Self-developing blob machines for spatial computing: the foundations

Frédéric Gruau, Christine Eisenbeis, Luidnel Maignan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Self-developing blob machines for spatial computing: the foundations

Frédéric Gruau — Christine Eisenbeis — Luidnel Maignan

N° ????

Février 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

Self-developing blob machines for spatial computing: the foundations

Frédéric Gruau * † , Christine Eisenbeis† , Luidnel Maignan†

Thème COM — Systèmes communicants
Équipe-Projet Alchemy

Rapport de recherche n° ???? — Février 2008 — 45 pages

Abstract: Technology can now produce massive hardware resources, large enough so that it becomes increasingly difficult to organize it in a centralized way. Spatial computing proposes to model such huge hardware as a relatively homogeneous computing medium satisfying a locality constraint: communication time is related to geometric distance. While the constraint is weak enough to allow arbitrary scalability, it is strong enough to make the tasks of programming and mapping significantly more complex; programming thus becomes the central problem. We propose a two level programming approach: at low level, a run time system layer runs on the computing medium and transforms it into a virtual machine, called the blob machine; at high level, programs run on this more expressive virtual machine.

The system layer is implemented on the computing medium as a local rule that manages distributed objects similar to membranes and filament channels; system calls are distributed primitives that allow objects to be created or deleted. The physical interpretation of the objects allows the implementation on arbitrary spatial computing medium. This layer is responsible for placing the objects in the space and does it by simulating physical forces. Each object is controlled by a FSA (Finite State Automaton) whose output actions are the object primitives (creation & deletion).

The set of FSAs together with the communication pathways implied by the topology of encapsulated membranes and channels, define a network of FSA that can self develop. This "self-developing network of FSA" is the formal definition of the blob machine. We present this blob machine, and how to program it using a higher level language description. We illustrate the execution of many examples of small and simple programs that cover a wide spectrum of parallel paradigms, including SIMD, data parallelism, Divide and Conquer and

* also LIRMM, Laboratoire d'informatique, de robotique et de microélectronique de Montpellier, 31 rue Ada, 34000 Montpellier, France and CEMS, University of the West of England, Frenchay Campus, Coldharbour Lane Bristol BS16 1QY, United Kingdom

† Projet Alchemy, Inria Futurs Saclay - Parc Orsay Université, 4, rue Jacques Monod, 91893 Orsay Cedex, France

Centre de recherche INRIA Futurs
Parc Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 Orsay Cedex (France)
Téléphone : +33 1 72 92 59 00

pipelining. Usually, if the problem needs a space of $\mathcal{O}(n)$, the time complexity can be reduced to $\mathcal{O}(n^{1/d})$ where d is the dimensionality of the computing medium: 2D, or 3D. In some cases the same program runs optimally for any value of d, in other cases, the dimensionality must be considered.

Citation about a computer using physical objects as primitives, like blobs and channels, which has given us inspiration: *"The kinematic model [of self reproducing machine] deals with the geometric-kinematic problems of movement, contact, positioning, fusing and cutting"* John Von Neumann [Neu66]

Key-words: No keywords

Machines blob auto-développantes pour le calcul spatial: fondations

Résumé : Les progrès technologiques permettent de produire des composants matériels en nombre tel qu'il est de plus en plus difficile de les organiser de manière centralisée. Le calcul spatial propose de modéliser cette multitude de ressources matérielles comme un medium de calcul relativement homogène qui vérifie un critère de localité: le temps de communication est lié à la distance géométrique. Cette contrainte est assez faible pour rester vérifiée à grande échelle, mais elle est assez forte pour rendre complexes les tâches de programmation et de placement. La programmation devient le coeur du problème. Nous proposons une programmation à deux niveaux: à bas niveau, une couche système run-time transforme le medium de calcul en machine virtuelle, appelée la machine blob; à haut niveau, les programmes tournent sur cette machine virtuelle, plus expressive.

La couche système est implémentée sur le medium de calcul par une règle locale, qui considère les objets distribués comme des membranes et des canaux filaments; les appels système sont des primitives distribuées qui permettent la création et la destruction d'objets. L'interprétation physique de ces objets permet de les implémenter sur n'importe quel medium de calcul. Le placement de ces objets dans l'espace est réalisé par la simulation de forces physiques. Chaque objet est contrôlé par un FSA (Finite Sate Automaton - automate à états finis) dont les actions en sortie sont des primitives de création et destruction d'objets.

La combinaison de ces automates avec les canaux de communication résultant de la topologie des membranes et filaments forme un réseau d'automates qui peut s'auto-développer. Ce réseau auto-développant d'automates à états finis est précisément la définition formelle d'une machine blob.

Nous présentons la machine blob, ainsi que son mode de programmation à l'aide d'un langage de haut niveau. Sur de nombreux examples de petits programmes simples, nous montrons que différents paradigmes d'exécution parallèle peuvent être exprimés: SIMD, parallélisme de données, "Diviser pour régner", pipeline. En général, si un problème requiert un espace en $\mathcal{O}(n)$, la complexité en temps peut se ramener à $\mathcal{O}(n^{1/d})$, où d est la dimension du medium de calcul (2d ou 3d). Dans certains cas, le même programme est optimal pour toute valeur de d, dans d'autres, il peut dépendre de d.

" Le modèle cinématique [de la machine auto-reproductrice] traite des problèmes géométrico-cinématiques liés aux mouvement, contact, placement, fusion et découpage." John Von Neumann [Neu66].

Mots-clés : Pas de motclef

1 Introduction

1.1 Plan

Technology can already produce chips with billions of transistors. Research on unconventional computing consider building hardware yet bigger order of magnitude bigger, by using advanced nano-technologies. To harness such hardware, we advocate a change of perspective on how we think about hardware, and how we tackle programming. Thus we begin by giving two new points of view, involving some new concepts This does make an involved introduction however those concepts are sufficiently generic to be presented independently.

For hardware, a framework called spatial computing, proposes to abandon the idea of a number of Processing Elements (PEs) connected all to all by a router, but instead, think of it as a spatially extended homogeneous computing medium. Computation and data have to be distributed in a 2D or 3D space and the particular spatial arrangement will be intimately connected with the performance. We sketch the state of the art of spatial computing in order to position the blob computing framework as a vertical approach to spatial computing. We present a key property of spatial computer, namely: "hardware freedom" which allows machine configuration — blobs, data, code or communication links — to move continuously through the computing medium.

The second part of the introduction considers the problem of mapping a task graph to a computing medium. We do not want the programmer to be concerned about the exact spatial location of the program's data and computation, yet we do have to efficiently map the programmed run-time configuration on a 2D or 3D space. We analyse the nature of this mapping problem in order to support our view that this is the fundamental problem of parallelization. We describe how it is usually tackled, and contrast this with our new solution. This solution involves the concepts of Self Developing Automata network (SDA network), and self-placement. Part of the mapping is the responsability of the programmer who makes a preliminary architecture-independent mapping of a task graph which is a directed graph, onto a SDA network which can be seen as a smaller task "network" with recurrent links and dynamically evolving architecture. The remaining part is controlled by the run time system, which supports a "hardware free" representation of the SDA network, meaning the placement of the SDA network node can continuously change on hardware without modifying the semantics. This plasticity can then be exploited by the run time system, to do "self placement", that means continuously update and optimize placement, by minimizing some criterion related to performance.

In section 2 we present formally the blob machine and the state of the art of its implementation. The formal part uses a simplified model called the binary blob machine. The detailed semantics of this model is reported, and its confluence is proved. In the implementation part, we explain how to map membranes and channels on a fine grain 2D cellular automaton, and define a "dDcomplexity" model as a set of three constraints that the implementation should meet. We put forward a kind of development called uniformly dividing, which can be efficiently executed in the dDcomplexity. The complete blob machine is not yet implemented, however the most difficult instructions are, and comply with the dDcomplexity.

Section 3 introduces language constructs to describe development in a compact way. Blobs are multisets of their inner blobs, so a collection of data such as an array must be coded as a multiset in order to be distributed on blobs, and updated in parallel. We show that optimal performance can be attained, for classic parallel algorithms such as sorting and matrix multiplication, by programming uniformly dividing developments.

1.2 Spatial computing

1.2.1 Spatial computing on a computing medium

Spatial computing is an umbrella term regrouping different approaches, all based on the observation that future computing platforms — whether VLSI, bio, or nano — will be made of a huge number of Processing Elements (PEs) homogeneously embedded in a 2D or 3D space, where huge means that the programmer will face a *locality constraint*: each PE has a specific location in the space and communication time will be related to metric distance in that space. For example, in the — classic — VLSI complexity model [Len90], this relation is linear. Communication costs have always been a main concern in parallel computing. In early parallel architectures such as the Transputer or the Connection Machine routing was made by each computing node through its input and output links. In the middle of the eighties came the idea to think of the communication network as a black box linking each processing element. The performance of an all-to-all communication (e.g. a permutation) is more efficient using this approach, relying for example on network architecture developped for the phone network. However, such a black box router abstracts away spatial location.

Architectures embedded in space, that we call *"computing media"* include *regular* classic models, such a as cellular automata, systolic arrays, FPGAs; but also *irregular* models, relaxing the constraints of lattice tiling of space, and synchronism in time as exemplified in the amorphous model [ADC+00]. Spatial computing was the subject of a recent workshop [AD06]. A complexity model of computing media called "spatial machines" is presented in [FS92].

Spatial computing calls for a departure from computing in time, i.e. using a traditional centralized programming approach, with a step-by-step modification of some global state. Intuitively, to exploit the space, computation should unfold in space, by dynamic construction of spatial entities such as circuits. Spatial computing calls for new architectural designs. For example, many classic building blocks of parallel architectures such as shared memories or all-to-all routers, are designed to abstract space away, and re-establish a Uniform Memory Architecture (UMA). They cannot however organize communication through an entire arbitrarily large spatial computer, because their scalability is usually studied for a limited number of PEs, in which case a unit delay can be assumed on wires, the time taken for a signal to traverse the length of the wire is not taken into account.

Consider the example of a router abstracting away spatial location, and use the router's diameter to measure the performance of that router. If communication time between any pair of PEs does not depend on the communicating PEs, it has to be this diameter, which is the worst case. To scale for an arbitrary large space, communication has to be optimized, by taking into account physical distance. The hierarchy of different long line size used in FPGAs is an

example of a scalable building block that reduces the global network diameter. Spatial computing is a broad subject covering a large part of unconventional computing. We propose the following classification sampling the vertical scale from hardware to software.

1. Research on hardware itself : development of new technologies that enable spatial computing. Here *space matters for scalability*. Nano-technologies includes nano-tubes [PDLS04] [GRS05], DNA computing [Win06], chemical reactions [Adl98]. In the programmable matter framework of Goldstein [GLCP06], processing elements themselves can move.

2. Research on architectures: designing structures on existing spatial computing platforms such as FPGAs, so that as the hardware resources increase the same given program is able to exploit the added space and increase its performance. Here, *space matters for performance*. Dehon [CCH$^+$00] proposes a framework called SCORE that allows run time unfolding of pipelined circuits in space and is able to exploit different sizes of hardware by trading time for space. The "poetic" group at EPFL [TSF$^+$03] develops a chip specialized in bio-inspired algorithms including evolution, development and learning.

3. Research on algorithms: development of *spatial primitives* that use space and compute information about space. In turn, they allow to program in space in a more abstract way. Here, *space matters for functionality*. For example, the MIT amorphous group [ADC$^+$00] shows how to compute a set of coordinates for each PE elements of an amorphous medium, Eric Rauch [Rau03] simulates wave propagation, which can support communications on amorphous computers. Using reaction diffusion computers, Adamatzky [ACA05] computes the Voronoï tessellation, which partitions the space and also installs a network by using the Delaunay triangulation.

4. Research on spatial languages: here the notion of space is explicitly used as a programming metaphor and semantics; *space matters for expressiveness*. Giavitto and Michel [GM01] [Gia03] use the data structure itself as the computation space. Their MGS language does show that reasoning in space leads to compact programs if the task has a spatial formulation. The Gamma formalism [BM93] exploits the parallelism inherent in chemical reactions by avoiding any possible artificial constraint implying a sequential execution; this also leads to very concise programs. The idea of using encapsulated membranes has stimulated two projects of specific languages: Paun [Pau02] studies the formal language part of a model called P-system, Cardelli [RPS$^+$04] programs algebraic systems of membranes focused on simulating the real biological cell.

1.2.2 Vertical approach to spatial computing

Ideally, the overall goal of spatial computing is to encompass these four levels in a full vertical model that combines both a language and a machine: program in an *expressive* high level language based on a *functional* library of *spatial primitives*, running on an adequate *performant architecture*, defined for a scalable hardware. Several research works already address this grand challenge: Nagpal [R.N01] proposes a language based on primitives for folding a sheet of paper

as in Origami; Coore [Coo99] develops the Growing Point language, based on primitives that manipulates particles which emit gradients and move according to those gradients. Both Nagpal and Coore use amorphous computers. They develop patterns or circuits on top of the amorphous medium. However, once the structure is laid out, it cannot evolve any more, this limits the programming expressiveness. Toffoli [Tof99] proposes a programming framework also called "programmable matter" (like Goldstein), focused on efficient compilation towards cellular automata with block rules. However, the language is not very high level, covering mainly simulation in physics with problems such as noise filtering, simulation of polymers. Dehon [CCH+00] also works on both a language and an architecture. Its strong point is performance. However, it seems to target especially stream processing.

In a vertical framework the requirements of performance and expressiveness are necessary, but very difficult to achieve simultaneously. In practice, research on spatial languages does not consider efficient parallelization on a computing medium, and performance-oriented research is happy enough to use languages whose expressiveness is limited to a specific niche of applications, as long as significant improvements in speed performance are obtained.

1.2.3 Blob machine for spatial computing

The blob machine is a vertical framework for spatial computing trying to obtain both expressiveness and performance by using simple building blocks for managing space: **blobs** and **channels**. On the performance level, blobs and channels are like physical 2D/3D objects — membranes and filaments — and can thus be parallelized by the use of discretized physical laws on an arbitrary large 2D/3D computing medium. On the expressiveness level, blobs and channels can be dynamically developed. This allows dynamic data structures to be programmed. In general, blobs abstract the concept of compartmentalization that frees the programmer from having to consider the particular 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 to happen 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. Such a compartmentalization is not required if one considers pure *spatial primitives*, such as setting up a gradient in order to measure distance to a given point, in which case, the problem can be solved by having all participating PEs execute globally the same simple local rule. However, as soon as one needs to execute different types of tasks, and would like to run those in parallel, compartmentalization of the computing medium into disjoint connected regions allows different part of the medium to run different programs more efficiently: each connected region has its own thread of execution and needs to store and execute only the one program it runs.

Blobs embody dynamic development for dynamic computation. When programming, using dynamic data structures increases a lot expressiveness. One can colonize the memory with new data structures created on the fly, and then delete them. The use of single recursive function can result in the allocation of

arbitry large space on the stack. To obtain a similar expressiveness in spatial computing, the language description that develops and installs spatial structures on the computing medium, must also be able to delete them dynamically. Blobs embody this feature of dynamic development. Blob instructions can add or delete blobs at run time simply by creating or deleting compartments.

The location of the compartments needs to be adjusted at run time. Assume one has two tasks to execute, and decide to divide the medium in two, thus creating two compartments, one for each task. Now, if the tasks make some dynamic allocation, and the compartments further subdivide, one cannot predict at compile time, how much resources each task should be given, and thus, how big the compartment should be, and where to install the frontier between both compartments. If the frontiers of the compartments can move dynamically after they have been installed, then, it will allow the system to adjust the location of the separating membranes and by that, the amount of hardware resources allocated to each task. The system does a form of run time load balancing. Since blobs hold computation as well as data, moving blobs balances not only CPU load, but also memory occupation. Balancing communication load is also done by the system, by moving channels so has to homogenize their density. Note that hot spots are not avoided by redirecting messages through a fixed network, but by programming a virtual meta network made of channels on top of the existing networks linking PEs of the computing medium. This network is developed, along with blobs, and is moved over the computing medium to follow the blobs.

1.2.4 The perspective of hardware freedom

Moving blobs and channels is an example of a more general concept called hardware freedom, defined as follows: Consider some computing medium, and call a configuration, the content of the memory location controlling that hardware. It can be the bits configuring the Loop-Up Table (LUT), and the switch of an FPGA, or simply instructions and data stored in memory. In the blob machine we also consider the bits coding the presence or absence of a membrane or channel. A machine configuration is called *hardware free* if the particular location of its components can move continuously on the computing medium, without losing the semantics. Hardware freedom is relevant for spatial computing in general, because it allows us to deal with the locality constraint: to accommodate the fact that communication increases with the distance, hardware freedom suggests to move the threads on the hardware, so the threads that need to exchange messages may be brought closer together. When writing sequential code, software is viewed as being permanently tied to hardware. Let us illustrate how adopting the hardware freedom perspective can give a different fresh picture:

Example 1: *"Hardware free processes"*. A run time system performing dynamic CPU load balancing is able to migrate processes between processors without modifying the semantics, so as to balance the CPU load of each processor of the machine.

Example 2: *"Hardware free memory"*. If you store a data in a memory cell, you do not expect it to jump to the next memory cell, under the pressure of adjacent memory cells, as if they were elastic physical objects. It would

nevertheless be advantageous for the memory to self organize toward homogeneous occupation in the advent of over crowded and under crowded region. But instead of a globally indexed memory one would need an associative memory, using pattern matching on labels to bind addresses to data. Such a mechanism is robust to change in the particular location of the addressed data. On the other hand, it can be time expensive, because it implies searching a matching label on the entire memory. It is used by biological systems: for example, the beginning or the ending of genes within a genome are localized using specific markers.

Example 3: *"Hardware free circuit".* In FPGAs, once a circuit has been (re)configured on a sub rectangle of the chip, its inputs and outputs are bound to ports, or to adjacent circuits on the same FPGA. The net list of bits configuring the circuit is tied to hardware, and stays immobile on the chip, while the circuit executes. Moving it would imply a lot of additional control circuitry to save the current state, move the bits of configuration, restore the state, and reroute the inputs and outputs. Furthermore, today's FPGA are not conceived for local reconfiguration. It is usually a central external host that manages where and what to reconfigure. It would nevertheless be nice, to be able to locally "push" circuits on an FPGA by pressuring against each other as if they were elastic physical objects. This would permit the modification of already loaded circuits at runtime, without having to plan it in advance, and without having to reconfigure the whole FPGA. In this situation, installing an operator, or a circuit line between two operators, would loose the dramatic intensity of a decisive action that needs to be done as efficiently as possible. That operator or this line can always move after creation.

1.3 The mapping problem revisited for computing media

We propose to distribute the job of mapping software to hardware between the programmer and the Run Time System (RTS). Nowadays, when the target is a parallel computer, most of this task is done by parallelizing compilers. The range of programs that can be parallelized and the possible target architectures are restricted by the limited "intelligence" of those compilers. There exist two frameworks mapping software to hardware at run-time: a classic one: data-flow computers [Vee86] where mapping is done according to availability of tokens, and a recent one: grid computing [Jaf06] where mapping is done following the hardware availability. The latter is adapted to distributed applications where large chunks of computation can be done independently. Both of these approaches do not focus on optimizing mapping to minimize communication latency. This maybe a simple reason why they both need a high degree of parallelism to be effective. We are interested to obtain conplexity results valid for arbitrary large hardware of the computing media type, and considering parallel algorithms as opposed to distributed, i.e. programs whose task graph does imply constant communications. We identify the problem of minimizing communication latency as the main goal to attain, and we propose to share the effort between the RTS, and the programmer. The RTS exploits the regularity of the computing medium to minimize distance between communicating tasks by moving them, the programmer produces a folded form of the task graph to minimize

the number of tasks and communication links that need to be mapped. In our estimation, this represents the maximum possible human contribution, when considering only architecture independent formulations. The approach allows to parallalelize programs with dynamic task graphs, which as discussed in the landmark paper [GPKK82] (in the context of data-flow computing) is the area where static mapping is difficult and the overhead implied by dynamic mapping can be justified.

1.3.1 Parallelization is a "folding problem"

Why is parallelization difficult? Consider the *"Data Flow Graph"* (DFG) associated to a program execution: vertices represent arithmetic and logic operators and an edge connects an operator that produces data to an operator that consumes this data. A DFG is a directed graph without loops, every operator is used only once. In parallelism, one usually considers task graphs, which are a similar concept, but where tasks are coarser grain than just operators. Now, consider the graph of a parallel hardware, where vertices are Processing Elements (PEs) and edges are communication channels. To distinguish both graphs, we use the word edge and vertex, for DFG; nodes and links for the PE network. A parallel execution is defined by a mapping m associating each operator to a PE and a scheduling of the operators within a PE. This problem can be analysed independently from any programming language, although some language can ease the construction of the DFG underlying a program.

Assume first that the network has no router, so that only local communications are allowed, and forget scheduling for a moment. Operators that are linked need to be executed either on the same PE or on its direct neighbors, to support the communication of the data from the producer operator to the consumer operator. So the mapping m preserves adjacency and is therefore a *graph homomorphism* [gra] from the DFG, to the PE network. This homomorphism is usually called the mapping of a task graph to hardware. We propose to call it a *folding*, because the morphism is not one to one - in the general case - , otherwise each PE would execute only one operator. The graph homomorphism problem is a canonical NP-complete problem [BM95].

Secondly, if the target parallel hardware uses a network with routers, then we can define a similar more general concept of folding by mapping each edge to a communication path in the network of PEs. The problem of finding a legal execution is simplified by the router, but the global problem remains of the same complexity except it is now formulated as a problem of optimizing cost functions to balance load and minimize communications [NT93].

Thirdly, finding the optimal folding, is also dependent on the scheduling of operators within each PE. But the scheduling problem can also be grouped with the mapping problem, to be formulated as a graph homomorphism problem. One has to map the DFG to a target network defined as follows: the nodes are PEs indexed by integers, and p_t is linked to $p'_{t'}$ if $p = p'$ or p is neighbor of p' and $t' > t$.

In summary, we argue that the central difficulty of parallelization is the folding of a "programmed soft graph" on a given "hard graph", because it involves NP-complete graph homomorphism.

1.3.2 Folding on a static FSA graph

Because the folding problem is not tractable in the general case, the standard procedure is to consider restricted cases. On the software side, one uses specific language constructs to restrict the set of possible DFGs, on the hardware side, the architecture has to be chosen in a predefined family. One obtains a subset of DFGs and a subset of PE networks enough tuned to each other so that efficient folding becomes tractable.

Furthermore, the DFG is folded onto an intermediate graph by merging subsets of its vertices $n_1, \ldots, n_k$ into a single "macro-vertex" which schedules the k operators of $n_1, \ldots, n_k$ in a particular sequence. This macro-vertex must therefere store a local state indicating which operators it performs, and a transition function to specify the next state (the schedule). In other words, after folding, the macro-vertex obtained hosts a Finite State Automata (FSA) instead of memoryless operators. That FSA has no inputs, it changes state each time an operator is performed. Thus we now have a graph whose nodes are labeled by FSAs instead of a DFG. The final folding to hardware is indeed simplified, because by mapping an automaton on hardware, one simultaneously maps all the operations that this automaton is doing.

Example 1: The grid community [Jaf06] directly programs task graphs, which is just a different name for automata graph. Since communication is particularly expensive on the Internet grid, the algorithms try to minimize intertask communications, and tasks must be sufficiently coarse grained to keep a node busy computing on local data, before it has to communicate.

Example 2: The reconfigurable community [CCH+00] also programs task graphs, but their language constructs allow data flowing in streams between the tasks. This enables one to exploit the pipelining capability of FPGAs, and draw the performance.

Example 3: The data parallel community access array within nestep loop using affine combination of loop indices. The DFG is a k-dimensional regular lattice [Dar99] embedded in a k-dimensional discrete space. A mathematical analysis of the dependency graph allows describing this lattice in a finitely parameterized form, exposing the parallelism. In particular, with High Performance Fortran [KK95], the programmer specifies "templates" to explicitly align all the arrays on such a virtual k-dimensional space. The lattice can be folded onto a regular array of PEs, such as a 2D grid of PEs, by using a simple projection on a 2 dimensional discrete space, respecting data dependencies. If the grid is not big enough, one needs to additionally wrap computation around it, or to tile the 2D grid, which constitutes the final folding. The computation of optimized projections, wrapping or tiling remains tractable as long as the number of instructions in loops is small. This can lead to either automatic synthesis of systolic arrays, or automatic parallelization on networks of PEs.

Folding programs on a fixed — static — automata graph has some limitations: first, algorithms where the shape of the DFG is itself data-dependent, cannot be folded on a static graph. More generally, this does not scale easily for big programs, which are typically sliced, and analyzed piece by piece. For example, in data parallelism, each loop nest is analyzed separately, and each

generates a particular mapping. Optimizing the different mappings across the different loop nests has been proved to be NP-complete [LC90].

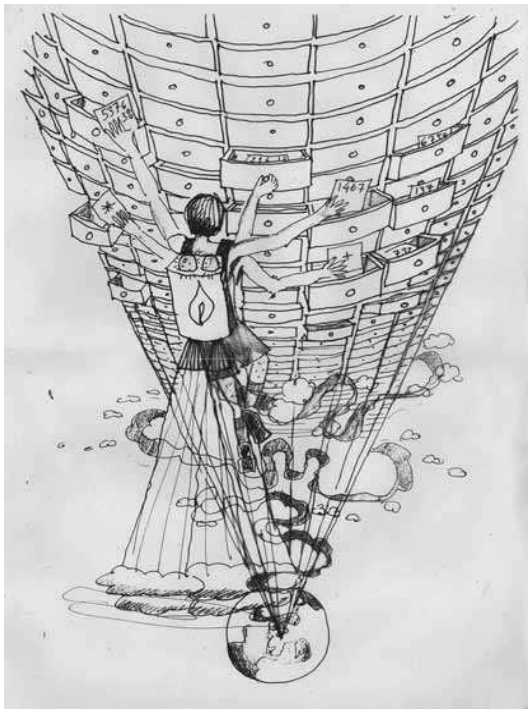


Figure 1: Artistic Illustration of the bizarre though well accepted dichotomy between processor and memory (courtesy of Paula Femenias)
.

1.3.3 Folding on a dynamic automata graph

Blob machines are also based on a match between a set of language constructs, and an architectural domain, but with the ambition of having a broader scope: arbitrary scalability on the hardware side, and no restriction on the software side: we want to handle the case of programs with dynamic taskfolding graphs.

- For the architecture, we target arbitrary computing media (see page 5). Their main advantage is that the locality constraint allows one to consider arbitrary large size.
- For the language, we fold on a dynamic instead of static automata graph (the architecture can evolve) which allows folding programs whose DFG is dynamic.

The von Neumann sequential model immediately implies a dichotomy between a huge passive part: the memory storing the global state, and a small very active part: the processor modifying that global state illustrated in figure 1. There exist alternative decentralized systems such as Cellular Automata

(CA), where the system elements have finite memory and can be controled by an FSA. Another less known interesting example is population protocols [AR07] which considers a collection of mobile agents with bounded memory which interacts with one another to carry out a computation. However both CA and population protocols are not very expressive, and hard to program outside specific algorithms. Our conviction is that the key to generalize programmation of decentralized systems is the ability to dynamically develop a network connecting the FSA, whose structure match the particular functionality to program. The Parallel Pointer Machine (PPM) introduced by Cook and Dymond [CD93] consists of a collection of FSA having a fixed size array of pointers to other FSA which determine an explicit architecture that is dynamically developed. An FSA a_1 has output actions which are instructions (it is a Mealy machine). It can execute instructions to create and initialize a new FSA, or to read the state or copy the pointers from another FSA a_2 it is connected with. This implies an implicit communication from a_2. This non-locality makes the PPM an inherently synchronous model which limits it scalability, because it implies a global clock.

In order to move one step towards decentralization, we are proposing in [Gru08] an asynchronous version of PPMs, called *"Self-Developing Machine"* (SDM) where FSA can execute their instructions independently, asynchronously, which clearly forbids direct access to the state of neighboring FSAs. Instead, communication is implemented by using links instead of pointers, and letting links carry a label, acting as a shared memory between both ends. The labels of a link can be modified at one end and read from the other end. An FSA does not access the links of its neighbors, but it can create links between two of its neighbors, and when an instruction creates a new FSA, is has to specify how to glue it to the neighbors. Formally, instructions are graph-rewriting rules, that can add or suppress vertices or edges in the graph, or modify the labels of vertices and edges. Some specific edges called *"ports"* are used for parallel input and output. The initial configuration is a single node called the ancestor, connected to the p port edges, and whose Mealy machine is the program of the SDM. During execution, the graph expands or contracts itself, according to this program, as the result of the instruction triggered by the automata. All the automata execute the same Mealy machine, but with a distinct state stored locally. Parallel execution is made possible by setting up some exclusion rules, this will be explained in the context of the blob machine, which is a particular example of SDM. An SDM is of course a virtual machine, since we do not assume that automata and edges are created or destroyed physically. A particular kind of SDM has already been explicitly introduced, to model self reproduction [KT02].

Using dynamic graph of automata gives more folding opportunities. For example, the quicksort program (figure 2) illustrates how the DFG of the quicksort of 4 values $n_1, \ldots, n_4$ is folded on a dynamic graph having up to 8 nodes (not counting the top "master node"). Each of $n_1, \ldots, n_4$ is stored on a distinct automaton $a(n_1), \ldots, a(n_4)$, and all the comparisons between a given n_i and the different pivots encountered during execution are folded, i.e. executed on the same automaton $a(n_i)$. This is made possible because the pivots are passed along a dynamically evolving intermediate network structure, made of 1, then 2, then 4 nodes. Those intermediate nodes pass the pivot, and store a temporary rank.

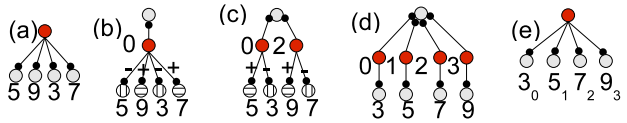


Figure 2: We illustrate snapshots of the development for the quicksort of four integers: 5,9,3,7 The program is given and explained page 32
.

1.3.4 Self mapping of self-developing machines

The parallel semantics of SDMs includes more than just exposing parallelism: communications are always local; an automaton communicates only with its direct neighbors. Note that this network is virtual, therefore it does not mean that two communicating automata can always be placed on neighboring PEs, if for example, an automaton is connected to n other automata, and each PE stores a single automaton, it needs 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 which is $n^{1/d}$ if the grid has d dimension. What it does mean is that there is no shared memory, or even a global name space. Who communicates with whom, this is thus clearly represented at any time by the network itself. This enables the run time system to automatically map the network. We consider architectures, which are computing media, embedded in a 2D or 3D space. That space is partitioned and each PE is responsible for one piece of it. Each vertex of the SDM has coordinates, which determine the PE responsible for hosting it and updating it. In order to be able to dynamically map the SDM, the run time system should implement two things:

- Firstly, it has to provide a hardware free (see page 8) distributed representation of the vertices and edges representing the self-developing graph: vertices must be able to freely move between neighbor PEs without interference with the underlying computation going on.
- Secondly, the run time system must determine the appropriate direction in which to move the vertices. For this purpose, it simulates physical forces of attraction between adjacent vertices so as to optimize communication latency, and 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 placed in equilibrium with respect to the fixed vertices used as ports. Whenever self-development occurs, vertices or edges are added, or suppressed, and the equilibrium is perturbed. Each vertex locally computes the force applied to it from its neighbors, and moves according to these forces, possibly migrating to a neighbor PE, if its coordinates are no longer in the area managed by its current owner PE. After some iterations, the situation stabilizes again, and computation can carry on.

Comparable techniques based on force simulation have already been developed: 1- For the placement and routing problem on VLSI [SM91], they are called force directed placement, starting from random initial states, wires linking gates act as springs to move the gates over the VLSI surface. In our case, the

combination with a step-by-step development intuitively diminishes the plague of local convergence. Indeed, the adjustment needed at each step is hopefully sufficiently simple, that vertices should be directly attracted toward their new optimal position. There are fewer chances to be trapped by local sub-optimal basins of attraction. We do not yet have computer simulation of development to support this claim. It should be interpreted in the light of the examples of development presented in this article which use planar graphs, simple enough to be convincing. Since for complex optimization problem, an optimal sub part of a solution may no longer be optimal when the global solution is considered, it should be possible to program more complex non-planar graph converging to a suboptimal solution.

2- In the MaRS dataflow machine [CCC+89], task density has been used for load balancing, this is similar to a repulsive force that homogenise density. However the network used is all to all, giving a non scalable simple space where each pair of PE is at the same distance.

2 The blob machine

The blob virtual machine is a particular self-developing machine (SDM), whose instruction set is designed to satisfy the opposing requirements of efficiency and expressiveness (see page 6). The instructions are simple enough to run on arbitrary computing medium, and expressive enough to program non trivial applications.

In part 2.1, we explain how the blob machines self develop, and illustrate this with an example of an SDA. We use a simplified blob machine called binary blob machine, which captures the essential features and simplifies the presentation. We give the precise semantics of the instructions, and prove their confluent behavior. In part 2.2, we present the state of the art of the implementation.

2.1 The formal binary model

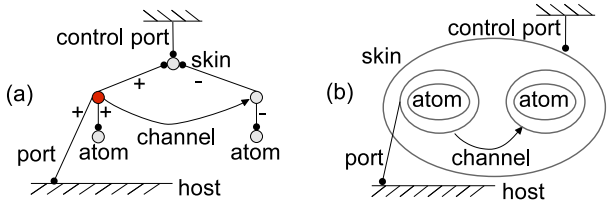


Figure 3: The two representations of a blob graph (a) network representation: arrows with round (resp. sharp) head represent vertical edge, (resp. channel) and their dynamic orientation. Static orientation of vertical edges is upward (b) topological representation, omitting vertical edge and automata.

2.1.1 Overview of the blob virtual machine

Blob graph: Since a blob machine is a SDM, a configuration of the blob machine is a graph — called *"blob graph"* — of nodes executing a Finite State Automaton

(FSA) with output actions that are the blob instructions. The FSA is called 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. At any time of the blob graph development, a spanning tree is always embedded in it, and defines a hierarchical ordering between all the nodes. A natural way to represent this hierarchy, is to draw nodes as if they were blob-like membranes, as shown in figure 3 (b). The blob at the root of the tree contains all the other blobs and is called the *"skin"*. The empty blobs at the leaves, are called *"atoms"*. An *inner blob* (resp. *outer blob*) of a blob, is a blob that is immediatly encapsulated (resp. encapsulating).

Topological and graph representation of blobs are like Dr Jekyll and Mr. Hyde, looking at one of the two, one tends to forget the other which is hidden. For example, the topological representation does not show that one automaton is associated to each blob. The network representation is a logical, exhaustive coding, used for programming. The topological representation is more "physical", it reflects the actual implementation on a computing medium, see figure 8. The topological representation recalls membrane systems called P-systems developed by Paun [Pau02]. The difference is that each blob is controlled by a simple FSA, whereas in Paun's system, each membrane hosts an unbounded number of molecules. The program is a finite set of rules modeling chemical interactions between molecules. Using atomic blobs to represent molecules, one can think of P-systems as a particular, "chemical flavoured" way to program blob systems which could exploit the parallelism inherent in chemical reactions, but it is not considered in this article.

There are three kinds of edges in the blob graph: *vertical edges*, *channels*, and *ports*. First, the vertical edges encode the blob hierarchy. They are oriented, so that a blob can distinguish between its unique outer blob, and its inner blobs. Vertical edges can represent blob graphs which are trees. This already enables the programing of non trivial algorithms such as the sorting algorithms presented in this article. However, some algorithms need non-hierarchical graphs, such as 2D-grids. In order to program those, we use *channels* which can connect arbitrary blobs and are represented as filaments on the computing medium, see figure 3 (b) and 8. Last, the blobs need to exchange input/output data with an external *host*. That host is modeled as a distinguished node whose behavior is externally controlled, 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 does not change during execution. It models the amount of parallelism available for the input and output. Like channels, port edges have to be represented as filaments on the computing medium.

Following the formalism of SDA, each blob is controlled by the same automaton but the automaton's state is distinct and stored locally on the node. Each blob automata can run in parallel, asynchronously. However, at a given time step, a blob can execute its automaton only if it satisfies two readiness predicates: the parallel readiness predicate and the confluence readiness predicate. The parallel readiness predicate is compulsory for parallelism, it provides mutual exclusion between adjacent blobs, both wanting to modify the edge connecting them. We use a dynamic orientation of the edges, which is distinct from the up/down static orientation, and which is defined for all edges, and not only vertical edges. Dynamic means that there exists an instruction called **flip** which can modify this orientation. Dynamic orientation is used to determine

that a blob owns an edge, if that edge is leaving the blob. The parallel readiness predicate states that a blob can modify only the edges that it owns. A blob is not ready if the next instruction to be performed needs to modify an edge that is not owned. A precise definition of "modify" is given in the next section. If it is not ready, the blob must wait that the neighbor blob, at the other end of the edges that should be owned, flips back those edges, or deletes them. This is the first condition. The second condition — the confluence readiness predicate — ensures confluence. It is more technical and is explained later.

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

- For communication, the polarity bit behaves as a shared register between a blob and its elements. The polarity of an edge is set from one extremity, using instruction **setp**, and tested from the other with instruction **testp**. Setting the polarity downward with instruction **setp down** $\pm$ is analoguous to a broadcast, since there can be arbitrary many inner blobs.

- Whenever a blob creates another child blob, they initially get the same state specified by the SDA transition function. However, the upper edge of the created blob (resp. creator blob) is negative (resp. positive) (see figure 5). Hence, by testing the polarity of the upper edges, creator and created blobs can execute their next transition to distinct states.

Furthermore, as we said, vertical edges also carry a static orientation: *up* (resp. *down*) refer to the outer (resp. inner) blob. Channel edges carry neither static orientation nor polarization. Port edges are treated like vertical edges: Communicating with (resp. moving) ports is like communicating with (resp. moving) inner blobs. In total, as shown in figure 5 upper left, there are 5 labels used to address the edges: *up*+, *up*−, *down*+, *down*−, *chan*.

The *initial configuration* is made of a single ready blob called the *ancestor blob*, connected to the port edges with polarity +, owned by the host (upper left of figure 4). One of the port called the control port, has a static orientation upward to the host, so that the ancestor, and later on, the skin, has also a unique up vertical edge, like every blob. All the other edges are linked to the host downward. The state of the ancestor is the initial state q_0 of the SDA.

Example of SDA: Figure 4 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 a development. The unique self developing instruction used is the **wrap** instruction that develops the blob graph vertically in a degenerated tree representing a stack. The SDA implements the "push" operation in an ordered stack. Values flow from the top down to some right location that makes the stack ordered. This stack works for integer values in general by binary encoding them as a sequence of polarities + or −. For sake of simplicity we present it for 0(−) and 1(+) values. Each blob evolves according to the automaton drawn in figure 4. The upper part of the 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 which is the initial stack bottom state. The host sends the value "0" by setting the polarity of the control port to − and give port ownership to b by the **flip down** instruction. Then b can receive this value using **testp up** to test this polarity and does a **wrap**. The inner blob becomes the new stack bottom, while the outer blob is a dynamically created stack element holding

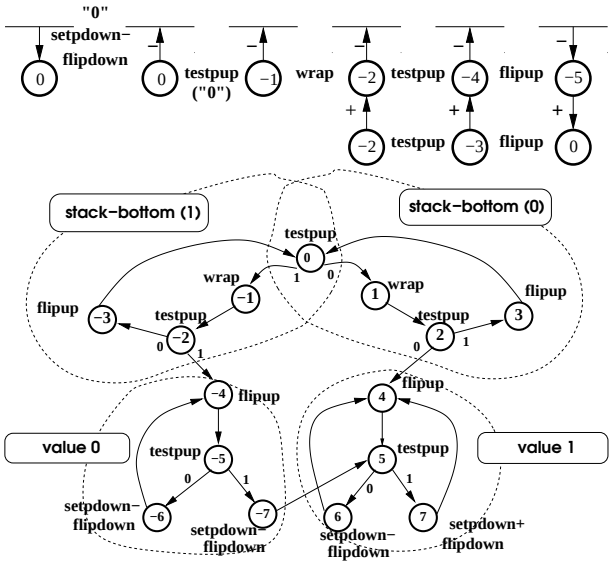


Figure 4: An example of basic self developing automaton (SDA) implementing a priority queue. For sake of simplicity values are only 0 (-) or 1 (+). The automaton has 3 parts, for the stack bottom, and the stack elements with value 0 and value 1.

Static			
Codop	**Semantics**	**modified**	**tested**
Flip dir	Changes orientation of edges dir	dir±	
Setp dir q	Polarizes edge dir	dir $\overline{q}$	
Testp dir	Tests presence of + edges		dir+
Move	Exchanges parent blob and channel	up±,chan	
Self-developing			
Codop	**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	idem	up±, chan, down+	down+

Table 1: The blob instruction set: 4 static instructions modifying edge labels, 4 developing instructions adding or deleting vertice and edges. Operand "dir" takes values in $\{up, down\}$; and operand "q" is a polarization + or -.

value "0", which corresponds to states -4 to -7. The symmetric states 4 to 7 correspond to a stack element holding value "1". To push a "1", the host makes a positive polarization instead of a negative one. When a stack element holding value v receives a pushed value v', it propagates downward $min(v, v')$ and stores $max(v, v')$. Values get 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** instruction synchronize the receivers with the senders. It changes the direction of the up/down edges. A pipelined parallelism is obtained, since a new value can be pushed every five clock cycles.

2.1.2 Semantics of the 8 binary blob instructions

The binary blob machine is defined by the 8 instructions listed in table 1, which develop a blob system starting from the initial configuration. Figure 5 com-

pletely describes the semantics of the 8 instructions, using the graph representation. Each instruction is formally a graph-rewriting rule. But those rules are not global rules acting on the whole blob graph. More precisely, the predicate triggering rewriting, as well as the rewriting itself can be computed locally by each blob, because only the edges adjacent to the blob are manipulated.

Triggering predicate. The possible left contexts are multisets of oriented labels carried by edges connected to the blob. The triggering predicate puts some restriction on the multiplicity of some labels. It is the conjunction of the two *readiness predicates*, which impose absence of some labels, plus an optional instruction-specific predicate used in instruction **merge** and **testp**: 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 predicated by $\mid down+ \mid = 0$ and $\mid down+ \mid > 0$. The two rules are conveniently summarized in one, by making the input to the SDA, a parameter computed from the context. The label *down+* is said to be tested by **testp down** and **merge down**. To prevent dead locking, if an instruction is ready, but the instruction-specific predicate is not verified, the instruction is considered to be **nop**, i.e. the neutral instruction that has no effect. This can happen only for instruction **merge down**.

Rewriting process The left member of the rewriting rule modeling an instruction represents the node n being rewritten in the center of a grey disk, and edges leaving n. There is at most one edge for each possible 5 labels, all represented in the upper left figure. An instruction does not necessarily use all the 5 labels. The right member contains a blob graph with a distinctive node called the *root* identified with n, which means two things: firstly, n's rewriting needs to create other nodes only if the right member contains other nodes than the root; secondly, n's edges carrying labels not present 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 l-location, that represents all the neighbors of b, connected with an edge labeled l, called l-neighbor. The l-location is used in the right member to specify how to establish 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 a node n (resp. another l_2-location), then such an edge labeled l_1 must be created to connect each l_0-neighbor to n (resp. to each l_2-neighbor). Potentially, the rewritting could add arbitrary many connections. However, a closer look shows that an instruction never add more than 2 edges: 1- Only the outgoing channel and the upper edge are duplicated, (by **divide** and **new chan**). But the number of upper edge (resp. outgoing channel) is always one (resp. smaller than one), This is true for the initial blob graph and is maintained by each instruction. 2- When direct connection between l-neighbors are established (by **mrg**), one of the two is *up*, or *down+* with multiplicity constrained to be 1.

Modified labels, used labels. From figure 5 we can state precisely the meaning of "modified label" used to define the parallel readiness condition: a label l is modified, if the context of l-neighbor changes after the instruction. This includes modifying the label of the edges leading to them, removing those edge, or adding new edges. The label represented in the left member, are call *used label*, they may include labels that are not modified. For example, with instruction **divide** the polarity of inner blobs is used to determine to which node they get connected. The *down+* and *down−* edges are used, but not modified. Also, by convention,

tested edges are considered to be used, i.e. they must appear in the left context. Table 1 indicates what are the edges modified and tested by each instruction.

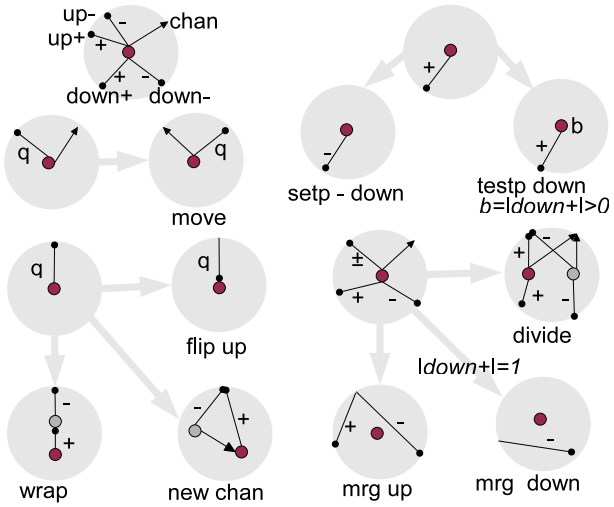


Figure 5: The complete semantics of the 8 binary blob instructions. Notation are the same as fig. 3. The root of the right member is represented using a darker gray. The instruction **setp**, **testp**, **flip** are symmetric with respect to *up* and *down* direction, so only one direction is represented. **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 well introduced. We comment further instructions that create, delete or move blobs, whose semantics is more obvious, using the topological representation, in figure 6.

Blob creation: **wrap** encapsulates a new blob, which gets all the inner blobs. It develops the blob hierarchy vertically, **divide** produces a brother blob toward which it moves all the negative elements. It develops the blob hierarchy horizontally; last, **new_chan** also creates a brother blob, but without elements, and linked to the creator via a channel edge; We will use an additional instruction **wrap-** which move the negative elements in the newly created encapsulated blob. It can be defined as a macro-instruction using a combination of **wrap**, **divide** and **mrg up**. In **divide** and **new_chan** the upper edge is duplicated. But port edge cannot duplicate, this implies that the skin, having always an upper edge which 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.

Blob deletion: Blob deletion occurs when all 5 labels are used, and the root is isolated. Being unconnected to the blob graph, it is automatically deleted. The instruction **mrg up** merges the blob with its outer blob, giving it all its inner blobs, but deleting its channel, if it has one. This amounts to just suppressing the membrane and the channel. The instruction **mrg down** is not symmetric to **mrg up**. The executing blob must have a single positive element whose membrane is noted $m+$. It moves all the other negative inner blob inside $m+$ and then deletes its membrane m and channel. To avoid moving blobs, we

delete $m+$ instead of m, as shown in figure 6. We need to replace the automaton controlling m by the automaton which was controlling $m+$, which the figure does not show. 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 blob through channels. Notice that all self developing instructions preserve the following invariant called 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, the blob b_0 may choose to move to become an element of b_1 using the instruction **move**. In the graph representation, **move** simply exchanges the labels of the unique owned channel if it exists, with the unique outer blob. The topological representation in figure 6 makes clear why this is called a movement, because to change outer blob means really to move within the membrane of the new outer blob. Flipping a channel is not allowed, this would break the channel invariant. This is why communication through channels is done exclusively by blob movement.

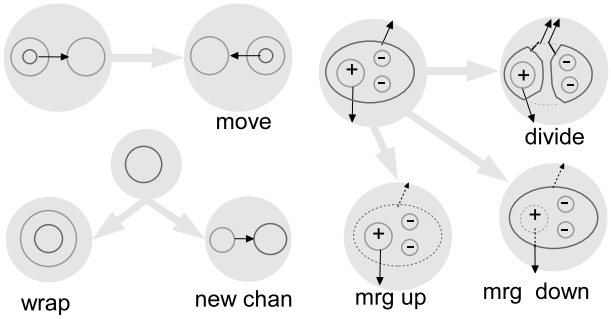


Figure 6: The developing instructions of figure 5, plus move, in the — more intuitive — topological representation

2.1.3 Confluent Asynchronous Parallel execution

Asynchronism suppresses the necessity of a global clock, which is important for scalability. In an asynchronous execution, only a subset (randomly selected) of all the ready nodes, execute their instructions simultaneously. We would like to have a *confluence* property ensuring that the execution order does not matter, and the system will always converge to the same configuration. For example, a data-flow network has an asynchronous execution but is confluent. It is difficult to convince oneself of the confluence of the blob model because one does not have a clear picture of what can be programmed with the 8 instructions until one has actually tried to program something, and the example given in 4 may not be sufficient. Consider the classic example of a non-deterministic merge of two input streams to an output stream which is not confluent: It is not possible to program such a gate in a blob network: A blob b_0 can receive inputs from two different sources, if it uses two inner blobs b_1 and b_2 which can themselves receive data from arbitrary other points of the network, using channels. The inner blobs b_1 and b_2 can each communicate one bit to b_0 using instruction **Setp up**. First of all, b_0 cannot receive anything before both b_1 and b_2 have flipped back the edge to their outer blob. Secondly, using instruction **Testp down**, b_0

does not read the two bits, but a logical OR of the bits. In order to receive both bits distinctively, b_1 and b_2 must agree to send their bit each one in turn, while the other sends a bit one. Those two points imply a bit-level synchronization which prevent non-deterministic merge. Data-flow graphs use non-deterministic merges while still remaining confluent, in the particular specific case where one of b_1 or b_2 will not receive any input stream at all. One wishes not to block the other to forward its input stream. In blob computing, additional non-confluent instructions can be added to increase expressivness, and ease the programming of such gates. Alternatively, to stick with the 8 instructions, one can forward to one of the inner blob, the information that it will not receive input streams, and let it send a flow of ones of appropriate length.

We analyse the confluence in a systematic way, that could be conducted for other self developing instruction set than just the blob instruction set: 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 connecting to n_0. We say that e_0 produces e_1. The figure 7 represents this production relation, which is defined from n_0's point of view. It uses oriented labels, which are pairs of a label and an orientation coded as 0 for incoming, and 1 for outgoing. For each oriented labels (l_0, d_0), it 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 if its context does not contain oriented labels that can produce labels used by i. To be ready, a node must simply check the absence of such labels 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.

We prove that the confluence predicate implies commutativity, which is stronger than confluence: commutativity is true if the result of executing two arbitrary nodes n_0 and n_1 which are simultaneously ready, does not depend on their relative timing. If n_0 and n_1 are not neighbors it is trivial. Otherwise, consider, as before, 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. Those edges are not used by i_0, otherwise it would contradict the confluence readiness predicate for i_0. Being not used implies that the label of $e_1 \dots e_k$ are: (1) not tested by i_0, (2) not appearing in the left member of i_0. (1) implies that the instruction executed by n_0 after i_1's execution is still i_0. (2) implies that when i_0 is executed, $e_1 \dots e_k$ remains connected to n_0, which also happens 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's execution.

From figure 7. We distinguish two extra readiness confluence predicates in the case of blob instructions: 1- Because $(up\pm, 0)$ generates $(up\pm, 1)$ and $(down\pm, 0)$ generates $(down\pm, 1)$, an instruction which uses *up* (resp. *down*) should own all the *up* (resp all the *down*) edges. If the use is a modification, the condition obtained was already specified by the parallel readiness condition. If the use is a test, i.e for instruction **testp** *dir*, confluence readiness states that **testp** *dir* must own edges *dir*. Indeed, this makes sure that the value will not be changed by the neighbor while reading. The automaton in figure 4 illustrates a direct consequence of this: to communicate bits, the edges must be

flipped back and forth by sender (state $q_{\pm 6}$,$q_{\pm 7}$) and receiver (state $q_{\pm 3}$, $q_{\pm 4}$) for each bit exchanged. 2- Because $(up\pm, 0)$ and $(chan, 0)$, generates $(down\pm, 1)$ an instruction which uses *down* should not have incoming channels, nor any incoming upper edges.

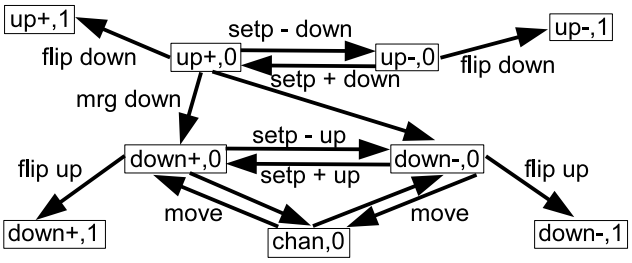


Figure 7: The production relation between edge labels. A label l_0 can produce another label l_1 if there is a path leading from l_0 to l_1. The arrows between nodes are labeled by the instruction responsible for the production. Some instructions are not present because they do not add new edges to this graph.

2.2 Implementation of the run time system

For the moment we have mainly considered only 2D or 3D Cellular Automata (CA) as target computing medium. The specific features of more realistic hardwares such as long lines in FPGA or memory in coarser grain hardware can be exploited to lead to a better implementation. However, if we consider calculating complexity for arbitrary big problem size, it is the spatial locality constraints, that will dictate the asymptotic behavior. At this stage, we are interested to derive this complexity, rather than study how to speed up by a constant factor by exploiting technologically dependant features. The 2D CA is a simple framework exhibiting these spatial constraints. We call Processing Element (PE) the automata of the 2D CA to avoid confusion with the SDA automata. Because PEs have finite small state, we can assume that the state of each node of the blob graph is stored on a distinct PE. In the basic blob machine, each PE runs the same automaton but on a different state. Alternatively, a more universal representation for running different automata is to use two levels for PE's 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 circuitry with flip flop registers and look up table, or more traditional micro programmed controller. Fine grain implementations such as CAs are also interesting for their own sake: 1- With finer grain, blob's membrane have a larger diameter counted in number of PEs, and this augments the parallelism of blob movement over hardware (corresponding to dynamic migration of code and data). The increased parallelism is double: pipelined (resp. data parallel) in the direction parallel to (resp. perpendicular to) the blob movement. 2- The problem of the finest possible granularity in poses a nice challenge: what is the simplest hardware building block that can be combined in *arbitrary* number, and provide a computing medium able to simulate *arbitrary* blob machine? The problem is the finite memory of this building block, which will force the partition of the SDA itself into several ele-

mentary SDA of fixed size whose state fit on a PE. This problem is addressed in [GLRT04].

2.2.1 Mapping of blob graph on a 2D CA

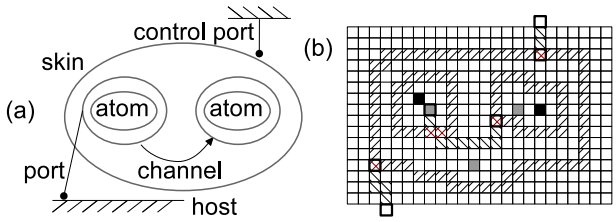


Figure 8: (a) a blob graph (b) its fine grain mapping. Atomic blob are black PEs, gray PEs store the automaton of a non atomic blob. Membranes are hatched PE, Channels and port's filament are also hatched, in a perpendicular directions. The PE at a filament's extremity has thick lines.

The blob graph as a whole is naturally mapped on a 2D CA by discretizing its topological representation: the 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 we said 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 in two connected components: the inside and the outside. The inside is called the *hardware support* 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 distinguished from other particles representing atoms of b. Let $N(b,t)$ be the number of automata contained in b, we have $|H(b,t)| > N(b,t)$. 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 to $H(b_1,t)$. 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 minimal 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 will be $\mathcal{O}(n)$. If the blob is round, the minimal 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 $H(b,t)$. This space saving is a key advantages of membranes, which justifies their use: membranes allow communication along vertical edges, without having to represent explicitly those vertical edges, by using pure flooding within $H(b,t)$. This means that to reach n nested membranes, we need only $n^{1/d}$ steps in d dimensions.

Hardware freedom necessitates allowing changing over time the hardware supports of blobs, automata, and channels, i.e. the membranes,particles and fil-

aments. This is why $H(b,t)$,$A(b,t)$ and $H(c,t)$ all depend on the time t. These movements must maintain the topological properties giving the functionalities. These properties, as just stated, include, in addition to connectedness, separating inside from outside for membranes and maintaining adjacency for filaments. In [GM02] we show that separation and adjacency can also be expressed using connectedness of specific sets of PEs defined by appropriate predicates over being membrane or filaments. We introduce the "*blob rule*" that specifies 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 is at the origin of the term *"blob"*. The blob rule not only establishes blobs, but also channels. In [GM02] we also prove that this local rule can be defined on arbitrary architectures, and asynchronously. This opens up the possibility to map blob graphs on irregular architectures, embodied by the amorphous scalable model of architectures.

2.2.2 The blob "dDcomplexity" model

Definition of the dDcomplexity The time taken to iterate an SDA node b includes the time needed to update the automaton state, and the time $T(i,b)$ to execute its instruction i. State updating is done locally on a PE, and needs one unit of time. On the other hand, most blob instructions need to modify the hardware support at time t: $H(b) = H(b,t)$ which is distributed in space. In general i needs to reach all the PEs of it hardware support under is control, let $T(b)$ be the time needed for that, $T(i,b) < T(b)$ and $T(b) = \mathcal{O}(D(H(b)))$ where the diameter $D(H(b))$ is defined as the maximum distance between any two PEs of $H(b)$, for the distance induced by $H(b)$ which is the hop count to go from one PE of $H(b)$ to another, while remaining in $H(b)$. Some instructions also use a channel c, and its associated filament at time t: $H(c) = H(c,t)$, in which case $T(b) = \mathcal{O}(D(H(b)) + D(H(c)))$. Let d be the dimension of the computing medium, $d = 2$ or $d = 3$. The "dDcomplexity" model is a set of three constraints that should be met by the implementation: *(i)* the density of particles is homogeneous throughout the skin, equal to α, as a result, $|H(b)| = \alpha^{-1}(N(b))$ *(ii)* $H(b)$ is always close to the disk or sphere: $D(H(b)) = \mathcal{O}(|H(b)|^{1/d})$ *(iii)* We target the best possible performance: $T(i) = \mathcal{O}(D(H(b)) + D(H(c)))$ For blob graph with channels of bounded length, by combining *(i)* with *(ii)* and *(iii)* using simple substitution, we obtain *(iv)* $T(i) = \mathcal{O}(N(b)^{1/d})$ which defines the time dDcomplexity $T(i)$ of a blob instruction independently from the mapping. This allows calculating the time complexity of an SDA execution knowing only the dimensionality of the computing medium. We will apply it in section 3 for a variety of SDA.

The problem of stalls Consider a second order blob b containing first order blobs $b_1, \ldots b_n$, all dividing simultaneously, thus doubling the density within b. The constraint *(iii)* takes into account the time needed to move particles within b_i, for b_i's division. It does not consider the time needed to inflate b's membrane, and relocate b_i's particles and b_i's membrane within b, in order to re-establish uniform density of α through b. This homogenization involves the whole blob system, because b itself needs to push on its neighboring blobs until the skin membrane is reached. Furthermore, if the blob b_i needs to continue

duplication several times, the rate of duplication will be greater than the speed of homogenization, and development can be stalled due to shortage of empty PEs: a particle that needs to duplicate, and has no empty PE in its neighborhood, is stalled. The complexity *(iv)* does not take into account the waiting time implied by stalls. Stalls can happen especially in the initial stage of development which start from a single ancestor blob and creates numerous 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 We consider algorithms including only one growing phase, followed by a stable phase and a diminishing phase. Furthermore, for any blob b that is created, we can dynamically evaluate the maximal size $N_{max}(b)$ that it will reach during its lifetime. In the cases we will consider, a variable of the program directly indicates how many sub-blobs will be contained in it. This is a weak constraint that is usually checked and does not mean that the algorithm has a static task graph (*e.g.* the quicksort).

In this restricted framework, a simple strategy for development enables correct and good complexity estimates: Upon creation, a blob that will later grow, will execute a specific instruction called **allocate** $N_{max}(b)$ that generates a repulsive force emanating from $A(b)$ that will inflate b's membrane so that it becomes large enough to contain the expected future $N_{max}(b)$ particles. The density of b will temporary fall lower than α so we need to modify constraint *(i)*, and *(iv)* by replacing $N(b)$ by the "virtual size" $N_{max}(b)$. This method must be applied first to the ancestor, whose very first instruction executed is **allocate** N, where N is the maximum number of blobs developed during the whole execution. This will inflate 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 instruction **allocate** 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 we can tolerate some bounded variability in density, blobs whose size is quasi preserved, i.e. varies between n and $2n$, do not need to allocate space. The following development called *uniformly dividing development* is naturally *quasi size-preserving*, without any use of **allocate**: a first order blob with initially n atoms, divides iteratively into two blobs sharing equally the atoms, until they contain a single atom. In terms of sets, a set is divided in two subsets iteratively, until singletons are obtained. After each division, a different processing can be applied to each subset, which is actually a standard method to process the elements of a set in a *non-uniform* way. We will use this method, and program uniformly dividing development 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 the divisions takes place has its sizes varying between n and $2n$ as is clearly shown in the figure of program 2. The outer blob's size 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 taken for a uniformly dividing development will be $\sum_{1 \leq i \leq n}(D(H(b))/(q^i))$ which is

$\mathcal{O}(D(H(b)))$. In other words, a uniformly dividing development within a blob b, takes a time $\mathcal{O}(T(b))$, i.e. the same time b needs to execute a single instruction. Uniformly dividing developments recalls the "nested parallelism" exploited on sequence in the NESL programming language introduced by Blelloch [Ble95], who also proposes a simple framework to evaluate the parallel complexity of its programs.

2.2.3 State of the art of the implementation

How can we implement on a particular computing medium a run time system checking the 3 constraints? Constraints *(i)* and *(ii)* corresponds to a set of background rule going on permanently through the CA. We have proposed such a rule set in [GM04] consisting of well known, very simple, CA rules discretizing simple physics, explained in [TM87]. The CA rule HPP models particles as a gas under pressure. This does not put an exact lower bound on it, but makes it highly improbable the creation of empty spots, so *(i)* will be true on average. The membrane is implemented as an elastic bubble on which the gas exerts a pressure. A constant outside pressure exerted on the skin will determine the average density α. The CA rule called voting rule (1 bit of state) ensures *(ii)* by modeling surface tension. Constraint *(iii)* must be checked instruction by instruction. Some instructions are natural to implement: communication is done by flooding inside membrane or through filaments; **Mrg** is done by suppressing membranes and filaments, **Wrap** splits the membrane as if it was a zipper. **Move** for an atomic blob makes a particle travel through the filament. The difficult instructions are **divide** and some new instructions, not present in the binary blob machine, that improve communication. We present their implementation when the executing blob b is first order, i.e. contains only atoms, $a_1, \ldots a_n$. The membrane contain $n+1$ particles: the atom's particle and b's master particle $A(b)$. Our results on communication can easily be extended to higher order blobs, whereas division of higher order blobs needs to move membranes through hardware which is not yet solved. The algorithms presented in this article uses only division of first order blobs.

Communication instructions We will use scalar instruction that can exploit the 2D topological representation to improve the performance of communications. First, when communicating upward the instruction, **Send** v_i done by an atom a_i can communicate a scalar value v_i to b instead of just a one bit polarity. In order to receive the n values $v_1, \ldots v_n$, b must execute an associative and commutative operator such as **sum** which retrieves $v_1 + \ldots + v_n$. Reduction is done by having a connected spot c, initially equal to $H(b)$, and let this spot shrink, while maintaining its connectedness using the blob rule of [GM02]. **Send** v_i generates a temporary particles p_{v_i}, containing v_i. Those temporary particles are collected by c, i.e. they stay within c as c shrinks. Eventually they will 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 will eventually have a hardware support of only one PE p_{end}, containing only one temporary particle, with the result of the reduction $r = v_1 + \ldots + v_n$. Finally, when c goes past $A(b)$, we let c also collect $A(b)$ so that $A(b)$ stays within c. $A(b)$ will also be hosted by p_{end}. $A(b)$ can test termination, and read the 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 needed to maintain connectedness of

c. The whole shrinking process, illustrated in fig. 9 takes a time proportional to the diameter of b, which fulfills constraint *(iii)*. This also an optimal time, in general, for a reduction on a 2D grid or 3D grid of PEs.

Secondly, when communicating downward, instruction **Broadcast v** broadcasts a scalar v instead of just a one bit polarity. Furthermore, instead of flooding a signal inside the membrane **Broadcast v** sends waves separated by constant intervals of space. This allows for a list of scalar to be broadcast with a constant throughput and avoid to wait the latency of the whole diameter implied by flipping back and forth. The instruction for receiving is called **Rec**.

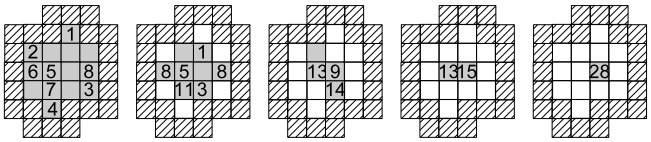


Figure 9: Four steps of the shrinking blob applying the reduction operator "sum".

Mathematical proof of division for rectangular blobs with static membrane. Consider a 2D CA, and a vertical rectangular membrane. Vertical division needs a mechanism to move the positive particles down and the negative particles up so that a new horizontal wall can separate them. This separation can be achieved by piling up positive downward (resp. negative upward) into two heaps with a 45 degrees slope. Piling up is sufficient if the particle density is less than 1/4. We have presented with Tromp [GT00] a simple piling up CA rule presented in figure 10. We give a proof (3 pages!) that the pile up time is less than three times the rectangle length, which fulfills constraint *(iii)*. This proof does not consider hardware free membranes. However, it is sufficient to justify our result on complexity, since the algorithms we will describe use exclusively uniformly dividing development, where hardware free membrane are not necessary, and rectangular membrane are sufficient.

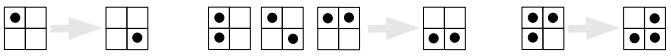


Figure 10: Tabulation of a 2D CA block rule piling up particles by moving them downward. It uses 1 bit of state representing particle occupation. The rules not represented are either quiescent, or can be obtained by horizontal symmetry.

Measurements of division with round blobs and hardware free membrane. In the case where the blob b has an hardware free round membrane, we have no proof but implementation results in [GM04] using the gas models for constraint *(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, using the piling up rule. This automatically creates a retraction in the equatorial zone. During this process, we use a centering shrinking blob that pushes the master particles toward the center of the dividing blob, and collects the information whether there are only plus and minus on each side. When $A(b)$ touches the two retracting walls, it waits until all the plus are on one side, and the minus on the other side; then $A(b)$ does the final cut and duplicate, which produces two blobs; see figure 11 (d). We tested division of blobs of up to 1600

atoms and measured that the time latency for dividing a blob of n particles, is proportional to $n^{1/2}$ which directly correspond to the dDcomplexity *(iv)*.

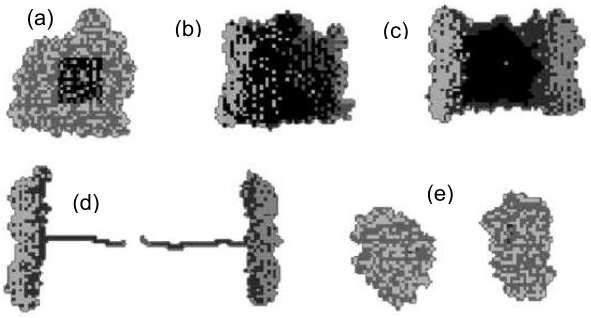


Figure 11: Snapshot of the blob division (a) start (b) regrouping the plus and the minus (c) wall retraction (d) final cut (e) end. The video is available on [sit].

3 Programming efficient blobs development

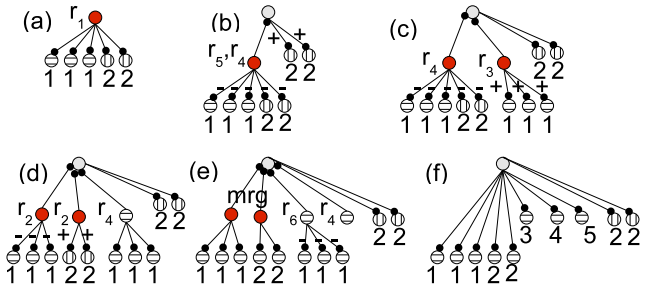
Since there is no order between the inner blobs of a given blob, a blob is functionally a multiset of its inner blobs. A multiset, also called "bag" is a set where elements can be repeated. It is a free monoidal data structure. This property has been exploited to describe parallelism in a concise way. In [BNTW95], Tanen defines algebraic high-level parallel operations such as reduction. In Haskell and SETL [SDSD86], language constructs allow bags to be defined in comprehension. The NESL programming language introduced by Blelloch [Ble95] implements nested parallelism on sequences which are collections, similar to bags. He manipulates sequence to implement divide an conquer algorithms, in a style very similar to ours. The work of Lisper [HL93] uses sets (data fields) to describe a generic data parallelism in a functional way. In [BCD95] the efficient parallelization of set based language is also considered. We are developing a blob language that also exploits the properties of bags. We present language constructs for describing SDA in a higher level that just finite state automaton. The description is compact enough so that we can present elaborate blob algorithms with optimal dDcomplexity. It is close enough to blob graphs, so that the programmer has 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, we use the OCAML [Ler05] syntax that enables automatic type inference. The type of the functions is inferred and given in italic, as is usually done. This is an ongoing work: the compiler for the blob language is not yet implemented, and the blob language itself is not yet stabilized.

The finite state constraint. An SDA in a finite state automaton therefore a blob can host only scalar variables, but not collections of data such as arrays or lists. Such collections must be distributed on different blobs, and must be encoded as bag hierarchies. So OCAML's arrays and lists are proscribed (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 bounded stack. With par-

allel languages, it is generally accepted that some constraints should be met in order to produce feasible code. The finite state constraint has a simple formulation and can be easily understood by the programmer. He is forced to use bags extensively. In order to obtain good performance, the program should generate uniformly dividing development, which corresponds to iterative bag division.

Code size. In a first approach we can assume that each blob has the whole program contained in it and can therefore access the code to be executed. Since the program is finite, we obtain indeed an FSA. By using the oriented object style, 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 of the scope of this article.

3.1 Basic manipulation of bag (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 start executing line 2, already in (b). The bag id X (resp. Y) is represented by horizontal (resp. vertical) hatching. Master nodes are labeled by the compilation rule which produce the code that they execute in the next step. Slave nodes are labeled by the integer element hosted. The round arrow head indicates the 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. The program 1 illustrates the development obtained when a blob b builds a new bag by making 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. This bag address is allocated by the primitive `newbag`. In the example, $X = \{1, 1, 1\}$ and $Y = \{2, 2\}$ thus b contains 3 atoms with $id = X$ and $elt = 1$ and 2 atoms with $id = Y$ and $elt = 2$. The blob b will send commands to its elt-blobs by broadcasting down a list of scalars $[S, f, p_1, .., p_k, end]$ including a bag address S, a function name f,

and scalar parameters $p_1, .., p_k$; it also does a **flip 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 all cases, they finish by doing a **flip up** to give back the 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;;let set_id x = id:=x ;;`[1] which sends up the element to b and sets 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 '*' matching all ids. The SDA program part executed by b is compiled using the following six rules[2]

```
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;
```

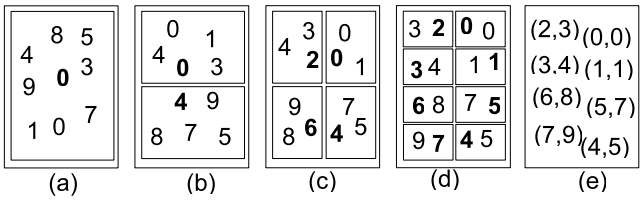
Rules r_1 compile the bag assignment `S<-exp1;exp2`. The code produced by `route(exp1,exp2)` routes inner blobs by analyzing liveness of bag variables present in exp_1 and exp_2. It commands a negative polarization (resp. positive polarization, duplication, deletion) to inner blobs representing bags which are live only in exp_1 (resp. 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 we assume it remains live 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 a blob b_1 that can evaluate the bag expression exp_1 in parallel with b which can continue evaluating exp_2. b_1 is called the *bag-ancestor*. The bag expression exp_1 is not compiled with `code` but with `code2` which has an additional second argument S carrying the address of the bag being computed. This parameter S is used in rule r_2 and r_3 in order 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 implements reduction. When a bag variable is no longer live, the inner blob representing it must be deleted. This

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

[2] 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 algorithm of quick sort

garbage collection is not completely described in the presented rules. For example, in the figure (e) of program 1, the blob executing r_6 must delete its inner blob, as it is shown in figure (f). One can define bag hierarchy such as `S<-{(1, {(2, {(3, {})})})}`, of type `blob-list = (int*blob-list) bag`. However, circular structures are not possible because they would break the blob hierarchy. An assignment like `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: Program of the quicksort. A development for this algorithm is shown in fig. 2, in network representation. Here, we show 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 index i in bold font, and the slave by their element being sorted.

The for-in construct Basic manipulation of bags authorizes only a fixed catalog of functions that elt-blobs can perform. The for-in construct `for s in S do exp done` allows a master blob b to remotely trigger an arbitrary computation 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 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 it is the case for the function `card` 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 done in 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 sum or forming a new bag. In the first case, one uses the addition to reduce the values, in the second case, one uses a kind of merge function to aggreate the values. Those are two instances of a bag homomorphism. However, a bag is a particular value: it is a collection that can take any large space, and needs therefore 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, the for-in's action is to have a side-effects which can be of two kinds: 1- modify the elements, as in function `Ainvert` (next paragraph) or generate communication as in function `Input` of program 4. Usually, accepting side-effects in functions that are asynchronously remotely called leads to a form of non-determinism that makes it very difficult to have confluence. However this is true only when side-effects apply to some global 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 a remote call f in the elt-blob of S. For example, the function `card` 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 call to quicksort `A <- qsort(X,n,i )` includes the cardinal n of the bag X of values to sort (to avoid recomputing it twice), and the starting index i to give to the smallest element. It returns an 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, arrays 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 with equal size (up to one elements if the size was 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 the elements of X smaller than the pivot. The result is defined as the union of bags produced by a recursive call done on Y and X, with new parameters for the starting index and cardinals. Recursive bag division is continued 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 forbids one of the calls to be terminally recursive. The blob framework allows a 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 an additional treatment to concatenate two sub-arrays for example. Being terminal, the recursion can be compiled with a bounded stack (independently from the data). This secures the finite state constraint. In fact, the stack of the blob doing recursion does not even store the element to be sorted, but only the three parameters X, nX, i of quicksort which are scalars.

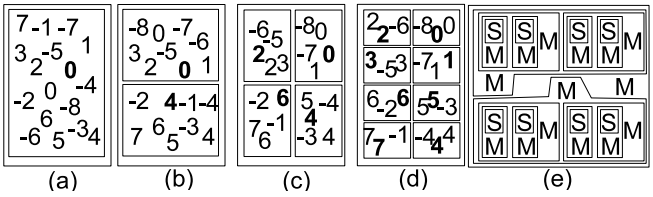
dDcomplexity If we are lucky, the pivot always splits X into two bags of equal size. In this case, one obtains a uniformly dividing development, and an optimal time complexity of $\mathcal{O}(n^{1/d})$ and space complexity of $\mathcal{O}(n)$. In [GLRT04] is presented an emulation of the blob machine, using the dDcomplexity. We measured the average time and space needed on a 2D grid, when elements to be sorted are randomly chosen, and found the same time complexity $\mathcal{O}(n^{1/2})$.

Technical notes about functions on bags: 1- A bag X can be passed as a variable to a 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 function `Adivide`'s side effect is to consume the lower half of X, whereas `qsort` has no side effect on the bag being sorted. Passing a bag as variable is also useful for sparing bag duplication, which is why it is used in function `card`. 2- Setting the id of bag parameters. A function f can also return a bag. In the assignment `X<-qsort(...)` it is the responsibility of the caller to 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 combine bags (data parallelism)

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 subject of data parallellism is to combine in parallel values from different arrays. For example, a computation such as vector sum, or scalar product needs to combine (x_i, i) with $(y_i, i), i = 1 \ldots n$. This is usually specified with sequential loops for accessing X and Y's values. In a grid of d dimensions the diameter of a blob array with n elements is $n^{1/d}$ hence the time needed to read a value will be also $n^{1/d}$. Such an expensive read precludes the use of sequential loops. The alternative solution is to use an iterative division similar to the one used for quicksort by using the following step: 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 function `Adivide`, 3- let b_1 divide itself: the first (resp. second) offspring contains the first (resp. second) half of X and Y. 4- Let both offsprings recursively divide in this way, until the cardinal of the remaining local X's parts becomes one. 5- This will generate n offsprings, $b_1 \ldots b_n$, where b_i contains (x_i, i) and (y_i, i) and can combine them. The iterative division is uniformly dividing and has 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 (resp.




```
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: Program for the bitonic sort. (a)-(d)illustrate the 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. Those 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 8 integer gets laid out. The letter S stands for bsort, and M for bmerge.

y) refers to (x_i, i) (resp. (y_i, i)). The "sum of vector" 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 single range, modifying the range index x or y will result in modifying X (resp. 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. We must define what happens if X and Y do not have the same number of elements: 1- If X has less elements that Y, then when the recursive division terminates, each of the b_i contains one element of X called x but still 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 that Y then y takes a predefined value called $\perp$ meaning "undefined". We adopt the conventions that $x > \perp$ is always true, and $\{\perp\} = \{\}$. These conventions work nicely in the case of `bitonic merge`: they allow the sorting of arrays of arbitrary size and not only sizes which are exact powers of two.

The bitonic sort algorithm. The 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: one can divide it into two halves, $[0, n)$ and $[n, 2n)$, such that 1- each half is a bitonic sequence, and 2- every element in half $[0, n)$ is less than or equal to each element in $[n, 2n)$. One simply compares elements in the corresponding positions in the two halves and exchanges them if they are out of order, the sequential loop is: `for (i=0;i<n;i++) { if (get(i)>get(i+n)) exchange(i,i+n); }`. The for-in of `bitonic merge` does exactly this processing. The `bitonic sort` works by using a double DVC mechanism 1- It sorts the lower half into ascending order and the upper half into descending order. This gives us a bitonic sequence. 2- It bitonic-merges the sequence, which gives us a bitonic sequence in each half and all the larger elements in the upper half. 3- It recursively bitonic-merges each half until all the elements are sorted. Bitonic merge is a terminal recursive function, but bitonic sort is not. However, as shown in the figure (e) of program 3, the two recursive calls are remotely called on two inner blobs, hence the stack's size remains constant and the finite state constraint is checked.

dDcomplexity Unlike quicksort, bitonic sort and bitonic merge of n elements always exactly divide the problem size in two. They both have a time complexity $t(n) = \mathcal{O}(n^{1/d})$ and a space complexity of $\mathcal{O}(n)$. Proof can be done by recurrence on n for bitonic merge, let the hypothesis of recurrence be $H(n)$: $t(n) < K_1 * (n)^{1/d}$ for some constant K_1. Consider the hypothesis true for $k < n$ and prove $H(n)$. The bitonic merge of n value involves two parallel recursive calls to merge $n/2$ values. From $H(n/2)$, they take a time, $t_1 = t(n/2) < K_1 * (n/2)^{1/d}$. The rest of the computation involves a uniformly dividing development. This takes a time $t_2 < K_2 * n^{1/d}$ for some constant K_2. By choosing K_1 big enough such that the inequality $K_2 + K_1 * (1/2)^{1/d} < K_1$ holds, we obtain $t(n) = t_1 + t_2 < K_1 * (n)^{1/d}$. The same reasoning can be repeated for bitonic sort. $\mathcal{O}(n^{1/d})$ is an optimal VLSI complexity as proved in [Tho83] for 2D grids and [Lei92] for 3D grids. There does already exist a sorting algorithm on a 2D grid reaching the optimal complexity [Lei92], but it is much more complex than the one presented here: for example, it needs to tile a 2D grid of PEs, with tile length of $n^{1/4}$. In [Lei92], is also shown another algorithm, optimal for 3D grids. What is elegant here, is that the same simple algorithm is optimal both for 2D grids and 3D grids.

3.4 Developing non hierarchical network by using channels (pipelined parallelism)

The channel master An arbitrary graph is not a pure hierarchical structure and cannot be coded using only blobs: a graph link is represented by using a channel c. This 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 extremity and the blob at the other end of the channel c, the target extremity. 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 will consequently become the source extremity of c. Two other `put` and `get` are added to the catalog of predefined functions for channel masters. The communication across the link from source to target is performed by 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 extremity, it can be read from the target extremity and is then available for another write from the source. The predefined

function `duplicate` 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` back and forth. The division for duplication is performed only if the duplicate command is issued from both extremities. This implements a synchronisation between the two extremities of a link when they want to duplicate the link.

Xblobs polarize the extremities The channel master does not have any id. But both source and target extremities of its channel (called Xblobs) have one. Thus each link is represented using three blobs: one channel master, one source and one target Xblob. Xblobs enable independent polarization of the link extremities. This is mandatory 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 by doing `broadcast [f,p_1,..,p_k]; flip down`.

Xpairs handle a target and a source coherently A bag of Xblobs has the special type "Xbag". It is practical to manipulate Xbags by using a pair of singletons $(\{i\}, \{o\})$ with `type Xpair = Xbag*Xbag`: i (resp. o) is a target (resp. source) extremity of an incoming link (resp. an outgoing link), and can receive (resp. send) data from (resp. to) the blob at the other extremity of the link. Firstly, when a link is created by the predefined function `new_link`, it returns an Xpair: the pair of extremities of the new link created. Secondly, communication and link creation are easily programmed with 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
```

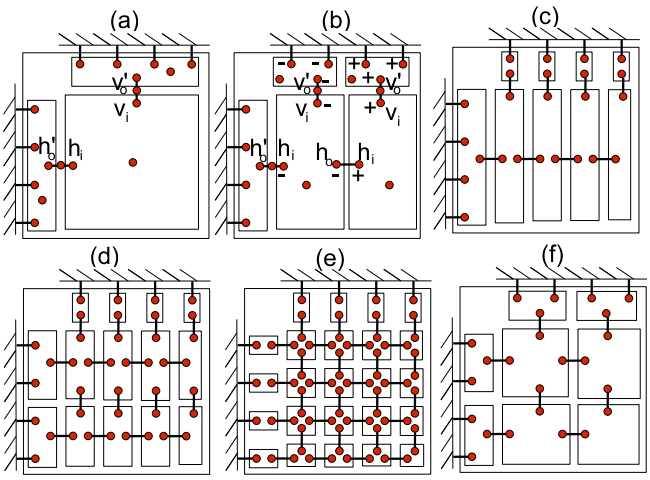
Ports Xblobs are also used for the control of ports. Each port edge is assigned a dedicated Xblob. The ports are indexed by arranging their Xblobs into a blob array with `type ports= (int*Xbag) bag`. The function `put` (resp. `get`) is redefined for those Xblobs that control ports so that they can output (resp. input) data through the port. `let put x = broadcast x ;; let get () = sum ;;`. The reduction `sum` 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 the blob x. This deserves explanation, because until now, we have only seen bodies of for-in using scalars. Unlike scalars, the Xblob x cannot be packed into a message and broadcast to inner blobs, so how do we compile this for-in? The compilation uses iterative division of a for-in ancestor b, exactly like the compilation of a for-in with dot product. Hence the range must be a master range. A copy of x is placed in b, and before each division, b commands a duplication to x so that each offspring gets its own

copy of x. A distinct copy of x will be available for each parallel evaluation of the for-in body. More generally, when compiling a for-in, a liveness analysis must determine whether some blobs $b_1, \ldots b_k$ are used in the body. If it is the case, a copy of $b_1, \ldots b_k$ is placed in a for-in ancestor, which will repeatedly duplicate $b_1, \ldots b_k$ 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. In order for a variable v to be eligible as a range, a divide function must be defined on v. 1- for blob arrays, the function `Adivide`, already introduced, divides an array; 2- for integer intervals, division just splits the interval in the middle; 3- for Xpairs, the function `Xdivide` is used. This case is a bit unusual, since `Xdivide` divides the input from the output, but also adds a new input and a new output, and the size of the Xpair does not diminish. Hence 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. Also, an Xpair h is always used as a non-terminal range, and so the range index must be skipped as said before. 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 (resp. inner) for-in on i (resp. j), h (resp. v) is divided, while v (resp. h) is duplicated. The Xpair h (resp. v) implements communication on the horizontal (resp. vertical) axis.

The Kung and Leiserson algorithm [Lei92] for multiplying 2D $n * n$ matrices is implemented by function `prodmat(left,up,n)` of program 4. In order to simplify the presentation, the two matrices to multiply are directly input 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 the phase 1, it develops the circuit, in the phase 2, the nodes of this circuit iterate a cycle of n systolic computations. In the 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 that 2D grid using Xblobs h' (resp v') connected to h (resp. to v). Because of the specific redefinition of `duplicate` for channel masters, h and h' (resp. v and v') are synchronized when they duplicate. Thus, the three parallel developments of phase 1 get synchronized. In phase 2, the body of the nested for-in implements the Kung and Leiserson algorithm: it accumulates the sum of the product of values input from the left (resp. the up) link, and forwards those values toward the right (resp. the down) link, in a pipelined way. The bodies of the `input`'s for-in feed the 2D grid with rows of A from left, and columns of B from up, 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 by 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 giving a virtual size to blobs. Each iteration of the nested for-in divides the virtual size by two, as for uniformly dividing development. However, the diameter does not decrease along with the size, 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




```
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 extremities 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 avoid overloading of the figure. (a) and (b) also show Xblobs id: the input (resp. output) id of h Xpair is named h_i (resp. h_o) idem for v, v', h'. Allocation of space has been done before the development starts. This causes a temporary low density in (a). The density gradually increases from (a) to (e).

of the tension induced by the channel the dividing blobs remain on an horizontal line during the whole horizontal phase. The width is divided by two, but the height remains constant, hence also the diameter (see fig (c) of program 4). In order to produce a development alternating between the horizontal and vertical directions, we need to program the division of an object (h, v) by combining a vertical and an horizontal link, so that it divides alternatively horizontally and vertically. `let hvdivide var (h,v) = (Xdivide v,h)`, and do the same with pair of indices. The for-in must be written `for (i,j) in (1..n,1 .. n) dot (h,v)`. Using this refined program, the diameter of blobs does get divided, along with the size, and the complexity of $\mathcal{O}((n^2)^{1/2})$ is reached for phase 1. The channels remain always of length $\mathcal{O}(1)$, they can be moved like particles, and do not increase the complexity. Their influence is limited to determining

whether the direction of division is horizontal or vertical, and which one is the plus and minus half. In phase 2, the data are sent from the ports and makes a full traversal of the grid. This takes a time equal to the diameter of the grid, which is also $\mathcal{O}(n)$. For a 2D grid of PEs, this is optimal with respect to VLSI complexity: see [Len90] : however, it is not optimal for a 3D grid of PEs; so unlike bitonic sort, the algorithm should be rewritten, so that it better exploits the third dimension by developing a 3D grid instead of a 2D grid. This shows that for some problems, the optimal algorithm depends on the dimensionality of 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. We show that it is feasible to program it with a variety of different parallel styles, with optimal complexity. The program specifies the "self-development" of a network of automata, which contains a high proportion of parallel semantics. On one hand, these parallel semantics makes it somewhat peculiar to program, on the other hand, it describes the spatial features, and can therefore be reused for different hardware platforms of the computing medium type. For example, although the present paper focuses on fine grain as a symplifying framework, granularity can range from FPGA to distributed memory multiprocessor machine with local connections. We also target irregular architectures embodied by the amorphous computing framework. Only the dimensionality of the computing medium has to be known by the programmer, 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 should "collapse" one of the dimensions into memory. Nevertheless there do exist algorithms that are optimal both in 2D or 3D computing media such as the bitonic merge sort of program 3 in section 3.3.

In this article, we choose to program a catalog of *classic parallel algorithms* in order to compare expressiveness and performance with existing models of parallelism and to motivate our approach in the parallel community. As for expressiveness, the programs that we describe rarely make more than four lines and often a single line. Our goal here is also to show that one can go beyond the spectrum of purely spatial algorithms and program algorithms that "compute something" in the ordinary meaning such as quicksort or matrix multiplication, as opposed to spatial-specific algorithms. We analyze performance in terms of asymptotic complexity and show optimality in the context of the VLSI complexity model. This is equivalent to say that we compare our results with the performance of parallel algorithms running on a 2D or 3D grid of processing elements. Using architectures with stronger connectivity such as the hypercube or a model of parallelism with weaker constraints, such as the BSP or the logP model, one can obtain a better time performance, but one loses the arbitrarily scalability of the hardware. BSP, LogP and many other models of parallelism are compared in [ST98].

However, the blob machine better fits *non classic parallel algorithms* having four features: *decentralization*, *structure*, *dynamicity*, *persistence*, which actually characterize many complex systems. *Decentralization* means that the algorithm

describes a parallelism that is explicit and potentially arbitrary large. Complex systems are made of many parts operating simultaneously. By *"structure"* we mean that the architecture, — i.e. the set of communication pathways between the parts — matches the function it has to performed. *Dynamicity* characterizes systems whose architecture can evolve at run time, depending on the particular inputs, and on the computation being done. The system parts can move, delete existing connections and establish new ones. *Persistence* happens when the rate at which the structure evolves is slow compared to the rate at which it computes. Those four criteria define a niche of applications suited for the blob machine for the following reasons: *decentralization* matches the blob machine's ability to manage arbitrarily large parallel resources; *structure* exploits the unique feature of the blob machine which is able to program the network structure itself. That network development does not restrict the shape of the developed circuits to the lattice structure of the task graphs associated to affine nested loops. *Dynamicity* is the weak point of parallelizing compilers which are more specialized in static (compile-time) analysis. Dynamicity is a basic property of blob machines. It is obtained through hardware free structuring and run-time self-placement. A classic example of a problem having a dynamic task graph is the quicksort which has been covered in this paper. *Persistence* is necessary to amortize the time cost induced by this self-placement. If the structure is modified too often then the time spent for re-placing it may overtake the time spent for computation. The canonical example of an application in the niche defined by those four features is Artificial Neural Networks (ANNs): They are obviously decentralized, the matching between architecture and function characterizes brain circuits [MA98], 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. The architecture is persistent because modifications are applied only when learning new tasks.

"Blob computing" is a long-term project and still needs a lot of work: 1- Our proof of complexity makes some hypothesis on the implementation of the blob machine; the current implementation does respect the hypothesis but it is only partial and needs to be completed. This implies the development and articulation of many spatial algorithms. 2- Our programs assume that there are always enough Processing Elements (PE), so that each automaton of the virtual machine's configuration can be hosted on a distinct PE. If there are not enough PEs then creating more blobs does not create more parallelism. It would then be more efficient to dynamically develop the blob system until all available resources are exploited, then stop the self-development and unfold the computation in time instead of space. 3- The run time system that implements a blob machine can be seen as a an operating system that handles the responsibility of updating the placement of many threads. It performs thread forking, computing, dying, and message exchange. This system is *itself distributed* and takes a fixed bounded percentage of the hardware resources One will be able to afford devoting hardware resources for self placement, only when the size of the available hardware resources reaches a critical threshold that remains to be determined by experimentation.

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

References

[ACA05] A. Adamatzky, B. De Lacy Costello, and T. Asai. *Reaction-Diffusion Computers*. Elsevier Science Inc., New York, NY, USA, 2005.

[AD06] F. Gruau A. Dehon, J.-L. Giavitto, editor. *Computing Media and Languages for Space-Oriented Computation 2006*, Dagstuhl international workshop 06361, 2006.

[ADC+00] H. Abelson, D.Allen, D. Coore, C. Hanson, G. Homsy, Jr. T. F. Knight, R. Nagpal, E. Rauch, G.J.Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.

[Adl98] L. Adleman. Computing with dna. *Scientific american*, 1998.

[AR07] J. Aspnes and E. Ruppert. An introduction to population protocols. *bulletin of the EATCS*, (93):98–117, 2007.

[BCD95] R. Bagrodia, Mani Chandy, and Maneesh Dhagat. UC: A set-based language for data-parallel programming. *Journal of Parallel and Distributed Computing*, 28(2):186–201, 1995.

[Ble95] G. E. Blelloch. Nesl: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, 1995.

[BM93] J. Banatre and D. Le Metayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993.

[BM95] R. Bacik and S. Mahajan. Semidefinite programming and its applications to np problems. In *COCOON '95: Proceedings of the First Annual International Conference on Computing and Combinatorics*, pages 566–575. Springer-Verlag, 1995.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. In *ICDT '92: Selected papers of the fourth international conference on Database theory*, pages 3–48. Elsevier Science Publishers B. V., 1995.

[CCC+89] Alessandro Contessa, Eric Cousin, C. Coustet, M. Cubero-Castan, Guy Durrieu, Bernard Lecussan, Michel Lemaître, and Paulino Ng. Mars, a combinator graph reduction multiprocessor. In *PARLE '89: Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, pages 176–192. Springer-Verlag, 1989.

[CCH+00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and Andre DeHon. Stream computations organized for reconfigurable execution (SCORE). In *FPL*, pages 605–614, 2000.

[CD93] S. A. Cook and P. W. Dymond. Parallel pointer machines. *computational complexity*, 3:354–375, January 1993.

[Coo99] D. Coore. *Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer*. PhD thesis, MIT, 1999.

[Dar99] A. Darte. *De l'organisation des calculs dans les codes répétitifs*. Habilitation à diriger des recherches, Ecole Normale Supérieure de Lyon, March 1999.

[FS92] Y. Feldman and E. Shapiro. Spatial machines: a more realistic approach to parallel computation. *Commun. ACM*, 35(10):60–73, 1992.

[Gia03] 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*, 2003.

[GLCP06] S. Goldstein, P Lee, J. Campbell, and P. Pillai. Scalable shape sculpting via hole motion. *International Conference on Robotics*, 2006.

[GLRT04] Frédéric Gruau, Yves Lhuillier, Philippe Reitz, and Olivier Temam. Blob computing. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 125–139. ACM, 2004.

[GM01] J. Giavitto and O. Michel. Mgs: a programming language for the transformations of collections. *LaMI technical report No 61-2001*, 2001.

[GM02] F. Gruau and P. Malbos. The blob: A basic topological concept for hardware-free distributed computation. In Cristian Calude, Michael J. Dinneen, and Ferdinand Peper, editors, *Unconventional Models of Computation, Third International Conference, UMC 2002, Kobe, Japan, October 15-19, 2002, Proceedings*, volume 2509 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2002.

[GM04] F. Gruau and G. Moszkowski. The blob division a "hardware-free", time efficient, self-reproduction on 2d cellular automaton. In Ijspeert Auke Jan, Murata Masayuki, and Wakamiya Naoki, editors, *Biologically Inspired Approaches to Advanced Information Technology: First International Workshop, BioADIT 2004, Lausanne, Switzerland*, volume 3141 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2004.

[GPKK82] D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, 15(2):58–69, 1982.

[gra] Definition of graph homomorphism. Wikipedia.

[GRS05] B. Gojman, E. Rachlin, and J. E. Savage. Evaluation of design strategies for stochastically assembled nanoarray memories. *J. Emerg. Technol. Comput. Syst.*, 1(2):73–108, 2005.

[Gru08] F. Gruau. Self-developing machines and parallel universality of the blob machine. *submited to computational complexity*, 2008.

[GT00] F. Gruau and J. Tromp. Cellular gravity. *Parallel Processing Letters*, 10(4), December 2000.

[HL93] Per Hammarlund and Björn 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*, pages 210–219. ACM, 1993.

[Jaf06] S. Jafar. *Programmation des systémes paralléles distribués : tolerance aux pannes, résilience et adaptabilité*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, jul 2006.

[KK95] Ken Kennedy and Ulrich Kremer. Automatic data layout for high performance fortran. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 76. ACM, 1995.

[KT02] S. Murata K. Tomita, H. Kurokawa. Graph automata: natural expression of self-reproduction. *Physica D: Nonlinear Phenomena*, 171(4):197–210, Nov 2002.

[LC90] J. Li and M. Chen. Index domain alignment: minimizing cost of cross-referencingbetween distributed arrays. In *Frontiers of Massively Parallel Computation*, pages 424–433, 1990.

[Lei92] F.T . Leighton. *An Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.

[Len90] Thomas Lengauer. Vlsi theory. In *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 835–866. MIT Press, Cambridge, MA, USA, 1990.

[Ler05] X. Leroy. The objective caml system release 3.08. Technical report, 2005.

[MA98] J. Szentagothai M. Arbib, P.Erdi. *Neural Organization - structure, function, and dynamics*. The MIT Press, 1998.

[Neu66] J. Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966. edited by A. Burks.

[NT93] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Comput. Surv.*, 25(3):263–302, 1993.

[Pau02] G. Paun. *Membrane Computing. An Introduction*. Springer-Verlag, 2002.

[PDLS04] J-P. Patwardhan, C. Dwyer, A. R. Lebeck, and 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.

[Rau03] E. Rauch. Discrete, amorphous physical models. *International Journal of Theoretical Physics*, 42(2):329–348, feb 2003.

[R.N01] R.Nagpal. *Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics*. PhD thesis, MIT, 2001.

[RPS+04] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[SDSD86] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., 1986.

[sit] The home site of blob computing (blob.lri.fr).

[SM91] K. Shahookar and P. Mazumder. Vlsi cell placement techniques. *ACM Comput. Surv.*, 23(2):143–220, 1991.

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[Tho83] C. D. Thomson. The vlsi complexity of sorting. *IEEE transaction on Computers*, 1983.

[TM87] T. Toffoli and N. Margolus. *Cellular automata machines: a new environment for modeling*. MIT Press, 1987.

[Tof99] T. Toffoli. Programmable matter methods. *Future Gener. Comput. Syst.*, 16(2-3):187–201, 1999.

[TSF+03] A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J. Moreno, J. Rosenberg, and A. Villa. Poetic tissue: An integrated architecture for bio-inspired hardware. In *Evolvable Systems: From Biology to Hardware 5th ICES conference*, volume 2606 of *Lecture Notes in Computer Science*. Springer, 2003.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.

[Win06] E. Winfree. Self-healing tile sets, architecture for dna-guided self-assembly of nanoelectronics. *Nanotechnology: Science and Computation*, 2006.

Centre de recherche INRIA Futurs
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399