Distributed execution of automata networks on a computing medium: introducing IfAny machines.

Gruau Frederic, Luidnel Maignan

Distributed execution of automata networks on a computing medium: introducing IfAny machines.

Frederic Gruau and Luidnel Maignan

Laboratoire de Recherche en Informatique
Bt 650 Universit Paris-Sud 11 91405 Orsay Cedex France
Laboratoire D'Informatique, de Robotique et de Microlectronique de Montpellier
frederic.gruau@lri.fr http://www.lri.fr

Abstract. A computing medium is a set of Processing Elements (PE) homogeneously distributed in space, with connections local in space. PEs are fine grain, and are therefore modeled as Finite State Machine (FSM). In this elementary framework, the interaction between PEs can be defined by a set of instructions, which return a value depending on the neighbor's state. That value is then used as an input to the FSM. This paper studies an instruction set reduced to a single instruction called "IfAny q" that tests IfAny of the neighbors has a given state q. This instruction puts a minimal requirement on hardware: there is no need for addressing channels, communication can be done by local radio broadcasting. An IfAny machine A running on a network tailored for a specific computational task can be executed in parallel on an IfAny medium whose network is fixed and reflects the locality in space. The execution involves an embedding of A's network, and a transformation of A's FSM, adding a 3 states register. We analyse the example of A realizing the addition of n binary numbers. With a carefully chosen network embedding, the resulting parallel execution is optimal in time and space with respect to VLSI complexity.
This work demonstrates that IfAny machines can be seen as a rudimentary programming method for computing media. It represents a first step of our long term project which is to realize general purpose parallel computation on a computing medium.

Keywords: Distributed computing, spatial computing, simulation, embedding, automata network, computing medium, cellular automata

1 Motivation and review

Spatial computing [3] models hardware called *"computing medium"* made of fine grain Processing Elements (PEs), distributed homogeneously in space, and communicating locally between neighbors. Cellular Automata [6] (CAs) is an emblematic example. The locality of communication is nice because it enables arbitrary large medium size. However, it also makes it difficult to abstract space away: most of the problems considered usually involves space both as an input and as an output; A typical example is the computation of the discrete Voronoi

diagram. Our goal is to achieve more general purpose computing, in order to better exploit the potentially enormous power brought by arbitrary scalable hardware. A first intermediate level of abstraction, is to program a virtual network of communicating agents. In particular, when the agents are restricted to logic gates, the programmed object is simply a circuit, and is often used to prove universality results: For example, for the so called "game of life" universal circuits are designed using gliders for signals [8], and for amorphous medium [2], Coore builds arbitrary circuits using gradients and particles. In our view, universality alone is not very meaningful, we consider computational qualities including programming expressiveness, space-time performance, and medium scalability. 1- Expressiveness: Our nodes are agents which are not executing a mere logic gate but an arbitrary Finite State Machine (FSM). 2- Time Performance: computation can be pipelined in space. Large distances between a producer and a consumer of a data augment the latency, which is unavoidable due to the locality of communication. But thanks to pipelining, it does not diminish the throughput. 3- Space Performance: our simulation consists in adding a 3-states register to each agents. In contrast universality demonstration for CA often requires so huge configurations that they are purely theoretical. 4- Scalability: we relax the constraint on cristal regularity and global synchronicity. We consider Amorphous medium [1] defined by scattering PEs in space, and using local broadcast radio communication.

In Coore's work the network can be dynamically instantiated, and this is a fundamental requisite for programming. This work reports only purely static network: the medium is assumed to be initially configured by an external entity in order to execute a desired network which remains fixed. The dynamic instantiation will be modeled through self development [4, 5].

2 IfAny Distributed Finite State Machine

2.1 Computing medium as Quasi-Synchronous Distributed FSM

To force the computation to spread in space, we prefer fine grain over coarse grain medium. This means that the PE's behavior is adequately modeled by a Finite State Machine (FSM). A distributed machine needs a set of instructions I allowing interaction between PEs, it can be modeled as a Moore machine $(\Sigma, Q, \delta, I, \gamma)$ where the input alphabet Σ is the set of possible values returned by the instructions. Q is the set of states, $\delta : Q \times \Sigma \mapsto Q$ is the next state function, $\gamma : Q \mapsto I$ is the output function. A PE executes the instruction $\gamma(q)$ of its current state $q \in Q$, and uses the returned value $v \in \Sigma$ as the input determining the next state $\delta(q, v)$.

Unlike generic asynchronous execution in distributed system, the PEs of a computing medium are identical, and therefore run with the same average speed, although they do not necessarily share the same global clock. A simple way to model those two features is to consider the same discrete time for each PE, but skip the update with a small probability, at each time step for each PE. This schedule that we call *quasi-synchronous* obliges the designer of programs

to engineer specific methods if he needs a behavior which is robust with respect to the non determinism introduced by the random skips.

2.2 IfAny Machine

The instructions `send` and `receive` are often used to exchange messages through point to point communication channels. However, since a computing medium's most important feature is its potential scalability, communicating by radio broadcasting in the local neighborhood is more relevant. IfAny machines takes into account the "rudimentary-ness" of such medium: neither PEs, nor communication channels need to be identified. We will see that a particular sub-class of these machine can indeed be implemented with radio broadcasting.

Definition 1 (IfAny Machine). *It is a distributed FSM with a single type of instructions $\exists Q'$ which tests ifAny of its neighbors has its state in $Q' \subset Q$.*

The atomic instruction of an IfAny machine is a test, therefore it returns a boolean value, and $\Sigma = \{0, 1\}$. In this work, we use IfAny machines both for modeling a computing medium and for programming. For modeling hardware, the network is homogeneous, similar to a 2D grid, and each PE has a bounded small number of neighbors. For programming, the PEs are virtual, we call them agents. The structure of the network reflects a specific pattern of communication needed for a particular algorithm, each agent can have arbitrary many neighbors.

When programming agents interconnected in a static network, we test presence of states among the neihgbors for two purposes: 1- the agent is awaiting a particular source signal $s \in S \subset Q$ to appear; 2- The agent is awaiting for inhibitory states $i \in I \subset Q$ to disappear. Sources model signal propagation, while inhibitors can delay this propagation, in order to make it deterministic despite the random skips. To increase the expressiveness, we consider macro-instruction written $\exists S \neg \exists I$ declaring both a set of sources S and inhibitors I. Let Q_N be the set of states represented amongst the neighbors, the agent executing the macro-instruction is blocked in the same state until there are no inhibitors $Q_N \cap I = \emptyset$ and at least one source $Q_N \cap S \neq \emptyset$. The unblocking of a macro-instruction can be programmed using one state with instruction $\exists I$, a transition 1 leading to itself, and 0 leading to a states with instruction $\exists S$. However, this is true only if the sources in the neighborhood are stable. Fortunately, this will be the case for the subset of machines considered in this paper.

Macro instructions define a new machine where $\Sigma = Q$ instead of $\{0, 1\}$. If a single source appears among the neighbors, $Q_N \cap S = \{s\}$ the transition is called *mono-source*; and we decide that s is the value returned by the macro instruction, i.e. the source is propagated. For *multi-source* transitions, the returned value must be computed from $Q_N \cap S$. We want to keep this computation as simple as possible to be able to execute it in parallel with fine grain PEs. We consider ordered IfAny machines, where an order is defined on Q, and decided that the value returned is the smallest element of $Q_N \cap S$. A mono-source transition, implements pure communication, merely copying states from neighbors, all the

computation has to be done within the PEs. In contrast, a multi-source transition needs to compute a minimum state over arbitrary many neighbors.

2.3 An IfAny online adder.

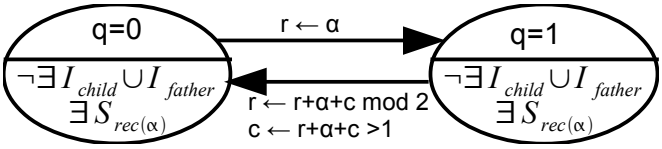

Fig. 1. Implementation of the IfAny adder, where α is a name used to bind the operand.

As an illustrating example, we propose an IfAny machine using only mono-source transitions for solving the computing task of adding p numbers using a binary tree of agents. The numbers are fed bits after bits at the leaves, least significant bits first, each branch agent adds two flows of bits representing integers that are intermediate sums. The result is available bits by bits from the agent at the root of the tree. For simplicity, the machine presented here just describes a permanent computing regime: we do not handle the initialisation nor the termination phase of the addition.

The agent's state is a 5-uplet (d, n, q, c, r) of registers. The registers c, r store bits used in the computations: and q is a control state. The transition graph has only two control states $q = 0$ and $q = 1$ and cycles between the two as indicated by fig. 1. The state $q = 0$ just collects the first input $r = a$, while $q = 1$ collects the second input b, it also updates the carry $c \leftarrow c+a+b \geq 2$ and simultaneously computes the next output bit $r = (c + a + b)$ modulo 2 .

The registers d, n store an address: d is the distance to the root, it can be stored modulo 3, in order to need only tree states, n is the child number: 0 or 1. Both control states use the same instruction parametrized by state registers d, n, q, which includes one set of sources : $S_{\mathrm{rec}(\alpha)}$ and two sets of inhibitors $I_{\mathrm{child}} \cup I_{\mathrm{father}}$. The source is used to collect successively each of the two input bits: $S_{\mathrm{rec}}(\alpha) = \{(d + 1, q, 0, -, \alpha), \alpha = 0, 1\}$, it reads the result α from child number q at distance $d + 1$. Inhibition from the father (at distance $d - 1$) is $I_{\mathrm{father}} = (d - 1, -, (n + q) \bmod 2, -, -)$, it ensures that a result will be used at least once by preventing to go from $q = 0$ to $q = 1$ (resp. from $q = 1$ to $q = 0$) if the father is reading (resp. not reading) the agent's output. Inhibition from the children $I_{\mathrm{child}} = (d + 1, \neg q, 0, -, -)$ ensures that a result is not used twice by preventing to go to $q = i$ if the i^{th} child still has an already used result. The inhibitions and source cause the pair of control states between a father f and its first child c_0 (resp. second child c_1) to follow the 4-cycle: $(q_{c_0}, q_f) = (0, 0) \rightarrow (0, 1) \rightarrow (1, 1) \rightarrow (1, 0)$ (resp. $(q_{c_1}, q_f) = (0, 1) \rightarrow (0, 0) \rightarrow (1, 0) \rightarrow (1, 1)$). In both cases, a father never updates simultaneously with any of its children and the order of updates is deterministic.

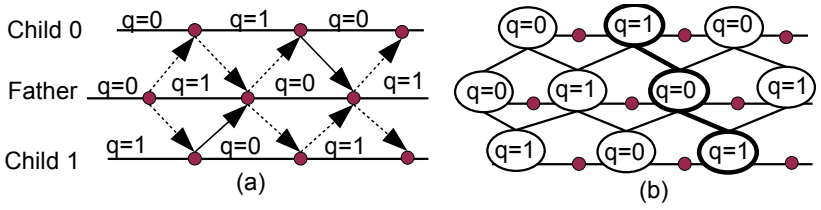

Fig. 2. The IfAny addition machine's execution shown on 3 agents (a) Space time diagram showing inhibitions (dotted arrows) and sources (plain arrows). (b) Execution graph with a reachable configuration in bold

2.4 Source-Deterministic IfAny Machines

In classical distributed computing, the space-time diagram of a distributed execution can be drawn. A horizontal line represents the progress of a particular PE; a dot indicates an event; a slant arrow represents a message transfer which defines causal precedence. We can represent the execution of an IfAny machine, by a similar diagram, as shown in fig. 2 (a). Events are state transitions, there are no message communications, but inhibitions and sources also lead to causal precedence. We decide to draw a slant arrow from an an event e_1 to an event e_2 if e_1 is causally before e_2, which is denoted $e_1 < e_2$ and means that e_1 cannot occur after e_2. A dotted arrows indicate that the state before e_1 was inhibiting e_2, thus e_1 must occur before e_2. Plain arrows indicate that the state after e_1 is a source for e_2. Plain arrows lead to a causal precedence only if the state preceding e_1 was not a source for e_2, and there is a single neighbor which can be source [1].

Recall that the set of sources present in the neighbors determines which transition is taken, while the inhibitions just delay it. Because of quasi-synchronous execution, the sources for a given transition in one run may come too late or leave too early to achieve their effect in another run. The preceding example shows how inhibitions can implement causal precedence preventing these happenings, leading to a deterministic behavior. More generally:

Definition 2. *An IfAny machine with given initial configuration, and FSM is source-deterministic, if whenever an agent in state a is source for a neighbor agent in state b , then* $\text{prec}_a < \text{next}_b < \text{next}_a$.

Here, prec_a is the transition leading to a while next_b is the next transition done after b. $\text{prec}_a < \text{next}_b$ implies that a is present before b's transition, and $\text{next}_b < \text{next}_a$ that it is still present after, together, it proves that a is present during b's transition. The adder is source-deterministic, the diagram 2 (a) clearly shows that the transitions preceding, (resp. following) the source state for a given transition e are causing, (resp are caused by) e. The definition implies that for each transition, the set of sources which is present at the time of the transition

[1] If two sources are present, one could be suppressed without impeding the transition to take place since one source is sufficient for triggering a transition.

is deterministic, therefore the transition itself is deterministic, and the whole execution is deterministic.

Given a deterministic machine, we can build the *execution graph* defined as follows: For each agent a, it has one node for each of a's transition, indexed by the transition number. The node a_t is labeled by the state taken after the t^{th} transition. For each t, a_t, is connected to a_{t+1}, and a_t is connected to $a'_{t'}$ if a and a' are neighbors, and there is a possible configuration of the system, in which agents a and a' have done respectively t and t' transitions. The execution graph of the addition machine is shown in fig. 2 (b). It compactly represents all the reachable configurations which are connected sub-graphs with exactly one node on each horizontal line.

2.5 Implementation of IfAny machines by radio broadcasting.

Our primary concern with medium is to take into account the locality, however, we should not ignore the specific robustness of the IfAny machine allowing to consider communicating by radio broadcast over an anonymous medium. In this case a PE cannot identify the sender of a message. Yet, this rudimentary framework can implement IfAny machines if they are source-deterministic. Each PE broadcasts its state at regular time-interval τ. When a message arrives, the date of arrival is stored together with the message. Messages that arrived more than $\tau + \epsilon$ time ago are discarded. Here, ϵ accounts for slight difference in clock frequency. A PE can maintain an image of the set of state present in its neighborhood, and be sure that each neighbor will be represented, up to a timing precision. Source-determinism imposes that the sources which determine a transition are causally before the transition, and their removal is caused by the transition considered. The lack of time precision does not change the fact that every PEs will safely get its sources, and the transition made by each PE will be deterministic.

3 Execution of IfAny machines on a computing medium

We now want to execute a given programmed virtual IfAny machine P having a specific network of agents (such as a binary tree of adders), on a given computing medium M. Due to the locality of connections in M, the number of links per PEs is upper-bounded by a small constant like 6 for the hexagonal lattice. In contrast the network of P can have arbitrary large degree. The simulation of circuits usually lays out wires for transmitting signals over arbitrary long distances between an operator that produces the signal, and another operator that uses it. We propose a different view in which it is the agent's support which can be arbitrary large in space, so that the support of communicating agents can be directly adjacent in space. For example, if an agent has n neighbors, and has a support without holes, it needs a large support of area $O(n^2)$, so that its perimeter has a length of $O(n)$ and can offer enough border length for touching all the n neighbors. The transmission of signals over wires will be replaced by the propagation of new state within a support.

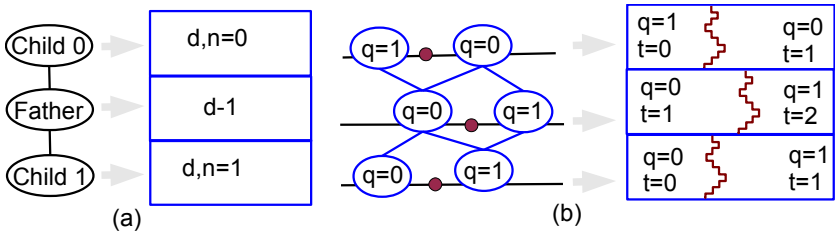

Fig. 3. Mapping on the medium. (a) Embedding the agent network, (b) Embedding a slice of the execution graph.

Definition 3. *An embedding of a network* (V_P, E_P) *in another* (V_M, E_M)*, is a mapping* $\phi : V_P \mapsto \mathcal{P}(\mathcal{V}_\mathcal{M})$ *such that (i)* $\forall a \in V_P, \phi(a)$ *is connected, (ii)* $\forall a_1, a_2 \in V_P$*,* $\phi(a_1) \cap \phi(a_2) = \emptyset$ *(iii)* a_1 *and* a_2 *are adjacent* $\Leftrightarrow$ $\phi(a_1)$ *and* $\phi(a_2)$ *are adjacent*

We will also consider "quasi-embedding" verifying only condition (ii) and (iii). The complement of $\phi(V_P)$ is called the background. Its role is to separate the different supports. Note that a completely connected network (cliques) are not good for targets, they can embed only cliques, because the supports will always be adjacent. Grids with 3 dimensions can embed any networks if they are large enough. Fig. 3 (a) represents an embedding of a binary tree reduced to three nodes, within a planar 2D networks. Adjacency in such a network is equivalent to adjacency in the underlying Euclidian 2D space, the underlying network can thus be omitted in the figure.

The state of an agent a of P is distributed on the PEs of a connected component $\phi(a)$ called its *support*. The coordination of those PEs needs some additional mechanisms. First a PE must be able to identify whether an adjacent PE belong to the same support or not. We define the equivalence relation $\sim$ by $q_1 \sim q_2$ if q_1 and q_2 can be taken by the same agent. The class of q is noted $[q]$. By definition, all the states that can be taken by an agent belong to the same state class, which is the agent's state class.

Definition 4. *An IfAny machine is called state-separated, if any two neighbors always have distinct state class.*

For example, the addition machine has six state classes, one for each value of d and n. Any two neighbors always have distinct value of d, so they are state-separated. Now, let us consider a computing medium M also executing the macro IfAny instructions with sources and inhibitors. We look for a restricted form of universality of M that could be called "network universality": using the same fixed network M can simulate a machine P having an arbitrary embeddable network, but the FSM of M is built according to the FSM of P.

Theorem 1. *An Ifany medium can simulate any embeddable, source-deterministic, state-separated IfAny machine. It needs only 3 times more states.*

Proof: Let P be the machine to simulate including an FSM A_P and a network, and M the medium along with ϕ an embedding of P's network, into M. The FSM A_M governing the medium's evolution is obtained from A_P by "enriching" the instructions and adding a fixed set of transitions. The proof run on two subsections, the first one considering the simpler case of mono-source transitions, and the second one dedicated to multi-source transitions

3.1 Mono-source transition

Because of locality, when simulating a transition for a PE p of P, the PEs of the support $\phi(p)$ cannot acquire the new state simultaneously. For a mono-source transition, only those PEs which are adjacent to the support of the source know what is the new state, and will be able to take it first. Once "born", the new state must then be propagated throughout the support. A PE of M in state q needs to identify whether a neighbor of the same support carries a new state that should be propagated. To this end, we add an "age" component to the state, which is simply the number of transitions already done by the agent, and assign the additional sources $S_{\{q,t\}} = [q] \times \{t+1\}$. The age of two PEs belonging to distinct support is not correlated, therefore, the original sources S (resp. inhibitors I), must be replaced by $S \times \mathbb{N}$ (resp. $I \times \mathbb{N}$) in order to take into account all the possible ages. The "death" of the previous state happens when the new state has invaded the whole support. Note that the presence of an inhibitor does not impede the new state to be born and propagate, but it does impede the previous state to die. The following summarizes the transition function δ_M with two cases for birth and propagation.

$$\delta_M((q,t),(s,t')) = \begin{cases} (\delta_P(q,s), t+1) & \text{if } \neg s \in [q] \quad (birth) \\ (s, t+1) & \text{if } s \in [q] \wedge t' = t+1 \quad (propagation) \end{cases} \tag{1}$$

Several state transition waves of increasing age can be pipelined through the support of an agent. We assign inhibitors $[q] \times \{t-1\}$ in order to isolates the wave aged $t+1$ from the $t-1$, and bound by one the age variation between two neighbor PEs of the same support. The age is needed only for distinguishing $t+1$ from $t-1$, knowing the age modulo 3 is obviously sufficient for this purpose, therefore only t modulo 3 is stored. This explains why only three times as many states are necessary for the execution. In summary, the rule 2 shows how to rewrite a mono-source instruction $q, \exists S \neg \exists I$ of A_P into an instruction of A_M

$$q, \exists S \neg \exists I \Rightarrow (q,t), \exists S_{\{q,t\}} \cup (S \times \mathbb{N})\ \neg \exists I_{\{q,t\}} \cup (I \times \mathbb{N}) \tag{2}$$

In order to prove the correctness, we need to characterize what are the possible configurations on M. Due to pipelined state transition waves, a configuration can simultaneously embed several snapshots of P's execution, as shown in fig. 3 (b). We prove by recurrence the following hypothesis $H(k)$: After k time steps there exist a quasi-embedding ϕ_k from a sub-graph G_k of the execution

graph, such that: (i) for each agent a, G_k contains a single continuous segment $a_t, a_{t+1}, \ldots, a_{t+k}$ whose image by ϕ_k partition the support $\phi(a)$. (ii) the state of PEs in $\phi_k(a_t)$ is (q,t) where q is the label of a_t.

The hypothesis $H(0)$ is trivial at the start; If $H(k-1)$ is true, consider the configuration after $k-1$ time steps, and a given PE, not in the background, with state (q,t), and let (q',t') be the state of a neighbor p'. Condition (iii) of def. 3 is equivalent to the following "neighbor-invariant property": either p' is in a different agent's support, and q' is a possible neighbor state in the simulated execution, or it is in the same support, and $q \sim q'$ and $|t-t'| = 1$. In all cases, the transition of rule 2 is defined and allow to define G_k and ϕ_k from G_{k-1} and ϕ_{k-1} by adding, or deleting nodes and connections; The birth (resp. death) event lead to the adding (resp. deleting) of nodes. Propagation events create (resp. deletes) connection when the border to a neighbor is reached, (resp, totally invaded). A border is usually not invaded in one time step, as a result, a connection moves from G_{k-1} to G_k in a "worm like" manner, it is duplicated first, before being deleted.

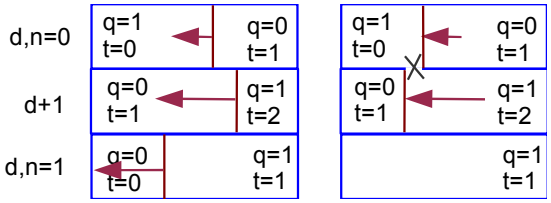

Fig. 4. Race condition: if the state $(q=1, t=2)$ of the father propagates faster than its source in child0 $(q=0, t=1)$, it may become adjacent to the previous state of the source $(q=1, t=0)$ which is forbidden.

G_k verifies (i) by "Reductio ad absurdum": if (i) was not true an entire intermediate component aged t would have updated, including PEs adjacent to the $t-1$ component, which is forbidden by the inhibition $[q] \times \{t-1\}$. In order to fullfil condition (iii) of def. 3 for ϕ_k we need to check the neighbor-invariant property for the new configuration. A problem happens if the next state $(q', t+1)$ propagates faster than its source (s,t') does, some PEs with state $(q', t+1)$ can become adjacent to PEs of the source's previous state $(s', t'-1)$, whereas s' is never adjacent to q'. This is illustrated in fig. 4. Due to source-determinism, the transition from s' to s causes the transition from q to q'. The solution is to apply a prior transformation of S by adding s' as an additional inhibitor of q, which does not modify semantic of S, but will prevent $(q', t+1)$ to propagate to a PE adjacent to $(s', t'-1)$ in the transformed system.

The mapping ϕ_k is a quasi-embedding, which means that it does not check condition (i) of definition 3. The support of a wave may be unconnected because birth events can happen independently throughout the boarder.

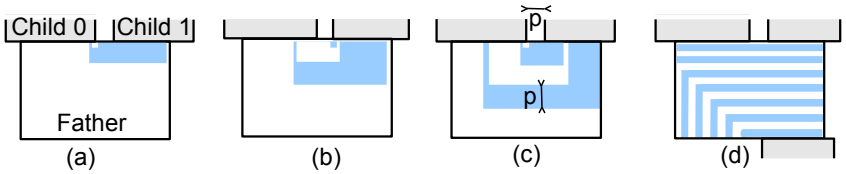

Fig. 5. Propagation of state waves within the support of the root agent: PEs with $q = 0$ (resp. $q = 1$) are colored blue (resp. white); (a)(b)(c) First 3 transitions (d) Buffering happening if the host is not reading.

Figure 5 illustrates the propagation of state waves in the support of the root node of the addition machine. The spatial period p of the waves is equal to the distance between both children, and is inversely proportional to the throughput. If the host is not reading the output, the computation can still go on on the medium, the space behaves as a buffer that stores the bits of the result.

3.2 Multi-source transition.

Consider now an agent a of P in state q, age t, doing an instruction potentially having k multiple sources $s_1, \ldots, s_k$, where the index $r = 1 \ldots k$ is the source rank i.e. $s_r < s_{r+1}$. We note $k_{\max}$ the maximal value of k, for all the possible multi-source transition of any agents. The agents in the support of p must compute the minimum rank among the sources actually present, because it is this value that will determine the transition's input.

Consider a PE p on the border of the support of a, it can compute the ranks of the sources present in its immediate local neighborhood and find out what is the minimum rank r_0, however this is only a local minimum. The computation of the global minimum over the ranks in the whole border needs a centralized processing: the ranks must be input at the leaves of a tree T and then propagated by letting each branch PE computes the minimum of its children. The global minimum $r_{\min}$ will then be available at the PE corresponding to the root of T. We will first assume the existence of such a tree T embedded in the support, whose leaves span the whole border, and then show how to install it.

We need to introduce additional rank states in order to store the rank while they are being propagated . Those states have three components: (c, r, t) where c ranges over the possible state class, and is needed to identify the support of the simulated agent a. $r \in \{1 \ldots k_{\max}\}$ is a possible rank and $t \in \{0, 1, 2\}$ is the age. If the network uses α state classes, the total supplementary number of rank states is $\alpha * k_{\max} * 3$[2].

Agents in the boarder memorise their local minimum r_0 by taking the state $([q], r_0, t)$. The ranks are then propagated within the support through T, We

[2] For the addition machine there is 6 state classes, but setting $c = d$ is sufficient to distinguish supports. A practical use of multiple source is to consider only two sources encoding a 0 and a 1, and compute a logical AND of arbitrary many neighbors. In this case, $k_{\max} = 2$. Thus 18 extra states are needed.

assign the additional sources $S_{min} = \{([q], r, t), r = 1 \dots k\}$ to the sources S_{R1} of rule 2. The input of a branch PE will be the local minimum by using the ordering: $([q], r, -) < (s_r, -) < ([q], r + 1, -)$. In fact, $([q], r, -)$ and $(s_r, -)$ are equivalent with respect to the next state, so $\delta_M((q, t), ([q], r, t)) = \delta_M((q, t), (s_r, -))$. This next state will be $([q], r, t)$ for branch PE. However, for the PE at the root of T, since $r = r_{\min}$, the next state can directly be the final new state $(\delta_P(q, s_{r_{\min}}), t + 1)$. Because of its higher age, this new state will thereafter be propagated by PE throughout the support which are in in a rank state (c, r, t). We reuse the propagation case of rule 1 by regrouping the first two components of rank states: $(c, r, t) = ((c, r), t)$.

A branch node PE should compute its minimum only if all its children have done so. We need to assign an additional inhibitors $I_{\text{not\_ready}}$ to prevent too early computation. In summary, the following rule 3 shows how to rewrite a multi-source instruction $q, \exists s_{i=1\dots k} \neg \exists I$ of A_P into an instruction of A_M.

$$q, \exists s_{i=1\dots k} \neg \exists I \Rightarrow (q, t) \exists S_{R1} \cup S_{min} \neg \exists I_{R1} \cup I_{\text{not\_ready}} \tag{3}$$

Let us now study how to implement the tree T, and compute $I_{\text{not\_ready}}$. First, note that the operation $a, b \mapsto \min(a, b)$ is idempotent: $min(x, x) = x$, a value may be used more than once in the computation of the global minimum, and a spanning Direct Acyclic Graph (DAG) can be used instead of a spanning tree, as long as it has a single root. A simple way to implement a DAG is to designate a leader PE in each support, and compute the hop count d to the leader using a separate distance layer on the medium. The relation between father and children is determined by successive distance values. The tree root is the leader itself. The inhibitors $I_{\text{not\_ready}}$ are agents on the same support, at greater distance, who do not carry ranks.

The computation of the hop count can be done in bounded state using a separate distance layer on the medium, the method is presented in details in [7], it also work in a dynamic environment, the resulting spanning DAG will then be automatically updated if the support and/or the leader move during execution. Note that however, the method assumes a synchronous update, it can nevertheless always be made to work on the quasi synchronous medium using standard synchronisation algorithms. The time cost will be bounded since the number of neighbors is upper bounded in a medium.

3.3 Optimality of the addition machine's execution on a 2D medium

We apply our transformation on the FSM of the addition machine which add n binary numbers, and execute it on a 2D computing medium. We evaluate the performance with respect to the VLSI complexity which states that moving a bit over one unit of space costs one unit of time The modified FSM simply needs an additional register to store the age modulo 3. The network to be simulated is a binary tree, where each node does an addition, bit by bit, and store the intermediate carry. The transition are mono-source and can be pipelined. The frequency at which the bits of the result are produced is the inverse of the

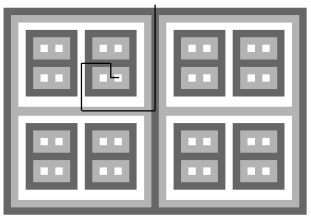

Fig. 6. 2D embedding of a binary tree of 2^k leaves, in $O(2^k)$ space, with constant euclidian distance between brother nodes. We color distinctly $d = 0, d = 1$ and $d = 2$. The bold path indicate the trajectory followed by one bit from leave to root.

the spatial distance between the support of two children. If that distance is constant, (independent of n), the frequency will also be constant. It is possible to map a binary tree with constant distance of only one PE between brothers, by encapsulating membranes, the membranes of the father contains the two membranes of the children as shown in fig. 6. Usually, the inputs are fed on the boarder of a circuit, but this optimised mapping forces the inputs to be fed directly at the right PEs within the medium. On the other hand, the output will be available throughout the whole boarder. The space needed is $O(n)$ and the latency, which is the length of the smallest path from input to output (leave to root) is $O(\sqrt{n})$. In summary, the transformation produces an asynchronous cellular automata rule that does a real computation in optimal time and space.

References

1. H. Abelson, D.Allen, D. Coore, C. Hanson, G. Homsy, Jr. T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.
2. D. Coore. *Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer*. PhD thesis, MIT, 1999.
3. A. Dehon, J.-L. Giavitto, and F. Gruau, editors. *Computing Media and Languages for Space-Oriented Computation 2006*, Dagstuhl international workshop 06361, 2006.
4. F. Gruau. Self developing networks, part 1: the formal system. Technical Report 1549, LRI, 2012. http://www.lri.fr/~bibli/Rapports-internes/2012/RR1549.pdf.
5. F. Gruau. Self developing networks, part 2: Universal machines. Technical Report 1550, LRI, 2012. http://www.lri.fr/~bibli/Rapports-internes/2012/RR1550.pdf.
6. Andrew Ilachinski and Zane. *Cellular Automata: A Discrete Universe*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2001.
7. Luidnel Maignan and Frédéric Gruau. Integer gradient for cellular automata: Principle and examples. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 321–325, Washington, DC, USA, 2008. IEEE Computer Society.
8. J. P. Rennard. *Implementation of Logical Functions in the Game of Life*. Springer-Verlag, 2002.