Frederic Gruau and Christine Eisenbeis

Programming self developing blob machines for spatial computing.

Abstract: This is a position paper introducing blob computing: A Blob is a generic primitive used to structure a uniform computing substrate into an easier-to-program parallel virtual machine. We find inherent limitations in the main trend of today's parallel computing, and propose an alternative unifiying model trying to combine both scalability and programmability. We seek to program a uniform computing medium such as fine grain 2D cellular automata, or more generally coarse grain 2D grids of Processing Elements, using two levels:

In the first "system level", a local rule or run time system is implemented on the computing medium. It can maintain global connected regions called blobs. Blobs can be encapsulated. A blob is similar to a deformable elastic membrane filled with a gas of atoms. (elementary empty blobs). Blobs are interconnected using channels, which act as a spring to bring connected blobs closer to each other. The system implements in a distributed way: movement, duplication and deletion of blobs and channels. It can also propagate waves to communicate signals intra-blob, or inter-blob.

In the second "programmable level", each blob and channel contains a finite state automaton, with output instruction triggering duplication or deletion . Execution starts with a single ancestor blob that duplicates and creates channels repeatedly, thus generating a network of automata. It installs a higher level virtual machine on top of a low level uniform computing medium. This "blob machine" is an example of "self developing automata network"

We present in detail, the blob machine, and how to program it using a higher level language called blob ml. We illustrate the execution of many examples of small programs They all exhibits optimal complexity results, under some reasonable hypothesis concerning the -not yet finished to implement – system level, and considering the model of VLSI complexity.

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 [1]

Table of Contents

I Introduction

We will first present blob machines as a natural approach to program « a computing medium », and as such, interesting for the community of people working on spatial computing. Then, we widen the scope of interest and motivates blob machines as a generic way of formulating a parallel algorithm, simplifying the problem of mapping software to hardware while remaining largely architecture independent.

A Blob machines as a vertical approach to spatial computing

1 Spatial computing on a computing medium

Spatial computing is an umbrella term regrouping different approaches, all making the hypothesis 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 it will not be possible to establish rapid communication between an arbitrary pair of components. More precisely, each PE has a set of coordinates in space and communication time will be on average more or less proportional to metric distance in space, which is just what *the VLSI complexity model* spells out [2]. Moreover, individual components are very likely to be subjected to faults or defects, to such a point that in some cases, it could be considered as a normal regime. Such an architecture, that we like to call a "*computing medium"* in this article, includes different models:

- Regular models: any 2D grid of PEs with only local connections such as Cellular automata, Systolic arrays, circuit with nano tubes.
- Irregular models relaxing the constraint of christal regularity and synchronism such as sensor networks in the amorphous model [3] bio films of bacteria [4] or DNA computers [5].
- Reconfigurable models: FPGA need a little adjustment of definition to be considered as computing media because they use long lines where communication does happen faster than just a time proportional to the distance. But because the long lines are costly, there are few of them. So, if one considers communication bandwidth averaged in space, i.e. between two groups of logic blocks, instead of two logic blocks, then the long line effect is smoothed out. The usefulness of such long lines for better efficiency or more robustness is pervasive in many network found in nature called "scale free" network [6], for example genetic regulation networks, protein interaction networks, or neural circuitry.

Spatial computing calls for a departure from computing in time, i.e. using a traditional centralized approach, with a step by step

Dagstuhl Seminar Proceedings 06361
Computing Media and Languages for Space-Oriented Computation
http://drops.dagstuhl.de/opus/volltexte/2007/1024

modification of a global state. Intuitively, to exploit the space, computation should rather unfold in space, by dynamic construction of spatial entities such as circuit, membranes of dividing cells, bacteria, or even network of moving processors. Also, many architectural building blocks such as shared memory or all to all routers used in traditional distributed computing cannot be used permanently to establish communication through an entire spatial computer, because they do not directly scale while complying with the VLSI complexity model.

2 The three directions of spatial computing research

In this new horizons, one can distinguish three levels of research depending on the time scales of available hardware:

1- Research that use « industrial » existing hardware platforms such as mainly FPGA. Here, *position matters for performance*. FPGAs already exhibit the feature of a computing medium, though in a bounded form, because a dice is bounded in area. To be efficient, this bound has to be dealt with in the first place. Existing architectural building blocks from the FPGA/distributed computing community are combined through a language description that allow compile time or run time unfolding of those building block in space. Bounded space implies a careful study of how much, when, and where to unfold, Dehon [7].

2- Research on not yet existing hardware done with simulations. Here, *position matters for functionality*. The underlying hypothesis is that technology, whether nano – bio or vlsi will offer hardware platform large enough so that reasoning purely in this space becomes the right thing to do. Rather than its bounded nature, the study focuses on space itself, and its local properties, such as being faulty, not synchronous nor regular. One can classify approaches on a vertical axis from hardware to software:

•Approaches dealing with the hardware itself, develop new technology enabling spatial computing such as nanotubes [8] or based on chemical reaction [5]. In the programmable matter framework of Goldstein [9], processing elements themselves can move: The poetic approach at EPFL [10] develops a chip specialized in bio inspired algorithm including evolution, development and learning.

•Approaches implementing some computational tasks of spatial nature using original models : the MIT amorphous group developed the fascinating amorphous computing model, and showed how to solve a number of different tasks such as finding a set of coordinates or simulating physical phenomena [3]. Using the chemical computing model, Adamatzky computes the voronoi tessellation [11]. Toffoli [12] proposes a model also called programmable matter focusing on efficient compilation towards cellular automata, he can remove the noise of images, generate textures.

•Some of the preceding approaches are more focused on choosing spatial tasks which can be considered as building block primitives – such as the propagation of a gradient, and movement according to a gradient. Those building block can be combined using a language description: Coore develops the Growing Point language [13], Nagpal proposes a language based on primitives for folding a sheet of paper as in Origami [14]

3-Research completely abstracting the hardware and focusing only on languages. Here, *position matter for expressiveness*. Giavitto and Michel [15] [16] propose to use as the computation space, the data structure itself. They show that reasoning in space can be profitable in itself, with regard to expressiveness The Gamma formalism [17] is based on the parallelism happening in chemical reaction. Gamma avoids any possible artificial sequentialization, which leads to very concise programs. Paun [18] develops the formal language theory of P-system made of encapsulated membranes, with chemical reaction taking place inside. In the "ambient" approach, Cardelli [19] programs alebraic systems of encapsulated membranes. Compared to P-system, the language is more focused on programming, and describing the real cell for simulation purpose.

Ideally, the overall goal of level 2 and 3 is to discover bottom up new principle so as to provide a vertical model fully organizing computation in a potentially unbounded space, from hardware building blocks to programming models, using a library of higher level primitives. We view it as perpendicular, and thus complementary to level 1, which goal is to implement the relevant space time trade off necessary to obtain challenging performance on existing bounded platform.

3 A vertical approach to spatial computing

Both the research of Coore, and Nagpal are in the spirit of a two stages vertical long term approach to spatial computing: 1- They implement a library of higher level primitives with spatial algorithm. 2- They develop a language to combine these primitives. However, for the moment, their language can describe only how to develop fixed structures. Once the structure is laid out on the computing medium, it cannot evolve any more. This severely limit the programing possibilities. The blob computing project follows the same spririt, but without such limitations. A blob can be seen as a generic distributed-computing primitive that installs a higher level virtual machine on top of a low level uniform computing medium. The machine has simple building blocks: blobs and channels, with a set of 14 primitive machine instructions (see table 1) realizing the antagonist requirements of generic efficiency, and expressivity:

- Downward, when they need to be implemented as spatial algorithms, they are sufficiently simple to be installed efficiently on an arbitrary computing medium. Although for the moment, it has only been partially implemented.
- Upward, when they need to be combined in a programming language description, they are sufficiently expressive for general purpose programming (e.g. Non spatial algorithm) to take place.

Furthermore, blobs and channels can move, and place themselves on the hardware. In the following paragraph, we do a step by step justification of why blobs and channels are necessary building blocks to use for spatial computing, in the long term, and why they should move. This allow to introduce a key concept of blob computing called hardware freedom.

Why blobs? Blobs allow to compartmentalize the medium. Blobs act functionally like membranes dividing the computing medium into different connected region. Such a compartmentalization is not required if one consider simple « spatial like » algorithm, such as setting up a gradient in order to measure distance to a given point, or simulating simple physical phenomena such as the wave propagation done by E. Rauch. In those cases, the problem can be solved by having all participating PEs execute globally the same simple local rule. This can be extended to algorithms actually computing something, such as doing a reduction, applying a commutative associative operator like "sum", on a collection of data. However, as soon as an algorithm 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 to store and execute only the one program it runs.

*Why channels? Channels install virtual communication link between remote compartments.*Without channels, a compartment could only talk to its directly adjacent compartment.

Why move blobs? To accommodate dynamic allocation. 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, 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 frontier of the compartment can dynamically move after it has been installed, then, it will allow the system to adjust the location of the separating membrane and by that, the amount of hardware resources allocated to each task. The system does a form of run time load balancing.

4 Hardware freedom

Blobs are software elements structuring the hardware computing medium. Dynamically moving blobs implies moving freely software on hardware. This is an example of an organization principle that we like to call freeing software from hardware or more simply « *hardware freedom* », which is one of the main concept of blob computing. Software is normally conceived as being permanently tied to hardware. Let us illustrates what it means to free it, and reason for freeing it with three simple examples:

•Example 1: *"Hardware free processes"*. A run time ystem doing dynamic load balancing is able to migrate processes between processors without modifying the semantic, so as to balance the load of each processors of the machine.

•Example 2: *"Hardware free memory"*. If you store a data in a memory cell, you do no expect it to jump to the next memory cell, under the pressure of adjacent memory cells, as if they where elastic physical objects. It would nevertheless be nice 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 adressed 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. Its inherent robustness makes it also useful when doing automatic programing using genetic algorithms as shown by Ray [20]. Note that the garbage collector of modern functional language partially implements hardware free memory, being able to defragment memory at run time.

•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, todays 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 where elastic physical objects. This would allow to locally modify already loaded circuit 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 by itself ater creation.

B Blob machine as a new approach to parallelism

1 Motivating a unified parallel model.

Sequential machines built after the Von Neumann architecture abstraction have a simple conceptual model of programming which could be just summarized as "a step by step modification of a global state". The state being stored in a memory and the modifying part being the processor. For parallel machine, there exists already many, highly relevant, models, adapted to different hardwares: multiprocessor machine, VLSI or FPGA circuit. There does not exists a unified simple model which could qualify has the counterpart of the sequential model. We look for such a model that matches the essential word "parallelism" and allows to think of any parallel hardware in a conceptual way.

On one hand, the search for a unified model could in fact seem useless. Indeed, if one consider only cost or performance criteria, then parallel models either originate from a given commercially viable parallel technology such as FPGA, PC farms, and the Internet grid, or else, they are conceived as a mere acceleration of a sequential model by speeding up loops operating on arrays. There does exist notable exceptions such as data flow graph and data flow computers [21], which did originate from new ways of conceiving computation in parallel.

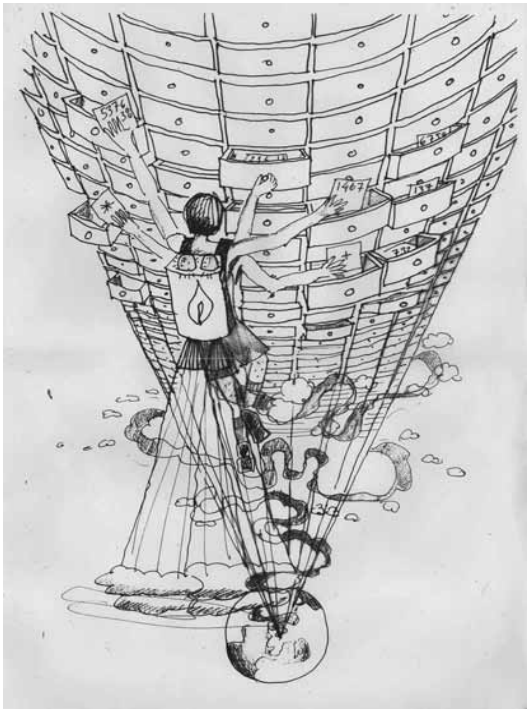


Figure 1: Artistic Illustration of the bizarre though well accepted dichotomy between processor and memory. (courtesy of Paula Femenias)

On the other hand, the biggest issue may not be performance, but the difficulty to program and parallelize algorithms running on parallel hardware. Here, the lack of a unified model is probably one of the big obstacles to considering parallel or distributed machine as "easy" or "standard" programmable devices. Therefore, a necessary prerequisite for a unified parallel model to be worthwhile, is to provide a simplified framework for parallel programing and parallelization.

A blob machine is a particular instanciation of a simple conceptual model called S*elf Developing Machine, or SDMs* which was born out of our strong desire to discover the parallel counterpart of the sequential model. SDMs, as we will see, make minimal assumptions as to the target parallel architecture, and propose new way to address the issue of parallelization.

2 Self Developing Machines (SDM)

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. SDM offer an alternative decentralized organization. The configuration of an SDM is a labeled graph whose vertices are automata. Each automata has only a

finite state, i.e. a small bounded memory, and has output actions that add or suppress vertices or edges in the graph, or modifies the labels of vertices and edes. Some specific vertices called *"port"* stay fixed during the development and make parallel input and output. Universality in SDM comes from the ability of the graph to self develop and expand in an active computing medium (as defined in page 1), instead of allocating data in a passive memory. While expanding, the graph exhibits more and more parallelism. An SDM is of course a virtual machine, since we do not assume that automata and edge are created or destroyed physically.

3 Parallelization is a "folding problem".

What is the central difficulty of parallelization? Call operator, an element of circuit which computes an arithmetic or logic operation. The execution of a computer program generates a "*compute graph*". Vertices represent the operators and an edge connects an operator that produces a data to an operator that consumes this data. Every operator is used only once: it is a directed graph without loops. In parallelism one usually consider task graphs which are a similar concept, but were 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 graph, we use the word edge and vertex, for the compute graph, 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 PEs. We argue that the mapping and scheduling are the central difficulty:

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 direct neighbor PEs, 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* [22]from the compute graph, to the PE network, the latter being completed by adding a recurrent channel from each node to itself so that each PE is adjacent to itself. This homomorphism is usually called the mapping of a task graph to hardware. We propose to call it a *folding*, because the morphism is usually not one to one, each PE having to execute more than one operator. The graph homomorphism problem is a canonical NP-complete problem. It generalizes various other well-studied problems such as graph coloring and finding cliques [23]. NP-completeness is a strong argument of pointing at the folding as being the essence of why programming parallel algorithm is more difficult

Secondly, if the target parallel hardware uses a network with router, then we can define a similar more general concept of folding by mapping each edge to a communication path in the network of PEs. In general, the folding should verify properties in order to guaranty a good performance, the formulation of which may vary depending on the target platform:

- [*distributed computing version*] map arrays and computation on a network of PEs so as to balance load and minimize communications.
- [*VLSI or FPGA version*] map a circuit on a die so as to minimize area and long wires.

The problem of finding a legal execution is simplified by the router, or by the ability of drawing long lines in VLSI or FPGAs, but the global problem remains of the same order except it is now formulated as a problem of optimizing cost functions on the folding: minimizing the communication path length to optimize communication latency, or homogeneously distributing the vertices to balance load.

Thirdly, finding the optimal folding, may also be dependent on the scheduling of operators within each PE. But the scheduling problem can be grouped with the mapping problem, to be also formulated as a graph homomorphism problem: one has to map the compute graph to a target graph defined as follows: the vertices are PEs indexed by integer, and an edge goes from p_t to $p'_{t'}$ if $p=p'$ or p is neighbor of p' and $t' > t$.

Last, one can consider further architechtural constainrs, and encode it in the target graph. For example, a node may not be able so send one data item through two links in a single time step. In summary, we argue that parallelisation is difficult because it implies matching a "soft graph" to a given "hard graph". Because this difficulty is inherent, there simply does not exists a nice programming language that will remove it in the general case. If this folding problem is indeed the guordian knot to cut, then a unified model of parallel spatial machine has to explicitely tackle it rather than hide it.

4 Preliminary folding on a static automata graph.

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

Furthermore, the folding is done in two steps: the compute graph is first folded onto an intermediate graph by merging subsets of its vertices $n_1, ...,n_k$ into a single vertex and locally scheduling the operations of $n_1, ...,n_k$ in sequence. Since it has now to perform many operations, rather than a single one, vertices need to be labeled by Finite State Automaton, (FSA) rather than simply memory less arithmetic operators. The FSA formalism includes the local scheduling. So what we obtain, now, is an FSA graph instead of an compute graph. The final folding to hardware is indeed simplified, because by mapping an automaton on hardware, one simultaneously map all the operations that this automaton is doing.

- Example 1: In distributed computing in general, one can describe a problem by its task graph, where tasks are bigger than simply operators. This is already an intermediate folded form of the underling compute graph. Each task regroups a big chunk of the whole compute graph, and by mapping a task to a PE, one simultaneously maps all its associated operators. The partition in tasks is given by the algorithm, and tries to minimize inter-task communication. This approach also commonly called "macro data flow" [24] can be adapted to execution on the Internet grid. Since communication is particularly expensive, 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 is also used to program FPGA with task graphs. They put the extra requirement that data flows as streams between tasks, so as to do pipelined computation: the SCORE project of A. Dehon [7] is representative of this approach. Language constructs specify "streamed circuits" where vertices have memory, and data streams between vertices. Here all the data is contained in the stream. The compute graph is already described in a consistent folded form in the streamed circuit, which can be mapped onto an FPGA, by adding all the necessary buffers and using standard CAD tools. In SCORE, the streamed circuit is itself specified in a parameterized way, so that it can unfold on the available hardware by trading space for time. The success of the system comes from having correctly identified stream has being an important key to easily obtain performance on FPGA, and giving language constructs to support those streams so as as we said, it defines a soft and a hard domain enough tunned to each other to ensure tractable efficient folding.

- Example 3: In data parallelism, one considers arrays and nested loops accessing those arrays with affine combination of loop indexes. The compute graph has a shape of a multidimensional lattice [25]. A mathematical analysis of the dependency graph allows to describe this lattice in a parameterized form, exposing the parallelism. This lattice is embedded in a k-dimensional discrete space. In particular, with High Performance Fortran [26],

the programmer himself specifies "templates" to explicitly align all the arrays on such a virtual k-dimensional space. This intermediate graph is not yet in a folded form, but its high regularity makes it easy to fold it onto a regular 2D grid of PEs. One uses 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 constitute the final folding. The computation of optimized projections, wrapping or tiling remains tractable as long as the number of loops is small. This can lead to either automatic synthesis of systolic arrays, or automatic parallelization on networks of PEs.

All those existing approaches preliminary fold the compute graph on a static circuit. This suits very well algorithms for signal processing or scientific computation using matrix algebra. However, many algorithms cannot be programmed into a fixed size, static circuit. In particular: algorithm with dynamic features such as quicksort , where the shape of the compute graph is data dependent. To provide more programmability, the blob machine offers an added mechanism, allowing dynamic instantiation of circuits.

5 Direct programing of a dynamic automata graph.

We are working out a similar match between a set of language constructs, and an architectural domain, but with the ambition of having a broader scope: we would like arbitrary scalability on the hardware side, and no limit in genericity on the software side.

•For the architecture, we only put the constraint of the spatial computing framework (see page 1). One of the main design concern of those architectures that we called "*computing media*" is precisely to scale in arbitrary size.

•For the language, the first idea is simple: given that the folding and scheduling are the difficult thing, let the programmer do them! So he will program directly a folded-scheduled form of the compute graph, that is to say: a graph of Finite State Automata (FSA). The second idea is that to be more generic than just static circuits, we use the Self Developing Machine (SDM) described page 3 where the automata not only perform some fixed arithmetic operations, but also locally modify the topology of the graph itself, by adding or removing vertices or edges using self developing instructions.

The language specifies the parallel development of a network, node by node. Thus, on one hand, it describes a spatially extended object, i.e a circuit, which is needed for exposing parallelism. On the other hand, it does not sacrifice the capability of dynamic instantiation, needed for programmability. Folding the compute graph on a dynamic circuit is illustrated in the quicksort program 1 p 10. Each integer n_1, .., n_k to sort is stored on a distinct automaton a(n_1), .., a(n_k), 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 communicated along a dynamically evolving network structure. Programming with SDMs present several advantages:

•*Matching structure to function:* The development does not restrict the shape of the developed circuits to the crystalline structure of the task graphs associated to affine nested loops. The structure is directly programmed and therefore adapted to the specific targeted functionality. The adequation between structure and function characterize brain circuits [27], and is obviously needed to build efficient dedicated VLSI circuits.

•*Reuse:* The parallel semantic of SDM embodies the difficulties and can then be reused for different hardware platform of the computing medium type. For example, the granularity can range from FPGA, to distributed memory. Only the dimensionality of the computing medium should be known by the programmer, since it restricts the set of feasible self developing graph. For example, it is not possible to efficiently map a 3D grid to a 2D computing medium, so one the programmer should manage to "collapse" one of the dimension into memory. Nevertheless, there does exists algorithms which are optimal both in 2D or 3D computing media, such as the bitonic merge sort of program 4.Worthwhile dimensions of the medium are either 2D or 3D, since 1D is not efficient, and 4D is not feasible.

•*Semantic with local communication.* The parallel semantic includes more than just exposing parallelism: communications are always local; an automaton communicates only with its direct neighbors. There is no shared memory, or even a global name space. Who communicates with who, this is thus clearly represented at any time by the network itself,. This enables the run time system to automatically map the network.

6 Automatizing the architecture-dependent folding

We consider architectures which are computing media: they are embeded 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 determines the PE responsible for 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 an hardware free (see page 3) distributed representation of the vertices and edges representing the self developing graph: vertices must be able to freely move between neigbor PEs without interference with the underlying computation going on.

•Secondly, the run time system must determine the appropriate direction where to move the vertices. For this purpose, it simulates physical forces of attraction between adjacent vertices, to optimize communication latency, and repulsion between nearby vertices, 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 edge are added, or suppressed, and the equilibrium is perturbed. Each vertex locally computes the force applied to it from its neighbor, 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 stabilize again, and computation can carry on.

Such techniques based on force simulation already exist, but here, the combination with a step by step development intuitively prevents the plague of local convergence. Indeed, the adjustment needed at each step is hopefully sufficiently simple, that vertices should be directly attracted towards their new optimal position. There are less chance to be trapped by local sub-optimal basins of attraction. Assuming a set of reasonable hypothesis (already tested in some classic cases such as sort, or matrix multiply), the theory predicts optimal asymptotic performance results (up to a constant factor), under the VLSI complexity model [2].

II The blob machine

A The formal model

The blob machine is a specific self developing machine that tries to answer to the question of what should be a minimal set of building blocks (types of vertex and edges) and self developing instructions working on them, to fill the following antagonist requirements:

•Provide scalability: building blocks should behave like simple physical 2D/3D objects to parallelize efficiently on an arbitrary large 2D/3D computing medium.

•Without compromising genericity: they should allow to represent arbitrary graph, while structuring it in a hierarchical way, to compile high level programming language.

We propose to use only two types of vertices: blobs and channels behaving like elastic membranes and springs, and two types of edges: horizontal and vertical.

1 Configuration of the blob machine

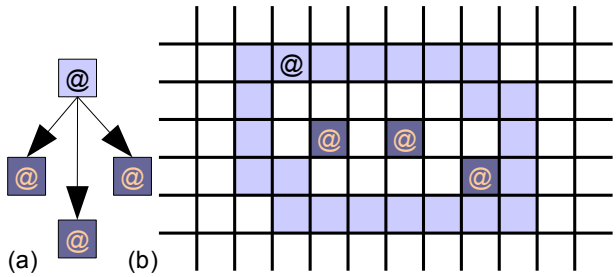


Figure 2: Representation of blobs (a) network representation of a blob containing three atomic blob. (b) topological representation on a 2D grid of PEs. The blob behaves as a mulit set, which is represented by a membrane that divides the grid in two connected components: the inside and the outside. The representation is hardware free, because elements can move freely inside as long as they do not cross the membrane. The membrane itself can move, while remaining connected. One of the membrane PEs must contain the automaton associated to the blob. Automata are represented by the character @. Here, the representation is very fine grain, since 22 PEs are wasted only to represent the membrane. On can choose a coarser grain representation, where a PE can support multiple membranes and automata.

Blobs and vertical edges: The vertical edges connect all the blobs together in a tree which defines a partial ordering on blobs. The root is called the *"skin"*, it is a maximum blob, and the leaves are empty blobs also called *"atomic blobs"* or *"atoms"*, they are minimal blobs. Blobs behave like elastic membranes regrouping and pulling together sub blobs called its elements. There are two reasons motivating to encode explicitely a tree structure:

•1- Encapsulated blobs provides a hierarchical representation enabling structured programming, such as the use of function or procedures, hierarchical data structures, hierarchy of objects.

•2- We do not distinguish between the different sub blobs of a blob, in other word, blobs are multi sets, and the tree represents a hierarchy of encapsulated multi sets. Such muti sets have a natural, hardware free representation, using membranes, as shown in figure 2. The edge of the tree are not needed to be represented, which can be a great economy of space and time.

In the topological representation, a blob can be represented using simply connected set of PEs; In [28] we demonstrate that hardware free blobs can be implemented on arbitrary network such as amorphous network. We prove that to maintain global connectedness while moving it is sufficient to maintain local connectedness.

Channels and horizontal edges: In practice, communicating along the edges of a tree is not sufficiently expressive. If for example, one wants to install a 2D grid, it is always possible to encode it in a tree, and label leaves with a pair *(x,y)* of coordinates, but that is not very practical nor efficient. One would have to implement a routing strategy so that vertices which are neighbor in the grid, communicate along the tree, with probably congestion near the root. This goes against one of the main design principles of SDM which is to always explicitely install a communication edge between vertices that need to communicate, so that the communication needs are constantly reflected in the machine configuration and the run time system can use this information to optimize placement of vertices. This motivates the use of channels that install a communication link between an arbitrary pair of blobs. A channel is connected to two blobs, called its extremities, using a left and a right horizontal edge. A channel behave like a spring, pulling together both its extremities to reduce communication latency. Furthermore, a channel may contain blobs being transferred from one extremitie to the other. These traveling blobs are connected to the channel using vertical edges, to distinguish them from the extremities.

Edges labeling: Because they are used as bidirectional link to communicate data between vertices, edges have two buffers for each directions. When an edge is duplicated, we need to distinguish the two copies. For this purpose, the edges have a polarization: positive or negative. One copy is polarized plus, and the other, minus.

Ports: Ports are specific atomic blobs connected to a host using a vertical edge downward, to which they can input or output data. Ports are present in the initial configuration, and remain fixed during the execution. Their number noted *p* is a feature of the machine.

Initial configuration: It is made of a single ready blob called the *ancestor blob*, containing the p ports. All edges are polarized positively, all the buffers are empty, all the automata are in their initial state. The ancestor also plays the role of the skin: it remains the root of the tree hierarchy, encapsulating all the other blobs at any time.

2 Instructions of the blob machine:

The blob instructions are not high level language constructs. They are comparable to elementary machine instructions, to facilitate their efficient implementation on an arbitrary computing medium. For example, they do not add or suppress more than one vertex or one edge at a time. The instruction set is in fact the output alphabet of the mealy machine [29] (FSA with output) executed by each vertex of the SDM. This mealy machine is the program executed by the SDM which is loaded in the ancestor and the ports. All the vertices executes the same mealy machine, but with a distinct state stored locally.

Static edge orientation for addressing: The edges are statically oriented: we distinguish outgoing and incoming edges using the addresses *up/down* for vertical edges, and *left/right* for horizontal edges. These addresses are used by instruction *send, rec, polarize, polarity, flip, move* to specify on which edges to apply the instruction.

Dynamic edge orientation for mutual exclusion. There exists also a dynamic orientation independent of the static orientation which determines *ownership* of edges: An edge is owned by a vertex, if it leaves the vertex. This orientation is called dynamic, because it can be change during execution, by the instruction *flip*. In order to execute a given instruction *i*, a vertex must own the edges *used(i)* used by instruction *i.* The third column of Table 1 shows what is *used(i)* for each instruction *i*. If the vertex does not owns all the necessary edges, then it is said to be not *ready*, and has to wait that the neighbors of those edges flip them back. This implements a mutual exclusion between instructions needing to use the same edges, and ensures the important property of *confluence* [30]. For example in figure 3, the port cannot do *polarize up* -, because it does not own its *up* edge. If it could do it, then the division happening after that would move the port to the negative child instead of the positive. So depending on wether polarization or division is executed first, one would obtain distinct systems, and confluence would be contradicted.

Asynchronous Parallel execution. At a given time step, all the vertices which are ready, can execute their instruction simultaneously, but they do not have to. If a ready vertex waits a few time steps, confluence ensures that the execution order does not matter, and the system will always converge to the same configuration. Asynchronous execution implies that a global clock is not needed, which is important for scalability.

Communications: During a communication on a given edge, both the sender and the receiver modify that edge: the first stores the data in the buffer, and the other pops a data out of the buffer. Therefore,

they both need to own the edge, and the sender has to flip the edge before the receiver can receive the data. The use of buffers on edges is not compulsory, it serves to avoid this constant back and forth flipping between sender and receiver, when many data have to be sent. One can implement a parallel *buffer*, where the send and receive may happen simultaneously (if the buffer is not empty). But, then, to duplicate or delete that parallel buffer, both the sender and receiver will have to synchronize.

		Codop	Semantic	Used
Classic	Blob and channel	Send *val adr*	Sends value val to edges *adr*	Adr
		Rec *adr*	Receives value from edge *adr*	Adr
		Reduce *op*	Sets reduction operator	
		Polarize *adr* +/-	Polarizes edges *adr*	Adr
		Polarity *adr*	Returns polarity of edges *adr*	Adr
		Flip *adr*	Changes orientation of edges *adr*	Adr
		Move adr	Moves sub blobs through edge *adr*	Adr,down
Self Developping	Blob	New blob	Includes a new blob	
		New chan	Includes a new channel	
		Divide	Divides blob	All
		Wrap	Encapsulates a new blob	Up
		Merge	Merges blob	Up
	Channel	Duplicate	Duplicates channel	All
		Delete	Deletes channel	All

Table 1: Deterministic instruction set with 7 instructions for communications and synchronization, and 7 instructions for self development; The operand " adr" can take the values up, down, left, right, to address a given type and orientation of edges. The operand "val" is a scalar: integer or float.

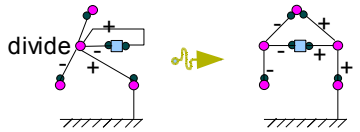


Figure 3: The division instruction. Blobs are represented by small gray disks and the channel by a square. There is a single port, represented by a blob with an edge to the ground of an electric system. The dynamic orientation of edges is represented by arrows whose extremity is a tiny black disk. The static orientation is implicit: up is upward, down is downward, left is leftward and right is rightward. In order to divide, the central blob needs to own all its edges, which are therefore all outgoing.

Parallel reduction Consider a vertex having several edges with the same address. For example, a blob with more than one sub blob, has several edges addressed by *down*. The execution of instruction *rec down* will return not one, but many values. All those values will be combined into a single one using a reduction. The reduction operator is selected prior to the receice, using the instruction *reduce op*. To preserve determinism, the operator mustbe associative and commutative such as sum or product. We sometimes needs th non deterministic operator *one_of* which randomly returns one of the values. To avoid dead lock, if one of the buffer is empty, then the neutral element is returned. For *one_of* the special value *undefined* is returned if no values are present. In some situation, the non deterministic instruction *prefix* is practical. It assumes an arbitrary order of the elements: $e_1, .., e_n$ and sends back to each element i the partial reduction done on values sent by $e_1, .., e_i$. Reduction and prefix are part of the blob machine, because they can be directly implemented efficiently in parallel, and are adapted to the multi set structure implied by blobs.

Self developing instructions. Whenever a vertex creates another child vertex, negative polarization is used to differenciate the created from the creator, which gets positive. Here, polarization of a blob is defined as its unique *up* edge, and polarization of a channel is the same on both it unique *left* and *right* edge. Creation is said to be symmetric, if creator and created are brothers. Blobs and channels have each their own way of symmetric creation called blob division, and channel duplication.

- Blob division is illustrated in figure 3. The creator moves all the negatives element to the newly created vertex. The elements must be polarized prior to the division. If it want to directly connect to the created, the creator just creates a new channel prior to the division (figure 4 (c)). Because the left and right edges of this new channel are of opposite polarity, creator and created will each receive one edge, as illustrated in figure 3.

- Channel duplication operates on empty channel, (without sub blobs being transferred) as in figure 4 (e). The created channels is connected to the same extremities as the creator; it also get a copy of the buffers with all the pending messages.

Blobs can also create a new blob in an asymmetric way: by wrapping around a new blob (figure 4 (a)) or adding a new atomic blob (figure 4 (b)); The creator owns the downward edge to the created. Note that the *new blob* instruction can be considered as syntactic sugar since it can be defined by combining *divide* and *wrap*. Nevertheless, defining it as a separate instruction is useful, because it does need to own any edges to be executed, as opposed to *divide*.

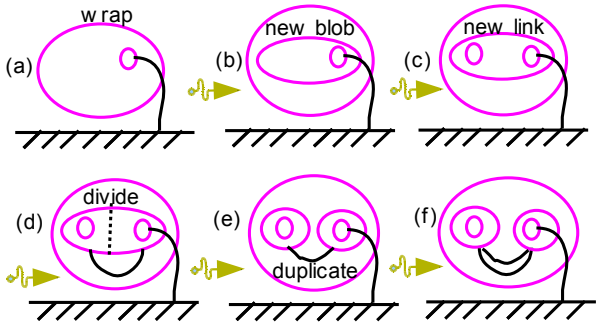


Figure 4: Example of self development: the smiley! Note that this example is only illustrative of the primitives, it is not the purpose of blob computing to generate shapes such as smiley's We use the topological representation omitting the edge orientation and polarity. The step (d) to (e) are the topological counterpart of figure 3

Removing instruction. The instruction *merge* merges the blob with its outer blob, giving it all its sub blobs and channels. This amounts to just suppress the membranes. The instruction *delete* for a channel removes the channel, if it is empty. The ancestor blob implementing the skin, and the port blob cannot divide nor merge to preserve the port number, and a unique root to the blob hierarchy.

Moving blobs. The instruction *merge (*resp. *delete)* is the exact reverse of *wrap* (resp *duplicate*). If a vertex does a *wrap* followed by a *merge*, then the system is back to where it was. We introduce the *move* instruction to be able to also reverse the effect of a *divide.* Reversion is possible, if the two blobs produced by the division, are connected by a channel. It is done by having one of the child move its element through that channel, back to the other child. The instruction *move adr* moves negative sub blobs to the neighbor connected by an edge with adress *adr* and positive. It works only if there is exactly one such neighbor.

3 The complexity model

Define by induction the *size* of an atomic blob or of a channel as one, and the size of a non atomic blob as one plus the sum of sizes of all its element . Call the *hardware support* of a blob, the connected set of PEs, on which the blob is distributed. The support must have an area in 2D (or a volume in 3D) at least proportional to its size. A

blob instruction has to be broadcasted throughout its hardware support, this takes a time proportional to the diameter. In order to obtain good performance, the run time system must ensure that this lower bound is also an upper bound: a blob instruction should take a time proportional to the diameter to complete, and furthermore, the blob shape should be maintained round, so that the diameter remain proportional the square(resp cubic) root of its size,on a 2D (resp. 3D) grid.

4 Implementation results

We have not yet implemented a complete run time system, but we do have studied the above complexity requirement for the *divide* instruction. We choose to focus on *divide* because it seemed the most difficult to implement. It needs a mechanism to regroup the positive and the negative element and install a separating wall. In the case of squared blob having only atomic sub-blobs simulated on 2D grids, we have presented with Tromp [31] a simple cellular automata block rules with one atom per PE, and only two bits of state representing atom occupation on a PE, and polarity. We give a complete three pages mathematical proof that the rule can do a horizontal division by separating the plus and the minus element in a time less than three times the height. The rules simply piles up atoms into a heap, with a 45 degrees slope. The obvious solution of pushing the positive (resp. negative) atoms downwards (resp. upwards) does not work, because it creates arbitrarily big vertical atom piles. The trick is to push preferably along the diagonals. In the case where the blob is allowed to have a round shape, we have done a simulation with Moskowski [32]of the complete division of a blob, and measured experimentally that the time for division is indeed proportional to the diameter.

With Lhuillier, Reitz and Temam [33], we have implemented a simulator of blobs using a decomposition of the FSA controlling the blob into elementary operators called particles, which are in fact atomic blobs. At the time, we considered a synchronous SDM , where every vertices had to execute at each instant. The simulator makes some performance prediction, assuming the specification above: an instruction time proportional to the diameter. The simulator confirms the time complexity of $O(\sqrt{n})$ for the quicksort of n values, which we will present in the examples. It does not however evaluates the time necessary for moving the blob, nor does it consider a distributed implementation of the blob system.

B Different incarnations of a blob machine

Consider a distributed memory parallel machine with many processing elements called PEs. The implementation of a runtime system on such a machine can be fine grain or coarse grain

1 Fine grain

On a fine grain machine, each PE will host one or zero automata. A typical fine grain implementation is a 2D cellular automaton, but it can also be an amorphous machine. The finite memory of the PEs will force to use a technique where a blob automaton itself is partitioned and distributed among several elementary automata stored into a set of atomic sub blobs. Such a technique is described in [33]. At the finest grain, one elementary automaton can be associated to each transition. Programming can be adapted to whatever processing resource is offered which can be as elementary as a couple of CLBs on an FPGA. The operators used will then be lookup tables operating on bits, rather than traditional arithmetic operations operating on bytes.

Finer grain augments the parallelism of the communications caused by dynamic migration of code and data, because blobs have a larger diameter counted in number of PEs. The increased parallelism is double: pipelined in the direction of the blob movement, and data parallel, in the direction perpendicular to the blob movement.

The problem of the finest possible grain model poses a fascinating scientific challenge: what is the most elementary hardware building block that can be combined in arbitrary number to build a computing medium implementing a blob machine?

2 Coarse grain

In coarse grain machines, each PE hosts a number of automata, and the resources needed to implement hardware freedom and dynamic migration can be reused for a couple of different automata instead of only zero or one automaton. One obtains a slower machine, but one that can run a bigger number of automata, big enough to do real algorithms. A simple implementation would be to simulate a connected set of PEs of the fine grain machine on a single PE of the coarse grain machine. This obvious method can be optimized to avoid calculating a distinct position for every single automata. We regroup the atomic blobs which are direct brother, and update only the position of the center of mass of this group, assuming a regular distribution within the group, around this center.

In a coarse grain machine, a blob or a channel hosted on a given PE is like a thread, and the runtime system is a kind of multi threaded operating system scheduled in a cooperative way. Each time a thread makes a call to a blob primitive, it temporarily stops so that another thread can start. Each thread computes a FSA. Since looping within a fixed FSA is not relevant to generate useful computation, cooperation is bound to happen at small finite intervals of time. Threads are given control in a round robin way. Not ready blobs are quiescent threads waiting to be waked up. For the sake of optimization, one can use an extra blob primitive instruction called "*pause"* that just let a thread pass it turn. It does not change the semantic, but modifies the order of execution.

III The blob ml language

Motivation: We define the bob ml language in order to program parallel algorithms into blob systems, in a concise and high level way. Programming directly the FSA of a self developing graph with the blob instruction set is difficult: firstly, FSA are even at a lower level of abstraction than a machine language, secondly, a blob cannot send a message selectively to one of its sub blob, all will receive it. The first advantage of a higher level language, is to manipulate sub blobs through identifiers. In essence, blobs are simply sets of sub blobs, so to compile a given host language into a blob sytem, one just need to add the "set" type, and language constructs to manipulate sets. We choose Ocaml as host language, extending it into blob ml. Ocaml brings in higher order functions, objects, and static type inference: Ocaml infers the type of each definition, and shows it, after it has been declared. We will write the inferred types, because it helps the understanding. Ocaml is also the target language towards which blob ml is planned to be compiled. The idea is to exploit the native Ocaml compiler, by preserving the part of the blob ml code which is pure Ocaml.

Notation: Our syntax follows the ocaml syntax, except for some slight addition: the "set" type is added, parenthesis with vertical bars '{| .. |}' are used to denote sets, and the @ is used for remote method calls on object hosted on distant blobs. The reader not familiar with Ocaml is strongly advised to glance at the excellent on line manual at caml.inria.fr/pub/docs/oreilly-book/ . In the figures illustrating development, we often prefer to use the network representation of figure 3, because it is more compact. To understand how the network is laid out, one must bear in mind the real machine configuration, which is the toplogical representation of figure 4 , that uses a set of encapsulated membranes.

The finite memory constraint An Ocaml program is compiled into a blob system from where each blob is controlled by a Finite State Automaton (FSA). The ancestor blob starts to evaluate the Ocaml program and allocates all the global variables. If it is a scalar, it can directly store it in its local memory. If it is an aggregate data structures such as arrays or lists, it should create sub blobs, to store it. More

precisely, as soon as a variable size depends on the problem size, then it must be distributed on a set of sub blobs. For example, an array, should have each element stored on a separate blob. While it complicates the processing, it opens up the possibility of parallel processing, since each of those blob can process concurrently its hosted array element. With parallel languages, it is normal that the programming style follow some constraints to produce efficient code. For example, when using imperative languages to describe hardware object for FPGAs, the use of variables generates possibly expensive communications. Usually, the constraints are hidden to the programmer. In our case, the constraint of finite memory is simple enough to easily fit in the programmer's mental representation.

Parallel data structures. While blob computing is designed for decentralized computation such as neural network development, we believe it is important to first demonstrate its expressiveness and performance on classic parallel algorithm. We consider the most frequently used data structures: arrays and linked lists, and show how they can be distributed on a blob system, to be used in parallel algorithm. One must think in advance how will the data flow within the data structure, and choose a blob representation accordingly. For example, a linked list can be implemented horizontally or vertically, depending wether the elements of the list are updated synchronously or hierarchically. Since sub blobs have no particular order within their outer blob, they naturally implement sets. So the set is the only basic data structure that can be directly distributed on sub blobs, all other data structures such as arrays or linked list will be encoded using sets.

A Sets

1 Specifying computation on sets

The Ocaml type system is completed with the aggregate parameterized type "'a set", meaning "a set of value of type *'a*," where *'a* is a parameter of type which can be anything. It is the type of the set element, which will be automatically inferred by usage. A set is represented using one sub blob per element. To generate a set by listing its elements, we use the following notation: let S= {| 2,8,2,5|}, creates four sub blobs each storing one of the four integers. Note that elements can be repeated, so blobs implement more precisely multi sets rather than sets. One can also program a set expression with the operator "set of <exp>" which creates an initial sub blob b_0 evaluating <exp>. If <exp> contains loops and division, b_0 can generate many elements. For example, the program: let S = set of let e = ref 0 in for i = 1 to k do divide in e := !e *2 and in e := !e*2 + 1 done; !e generates the set $0\{0, 1, ..., 2^k - 1\}$. The blob b_0 repetitively divides and loops, thus doubling the number of elements at each of the *k* iterations of the loop. Here, following the Ocaml notation, *e* is a reference to an integer and *!e* is the value pointed by *e*.

Master and slave The automaton associated to a set element implements a mater/slave relationship with respect to its outer blob. Theresulting parallelism is like the "*actor model*". Upon creation, a blob is in a quiescent state called *slave.* It is waiting to receive a command from its outer blob that will wake him up. Command include basic blob primitives, such as send, divide, polarize; or the evaluation of an Ocaml expression. Commands are sent down together with an id to identify which sub blobs are targeted. The id of a sub blob is simply the name of the associated set. It is given to the sub blob when the set is created.

Computing a scalar value from a set. One use directly reduction operators; if *S* is a set of integers, sum of S / card S computes the average, one of S returns a randomly chosen element. The reduction operators are directly available in the blob machine, so reducing *n* values can be donedirectly within the outer blobs and its *n* sub blobs. Allocating *n* memory cells would have contradicted the finite memory constraint. Here the command sent by the master is simply to send up the elements. To reduce those elements, the master sets the reduction operator, and makes a receive.

Compute several sets from a set. The construction forall e in S <exp> trigers a distinct computation on each elements *e* of a set *S*. The master sends the whole expression <exp> to its elements which evaluate <exp> and returns new elements forming new sets. For example, the expression forall e in S e when e< 10 denotes the subset of the element of *S* which are smaller than *10*; the function fun card S = sum of forall e in S 1;; *fun card 'a set -> int* computes the size of a set: each element creates a value "1", the master makes the sum reduction of this set of "1s". Several sets can be computed in a single forall loop. It is more concise, and more efficient. For example, the forall loop of the quicksort (resp. bitonic_merge) computes the two sets *L* and *R* (resp. *A* and *B*).

Blob polarization, cloning, and removal Those are system command which are automatically generated using an analysis of live set variables.

- Removal: A sub blob with id *S* will be automatically removed by its master, as soon as its associated set *S* is no longer a live variable of its master. If one uses reference to sets, such as in quicksort, this liveliness analysis may be done dynamically, similar to garbage collection.

- Polarization and cloning: When a blob divides, the liveliness analysis shows which sub blob is used in each child. If a given sub blob is used in both, it will be cloned, else it will be polarized according to which child is using it. For example, before the division occurring in the while loop of quicksort, L is polarized negatively, and R positively, because L is used in the first child, and R in the second child.

If a sub blob is atomic, deep cloning and removal is done simply by using the blob primitive divide and merge. If we use sets of sets, then sub blob contains themselves sub sub blobs. Deep cloning (resp. removing) a blob is done by recursively deep cloning (resp removing) all the sub blob hierarchy.

Compiler optimization Forall loops can be optimized in two ways:

1- *Avoiding cloning of sets* Cloning implies making a copy of all the sub blobs and that is expensive in time and space. Consider a forall loop applied on a set that is live after the loop. Instead of being cloned before the loop, the set should be preserved by the loop. For example, the function *card*, already mentioned, is best implemented by having sub blobs send up the value 1, and remain, rather than cloning the set, having all the elements compute a 1, and delete the newly created set right after that.

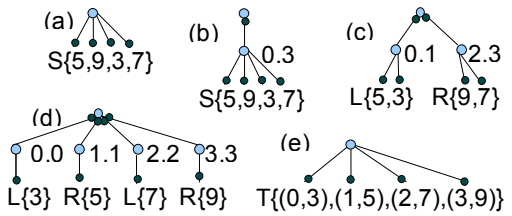
2- *Partitioning the automaton.* Compiling forall loops involves having the element perform some code, where is that code located? A first idea is to store the code in the master, and have the master send that code to the sub blobs. This is not very satisfactory, because it can generate expensive communications and big automata, since the master blob has to contains all the code. A better idea is to predict at compile time which code is executed by the elements, and directly compile it into their automaton. We will show how this can be done in a systematic way by storing objects in elements: the code is the object methods, and the automata associated to an object regroup all the object methods, and fields.

2 Iterative set division with quicksort

Motivation: The quicksort is a representant of the divide and conquer family of algorithm, simple and yet very efficient. Its task graph is dynamic, it depends on the data, which makes it difficult to handle using static compile-time method. It illustrates horizontal parallelism, as well as interleaving of computation and development. Last, quicksort is based on iterative division of sets, which is a basic technique that we are going to be using all the time.

Synopsis. In the following program 1, the function quicksort takes a set of integers, and returns an array represented as a set of records {i:int ;v:int }associating an integer index to a value. We will see later how such an array can be outputted in parallel. We here assume that all the values to sort are distinct. We see later how to handle the

general case using arrays.




```
Fun quicksort S = set of
  let pS= ref S and inf=ref 0 and sup = ref card of !ps in
  while !inf < !sup do
  let pivot = sum of !pS / card of !pS in
  let L,R = forall e in !pS (e when e<pivot, e when e >= pivot) in
  divide in pS:=L ;sup :=!inf + card !pS -1;
   and in pS:=R; inf:=!sup - card !pS + 1;
  done;
  {i= !inf ; v= one of !pS };;
fun quicksort int set -> (int *int) set
let T= quicksort {|3,9,5,7|} ;;
```

Program 1: The development illustrates the evaluation of the expression let T= quicksort {|3,9,5,7|} ;; *the notation follows figure 3, except that atomic blobs are omitted (a) the initial configuration. the root of the tree is the master evaluating the expression. (b) Because of the operator* set of, *a sub blob* b_0 *is created, and is given all the set elements 5,9,7,3 that are used in its computation. It is labeled by the two integers inf and sup storing the first index: 0 and the last index: 3 to be associated to the elements of the current set ; It also uses a variable pS which is a reference to the name of the current set. pS is initialized with S and is later updated to either L or R. (c) and (d) :the quicksort is not implemented as a recursive function but with a while loop, execute by* b_0 *.The loop body computes a pivot as the average of the elements, and partition the current set in two sets L,R of elements smaller, and greater than the pivot; It then divides , one child gets L, and the other R. They updates inf and sup, and loop again. After two iterations shown in (c) and (d), the loop exits, because only one element remains in the current set. The element is returned, in (e) with the computed index which is simply !inf = ! sup.*

Synchronization: The dynamic edge orientation used for synchronisation is represented by arrows whose extremity is a little black disk as in figure 3. As soon as the sub blob b_0 executing the forall loop body is created, the master blob can continue its execution. All the further computation will be done by b_0 's descendant. Since those descendant owns the link to their master, the master cannot control its sub blobs. It must wait until all b_0 's descendant.have flipped back the edge to him.

Complexity of iterative division When designing data structures, one must bear in mind the blob complexity model described page 7. We will consider only 2D grid of processor, for which, the time complexity of a blob operation is the square root of its size. Assuming that the pivot split the set in two subsets of equal size, the iterative division of a set of n elements has the complexity $O(\sqrt{n})$. In other word, doing an iterative set division takes the same time complexity than doing a single blob operation in the master blob.This is because the size is divided by two at each division step, so the total time is proportional to the initial diameter multiplied by the sum of $1 + q + q^2 + ... + q^k$, where q is $1/\sqrt{2}$. This sum converge to $1+\sqrt{2}$. If the pivot split the list in two sub lists which two sizes satisfy that the quotient of them falls between some fixed bound, sorting still preserves the time complexity of iterative set division: $O(\sqrt{n})$ and a space complexity of $O(n)$. This is the optimal VLSI complexity as demonstrated in [2]. We will also give the bitonic sort code which has this optimal complexity without having to make any hypothesis on the data.

Horizontal and Vertical parallelism. The blob framework distinguishes clearly between two kinds of parallelism: Vertical parallelism happens between a blob and its sub blobs, horizontal parallelism happen between brothers, sub blob of a given blobs. The quicksort example clearly illustrates how the forall construct enables horizontal parallelism: each element of the set computes simultaneously the comparison with the pivot. We will see an example of vertical parallelism with the queue example of program 5.

B Array

1 Basic function manipulating arrays

The program 2 summarizes the basic functions to manipulate arrays defined as sets. The type of an array is type 'a array = {mutable i: integer; mutable v:'a} set . As for sets, *'a* is a parameter of type, which is the type of the array element. The parenthesis are used on Ocaml to specify a record. Array are sets of records using two fields named *i* and *v*. The keyword mutable indicates that the value of a record field can be modified, therefore the record stores a state. An array is thus a set of "state full" elements which can be updated in place in a for all loop as illustrated by the function *1D_write*. This function return the type unit , because it works by a side effect on the array. The value of an element can also be "consumed", as illustrated in in the function *1D_divide.* The upper half of the array are consumed elements, which are withdrawn from the array. The array retains only the lower half of its elements.

Complexity Concatenating or dividing an array of size n is done in time $O(\sqrt{n})$ which is all right; On the other hand, reading or writing a single cell takes also a time $O(\sqrt{n})$ which is very expensive, compared to *O(1)* which is usually assumed. It forces the programmer to design algorithms where array elements are not accessed sequentially, but processed collectively. This is done by aligning arrays, as shown in the next paragraph.

```
type 'a array = {mutable i: integer; mutable v:'a} set
fun 1D_read A i = one of forall a in A a.v when a.i = i ;;
fun 1D_read: ('a array -> int )-> 'a
fun 1D_write A i v = forall a in A if a.i=i then a.v <- v ;;
fun 1D_write: ( 'a array -> int -> 'a ) -> unit
fun 1D_min A = min of forall a in A a.i ;;
fun 1D_min: 'a array -> int
fun 1D_middle = ( 1D_min A + 1D_max A) /2 i
fun 1D_moy 'a array -> int
fun 1D_divide A = let m=1D_middle A in
        forall a in A consume a when a.i > m
fun 1D_divide 'a array -> 'a array
fun 1D_translat B d = forall b in B b.i<-b.i + d ;
fun 1D_translat: ( 'a array -> int)-> unit
fun 1D_concat A B = 1D_translat B 1D_max A – 1D_min B; A union
B
fun 1D_concat: ( 'a array -> 'a array) -> a array
fun 1D_invert A= forall a in A a.i <- -i
fun 1D_invert 'a array -> 'a array
```

Program 2: Basic functions implementing the 1D_array data structure. If the elements are modified in the forall loop, the result is memorized in the same set. In this way, we can do update in place, for example, the write function directly modifies its argument using a side effect. The function 1D_divide returns the upper half, but has also a side effect on its argument, reducing it to the lower half, because it explicitly consumes the upper half. 1D_concat is written in a functional form for convenience in latter use, it uses the operator union *which makes a set union.*

2 Aligning arrays with extended forall loops

Iterative set division on arrays. Consider a computation on a set, that cannot be directly implemented with a forall loop on its elements. As we have seen with the quicksort example, we can also process sets by iteratively dividing the set in two subsets until the set is partitioned into singletons. In the case of arrays, the function

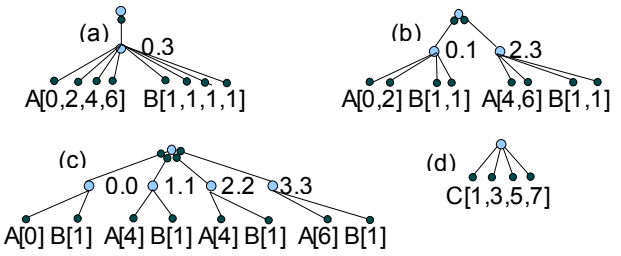
1D_divide of program 2 can divide the arrays in exactly two halves, using the array indexes. Ierative division can be applied simultaneously on two arrays A and B having the same size *n*. One will end up generating *n* sub blobs $b_0, .., b_{n-1}$, where blob b_i contains elements *A[i]* and *B[i]* and can compute in parallel an operation needing both *A[i]* and *B[i]* such as the scalar product, or the sum of vectors, illustrated in program 3. We call this process "*aligning"* A and B.

In order to easily align several arrays, we implement arrays as system data structure and exetend the semantic to forall loops applied on array borrowing the "dot" notation from the SISAL language: *forall* a_1 *in* A_1 *dot .. dot* a_k *in* A_k *exp*; allowing to align multiple arrays A_1 ,... , A_k . The variable a_i binds to A_i's element; it can be omitted in which case A_i will be accessed as an array. We have to omit it, when aligning to arrays of different size, as in the function *1D_output* of program 8. To build an array within a forall loops, the returned element is preceded by the keyword "*array of"* as in the function *1D_sum*. One can align arrays with other types of data structure that support division: a range can be:

- An interval *1,n* as in function *2D_multiply* of program 9.
- A 2X record, as will be explained in program 8
- A set, as in function *set2array*.

To iteratively divide a set, we need to divide it in two halves non-deterministically. We us a prefix computation with the *xor* operator. Every element send *true*, and with the prefix, *n div 2* out of the *n* elements get *true*, and the other *n – n div 2* get *false*. This boolean is then used to polarize the elements. The corresponding code is: fun set_divide S = forall e in S consume e when prefix xor TRUE.

Implementation and complexity. The extended forall uses an iterative set division as in the quicksort execution. It is illustrated in the figure of program 3 : a sub blob b_0 is created. It does the iterative division of all the ranges and divide itself, until the first range is reduced to a singleton. Finally all the descendant of b_0 evaluate in parallel the body expression of the forall. The iterative division of a set of *n* elements has the complexity $O(\sqrt{n})$ and a space complexity of *O(n),* as we have already seen with the quicksort example. Scalar product, sum of vector and concerting a set to an array, all use a forall loop, and have the same complexity. For synchronisation, the situation is similar to quicksort. The master calling the function *1D_sum* can resume execution as soon as the sub blob executing the iterative division has been created.

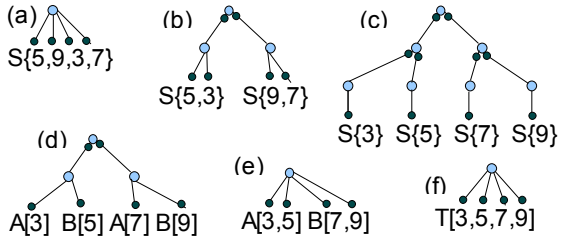



```
fun scalarprod A B = sum of forall a in A dot b in B a*b ;;
fun scalar ( int array * int array )-> 'int
fun 1D_sum A B = forall a in A dot b in B array of (a+b) ;;
fun 1D_ sum ( int array * int array )-> int array
fun set2array S:set = forall e in E array of e
fun 'a set -> 'a array
```

Program 3: The figure illustrate the execution of the code let C = 1D_sum [|0,2,4,6|] [|1,1,1,1|];; *The arrays are divided in two, iteratively, until they have only one element. By default, the forall returns a set. It can be preceded by a reduction operator as in the function scalar prod. To re-build an array, one use the keyword "*array of *" as in 1D_sum. The last function set2array use a set as range.*

3 Parallel divide and conquer

The program 4 present two algorithms using the divide and conquer strategy, where an array is splited in two, and the results are concatenated. The parallelism is specified using the Ocaml construct *let A= exp1 and B =exp2* where the expression *exp1* and *exp2* can be evaluated in parallel. On order to obtain a parallel execution, two sub bobs are created, one evaluates *exp1* and the other *exp2.* The expressions *exp1* and *exp2* contain recursive call which are done on the virgin stack of the two created sub blobs. Incidentally, it avoids to do the recursive calls on the stack of a single blob, which would have contradicted the finite memory hypothesis.



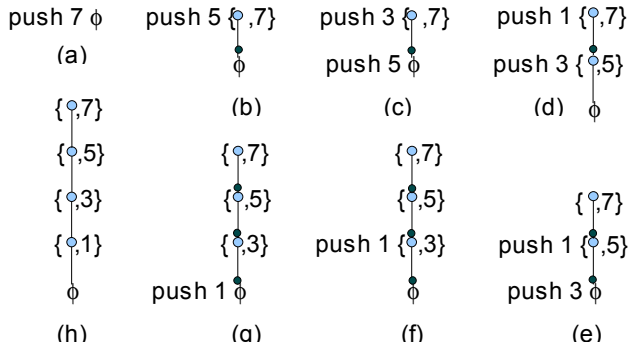

```
fun quicksort S =
let pivot = sum of S / card of S in
if pivot = min of S then set2array S
else let L,R= forall e in S (e when e < pivot , e when e > pivot ) in
 let A = quicksort L and B = quicksort R in 1D_concat A B ;;

fun bitonic_sort S =
 if card S < 2 then set2array S else let T = set_divide S in
 let A = bitonic_sort S and B = bitonic_sort T in
 1D_invert B; bitonic_merge 1D_concat A B ;;
fun bitonic_sort: int set -> int array

fun bitonic_merge A =
 if card A < 2 then A else let B = 1D_divide A in
 let A,B = for all a in A dot b in B (array of min a b,array of max a b)
 in let A =bitonic_merge A and B = bitonic_merge B
 in 1D_concat A B ;;
fun bitonic_merge: int array -> int array
```

Program 4: Divide and conquer parallelism. The figure illustrates the call to quicksort {|3,9,5,7|}. *While the program presented here is easier to write, its execution produces encapsulated blobs of depth log(n) sorting n number. Bitonic _sort is here programmed for list of arbitrary size. Bitonic_ merge divides an array A of size n in two sub arrays A and B which size may differ by one, if n is odd, which will happen when we sort sequence that are not a power of two in length. If n is odd, one of the blob generated by the forall loop will have one A element but no B elements to compare with. The program still works if min and max are programmed so that, if y is undefined: min x y returns x , and max x y returns "undefined", and "undefined" does not generate new elements.*

First the program 4 apply this technic to implements quicksort. We obtain a more compact algorithm, and also more general, since it can sort sets including repeated elements. Secondly, we program the bitonic sort, which as we will show, has an optimal VLSI complexity when run as a blob system on a 2D grid. How does the bitonic sort works? A sorted sequence is a monotonically ascending (or descending) sequence. A bitonic sequence is composed of two subsequences, one ascending and the other descending. Bitonic sequences have the following property that is used in bitonic sort: suppose you have a bitonic sequence of length *2n,* that is, elements in positions [0,2n). You can easily 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). What is this easy method? Simply compare elements in the corresponding positions in the two halves and exchange them if they are out of order: f*or (i=0;i<n;i++) { if (get(i)>get(i+n)) exchange(i,i+n); }*. So here's how we do a bitonic sort: 1- We sort the lower half into ascending order and the upper half into descending order. This gives us a bitonic sequence. 2- We perform a bitonic

merge on the sequence, which gives us a bitonic sequence in each half and all the larger elements in the upper half. 3- We recursively bitonically merge each half until all the elements are sorted.

Complexity The bitonic merge, and the bitonic sort of n elements have a 2D time complexity of $O(\sqrt{n})$ and a space complexity of $O(n)$. Demonstration can be done by recurrence on n. The non recursive part of bitonic merge takes a time t1 < K1* $\sqrt{n}$ and the recursive part, by recurrence, t1 < K2* $\sqrt{n/2}$. It suffice to choose K2 big enough such that the inequality K1+ K2* $\sqrt{1/2}$ <K2 holds. The same reasoning can be repeated on bitonic sort. This is an optimal VLSI complexity as proved in [34] . We do not need to make any hypothesis on data distribution, because unlike quicksort, the array size is always divided by two. There does already exist a sorting algorithm on a 2D grid reaching the optimal complexity [35], 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 $\sqrt[4]{n}$.




```
type 'a queue = { mutable next:'a queue set ;mutable v:'a;
      mutable empty: boolean} ;;
fun push v q = if not q.empty and v< q.v
  then forall q1 in q.next push q1 v
  else begin q.next= {| q |} ; q.v<-v; q.empty=FALSE; end
push ('a -> 'a queue) -> unit,
method pop q= if q.empty then Raise Empty_Stack
    else let v1= q.v and q1= one of q.next in
    q.next=q1.next;q.empty=q1.empty;q.v=q1.v; v1
 pop: 'a queue -> 'a
let q0 = {v=undefined;empty = TRUE; next = NOBLOB} ;;
```

Program5: Using vertical linked list to implement a parallel priority queue. The figure illustrate the execution of push 1 (push 3 (push 5 (push 7 q0)));; *We do not represent the* empty *field. A queue is a reference to a record with three mutable fields. The symbol* ϕ *represents the initial record* q0. *When push is executed on a queue q hosted by a blob b it possibly generates a recursive call in the sub blob q.next of b, which can run concurrently once it is launched, so that another push command can be executed by b. The insertion of a new element* q.next= {| q |} *takes two steps: 1- a new sub blob* b_0 *is created, using the "new blob" instruction, 2- all the fields of the record, including the sub blob q.next are moved to* b_0, *using the "move down" instruction. Both instructions are ready, as table 1 makes precise.*

C Linked list

To distribute a linked list of arbitrary length, one links together a 1D sequence of blobs $b_1, ..., b_n$. The linked list is vertical (resp. horizontal) if the edges between the blobs are vertical (resp. horizontal, using channels). Vertical list are used when the processing is done in a top down hierarchical way, b_i, contains b_{i+1} and is the master of b_{i+1}. Horizontal lists are used when blobs $b_1, ..., b_n$ play a symmetric role, and are updated in a synchronized way.

1 Vertical list

A vertical queue of n elements is coded by an encapsulation of n blobs, as shown by the figure of program 5. The figure illustrates a vertical parallelism (see page 10 for definition). It is a "pipelined" type of parallelism, where at each new time step, a new element can be inserted to the queue.

Synchronization Consider the moment when push is recursivelly called using a forall loop, on the sub blob *q1 in q.next*, that sub blob does not need to keep the ownership of the edge to its outer blob, because the push function does an update in place. Each blob can keep ownership of its sub blobs.

Complexity. Each blob contains only one sub blob. To obtain a space complexity of *O(n)* for representing a queue of n elements, the queue should not be internally represented using the topological representation of encapsulated membranes. One must instead represent the linear network structure shown in program 5. In general, it is not worth representing membranes, as soon as the number of sub membranes in known to be bounded at compile time; binary trees used in program 3 are another example of this rule.

2 Vertical list using Set of objects

The use of set of objects can simplify the preceding program. Sets of objects are enough important to motivate a specific syntax. An object that is systematically used as a set element is declared using a specific syntax "*blob class*", and is called a "blob object" such as the queue of program 6 . The instantiation of a blob object such as *O:= new queue* will first create a sub blob *b*, and instantiate an object of class *queue* within *b*. As a result, a set (singleton) of objects named *O* is created. Because objects have state, we can do update in place within set of objects. Instead of writing *forall o in O o#m exp*, we use the shorter notation *@O#m exp.* This notation is read: "launch method *m* in all sub blobs named *O* with parameter *exp*;" it is comparable to a remote method call. Remote calls can be done at arbitrary depth by chaining "@": *@a@b#m exp launches m* in sub blobs *b* of sub blobs *a*... It is also possible to do a remote call in parent blobs, or through channels although we do not cover this in the present article.

Using self There cannot be any pointer to a blob object, it can only be accessed through remote method call. As a result, a blob object can consistently replace itself using the instruction *self<- exp*, where *exp* must be of the same type as itself. It can also pass a copy of itself, see for example, the instruction *next<-self* of program 6.

Partitioning the code on automata Using blob objects as an other big advantage: the code associated to the method such as push or pop, can be compiled in the blob automaton controling to the blob hosting the object, since only that blob will use it. The code is partitioned and distributed it among the blob objects. One can compile many small automata, rather than a single big automaton including the whole program.

```
blob class ['a] queue
val mutable empty = TRUE
val mutable v:'a
val mutable next = NOBLOB
method push v0 = if not empty and v0< v then @next#push v0;
    else begin next <- self; v<- v0; empty = FALSE end
method pop = if empty then Raise Empty_Stack
     else let v1= v in self <-next; v1
<push 'a -> unit, pop: unit -> 'a>
```

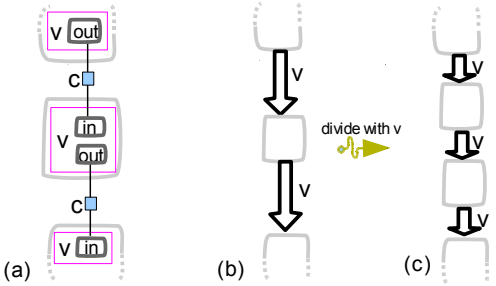
*Program6: Implementing the same queue as preceding program, using blob objects. In Ocaml objects, the data fields are declared with the keyword "*val*". The instruction* next <- self; *has the effect of encapsulating the whole queue in itself, naming it as next. Next is not duplicated when self is passed, because it is not live after that instruction, being obviously redefined.*

Blob primitives implemented as method. Any blob object directly inherit the blob primitives as methods, adapted with a functional syntax and semantic. Consider for example *merge* and *divide*. The call to "*merge b*" merges another blob *b* with self, it must checks that *b*'s type is the same as self and that the sub blob of *self* and *b* have the same identifier, which is true if all sub blobs are data fields

of type *set*. The call to *divide* divides the blob, and return a pair of blob, if called from outside. For example, the instantiation of a *1X* blob object of program 7 returns a pair of blobs, because the initializer of *1X* makes a *divide*.

3 Channel Implementation

Who is the master of a channel? A channel is programmed as a set element, except that it belongs at the same time to the two sets of channels controlled by the left and and the right extremity blob. It can obey to only one extremity, which is called the *master extremity* and is owned by that extremity. Initially, when a master creates a new channel, by convention, the left edge is polarized - and the right +, and the master extremity is the right. Both left and right edges are owned by the master. This is a just a necessary transitory state. In the permanent regime, the channel must own the non master extremity, so as to execute primitives such as delete of duplicate. In the initializer of the 1X class of program 7, the permanent regime is reached in the following way: the master creates the channel, divides, and the negative child flips its left edge back to the channel.




```
blob class 1X
 val c=new channel;
 initializer = divide in nop and in flip left;
 val mutable owner = polarity(out)
 method send m = if owner then @c#send m
 method rec= if owner then rec
 method flip= if owner then @c#flip; owner :=not owner
 method deep_clone=if owner then @c#duplicate; divide; @c#flip;
 method deep_remove= if owner then @c#delete; merge

type 2X= {in:1X; out:1X}
fun 2X_divide c=let l,r=new 1X in ({in= c.in;out=r},{in= l;out= c.out})
fun 2Xdivide: 2X-> (2X* 2X)
fun 2X_send c val = @c.out#send val
fun 2X_rec c = @c.in#rec
fun 2X_shift c = 2X_send c (2X_rec c)
```

Program7: Channel implementation. The class 1X allows to independently polarize (and therefore move) both extremities of the channel. The initializer "divide" ensure that 1X are created by pair, one for each extremities. Because of this divide, a call to new 1X returns a pair of 1X, extremities of the same channel. The flag "owner" stores which of the two extremities is the owner. The type 2X is a record allowing to manipulate two 1X in a coherent way (a) represents an horizontal list of three blob connected using two 2X called v, (b) is a compact notation where channels and 1X blobs are omitted The name of the 2X record directly label the connections, represented as a big arrow from out to in. This representation assume that all the node participating to the list, use the same name for their local 2X value. From (b) to (c), the central blob executes "divide with v" which is a short notation for let v1,v2 = 2X_divide v in divide in v=v1 and in v = v2; The result is to insert a new element in the 1D grid.

Channels primitives. Channels are implemented as objects with a fixed repertoire of 6 methods: *duplicate, delete, send, move, flip, obey*. A remote method call to a channel works like a remote method call to a blob object. Channels are used only as client object: they cannot be inherited from. Thus unlike blob class, there is no need to have channel class. Managing links is always done by triggering a combination of calls to the 6 primitive method, from the master extremity, plus the possibility to directly command the non master extremity. The channel methods implement the channel primitive instructions: *delete* (res. *duplicate*) deletes (resp. duplicates) the channel; *send m* forward a message m to the other extremity which can then call *receive* to receive *m*; The method *move, obey* and *flip* are used for transiting blobs through channels, and for synchronization

Blob movement through channel. The method *move b* migrates a blob *b* to the other extremity. A master b_0 can move blob b_1 through a channel only if it is itself an object, and b_1 is an object field, and since both extremities have always the same type, (channels are allowed to be moved only through merge that preserve the type); type coherence is preserved. The method *obey* make one extremity blob takes it order from the other. Remote call of methods through channels also generates blob movement through channel, when blobs are passed as parameters.

Synchronization and mutual exclusion: The master extremity is changed when the master calls the method *flip*. Synchronization between both extremities is done simply by a back and forth flipping. The non-master extremity is not owned by the blob connected to it. Therefore, that blob cannot execute a blob instruction needing to own the horizontal edge such as division or polarization of horizontal edges . Thus, channels implement an implicit mutual exclusion between the blobs at its extremities. For example, the extremities cannot move simultaneously their channel edges, because movement is done by polarization.

The 1X class The mutual exclusion just mentioned is over constraining. It is possible in principle to polarize and move a channel independently at each extremity, by separately polarizing the edges, without compromising determinism. That could be done using a non deterministic blob primitive allowing the channel automaton to probe whether or not it owns a given extremity; and then get separate commands, and apply polarization independently for each extremity. However, there is a simpler implementation that does not need non determinism to be introduced: one encapsulates each extremity into a 1X blob object defined in program 7, that just propagates commands to the channel. The two 1X associated to a channel can then be polarized and moved independently

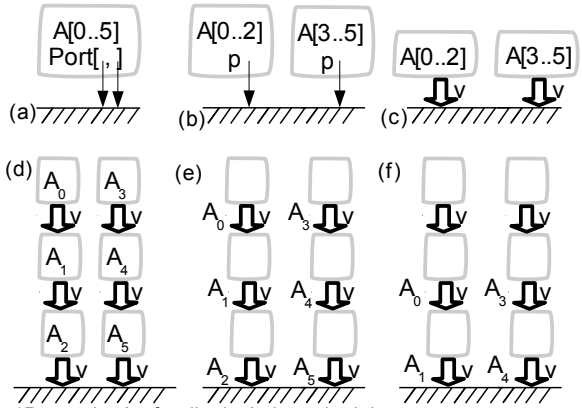
Deep cloning and removal. For blobs using channels, the recursive application of deep_clone and deep_remove mentioned in page 9 must be redefined, to make sure only the master extremity duplicates or deletes the channel. For this purpose, deep_cloning and deep_remove are implemented in blob ml, as object method that can possibly be redefined. In 1X, cloning implies also an implicit synchronization: after the master extremity has duplicated the channel, the non master extremity should also immediately divide while the two channels duplicata are polarized differently. This avoids that two channels have two extremity reaching the same 1X blob. To be able to divide, the non master extremity must temporary be given ownership of the channel. Thus, cloning (and removal) must be issued at each extremity This allows an entire blob hierarchy connected using 1X blob object, to deep_clone or deep_remove when needed. For example, if the horizontal list in the figure of program 7 , is contained in a blob that is deep_cloned, the whole list is deep cloned.

The 2X type for horizontal lists It manipulates two *1X* blob objects within a record, representing an horizontal linked list: the *in* (resp. the *out) 1X* connects to the previous (resp. the next) element. A list of blobs linked using a 2X value called *v* is represented in a compact way, as shown in the figure of program 7. Division of 2X is defined so that a blob of the list can insert a new blob just after it, just by dividing *v* before it divides itself.

4 Horizontal list for input / output

Consider the problem of outputting an array *A* of *n* values through a single port. Using a simple loop that reads the elements one by one would result in a poor time complexity of $O(n\sqrt{n})$ because each read costs $O(\sqrt{n})$. In order to obtain a good complexity of *O(n),* we install an horizontal linked list with 2X records, and shift the array value along that list.

Distributing an array A in a horizontal list. The type 1X and 2X are implemented as system type, since 2X can divide, it is possible to use a 2X value as a range in a forall loop. Consider an initial 2X value $c = \{ in = c_{in} ; out = c_{out} \}$, The iterative successive divisions generated by *forall A dot c* , also operating on *c,* will naturally insert a link between each value of *A* to be outputted, The resulting list will be attached at its extremity to c_{in} and c_{out} .




```
fun 1D_send c A= forall a in A dot c dot i :integer
      send c a; for j = 1 to i do shift c ; done
fun send : int array -> 2X -> unit

fun 1D_output A = forall  p in port dot A
      let v={ in= NOBLOB; out = p} in send v A;
fun output : int array -> unit
```

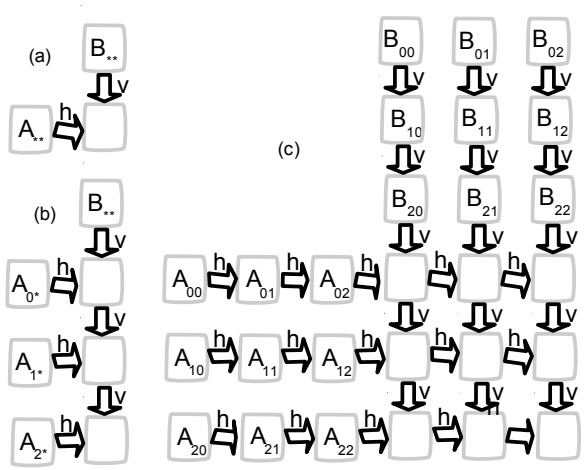
Program8: Parallel output of a 1D array of integers. The first function 1D_send A c sends one by one all the element of A to c.out. The second function 1D_output A outputs in parallel the element of A, using all the available ports. The figure illustrates an execution with an array of 6 elements, through 2 ports: (a) initial situation; (b) the forall loop defined in 1D_output aligns the arrays on the ports, which means making 2 sub arrays, since there are two ports; (c) a 2X record v is created from the port; (d) the forall loop defined in 1Dsend produces an horizontal list of three nodes; (e) the array values are sent through the 2X record, A_2 and A_5 are outputted in parallel. (f) the first of the two necessary shifts is performed by the lowest four blobs, A_1 and A_4 are outputted in parallel.

Using ports. Ports are implemented as system blob objects whose type is compatible with 1X blob object: the methods have the same profile. The method send (resp. rec) result in outputting (resp. inputing) a value through the port. The method deep_remove and deep_clone have no effect on ports, because ports cannot be duplicated or deleted. If one want to use the preceding forall loop to output to a port, if suffices to bind c_{out} to a port. Ports are provided as an array, in a global variable called "port". When a blob containing ports is removed, the ports will automatically be recovered and be reusable under the same global name. When one does parallel computation on a machine, it is of crucial importance to be also able to do parallel input and output on several ports. The program 8 illustrates how to do this using once more alignement of arrays and ports.

D Multidimensional grid and arrays

Multidimensional grid. We have seen how to generate one horizontal list using one 2X record. As a natural generalization, with 2 distinct 2X records *h* and *v*, one can build a 2D grid, by encapsulating 2 forall loops: forall i in 1,n dot v forall j in 1,m dot h exp. Here, the body of the loop: *exp* can use *h* and *v*, to communicate in the 2D grid, along the *x* and the *y* axis. This can be rewritten in a single forall loop sing the keyword "*cross"* borrowed from Sisal that makes a product of range: forall i in 1,n dot v cross j in 1,m dot h exp. First the list along the *y* axis is developed by iteratively dividing *v* , and then each element of that list generates another list along the x axis, by iteratively dividing *h* while deep cloning *v*. As usual, this deep cloning is automatically generated from liveliness analysis, since *v* is used in *exp* . This can be easily extended to grid of *k* dimensions by making the product of *k* ranges.

Multidimensional array. We implement an array *A* of *k* dimensions, and extend the semantic of forall loops, so that forall *B* in *A* will instantiate an array *B* of *k-1* dimensions, by distributing the first dimension. One can also distribute along another chosen dimension instead of the first, as in the function transpose of program 9 which distribute its parameter on dimension 2. The type of nD arrays is a generalization of 1D arrays, the record representing an element is completed with a vector coord of *k* coordinates. In this way, all the functions defined on 1D arrays can be reused, we just need to select the dimension considered using the function *select*_dim.




```
type 'a nD_array = {mutable i: integer; mutable v:'a;
        mutable coord: int vect} set
fun select_dim A k = for all a in A a.i <- a.coord.(k)
fun 2D_send A c = for all B in A 1D_send B c return
fun 2X_pair = 2Xdivide {in= NOBLOB; out = NOBLOB}
fun transpose A = for all B in A:2 return array of B

fun 2D_multiply (A, B) =
 let n= card A and k = card B and B = transpose B in
 let m = card B and (hA,hC) =2X_pair and (vB, vC) = 2X_pair in
 let _ = 2Dsend A hA and _= 2Dsend B vB and result =
 forall 1,n dot vC cross 1,m dot hC
  let c=ref 0 in for i= 0 to k do /* accumulation loop */
  let a= @hC#rec and b = @vC#rec in
  c:=!c + a * b; @hC#send a; @vC#send b; done;
 array of c in result
fun 2D_multiply:(int nD_array*int nD_array)-> int nD_Darray
```

Program9: Steps of the development of the Kung and Leiserson 's circuit for multiplying two matrices A and B.(a) Initially, the parallel let..and.. creates three sub blobs which will develop respectively the data grid containing A and B, and the computing grid containing the result C. (b) the development of the first forall loops along the vertical axis is performed. The B grid has not started to develop yet, because it must synchronize with the horizontal development of the C grid (c) The network is laid out, while A and B just forward their value in a pipelined way, C's elements iteratively receive, multiply, accumulate and forward.

The program 9 uses 2D arrays, and 2D grids to implement the kung and leiserson algorithm for multiplying 2D matrices. It is optimal with respect to VLSI complexity: to multiply two *nXn* matrices, *A* and *B,* it takes a space $O(n^2)$ and a time $O(n)$. The development installs three grids storing *A, B* and the resulting matrix *C.* It takes place first vertically, for *A* and *C* then horizontally for *A,B,* and *C*, and finally, again vertically for *B.* It would be more efficient to interleave steps of horizontal and vertical development, then the tree matrices would be developing simultaneously.

IV Conclusion: paying the price for programmability.

One can summarize the spirit of blob computing as paying space and time to obtain a clean abstraction of a computing medium. At the root of blob computing is the hypothesis of a lower level system able to constantly move the blobs and channels across the hardware so that communicating blobs are place nearby and density of blobs is homogenized This freedom of software from hardware implies a significant extra price of space and time to pay: Each Processing Element (PE), be it a group of clbs in an FPGA, or a small processor, has to be completed with an hardware or a run time system implementing the hardware freedom. This new functionality has indeed to constantly update the position of every blob and migrate them if necessary between adjacent PEs. It will take some fixed percentage of the space, and slows down the execution by some constant factor. Sure enough, if the goal is to exploit the most out of a piece of reconfigurable computing, or a distributed memory multiprocessor, then programming using specific ad hoc circuit or message passing libraries, is going to make the most performance out of it. So, why should one want to pay an extra price for hardware freedom? The answer is that using a simple, clean abstraction such as blob computing can facilitate the job of programming parallel computers, and especially computing media used in the spatial computing community.

In the world of sequential computing, one is already willing to pay a price for programmability. It is now well accepted that programming directly in a machine language is no longer necessary except for very specific case where performance is crucial. One can program in higher level language such as C, but, also in yet more high level object oriented style such as java, which while simplifying the programmer's job, further diminishes the performance. Java run on any system, and this is why people use it. So there is a trade off between performance and programmability which is shifting towards programmability, as the power of our machine augments.

Likewise for parallelism, when sufficiently enough hardware is available, we may be willing to pay the price to obtain portability across different platforms and free the user to worry about the specificity of the parallel architecture and how to map software to hardware. In our framework, the user has to "fold" a task graph into a "self-developing automata network" which is distributed by the machine itself, at run time. A program written using the blob primitives does not make hypothesis on hardware other than its dimensionality, i.e. whether it is 2D or 3D. The same program could run on an FPGA, or a 2D grid coarse grain processors, as long has the system layer, representing blobs and filaments, and implementing the blob primitive, is installed on top of it, and turn it into a virtual blob machine.

Bibliography

1 John Von Neuman Theory of self-reproducing automata, 1966 http://www.walenz.org/vonNeumann/page0111.html

2 T. Lengauer VLSI theory 1990 in Handbook of Theoretical Computer Science

3 H. Abelson D. Allen D. Coore C. Hanson G. Homsy T. Knight, R. Nagpal Amorphous computing *communication of the ACM* 2000

4 Flikkema, P.G. Leid, J.G. Bacterial communities: a microbiological model for swarm intelligence 2005 in Swarm Intelligence Symposium

5 L. Adleman Computing with DNA *scientific american* 1998

6 A. Barabasi Taming Complexity *Nature Physics* 2005 http://www.nd.edu/~networks/

7 A. Dehon et all Stream Computation Organized for Reconfigurable Execution *Microprocessors and Microsystems* 2006

8 B. Gojman E. Rachlin J. Savage Evaluation of design strategies for stochastically assembled nanoarray memo *ACM Journal on Emerging Technologies in Computing* 2005

9 Goldstein, P. Lee, J. Campbell, & Padmanabhan Pillai Scalable Shape Sculpting via Hole Motion 2006 in International Conference on Robotics

10 Tyrrell, Sanchez, Floreano, Tempesti, Mange, Moreno, Rosenberg, Villa POEtic Tissue: An Integrated Architecture for Bio-inspired Hardware 2003 in Evolvable Systems: From Biology to Hardware

11 Andrew Adamatzky Reaction Diffusion Computers, 2005

12 T. Toffoli Programmable matter methods *Future Generation Computer Systems* 1999

13 D. coore A Developmental Approach to Generating Interconnect Topologies, phd thesis 1999 http://www.swiss.csail.mit.edu/projects/amorphous/

14 R. Nagpal Programmable Self-Assembly, phd thesis 2001 http://www.swiss.csail.mit.edu/projects/amorphous/

15 J. Giavitto O. Michel MGS: a Programming Language for the Transformations of Collections, LaMI technical report N° 61-2001 2001 http://mgs.lami.univ-evry.fr/PUBLICATIONS/publicat

16 J.Giavitto Topological Collections, Transformations and Their Application ... in *Rewriting Technics and Applications* 2003

17 J.-P. Banatre and D. Le Metayer. Programming by multiset transformation. *CACM* 1993

18 G. Paun Membrane Computing. An Introduction, 2002

19 L. Cardelli BioAmbients: an abstraction for biological compartments *Theoretical Computer Science* 2004

20 T. S. Ray An evolutionary approach to synthetic biology in *Advances in evolutionary computing: theory and app* 2003

21 J. Dennis First version of a data flow procedure language, 1975

22 Definition: , http://en.wikipedia.org/wiki/Graph_homomorphism

23 Roman Bacik, Sanjeev Mahajan Semidefinite Programming and its Applications to NP Problems in *Electronic Colloquium on Computational Complexity* 1995 http://eccc.hpi-web.de/eccc-reports/1995/TR95-011/

24 S. Jafar Programmation des systèmes parallèles distribués : tolérance aux pannes, .., PhdThesis 2006

25 A. Darte De l'organisation des calculs dans les codes répétitifs, HDR 1999 ftp://ftp.ens-Lyon.fr/pub/LIP/Rapports/HDR/HDR1999

26 K. Kennedy U. Kremer Automatic data layout for high performance Fortran 1995 in 1995 ACM/IEEE conference on Supercomputing

27 M. Arbib, P.Érdi, J. Szentágothai Neural Organization- structure, function, and dynamics, 1998

28 F. Gruau, P Malbos The Blob: A Basic Topological Concept for distributed computation in Unconventional Models of Computation LNCS 2509

29 Definition: , http://en.wikipedia.org/wiki/Mealy_machine

30 Definition: , /wiki/Confluence_%28term_rewriting%29

31 F. Gruau, J. Tromp Cellular Gravity *Parallel Processing Letters* 2000

32 F. Gruau, G. Moszkowski The Blob Division: A hardware-free, time efficient, self-reproduction on 2D 2004 in biologically inspired approaches to advanced ...Pr

33 F. Gruau, Y. Lhuillier, P. Reitz, O. Temam Blob Computing 2004 in ACM conf. on Computing Frontiers

34 C. D. Thomson The VLSI complexity of sorting *IEEE transaction on Computers* 1983

35 F.T . Leighton An Introduction to Parallel Algorithms and Architectures, 1992