channel. In (c)-(f), particles representing the blob and channel master are omitted to simplify the illustration. (a) and (b) also show Xblob id: the input (or output) id of Xpair h is named h_i (or h_o), as is the case for v, v', h'. Space is allocated before development starts. This causes a temporarily low density in (a), which gradually increases from (a) to (e).

The Kung and Leiserson algorithm [40] for multiplying 2D $n * n$ matrices is implemented using the function `prodmat(left,up,n)` in Program 4. To simplify the discussion, the two matrices to be multiplied are input directly from two blob arrays of n ports called $left$ and up. The program not only specifies the Kung and Leiserson circuit, but also how to develop it. In Phase 1 it develops the circuit, while in Phase 2 the nodes of the circuit iterate a cycle of n systolic computations. In Phase 1 three developments are launched in parallel. The main nested for-in `for i in 1..n dot h for j in 1 .. n dot v` generates a 2D grid, and the two calls to the `input` function connect the left and up ports to the 2D grid using Xblobs h' (or v') connected to h (v). Because of the specific redefinition of `duplicate` for channel masters, h and h' (v and v') are synchronized when they duplicate, thereby synchronizing the three parallel developments of Phase 1. In Phase 2, the body of the nested for-in implements the Kung and Leiserson algorithm. It totals the sum of the product of values input from the left (or up) link, and forwards those values to the right (or down) link, in a pipelined way. The bodies of the `input` for-in feed the 2D grid with rows of A from the left, and columns of B from the up direction, where A and B are the matrices to be multiplied that must be sent by the host. The resulting matrix is returned as a blob array indexed using pairs of integers.

dDcomplexity. Setting up the circuit generates many particles. As a result, the **allocate** instruction must be inserted to produce a size-preserving development by assigning a virtual size to blobs. Each iteration of the nested for-in divides the virtual size by two, as for uniformly dividing development. Diameter does not decrease with size, however, because of the channels. The nested for-in `for i in 1..n dot h for j in 1 .. n dot v` has two phases: the outer for-in divides first horizontally and then the inner for-in vertically. Because of the tension induced by the channel, the dividing blobs remain on a horizontal line throughout the entire horizontal phase. Width is divided by two, but height remains constant, as is the case for diameter (see fig (c) of Program 4). To produce a development that alternates between a horizontal and vertical orientation, it is necessary to program the division of an object (h, v) by combining a vertical and a horizontal link so that it alternates between horizontal and vertical division, `let hvdivide var (h,v) = (Xdivide v,h)`, and then do the same using a pair of indices. The for-in must be written `for (i,j) in (1..n,1 .. n) dot (h,v)`. Using this refined program, both the diameter and size of the blobs are divided, and the complexity of $\mathcal{O}((n^2)^{1/2})$ is reached for Phase 1. Channel length remains $\mathcal{O}(1)$. Channels can be moved like particles and do not augment complexity. Their influence is limited to determining whether the direction of division is horizontal or vertical, and which half is the plus or minus half. In Phase 2 data are sent from the ports, completely crossing the grid. This takes an amount of time equal to the diameter of the grid, which is also $\mathcal{O}(n)$. For a 2D grid of PEs, this provides the best performance with respect to VLSI complexity (see [1]), but is not the best solution for a 3D grid of PEs. Consequently, unlike bitonic sort, the algorithm must be rewritten to improve processing in three dimensions by developing a 3D grid instead of a 2D grid. This shows that for some problems, the best algorithm depends on dimensionality in the computing medium.

4. Conclusion and perspectives

The purpose of this article is to give a detailed formal presentation of a virtual machine — the blob machine — whose primitives are sufficiently simple to be implemented on an arbitrary computing medium. The discussion shows that it is feasible to program the machine using a variety of different parallel styles, while achieving optimal complexity. The program specifies how "self-development" of a network of automata takes place through widespread use of parallel semantics. While parallel semantics introduce a particular twist to programming in this context, they also describe

spatial features and can therefore be reused for different hardware platforms based on the same type of computing medium. For example, although the present paper focuses on a fine-grain context for the sake of simplicity, granularity can range from FPGAs (reconfigurable architectures) to distributed memory multiprocessor machines with local connections. This work also targets irregular architectures encountered in amorphous computing. The only property of the computing medium that must be known to the programmer is dimensionality, since it restricts the set of feasible self-developing graphs. For example, since it is not easy to efficiently map a 3D grid to a 2D computing medium, the programmer may have to "collapse" one of the dimensions into memory. Nevertheless, there are algorithms that perform well on both 2D and 3D computing media, such as the bitonic merge sort in Program 3 in Section 3.3.

This article has chosen to program a catalog of *classic parallel algorithms* in order to compare expressiveness and performance with existing models of parallelism and promote our approach in the parallel computing community. As for expressiveness, the programs described rarely consist of more than four lines, and can often represent just a single line. The aim here is also to demonstrate the ability to go beyond the spectrum of purely spatial algorithms and program algorithms that "compute something" in the ordinary sense, such as quicksort or matrix multiplication, as opposed to spatial-specific algorithms. Performance is analyzed in terms of asymptotic complexity and shows optimality in the context of the VLSI complexity model, i.e., results are comparable to the performance of parallel algorithms running on a 2D or 3D grid of processing elements. Using architectures with greater connectivity, such as hypercubes, or a parallelism model with looser constraints, such as the BSP (Bulk Synchronous Parallel) or logP model, better time performance can be achieved, but at the cost of arbitrary hardware scalability. BSP, LogP and many other models of parallelism are compared in [41].

The blob machine is more appropriate for *unconventional parallel algorithms* that feature *decentralization*, *structure*, *dynamicity*, and *persistence*, which actually characterize many complex systems. *Decentralization* implies an algorithm describing a type of parallelism that is explicit, with a potentially arbitrary size. Complex systems consist of many parts operating simultaneously. *Structure* implies that the architecture (the set of communication pathways between the parts) matches the function it must perform. *Dynamicity* characterizes systems whose architecture can change at runtime, depending on input and the type of computation being performed. The system parts can move, delete existing connections and establish new ones. *Persistence* occurs when the rate at which the structure evolves is slow compared to the rate at which it computes. These four criteria, typical of a specific application niche, can be met by the blob machine's properties. *Decentralization*: the blob machine can manage arbitrarily large parallel resources. *Structure*: a unique feature of the blob machine is the ability to program network structure itself. This network development capability does not restrict the shape of the developed circuits to the lattice structure of the associated task graphs used to refine nested loops. *Dynamicity*: the weak point of parallelizing compilers, which are more specialized in static (compile-time) analysis. Dynamicity is a basic property of blob machines, achieved through hardware-free structuring and run-time self-placement. A classic example of a problem with a dynamic task graph is quicksort, presented in this paper. *Persistence*: necessary to absorb the time cost induced by self-placement. If the structure is modified too often, then the time spent re-placing it may exceed the time spent on computation. A recognized example of an application in the niche defined by those four features is Artificial Neural Networks (ANNs): they are obviously decentralized; the match between architecture and function characterizes brain circuits [42]; many ANN algorithms include a dynamic feature, adding or deleting neurons or synapses according to under-fitting or over-fitting of the task to be learned; and the architecture is persistent because modifications are applied only when learning new tasks.

"Blob computing" is a long-term project and still requires extensive work. The demonstration of complexity stipulates certain assumptions regarding blob machine implementation, and while the current implementation complies with these assumptions, it is not yet complete. This requires the development and articulation of many spatial algorithms. Furthermore, the programs developed thus far assume that there are always enough Processing Elements to allow each automaton of the virtual machine's configuration to be hosted on a distinct PE. If there are not enough PEs, then creating more blobs does not extend parallelism. It would then be more efficient to dynamically develop the blob system until all available resources are exploited, then stop self-development and deploy computation in time instead of space. Finally, the runtime system that implements a blob machine can be seen as an operating system responsible for updating the placement of many threads. It performs thread forking, computing, dying, and message exchange. This system itself is *distributed* and occupies a fixed bounded percentage of the hardware resources. Devoting hardware resources to self-placement will only be regarded as profitable when available hardware resources reach a critical size that remains to be determined by experimentation.

Acknowledgment. We thank rewiewers for their quite significant work and encouragements, and Donald Weston for his careful proofreading. We acknowledge support from EPRC grant EP/F003811/1 on general purpose spatial computation.

References

[1] T. Lengauer, Vlsi theory, in: Handbook of theoretical computer science (vol. A): algorithms and complexity, MIT Press, Cambridge, MA, USA, 1990, pp. 835–866.

[2] H. Abelson, D.Allen, D. Coore, C. Hanson, G. Homsy, J. T. F. Knight, R. Nagpal, E. Rauch, G.J.Sussman, R. Weiss, Amorphous computing, Commun. ACM 43 (5) (2000) 74–82.

[3] F. G. A. Dehon, J.-L. Giavitto (Ed.), Computing Media and Languages for Space-Oriented Computation 2006, Dagstuhl international workshop 06361, 2006.
URL
http://drops.dagstuhl.de/portals/index.php?semnr=06361

[4] Y. Feldman, E. Shapiro, Spatial machines: a more realistic approach to parallel computation, Commun. ACM 35 (10) (1992) 60–73.

[5] J.-P. Patwardhan, C. Dwyer, A. R. Lebeck, D. J. Sorin, Circuit and system architecture for dna-guided self-assembly of nanoelectronics, in: Foundations of Nanoscience: Self-Assembled Architectures and Devices (FNANO), 2004.

[6] B. Gojman, E. Rachlin, J. E. Savage, Evaluation of design strategies for stochastically assembled nanoarray memories, J. Emerg. Technol. Comput. Syst. 1 (2) (2005) 73–108.

[7] E. Winfree, Self-healing tile sets, architecture for dna-guided self-assembly of nanoelectronics, Nanotechnology: Science and Computation.
URL http://www.dna.caltech.edu/DNAresearch_publication

[8] L. Adleman, Computing with dna, Scientific american.

[9] S. Goldstein, P. Lee, J. Campbell, P. Pillai, Scalable shape sculpting via hole motion, International Conference on Robotics.

[10] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, A. DeHon, Stream computations organized for reconfigurable execution (SCORE), in: FPL, 2000, pp. 605–614.

[11] A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J. Moreno, J. Rosenberg, A. Villa, Poetic tissue: An integrated architecture for bio-inspired hardware, in: Evolvable Systems: From Biology to Hardware 5th ICES conference, Vol. 2606 of Lecture Notes in Computer Science, Springer, 2003.

[12] E. Rauch, Discrete, amorphous physical models, International Journal of Theoretical Physics 42 (2) (2003) 329–348.

[13] A. Adamatzky, B. D. L. Costello, T. Asai, Reaction-Diffusion Computers, Elsevier Science Inc., New York, NY, USA, 2005.

[14] J. Giavitto, O. Michel, Mgs: a programming language for the transformations of collections, LaMI technical report No 61-2001.
URL http://mgs.lami.univ-evry.fr/PUBLICATIONS/publicat

[15] J. Giavitto, Topological collections, transformations and their application to the modeling and the simulation of dynamical systems, 14th Int. Conf. on Rewriting Technics and Applications.

[16] J. Banatre, D. L. Metayer, Programming by multiset transformation, Commun. ACM 36 (1) (1993) 98–111.

[17] G. Paun, Membrane Computing. An Introduction, Springer-Verlag, 2002.

[18] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, E. Shapiro, Bioambients: an abstraction for biological compartments, Theor. Comput. Sci. 325 (1) (2004) 141–167.

[19] R.Nagpal, Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics, Ph.D. thesis, MIT (2001).

[20] D. Coore, Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer, Ph.D. thesis, MIT (1999).

[21] T. Toffoli, Programmable matter methods, Future Gener. Comput. Syst. 16 (2-3) (1999) 187–201.

[22] F. Gruau, Self-developing machines and parallel universality of the blob machine, submited to computational complexity.

[23] S. M. K. Tomita, H. Kurokawa, Graph automata: natural expression of self-reproduction, Physica D: Nonlinear Phenomena 171 (4) (2002) 197–210.

[24] K. Shahookar, P. Mazumder, Vlsi cell placement techniques, ACM Comput. Surv. 23 (2) (1991) 143–220.

[25] A. Contessa, E. Cousin, C. Coustet, M. Cubero-Castan, G. Durrieu, B. Lecussan, M. Lemaître, P. Ng, Mars, a combinator graph reduction multiprocessor, in: PARLE '89: Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Springer-Verlag, 1989, pp. 176–192.

[26] F. Gruau, Y. Lhuillier, P. Reitz, O. Temam, Blob computing, in: CF '04: Proceedings of the 1st conference on Computing frontiers, ACM, 2004, pp. 125–139.

[27] F. Gruau, P. Malbos, The blob: A basic topological concept for hardware-free distributed computation, in: C. Calude, M. J. Dinneen, F. Peper (Eds.), Unconventional Models of Computation, Third International Conference, UMC 2002, Kobe, Japan, October 15-19, 2002, Proceedings, Vol. 2509 of Lecture Notes in Computer Science, Springer, 2002, pp. 151–163.

[28] G. E. Blelloch, Nesl: A nested data-parallel language (version 3.1), Tech. Rep. CMU-CS-95-170 (1995).

[29] F. Gruau, G. Moszkowski, The blob division a "hardware-free", time efficient, self-reproduction on 2d cellular automaton, in: I. A. Jan, M. Masayuki, W. Naoki (Eds.), Biologically Inspired Approaches to Advanced Information Technology: First International Workshop, BioADIT 2004, Lausanne, Switzerland, Vol. 3141 of Lecture Notes in Computer Science, Springer, 2004, pp. 317–337.

[30] T. Toffoli, N. Margolus, Cellular automata machines: a new environment for modeling, MIT Press, 1987.

[31] F. Gruau, J. Tromp, Cellular gravity, Parallel Processing Letters 10 (4).

[32] The home site of blob computing.
URL http://blob.lri.fr

[33] P. Buneman, S. Naqvi, V. Tannen, L. Wong, Principles of programming with complex objects and collection types, in: ICDT '92: Selected papers of the fourth international conference on Database theory, Elsevier Science Publishers B. V., 1995, pp. 3–48.

[34] J. T. Schwartz, R. B. Dewar, E. Schonberg, E. Dubinsky, Programming with sets; an introduction to SETL, Springer-Verlag New York, Inc., 1986.

[35] P. Hammarlund, B. Lisper, On the relation between functional and data parallel programming languages, in: FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture, ACM, 1993, pp. 210–219.

[36] R. Bagrodia, M. Chandy, M. Dhagat, UC: A set-based language for data-parallel programming, Journal of Parallel and Distributed Computing 28 (2) (1995) 186–201.

[37] X. Leroy, The objective caml system release 3.08, Tech. rep. (2005).

[38] F. Gruau, C. Eisenbeis, L. Maignan, Self-developing blob machines for spatial computing: the foundations, Research report, Inria (February 2008).
URL http://hal.inria.fr/inria-00258845/en/

[39] C. D. Thomson, The vlsi complexity of sorting, IEEE transaction on Computers.

[40] F. . Leighton, An Introduction to Parallel Algorithms and Architectures, Morgan Kaufmann, 1992.

[41] D. B. Skillicorn, D. Talia, Models and languages for parallel computation, ACM Computing Surveys 30 (2) (1998) 123–169.

[42] J. S. M. Arbib, P.Erdi, Neural Organization - structure, function, and dynamics, The MIT Press, 1998.