Annals of Mathematics and Artificial Intelligence 16(1996)285–309

Computations on one-dimensional cellular automata

Jacques Mazoyer[1)]
Ecole Normale Supérieure de Lyon,
Laboratoire de l'Informatique du Parallélisme,
46 Allée d'Italie, 69364 Lyon Cedex 07, France
and *Institut Universitaire de Formation des Maitres de Lyon,*
5 rue Anselme, 69317 Lyon Cedex 04, France
E-mail: mazoyer@lip.ens-lyon.fr

Cellular automata may be viewed as a modelization of synchronous parallel computation. Even in the one-dimensional case, they are known as capable of universal computations. The usual proof uses a simulation of a universal Turing machine. In this paper, we present how a one-dimensional cellular automata can simulate any recursive function in such a way that composition of computations occurs as soon as possible. In addition, this allows us to show that one-dimensional cellular automata may simulate asynchronous computations.

1. Introduction

For a long time, one-dimensional cellular automata have been known as being capable of computations. They can simulate any Turing machine [13], but simulation on a parallel device loses the speed-up induced by parallelism. Various authors have given nice full-parallel algorithms on one-dimensional cellular automata [8] and have studied the computational power of such a network [3]. In fact, cellular automata have essentially been studied as language recognizers. Computing functions on cellular automata induces a definition of how inputs and outputs occur [10]. In this paper, we do not deal with these questions on inputs and outputs. We

[1)]This work was supported by the Programme de Recherches Coordonnées Mathématiques et Informatique and the Esprit Basic Research Action "Algebraic and Syntactical Methods in Computer Science".

always assume that inputs are given to the network on the first diagonal and that outputs occur on the first cell of a semi-infinite line. Giving inputs on the first diagonal is not usual, but in [10], we show that the two usual cases (inputs externally distributed on cells at the initial time, and inputs given sequentially to the first cell) can be quickly reduced to our choice.

In this paper, we aim to define computations on one-dimensional cellular automata in an intrinsic way. Let us first observe how authors set-up cellular automata which have a given computational behavior:

(i) First, they define elementary bits of information (usually the inputs).

(ii) Then, these bits of information are moved, via the informal notion of signal, in such a way that meaningful bits meet themselves in order to generate new meaningful bits (synchronization problems).

(iii) Such previous processes are repeated so as to obtain outputs. This repetition may be a recursive call to the same process or the initiation of a new process.

To define elementary meaningful bits of information and their moves, we need to construct the algorithm itself. In this paper, we define a framework in which the previous construction can easily be done. In particular, we emphasize the notion of signal. Informally, a signal is a state or a bounded set of states. We take "signal" as a basic notion (a meaningful bit of information) and we define the notion of state as a particular signal. This corresponds to the following point of view: in a cellular automata, we emphasize what is communicated by the wires and not what is in the memory of the cells.

In this paper, we present how to use this notion of signal in order to move computations on the space-time diagram of a cellular automaton, to compose computations, and hence to show that any recursive function may be computed "as soon as possible".

2. Computations by signals

2.1. SIGNALS

A one-dimensional cellular automaton is a semi-infinite line of finite automata which interacts. At each unit of time, the cellular automaton, located on cell j, has a state (an element of a finite set Σ), it knows the states of its two neighbors, located on cells $j-1$ and $j+1$, and it computes its new state. Thus, a cellular automaton is only formalized as a finite automaton (Σ, δ) when $\delta : \Sigma^3 \to \Sigma$ is the state transition function. The parallel character of the device is induced by the notion of configuration which is the set of all states of a half line of automata at some time t (a mapping $C_t : \mathcal{N} \to \Sigma$); then the evolution of this parallel device is due to the global function G, which sets up a new configuration from an older one: $C_{t+1}(n) = \delta(C_t(n-1), C_t(n), C_t(n+1))$.

Now let us introduce our notion of signal: During the evolution, a cell receives some information from its two neighbors. We can, usually, make a difference between the state of a cell (element of Σ) and what it sends (element of Ξ). If we consider that a cell has no more state but sends to itself the set of meaningful information, our definition of the state transition function becomes an application of Ξ^3 into Ξ^3 in which we distinguish the information sent to the three possible cells (included itself).

The information sent is made up of various elementary bits of information. We distinguish one part which is the data, computed with the algorithm from the part which controls the execution of the algorithm. In some way, we distinguish the code from the data.

But what may be the (local) code on a line of automata? Looking at a cell as a small processor, we consider that the code indicates the next cell to which to send data. In order to construct small and procedural pieces of code, we consider that the information sent may be split into smaller (locally) irreducible parts called signals. In this way, the previous set Ξ becomes a set of signals $\Xi = \{(\xi_i, d_i) : i \in \{0,\ldots,k\} \text{ and } d_i \in D\}$ (where D is a finite set). We look at ξ_i as the code and d_i as the associated data.

Now what may happen on a cell? Our finite automata receives from itself and its two neighbors three subsets of Ξ and it constructs a new subset of Ξ. We consider that for each signal (ξ_i, d_i), a method to compute the new signal (ξ_i, d_i) sent to the cell itself (denoted by St) is attached to our device, to its left neighbor (denoted by L) and to its right neighbor (denoted by R). Hence, we can replace the state transition function by $3 \times k$ new ones: f_i^m indicates if the signal, numbered i, is sent to the cell indicated by m, in $\{St, L, R\}$, and what data is associated to it. Formally, f_i^m is an application of Ξ^{k+1^3} into $(\{0, 1\} \times D)$; the value of $\{0, 1\}$ indicates if the signal is sent to some cell and the elements of D correspond to the associated data. If we suppose that each signal can occur at any time on any cell, the value of the first component of the functions f_i^m indicates if it is really present. We may view this set of functions f_i^m as the global code of our finite automaton itself.

Another question arises: do the values of the data modify the moves? We distinguish the two cases. If not, we say that we have a cellular automaton with moves independent of data.

To show the evolution of a cellular automaton, we use a space-time diagram. On a quadrant of plan, the point with coordinates (n, t) represents the cell n at time t and we join (n, t) to $(n - 1, t + 1)$ or $(n, t + 1)$ or $(n + 1, t + 1)$ by lines indicating if a signal is sent and what is the carried data. We call a "threadlike signal" a signal that we may visualize as a wire in the space-time diagram. This leads us to the following definition:

DEFINITION 2.1

(1) A cellular automaton (defined by signals) $\mathcal{A}$ is

$$\{D, k, \{f_i^L, f_i^{St}, f_i^R;\ i \in \{0, \ldots, k\}\}\}$$

with

- $D = \{d_0, \ldots, d_\ell\}$ is a finite set of elements (called data),
- k is an integer,
- f_i^L, f_i^{St}, f_i^R are the left, stationary and right transition functions associated to the ith signal. They are functions of $(\{0, 1\} \times D)^{k+1^3}$ into $(\{0, 1\} \times D)$.

(2) A line of automata (defined by signals) $\mathcal{L}_\mathcal{A}$ is made of $\mathcal{N}$ copies of the automaton $\mathcal{A}$. A configuration C of a line $\mathcal{L}_\mathcal{A}$ is an application of $\mathcal{N}$ into $(\{0, 1\} \times D)^{k+1^3}$.
The image of n by C is denoted by $(C_L(n), C_{St}(n), C_R(n))$.

(3) From a configuration C, we get another configuration C^* by the global function G defined (for $m \in \{L, St, R\}$) by:

if $n \neq 0$, $G(C)_m(n) = \{f_i^m(C_L(n-1), C_{St}(n), C_R(n+1))\}$,
if $n = 0$, $G(C)_m(0) = \{f_i^m(\{\emptyset, C_{St}(0), C_R(1))\}$.

In addition, f_0^m is always $(1, d_0)$; we identify the signal labeled 0 as the quiescent signal which always occurs but is never significant (d_0 is the quiescent data).

(4) Thus, a line $\mathcal{L}_\mathcal{A}$ evolves in discrete times, from the initial configuration C_0 (at time 0) to another one C_t (at time t) by $C_{\theta+1} = C_\theta^*$.

(5) When all the functions f_i^m are a couple (σ_i^m, δ_i^m) with:

$\sigma_i^m : (\{0, 1\})^{k+1^3} \to \{0, 1\}$,
$\delta_i^m : D^{k+1^3} \to D$,

we say that moves do not depend on data. In this case, we may express the function σ as a Boolean formula using the variables i_m, indicating if a cell receives the ith signal from itself or one of its neighbors.

(6) The signal i_0 is threadlike if only one of the first components of the $f_{i_0}^m$ has value 1.

2.2. COMPUTATIONS

Now we deal with the notion of computations. Formalizing the computations of functions on a parallel device is not simple. For cellular automata, in the literature we may find two ways to give inputs to the device. The first one distinguishes a cell and data is given sequentially (at each time step) to this particular cell (see [3]). In the second one, at time 0 we distribute inputs on the cells; this may be seen as parallel inputs. In [10], we have shown that in the two cases we may obtain the

same states on the diagonal (cells n at time n). We can arbitrarily choose that our inputs will be given on this diagonal. In our model, a label (a number) and some associated data are given by a complete signal.

Defining where are the outputs is a little more complicated. We briefly recall some kinds of outputs. First, they can appear on a distinguished cell at successive times and this leads us to sequential outputs. Second, they can appear, at a fixed time, on all active cells and this is called parallel outputs; in this case, cells may or may not know that the computation is achieved. Sequential outputs are interesting if our device communicates with a sequential "outside", while parallel outputs are concerned with composition of parallel devices. We decide to consider sequential outputs. Since we have defined signals as (local) code and data, we add to the inputs the information needed to set-up the (local) code. This leads us to the following definition:

DEFINITION 2.2

Let S be a finite alphabet and f a partial function of S^* into S^*, a cellular automaton $\{D, k, \{f_i^L, f_i^{St}, f_i^R; i \in \{0,\ldots,k\}\}\}$ (defined by signal) with $S \subset D$, computes f if there exists an application ϕ_f of $\mathcal{N}$ into $(\{0\ 1\} \times D)^{k+1}$ such that whatever $\bar{x} = x_0 \ldots x_{|\bar{x}|+1}$ in S^* is:

- $\forall i \in \{0,\ldots,k\}$, $\forall n \in \mathcal{N}$, if $i \neq 1$, then the ith component of $\phi_f(n)$ has value $\emptyset$ in its second component. In other words, signals set up by ϕ_f and indexed by a value other than 1 carry no data.
- The data value associated to the jth integer n_j, for which the first component of the 1st component of $\phi_f(n_j)$ is 1, is x_j. In other words, the signal numbered 1 carries the input (the jth input is the data associated to the jth signal 1 which starts from the diagonal).
- We denote $f(\bar{x})$ by $y_0 \ldots, y_m$. Starting with an initial configuration evolving such that, $\forall n \in \mathcal{N}$, $C_n(n)_R$ is $\phi_f(n)$, the jth signal, numbered by 1, appearing in a $C_t(0)$ (its first component is 1) has y_j as associated data.

2.3. AN EXAMPLE: MULTIPLICATION

In order to illustrate the two previous definitions, we give an algorithm to compute the multiplication of two integers using a cellular automaton with moves independent of data.

We use the usual human algorithm with number written in base 2. An example is depicted in fig. 1: numbers are written the most significant bit in first (at the left end) and a special mark * is used to indicate the end of the word. We execute products of the multiplier by successive bits of the multiplicand (the results is either 0 (if the bit is 0) or the multiplier itself (if the bit is 1)); we shift these partial

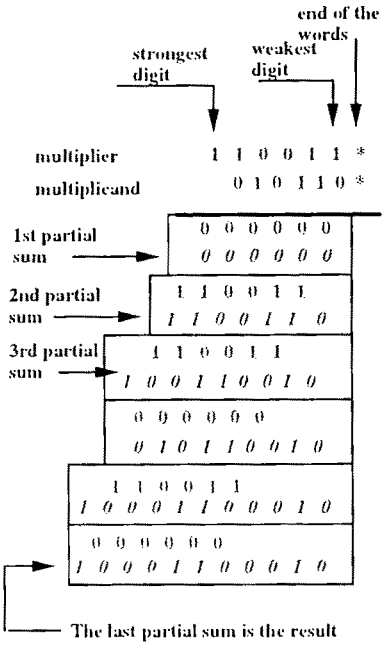


Fig. 1. A human multiplication.

products one position to the left per multiplicand bit. If the usual human algorithm carries out only one sum with as many factors as there are bits in the multiplicand, then we execute one addition (between the current partial sum and the new factors) per bit of the multiplicand. The final result is the partial sum of the last line.

Let $a_0 + a_1 2^1 + \ldots + a_i 2^i + \ldots + a_p 2^p$ and $b_0 + b_1 2^1 + \ldots + b_j 2^j + \ldots + a_q 2^q$ be the multiplier and the multiplicand, the jth partial sum is $c_{0,j} + c_{1,j} 2^1 + \ldots + c_{k,j} 2^k + \ldots + c_{p+j,j} 2^{p+j}$, with

$$c_{k,j+1} = c_{k,j} + b_j a_{k+j} + r_{k-1,j+1}, \tag{1}$$

where $r_{\alpha,\beta}$ is the remainder of $c_{\alpha,\beta}$ by 2.

We choose the following inputs (they allow more readable figures), interliving strings with odd cells for the multplicand and even cells for the multiplier: the jth bit of the multiplicand is given to the cell $2j + 1$ at time $2j + 1$ and the ith bit of the multiplier is given to the cell $2i + 2$ and time $2i + 2$. We easily obtain a cellular automaton in which inputs are given on sites of the diagonal by grouping cells two by two.

The feaure of the algorithm (illustrated in fig. 2) is the following:

- bits of the multiplier always remain on their initial cell;

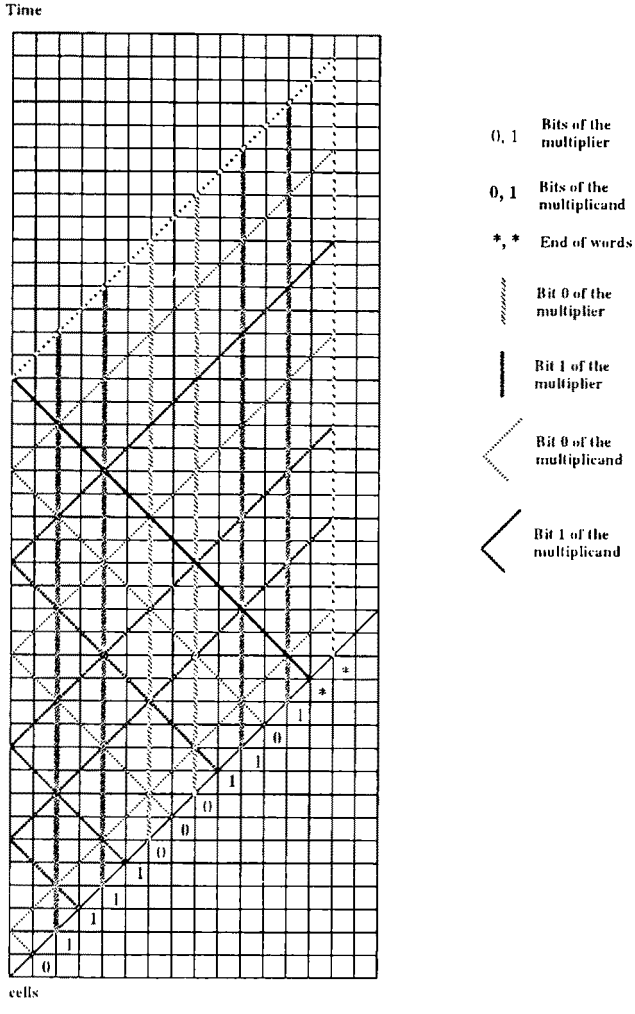
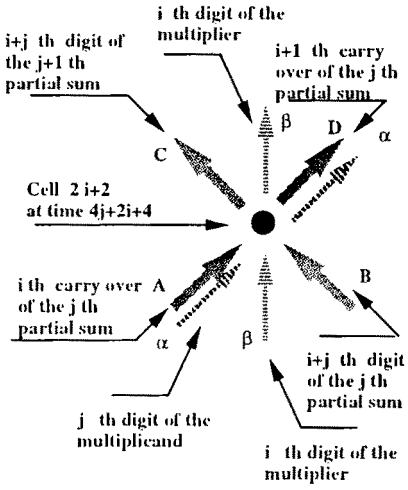


Fig. 2. Moves of bits in a multiplication.

- bits of the multiplicand are first sent at maximal speed (one cell per unit of time) to the left and, after their meeting with cell 0, they return to the right at maximal speed.

In this way, the jth bit of the multiplicand moves (to the left) along the diagonal $\bar{D}_j$, with $\bar{D}_j = \{(k, 2(2j+1)-k);\ k \in \{2j+1,\ldots,0\}\}$. Then it returns (to the right) along the diagonal $\bar{D}_j$, with $\bar{D}_j = \{(k, 2(2j+j)+k);\ k \geq 0\}$. On this diagonal $\bar{D}_j$, it meets the ith bit of the multiplier on the site $(2(i+1), 2(i+1) + 2(2i+1))$. On this site, it computes the values defined in (1), sending $c_{i,j}$ to the



One cell out of two computes one time out of two :
$C = (\alpha \wedge \beta) \oplus (A \oplus B)$
$D = (\alpha \wedge \beta \wedge A) \vee (\alpha \wedge \beta \wedge B) \vee (A \wedge B)$.

Fig. 3. Computations done on one cell out of two, one unit of time out of two.

site $(2(i-1), 2(i+2)+2(2i+3))$ via the left diagonal $\tilde{D}_j$ (and we obtain the left unit shift). This process is initialized: the cell 0 always sends the bit 0 to the sites $(1, 1+2j)$ (with $j \geq 2$) as the value of $t_{-1,j}$. Figure 3 illustrates the computation done on such a site.

Looking at fig. 2, we observe that two (local) codes are needed, corresponding to the moves of the bits of the multiplier and of the multiplicand. We add two others, corresponding to the first diagonal and to the first cell. The functions σ_i^m of definition 2.2 become:

- *Moves of the quiescent signal.* On all inputs, $\forall m \in \{L, St, R\}$, $\sigma_0^m = 1$. The quiescent signal is always present.
- *Moves of the "first diagonal" signal.* If the signal indicating the first diagonal comes from the left neighbor, it is sent to the right neighbor. Thus, on all inputs, we have $\sigma_1^L = 0$ and $\sigma_1^{St} = 0$. But $\sigma_1^R((\ell_0,\ldots,\ell_4), (s_0,\ldots,s_4), (r_0,\ldots,r_4)) = 1 \Leftrightarrow \ell_1 = 1$.
- *Moves of the "first cell" signal.* If the signal indicating the first cell is on a cell, it remains on this cell. Thus, on all inputs, we have $\sigma_2^L = 0$ and $\sigma_2^R = 0$. But $\sigma_2^{St}((\ell_0,\ldots,\ell_4), (s_0,\ldots,s_4), (r_0,\ldots,r_4)) = 1 \Leftrightarrow s_2 = 1$.

- *Moves of the "multiplier" signal.* If the signal indicating the bits of the multiplier is on a cell, it remains on this cell. Thus, on all inputs $\sigma_3^L = 0$ and $\sigma_3^R = 0$. But $\sigma_3^{St}((\ell_0,\ldots,\ell_4), (s_0,\ldots,s_4), (r_0,\ldots,r_4)) = 1 \Leftrightarrow s_3 = 1$.
- *Moves of the "multiplicand" signal.* If the signal indicating the bits of the multiplicand reaches a cell coming from one of its neighbors, it is sent to the other neighbor, except on the first cell. The first cell is indicated by the signal labeled 2 and the multiplicand signal is reflected on this cell. Thus, on all inputs $\sigma_4^{St} = 0$. But $\sigma_4^L((\ell_0,\ldots,\ell_4), (s_0,\ldots,s_4), (r_0,\ldots,r_4)) = 1 \Leftrightarrow r_4 = 1 \wedge s_2 = 0$. And $\sigma_4^R((\ell_0,\ldots,\ell_4), (s_0,\ldots,s_4), (r_0,\ldots,r_4)) = 1 \Leftrightarrow (\ell_4 = 1 \wedge s_2 = 0) \vee (r_4 = 1 \wedge s_2 = 1)$.

In order to define the functions δ_i^m, we use the computations shown in fig. 3. Thus, we consider that the data is a couple. The data associated to the "multiplier" signal is $(\beta, \emptyset)$, where β is a bit of the multiplier. The data associated to the "multiplicand" signal is a couple (α, A), where α is a bit of the multiplier and A is a carry over if the data is carried to the right, a bit of the partial sum else. The new values of A are given by the formulas of fig. 3. In order to indicate the end of the input word, the alphabet used for the values of α, β and A is $\{0, 1, *\}$.

The initial configuration is such that, on the inputs $a_0 \ldots a_p$ and $b_0 \ldots b_q$, the following signals appear:

- At $C_0(0)$, only σ_0^m, σ_1^R and σ_2^{St} have value 1 with data 0.
- At $C_{1+2h}(1 + 2h)$ $(h \in \{0,\ldots,p\})$, $\sigma_4^L = 1$ with data $(a_h, *)$ (* is used to indicate the lack of information). At $C_{3+2p}(3 + 2p)$, $\sigma_4^L = 1$ with data $(*, *)$.
- At $C_{3h}(h)$ $(h \in \{0,\ldots,q\})$, $\sigma_3^{ST} = 1$ with data $(b_h, *)$. At $C_{2+2q}(2 + 2q)$, $\sigma_3^{ST} = 1$ with data $(*, *)$.

The evolution on such a line is depicted in fig. 4. We observe that the outputs defined by the second component of the data of signal 4 reach the first cell one unit of time out of four (this is due to the fact that the bits of the multiplicand are given, on the diagonal, one cell out of two).

3. Grids

3.1. MOVES

In the previous section, we have expressed the possible moves by "go to the right neighbor", "go to the left neighbor", or "stay on the cell". As previously remarked by Čulik [4] and Gruska [5], two moves only are needed: the right and left ones; the stationary one may be expressed by a right one followed by a left one. Any computation occurs on such a grid (see, for instance, fig. 4). If we define another grid, and define the computation as occurring on this new grid, we obtain

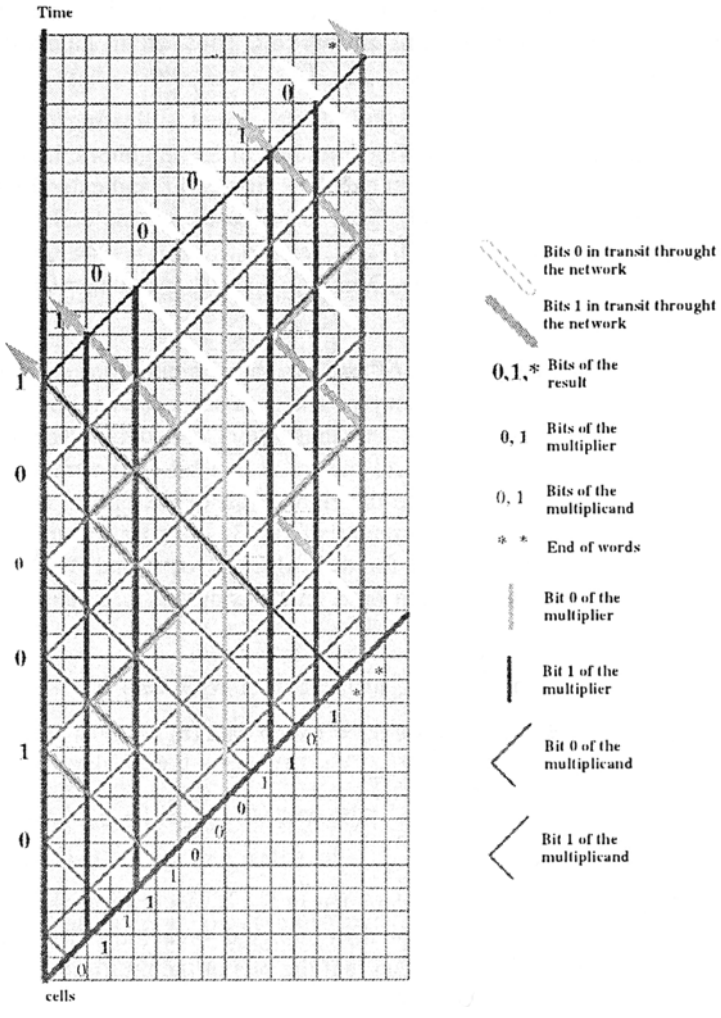


Fig. 4. Multiplying 110011 by 10110.

the same computation but on another area of the space-time diagram. This trick allows us to move the computational area in the space-time diagram.

The first point is to define a new grid (or a new half grid). Figure 5 shows a new grid with regularly distributed holes (non-meaningful cells). In order to define a new grid, we forget that a cell may send information to itself; thus, in definition 2.1 we suppress the function f_i^{St}. We distinguish two signals, numbered by $i_{\rightarrow}$ and $i_{\leftarrow}$, corresponding to the right and left moves. We define the evolution of these two signals $i_{\rightarrow}$ ($i_{\leftarrow}$) in such a way that a left move of $i_{\rightarrow}$ immediately

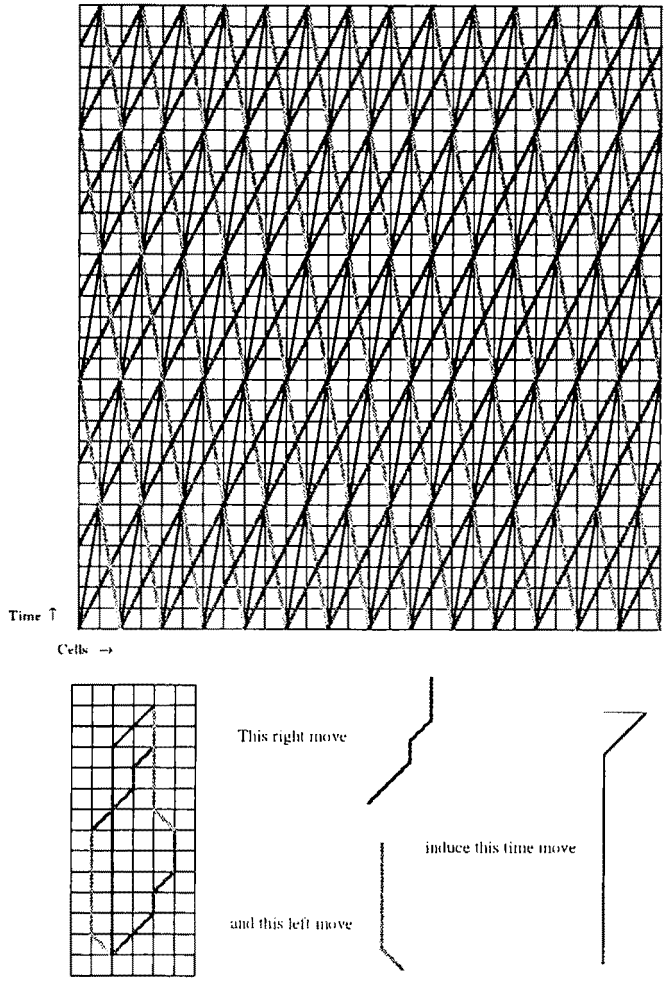


Fig. 5. A regularly twisted grid.

follows a right one, and we consider that a right move, followed by a left one, is a stationary move of $i_{\rightarrow}$ on the initial grid (a left one followed by a right one is a stationary move for $i_{\leftarrow}$). If we want to obtain a half grid, we send from the origin a signal $i_{\uparrow}$ which is only (new) temporal moves (a right move followed by a left one) and its marks the "middle" of the grid. Clearly, this process may be iterated; the two signals $i_{\rightarrow}$ and $i_{\leftarrow}$ may be viewed as the basic signals of the initial grid. This notion of a grid is not easy to formalize. A complete description is given in [10]. If moves depend on data, the best way is to define the data associated with

$i_{\rightarrow}$ and $i_{\leftarrow}$ as a couple, the first component of which is understood as the states of a finite automaton. In the other case, we replace the signals $i_{\rightarrow}$ and $i_{\leftarrow}$ by a finite set of signals. In this paper, we only give an example.

3.2. AN EXAMPLE

As an example, we define the following grid: we use signals 0 (the quiescent one), 1 (the "first diagonal"), 2 (the "first cell"), $\rightarrow_1$ (the first move of the "right move" signal), $\rightarrow_2$ (the second move of the "right move" signal), $\rightarrow_3$ (the third move of the "right move" signal), $\leftarrow$ (the move of the "left signal"), and $\uparrow_1$, $\uparrow_2$, $\uparrow_3$, $\uparrow_4$ (the new "stationary signal").

- Signals 0, 1 and 2 are defined as in section 2.3.
- The signal $\leftarrow$ is defined by $\sigma^R_{\leftarrow} = 0$ and $\sigma^L_{\leftarrow}((\ell_0,\ldots,\ell_{\uparrow}), (r_0,\ldots,r_{\uparrow})) = 1 \Leftrightarrow (r_{\leftarrow} \vee \ell_1) \wedge \neg r_{\uparrow_4}$. That is to say that the signal $\leftarrow$ is created on all sites of the diagonal and then always goes to the left until it meets the signal $\uparrow$. Thus, the signal "left move" is in fact the initial left move.
- The signals indicating the right moves are defined by

  $\sigma^L_{\rightarrow_1} = 0$ and
  $\sigma^R_{\rightarrow 1}((\ell_0,\ldots,\ell_{\uparrow_4}), (r_0,\ldots,r_{\uparrow_4})) = 1 \Leftrightarrow r_{\uparrow_4} \vee r_{\rightarrow_3}$.

  $\sigma^L_{\rightarrow_2} = 0$ and
  $\sigma^R_{\rightarrow 2}((\ell_0,\ldots,\ell_{\uparrow_4}), (r_0,\ldots,r_{\uparrow_4})) = 1 \Leftrightarrow \ell_{\rightarrow 1}$.

  $\sigma^R_{\rightarrow_3} = 0$ and
  $\sigma^L_{\rightarrow 3}((\ell_0,\ldots,\ell_{\uparrow_4}), (r_0,\ldots,r_{\uparrow_4})) = 1 \Leftrightarrow \ell_{\rightarrow 2}$.

  Thus, a right move is created on the temporal move of the first cell and then one new right move is made of two initial right moves followed by a left one.
- The signals indicating "stationary moves" is the "concatenation" of signals $\rightarrow_1$, $\rightarrow_2$, $\rightarrow_3$, $\leftarrow$.

If we put the algorithm of the multiplication defined in section 2.3 on this new grid, we obtain the exchange of information of fig. 6.

3.3. HOW TO CONSTRUCT GRIDS

In [10], we study how to construct a new grid from an older one. The main facts are the following:

1. If the new grid is regular (all its new right and left moves are the same up to a translation), it is always possible to construct it and also to construct its right half part. This construction of a new hole involves only six signals (indicating is the new right (left) move use a right (left), stationary move or corresponds to a wire of the new grid).

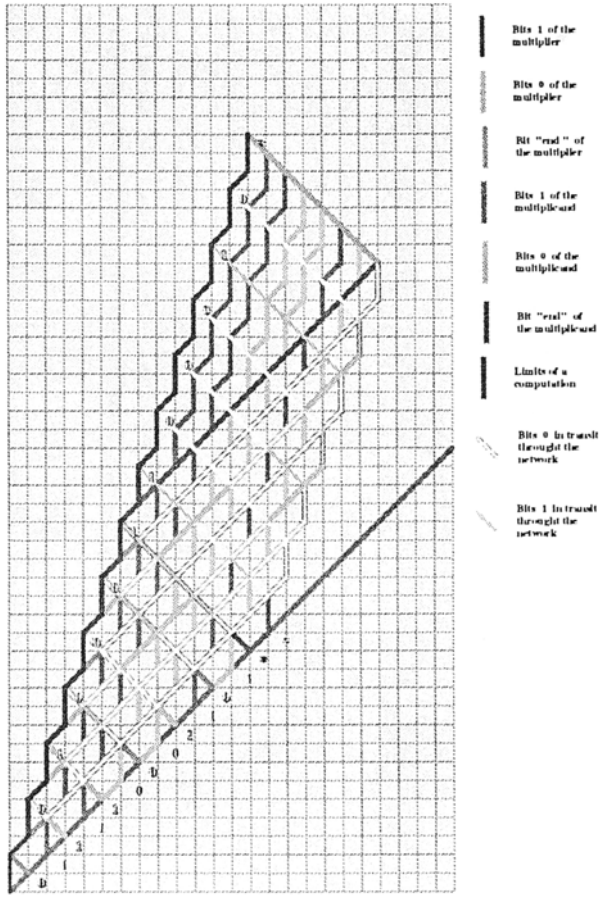


Fig. 6. Computing a multiplication with the outputs on the line $\Lambda_{(0,1)}$.

2. Sometimes (in fact, in the general case) it is possible to define new grids that are not regular. The interesting case is when the left move consists only of previous left moves. And in this case (with few extra conditions), it is possible to construct the new half grid only from indications given on the first new right wire, the construction involving only the area of the new half grid.

3. There exist not regular but recursive grids that are impossible to construct within this area.

All these constructions are long and tedious: they involve technical use of the notion of signal (see [10]).

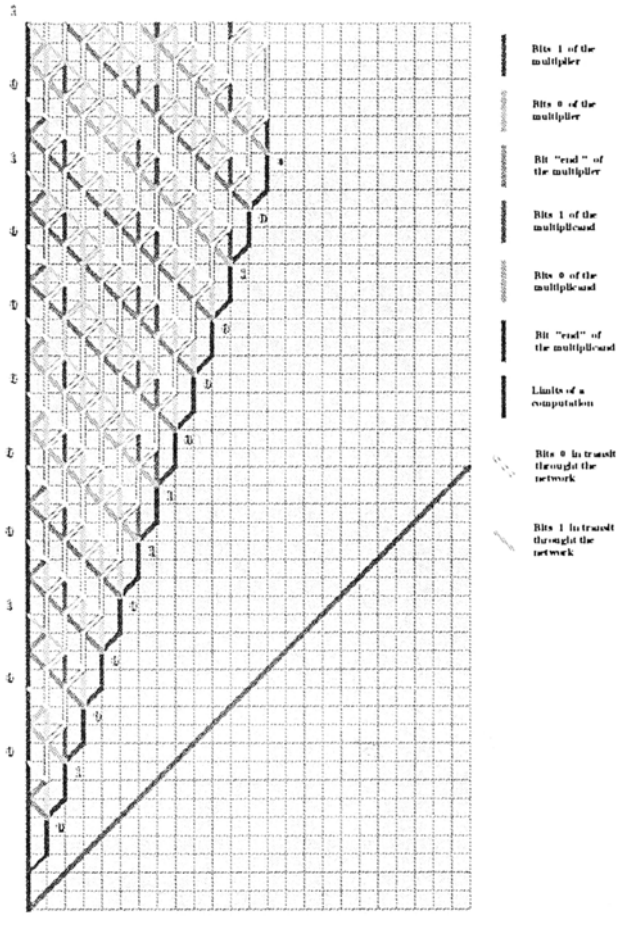


Fig. 7. Computing a multiplication with the inputs on the line $\Lambda_{(0,1)}$.

3.4. GRIDS AND COMPOSITION OF COMPUTATIONS

In fig. 7, we show the algorithm of the multiplication of section 2.3, in which we have indentified the multiplicand and the multiplier in order to get the square function on another grid: the new left move is the initial one; the new right one is made of an initial right move, and initial right move, an initial left move, an initial right move and an initial left move. This new right move corresponds to a stationary move of the grid of section 3.2 and is shown in fig. 6. If we put figs. 6 and 7 in the same picture, we obtain fig. 8, and we see that the new cellular automaton computes the function $(a, b) \rightarrow (ab)^2$.

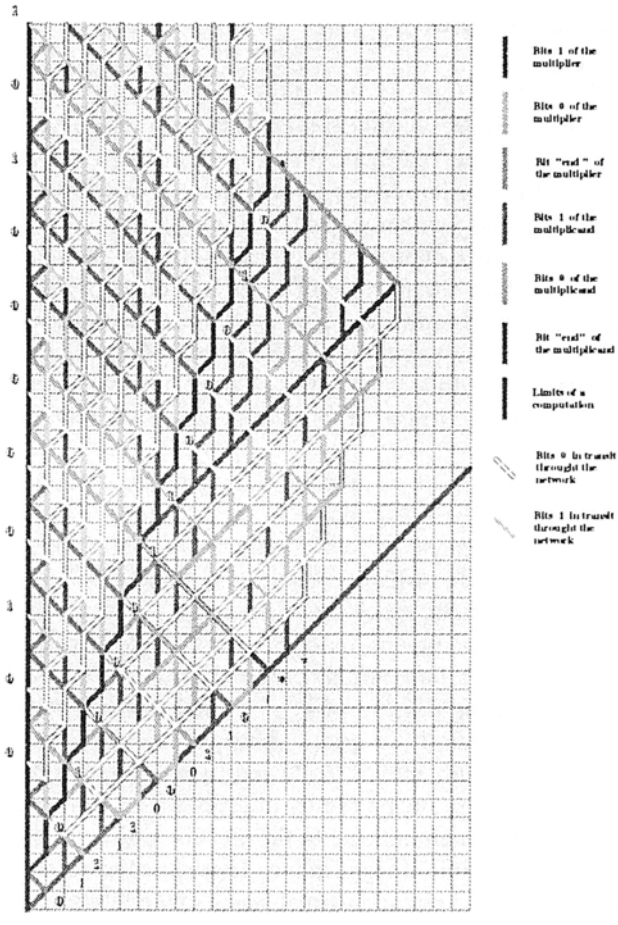


Fig. 8. Computing $(ab)^2$.

This can be generalized: to obtain the composition of the computations of two functions, it is sufficient to move the computation of the first one on a grid between the diagonal and another line (made from the stationary moves of the first cell) and to move the computation of the second one on a new grid (the initial right wire, corresponding to the diagonal is the stationary wire of the first one and its stationary moves are the initial ones). Outputs of the first computation become inputs of the second one. Thus, roughly speaking, to compose functions is to consider that stationary moves become right ones.

In addition, the two examples of figs. 6 and 7 show that we may put the computation in any area of the space-time diagram (between two rational lines).

4. Infinite families of grids

4.1. A CONSTRUCTION

In the previous sectiom, we have indicated that we were able to construct new grids and to put them in any "rational" place on the space-time diagram. The last example shows that it is possible to put a finite number of them in such a way that they correspond to a finite number of compositions.

Here, we aim to construct an infinite number of such grids in order to achieve new computations in which the number of steps is induced by the inputs. How to construct such a family of grids using only a finite number of signals is not obvious. Fortunately, in the studies of the Firing Squad Synchronization Problem (see [2,9,17], for instance), infinite families of lines have been set up. We use this method to set up infinite families of grids. In this paper, we only give one example, that of fig. 9.

We describe the exchange of signals, setting up the family $\Gamma_{n\in\mathcal{N}*}$ of fig. 9. The left space move of the nth grid is the initial left move. The initial right space move of the first grid Γ_1 is made by two initial right moves. In this way, we obtain a stationary move for grid Γ_1, which is made of a (initial) left move followed by two right ones. Then the right space move of grid Γ_2 is made by two stationary moves of grid Γ_1; thus, the stationary move of grid Γ_2 is made by three times a right (initial) move followed by a left (initial) one, all followed by a right (initial) one. Repeating this process (a right move of Γ_{n+1} is made by two stationary moves of Γ_n), we obtain an infinite family of grids. The right move of the nth grid is made by $2^n - 1$ initial right and left moves, followed by an initial right move.

The main idea to set up such a family of grids with a finite number of signals is to send special signals which indicate to the signals setting up the nth right move to finish their move by an initial right move. These special signals are created on any point of the diagonal and run at maximal speed to the left. The signal "first cell" of a grid suppresses one out of two of them.

4.2. AN EXAMPLE: EXPONENTIATION

In order to illustrate the previous construction, we describe how to compute the exponentiation.

We write x and y in binary in basis 2. Let x be $a_0 + 2a_1 + 2^2a_2 + \ldots + 2^ha_h$ and y be $b_0 + 2b_1 + 2^2b_2 + \ldots + 2^kb_k$, we use the usual algorithm, writing x^y as $\prod_{i=0}^{i=k} \rho(b_1, x^{2^{i+1}})$, where $\rho(\alpha, \beta)$ is β if a is 1, 1 else. We denote the value of $\prod_{i=0}^{i=j} x^{2^i}$ by π_j.

Thus, we must make $2k$ multiplications:

- computation of $x^{2^{j+2}}$, which is $x^{2^{j+1}} \times x^{2^{j+1}}$,
- computation of π_{j+1}, which is π_j if $b_{j+1} = 0$ or $\pi_j \times x^{2^{j+1}}$ if $b_{j+1} = 1$.

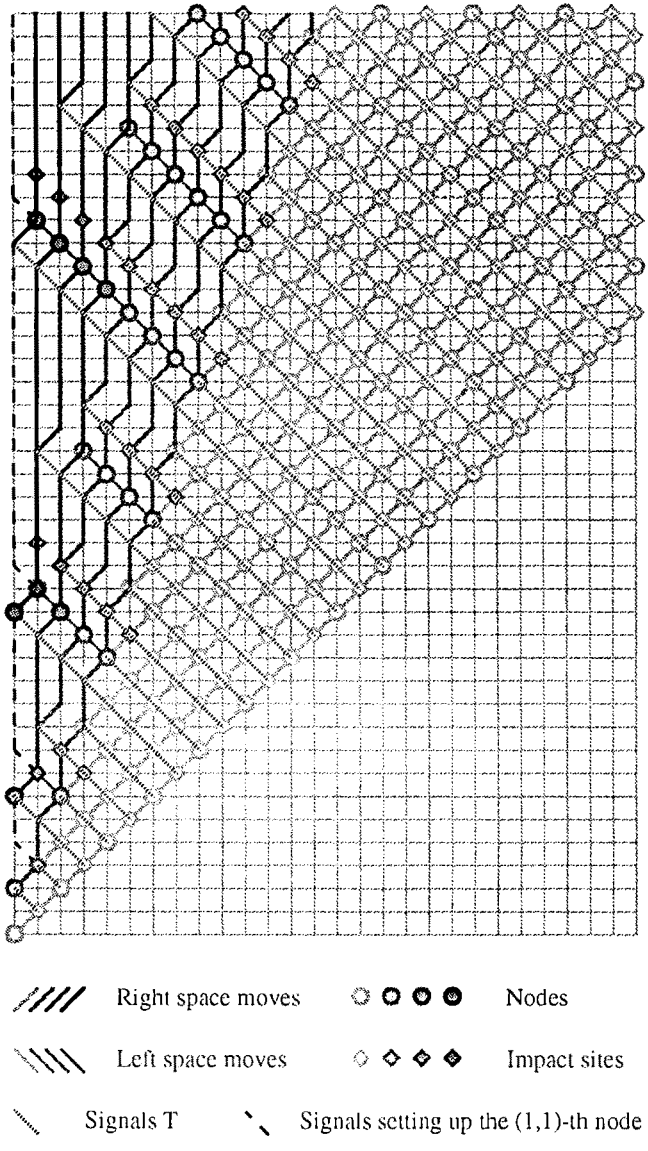


Fig. 9. Setting up an infinite family of regular safe grids (the darkness of the grid indicates its rank).

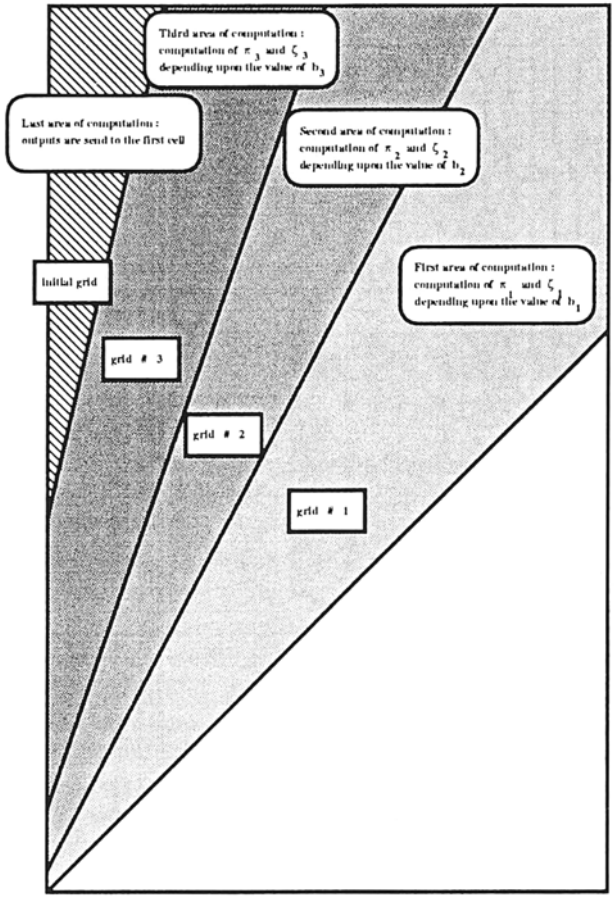


Fig. 10. Areas of computation of exponentiation: the nth area corresponds to the computation of the values of the nth values of η and π in the sequential algorithm.

We observe that we need only one recurrence with k steps and, at most, two multiplications by step. The sequential algorithm is:

$\pi := x$;
$\eta := x$;
for $j := 1$ to k **do**
begin
  $\eta := \eta \times \eta$;
  if $b_j = 1$ **then** $\pi := \pi \times \eta$;
end.

On our infinite family of grids, we put the computation of the nth values of η and π on the nth grid. We give the inputs as in the example of section 2.3. It is easy to obtain the values of x^{2^j}: we start with x as the input on grid Γ_1 and compute x^2 on Γ_1; the result is the input of grid Γ_2 on which is computed x^{2^2} and so on. Computing the values of π_{j+1} is a little more complex; we need to multiply π_j by 1 or $x^{2^{j+1}}$ on the grid $j+1$. So we start the computation on grid Γ_2, and grid Γ_1 only sends x as output. In order to compute π_j (on Γ_{j+1}), all the signals involved in grid Γ_{j+1} must know the value of b_j (choice between the two possible multiplications). Giving the value of b_j to the signals involved in Γ_{j+1} is a little complicated. The idea is to distinguish the first right wire of the grids; and, then, on any grid, bits b_j make a left move followed by stationary moves until they reach the "first cell" signal of the next grid. In this way, all the bits b_j move one cell to the left per grid and the first bit met by the first right wire of a grid gives the needed value b_j, needed by the whole grid.

To achieve the previous process, we mark the second cell (numbered 1 by a signal which has only stationary initial moves; and, when a first right wire of a grid does not meet any bit (of y) on this cell (but the symbol $*$ ending the input word), it stops the process of construction of the family of grids and the least computed value π_k is sent to the first send among all points of this initial wire.

4.3. CONSEQUENCES

The previous computation (we observe that the moves may be viewed as independent of data if we introduce new data indicating that the proces is achieved) is in fact an example of the computation of a function defined by primitive recursion. In [10], we proved the following theorem:

THEOREM 4.1

Any recursive function is computable ("as soon as possible") by a cellular automaton defined by signals with moves which do not depend on the carried data.

In this theorem, "as soon as possible" means that a new computation starts as soon as all the needed data have been computed as in the example of fig. 8. The proof is long and tedious, and quite similar to the example of section 4.2. The only tricky point is to define how inputs of a function with several variables are given and then to reorder these inputs.

We now make some comments. Cellular automata are ("synchronous?") parallel devices. But the definition of recursive functions by composition, primitive recursion and minimization is basically sequential: the two schemes of primitive recursion and of minimization are sequences of the first one: the composition. By our implementation of the composition, we do not need synchronization; more precisely, we do not need to achieve the first computation to start the second. In some ways, our local computations are done "as soon as possible".

Our construction of the composition of functions (fig. 8) induces the construction of a new grid, the right moves of which are the temporal moves of the second one. Thus, the composition of functions can be summarized by

"Time becomes right space"

But this fact induces that the holes of the second grid are greater than the holes of the first one and, thus, we must waste time in the second computation. Pursuing the same idea, if we look at the family of grids of section 4.1, we observe that the size of the holes is exponential in the number of the grid. Decreasing the simulation time is a problem similar to obtaining more efficient infinite families of grids on which the size of holes increase in a polynomial or linear ratio.

5. Moves depending on data

In all the previous sections, we have indicated how to construct a cellular automaton with moves that do not depend on data. When we allow moves to depend on data, we may do some computations which are asynchronous. We give two examples: in the first one, the time of computation may be long; in the second, the time of communication may be long.

5.1. LONG COMPUTATION TIME

We consider the multiplication given in section 2.3, and we set up a (unnatural but convenient) time of computation: the needed duration is the carried bit of the multiplier times the carried bit of the multiplicand plus the carried bit of the partial sum and plus the bit of the carryover.

The first idea is to use a "time delayed" grid (see fig. 11, part b) in which a left move (corresponding to the moves of the bits of the partial sum) is made by a left initial move followed by three initial stationary moves (thus, any computation has, on a cell, a duration of three units of time).

The second idea is: when signals reach a cell on which computations must be done (even machines receiving a bit of the multiplier), looking to the carried values, they wait on this machine until all needed signals have reached it. When all needed information has arrived, the machine knows if the delay of the local computation is of 0, 1, 2 or 3 units of time. Thus, it sends to the right and to the left two signals ϕ_R and ϕ_L, freezing the evolution of the line (with the needed time of computation as data). These signals return to the cell which has created them, after 0, 1, 2 or 3 units of time. In this case, we observe that the bits of the result are obtained in an irregular manner. We see that a long computation time does not imply a delay for all forthcoming computations (for example, in some cases two delays have the same result as only one). We also observe that the bits of the result are obtained faster than in the first process.

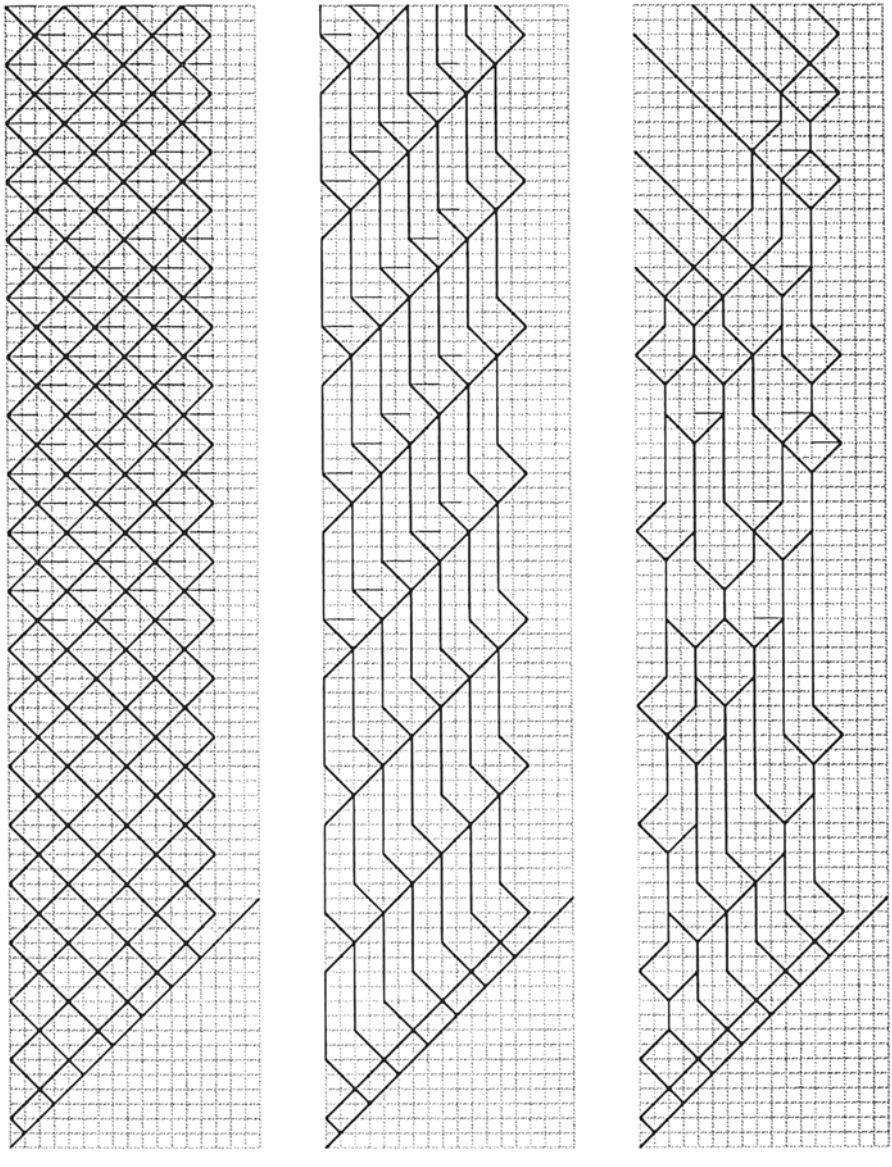


Fig. 11. Multiplication with long computation time. Part a: the grid of fig. 4; part b: the time delayed grid; part c: the dynamics grid.

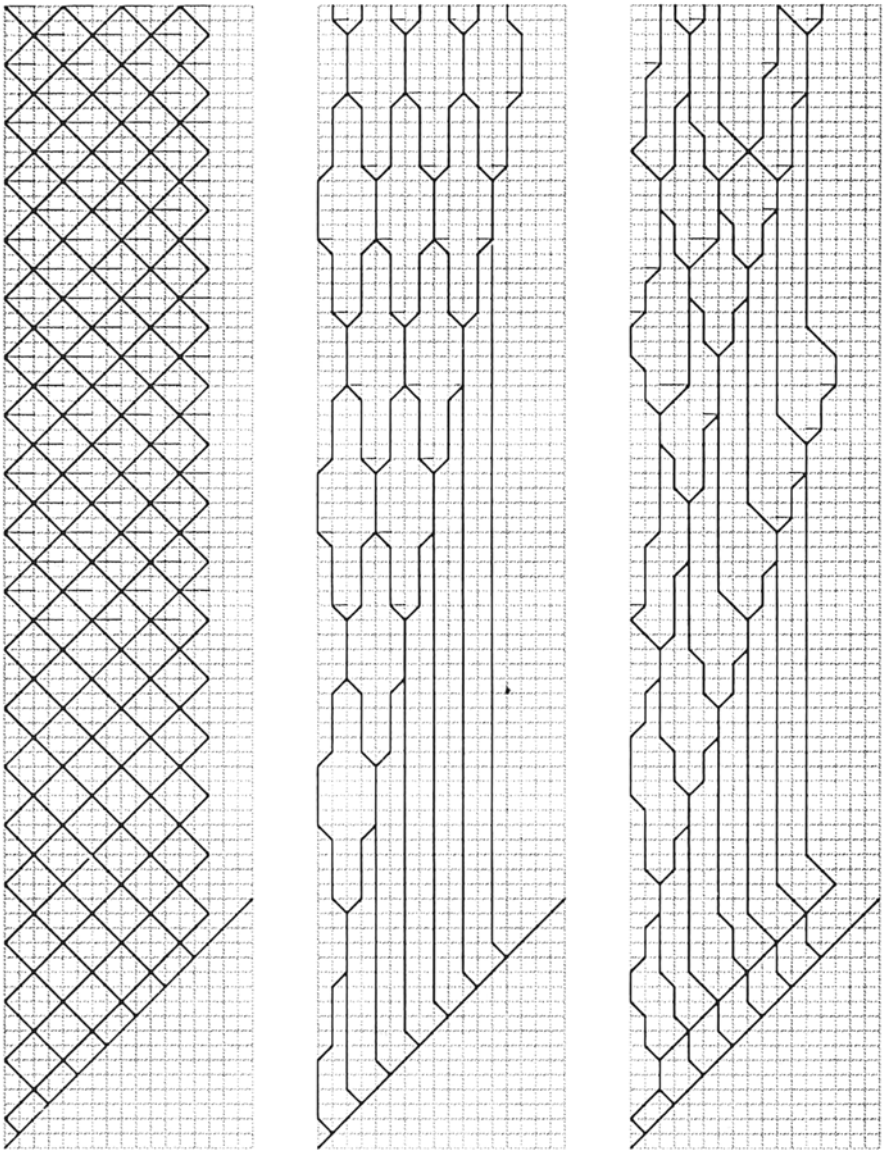


Fig. 12. Multiplication with long communication time. Part a: the grid of fig. 4; part b: the time delayed grid; part c: the dynamics grid.

How to interpret the last algorithm? One way is to consider that the moves of signals always occur on a grid, but that this grid is now set up dynamically and depends on the data (see fig. 11, part c).

Clearly, this example leads to some questions. In this example, two freezing areas never intersect. When such an intersection occurs, we only take the union of the freezing areas. The important question is: may this method be generalized to arbitrary complex algorithms?

5.2. LONG COMMUNICATION TIME

We consider the multiplication given in section 2.3, and we set up a time of communication which is the sum of the carried bits.

The first idea is also to use a "time delayed" grid: right and left space moves are made by a right (left) initial move followed by two initial stationary moves, the maximal possible times (see fig. 12, part b).

The second idea is the following. When signals move between two machines on which a computation must be done (even machines marked by a signal σ carrying a bit of the multiplier), they must move at a speed which depends on the carried data. From a physical point of view, the channel (the wire on which a signal is sent) corresponding to the involved signal is not available during some units of time. As usual, we introduce a new signal, indicating that the channel is not available. If the channel is not available for h times, it must indicate this fact to the signals which may use this signal in the next h units of time, and thus up to the h cell in the opposite direction of its move. Thus, we induce a new signal *Inhibit* with data h on both sides. When the communication is achieved, a new signal *InhibitEnd* is sent on both sides. The area involved by the signals $Inhibit_R$ and $InhibitEnd_S$ may be viewed as a freezing area: all other signals have only temporal moves in this area. We obserse that the bits of the result are obtained in an irregular manner. We see that a long time of communication does not imply a delay for all forthcoming communications (for example, in some cases, two delays have the same result as only one). We also observe that the bits of the result are obtained faster than in the first process.

How to interpret the last algorithm? Also in this second case, one way is to consider that the moves of signals always occur on a grid, but that this grid is now set up dynamically and depends on the data carried by the signals (see fig. 12, part c). Clearly, this example leads to the same question as in section 5.1.

6. Conclusion

In this paper, we have considered that a cellular automaton is made of basic bits of information (called signals), which can move in any "direction".

The notion of "direction" is the one introduced by the treillis automata. This notion allows us to define the moves of the signals on an underlying grid, which defines all the allowed moves. The interest of this notion of grid is due to the fact that we can construct one grid Γ on another, underlying grid Γ^*. All the computations associated with Γ are moved onto this new constructed grid.

By this fact, we consider that a computation defined by signals on the usual treillis is a piece (a "procedure") of computation occurring in some area of the space-time diagram and we can move and twist this area, putting it in another part of the space-time diagram (using Γ^*). The main consequence is that it is possible to define the composition of computations "as soon as possible", the temporal border of the first computation becoming the right border of the second.

However, all computable functions cannot be defined from a finite set of basic function and multiplication; we need to use recursive calls (modelized by the primitive recursion and the minimization). If on a sequential device (a Turing machine, for instance) this recursive call is done on a single machine (with a potentially infinite memory), in our framework in which a machine has only a finite memory, this recursive call is obtained by a (potentially) infinite number of grids. We use the possibility to put an infinite number of grids in a quarter of the plane and we may define an infinite family of grids. Using this trick, we may achieve computations involving primitive recursions and minimizations. Finally, we obtained the possibility of finding a direct, uniform, computable translation of any recursive scheme in an automaton computing the same function.

How much does the parallel device (defined by cellular auromata) speed-up the sequential time of computation? From our point of view, this depends on the number of recursive calls and on the different infinite family of grids we may define. In this paper, our infinite family of grids is very expensive: the size of holes grows exponentially with the index of the grid. We do not know if it is possible to define an efficient infinite family of grids that grows polynomially with its index.

All previous facts have been set up with a strong constraint: our signals move carrying some data and this data never interacts with the moves of the signals. Thus, we must explicitly construct all the grids that may be used in the execution of the algorithm To construct all these grids requires the prediction of all possible evolutions of the algorithms on all possible data, and this fact may lose a lot of time. In the last section, we study examples of the multiplication when the computational time or the communication time depends on the carried data. The idea is to associate to any signal some "flag" signals, indicating to the outside that some area is not available (it is "frozen"). It may be interesting to generalize these ideas in the general case. From two points of view: first, when both computation and communication take more than one unit of time; and second, when the needed time is not known "a priori". From the second point of view, the feature is to distribute the input one node out of two, which allows us to have signals faster than the inputs and to send a "freezing" signal killed by a faster "unfreezing" signal. Such a study remains to be done.

References

[1] A.J. Atrubin, An iterative one-dimensional real-time multiplier, Term Paper for App. Math. 298, Stanford University (1962).

[2] R. Balzer, An eight-state minimal time solution to the Firing Squad Synchronization Problem, Inf. and Contr. 10(1967)22–42.

[3] S. Cole, Real-time computation by n-dimensional arrays, IEEE Trans. Comp. 4(1969)349–365.

[4] C. Choffrut and C. Culik II, On real-time cellular automata and trellis automata, Actae Informaticae (1984) 393–407.

[5] C. Čulik II, J. Gruska and A. Salomaa, Systolic treillis automaton (for VLSI), Research Report CS-81.34, Department of Computer Science, University of Waterloo (1981).

[6] C. Čulik II, Variation of the Firing Squad Synchronization Problem, Inf. Proc. Lett. (1989) 152–157.

[7] C. Dyer, One-way bounded cellular automata, Inf. and Contr. 44(1980)54–69.

[8] P.C. Fisher, Generation on primes by a one-dimensional real-time iterative array, J. ACM 12(1965)388–394.

[9] J. Mazoyer, A six-state minimal time solution to the Fring Squad Synchronization Problem, Theor. Comp. Sci. 50(1987)183–238.

[10] J. Mazoyer, Computations on one-dimensional cellular automaton: Extended version, Preprint (1993).

[11] J. Mazoyer and V. Terrier, Signals on one-dimensional cellular automata, Preprint.

[12] N. Reimen and J. Mazoyer, A linear speed-up theorem for cellular automata, Theor. Comp. Sci. 101(1992)59–98.

[13] A.R. Smith, Cellular automata theory, Technical Report 2, Stanford University (1960).

[14] A.R. Smith, Real time language recognition by one-dimensional cellular automata, J. ACM 6(1972)233–235.

[15] M. Minsky, *Finite and Infinite Machines* (Prentice–Hall, 1967) pp. 28–29.

[16] R. Péters, *Recursive Functions* (Akadémiai Kiadó, Budapest, 1967).

[17] A. Waksman, An optimal solution to the Firing Squad Synchronization Problem, Inf. and Contr. 9(1966)66–87.