A speed-up of oblivious multi-head finite automata by cellular automata

Alex Borello[1], Gaétan Richard[2], and Véronique Terrier[2]

1 **LIF (Laboratoire d'Informatique Fondamentale)**
39, rue Frédéric-Joliot-Curie, 13453 Marseille, France
alex.borello@lif.univ-mrs.fr

2 **GREYC (Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen)**
Campus Côte-de-Nacre, Boulevard du Maréchal-Juin, 14032 Caen, France
gaetan.richard@info.unicaen.fr
veronique.terrier@info.unicaen.fr

Abstract

In this paper, we present a parallel speed-up of a simple, yet significantly powerful, sequential model by cellular automata. The simulated model is called oblivious multi-head finite automata and is characterized by the fact that the trajectory of the heads only depends on the length of the input word. While the original k-head finite automaton works in time $O(n^k)$, its corresponding cellular automaton performs the same task in time $O(n^{k-1}\log(n))$ and space $O(n^{k-1})$.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases oblivious multi-head finite automata, cellular automata, parallel speed-up, simulation

Digital Object Identifier 10.4230/LIPIcs.STACS.2011.273

Introduction

Cellular automata (CA for short) are recognized as a major model of massively parallel computation. Their simple and homogeneous description as well as their ability to distribute and synchronize the information in a very efficient way contribute to their success. However, to determine to what extent CA can fasten sequential computation is not a simple task.

As regards specific sequential problems, the gain in speed by the use of CA is manifest [1, 2, 3]. But when we try to get general simulations, we have to face the delicate question of whether parallel algorithms are always faster than sequential ones. An inherent difficulty arises from the fact that efficient parallel algorithms make often use of techniques radically different from the sequential ones. Also there might exist a faster CA for each singular sequential solution whereas no general simulation exists.

Hence, no surprise: for Turing machines, model of sequential computation, the known simulations by CA provide no parallel speed-up. The early construction of Smith [9] simulates one step of the Turing machine by one step of the CA. Furthermore, no faster simulations have been reported yet even for restricted variants. In particular, we do not know whether any finite automata with k heads can be simulated on CA in less than $O(n^k)$ steps, which is the sequential time complexity.

In a step toward addressing such issues, we shall examine here a simple sequential model, called data-independent multi-head finite automata. This device was introduced by Holzer in [6] as multi-head finite automata with an additional constraint of obliviousness: the

28th Symposium on Theoretical Aspects of Computer Science (STACS'11).
Editors: Thomas Schwentick, Christoph Dürr; pp. 273–283

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


trajectory of the heads only depends on the length of the input word. As emphasized in [6], such finite automata lead to significant computational power: they characterize parallel complexity NC^1 – which explains they can be efficiently parallelized. Their properties have been further discussed in [7]. We propose below a simulation of these data-independent multi-head finite automata by CA which gives rise to an efficient parallel speed-up.

This paper is organized as follows. The next section introduces the two models considered. Section 2 displays some of their features and abilities. Section 3 presents the simulation algorithm and its time and space cost.

1 Definitions

1.1 Multi-head finite automata

Given an integer $k \geq 1$, a two-way k-head finite automaton is a finite automaton reading an input word written between two end-markers using k heads that can move in any direction provided they do not go beyond these markers.

▶ **Definition 1.** A (deterministic) *two-way multi-head finite automaton* (2DFA(k) for short) is a tuple $(\Sigma, Q, \triangleright, \triangleleft, q_0, q_\mathrm{a}, q_\mathrm{r}, k, \delta)$, where Σ is a finite set of *input symbols* (or *letters*), Q is a finite set of *states*, $\triangleright \neq \triangleleft \notin \Sigma$ are the left and right *end-markers*, $q_0 \in Q$ is the *initial state*, $q_\mathrm{a} \neq q_\mathrm{r} \in Q$ are respectively the *accepting state* and the *rejecting state*, $k \geq 1$ is the *number of heads* and $\delta : Q \times (\Sigma \cup \{\triangleright, \triangleleft\})^k \to Q \times \{-1, 0, 1\}^k$ the *transition function*; -1 means to move the head one letter to the left, 1 to move it one letter to the right and 0 to keep it on its current letter. For the heads to be unable to move beyond the end-markers, we require that if $\delta(q, a_1, \ldots, a_k) = (q', m_1, \ldots, m_k)$, then for any $i \in [\![1, k]\!]$, $a_i = \triangleright \Rightarrow m_i \geq 0$ and symmetrically $a_i = \triangleleft \Rightarrow m_i \leq 0$.

A *configuration* of a 2DFA(k) on an input word w at a certain time $t \geq 0$ is a couple (p, q) where $p \in [\![0, |w| + 1]\!]^k$ is the position of the multi-head and q the current state.

The computation of a 2DFA(k) on an input word $w \in \Sigma^n$ starts with all heads on the left end-marker, and ends when the automaton reaches the accepting or the rejecting state. In the former case, the word is said to be accepted, while in the latter it is rejected. For some words, none of these cases happen and hence the automaton will enter a loop eventually. The language $L(\mathcal{F})$ *recognized* by a 2DFA(k) $\mathcal{F}$ is the set of the words accepted by $\mathcal{F}$. One can notice a 2DFA(k) necessarily enters a loop if it has not accepted nor rejected the input at step $|Q|(n+2)^k$ steps, which is the number of configurations featuring w; so, we say the computation is over if we reach this step (it may of course take less).

We will focus now on the data-independent 2DFA (2DIDFA), a particular class of 2DFA for which the path followed by the heads only depends on the length of the input word, not on the letters thereof.

▶ **Definition 2.** Given $k \geq 1$, a 2DFA(k) $\mathcal{F}$ is said to be *oblivious* (or *data-independent*) if there exists a function $f_\mathcal{F} : \mathbb{N}^2 \to \mathbb{N}^k$ such that the position of its multi-head at time $t \in \mathbb{N}$ on any input word w is $f_\mathcal{F}(|w|, t)$.

1.2 Cellular automata

A cellular automaton is a parallel synchronous computing model consisting of an infinite number of finite automata called *cells* which are distributed on $\mathbb{Z}$ and share the same transition function, depending on the cell itself and its two neighbors.

▶ **Definition 3.** A *cellular automaton* is a tuple $(\Sigma, Q, \#, q_a, q_r, \delta)$, where Σ is the finite set of *input symbols* (or *letters*), $Q \supset \Sigma$ is the finite set of *states* and $\delta : Q^3 \to Q$ the *transition function*. $\# \in Q \setminus \Sigma$ is a particular quiescent state, verifying $\delta(\#, \#, \#) = \#$. $q_a \neq q_r \in Q$ are respectively the *accepting state* and the *rejecting state*. These are *persistent*, which means that a cell in such a state will never switch to another state: for any q,$q' \in Q$, $\delta(q, q_a, q') = q_a$ and $\delta(q, q_r, q') = q_r$.

A *configuration* is a function $\mathfrak{C} : \mathbb{Z} \to Q$. A *site* is a cell at a certain time step of the computation; $\langle c, t\rangle$ will denote the state of the site $(c, t) \in \mathbb{Z} \times \mathbb{N}$. The computation of a CA $\mathcal{C}$ on an input word w of size $n \geq 1$ starts at time 0 with all cells in state # except cells 1 to n where the letters of the word are written. This is the initial configuration $\mathfrak{C}_w$ associated to w. Then the cells update in parallel their respective states according to δ: for all $(c, t) \in \mathbb{Z} \times \mathbb{N}$, $\langle c, t+1\rangle = \delta(\langle c-1, t\rangle, \langle c, t\rangle, \langle c+1, t\rangle)$.

The input word is accepted (resp. rejected) in time $t \in \mathbb{N}$ if and only if cell 1 enters the accepting state q_a (resp. the rejecting state q_r) at time t (and hence at any time $t' \geq t$). The language $L(\mathcal{C})$ *recognized* by the automaton is the set of the words it eventually accepts. $L(\mathcal{C})$ is said to be recognized in time $\tau : \mathbb{N} \to \mathbb{N}$ if and only if any word w is accepted or rejected in time $\tau(|w|)$.

2 Preliminaries

Our concrete question is the following: How long does it take to simulate a data-independent k-head finite automaton on CA? Regarding general multi-head finite automata, one can recall that a 2DFA(k) can be simulated on a deterministic multi-tape Turing machine in time $O(n^k)$, where n is the length of the input word [10]. Besides, it is well-known that CA are able to simulate in real time any deterministic (even multi-tape) Turing machine [9, 4]. A simple simulation consists in representing each tape of the Turing machine by two stacks – several stacks can be simulated simultaneously by a CA without loss of time. An upper bound in $O(n^k)$ for the time required by a CA to simulate a 2DIDFA(k) follows immediately. Now, how to reduce this time bound? As yet, no parallel speed-up is known to simulate Turing machines on CA; and no faster simulation of DIDFA on Turing machines taking the obliviousness constraint into account has been proposed. Here we will present a direct simulation which will take advantage of the oblivious feature of the DFA and so allow us to parallelize its computation.

2.1 Facts about multi-head finite automata

Let $\mathcal{F} = (\Sigma, Q, \triangleright, \triangleleft, q_0, q_a, q_r, k, \delta)$ be a 2DIDFA, $n \geq 1$ be an integer and $w \in \Sigma^n$ be a word of size n. Let us look at the computation of $\mathcal{F}$ on input word w. The multi-head (composed of k heads) can be regarded as a device moving one point at a time in any direction within the set $\mathcal{W} = [\![0, n+1]\!]^k$.

As $\mathcal{F}$ is data-independent, we can separate the path taken by the multi-head from the consecutive states of the automaton (depending on the letters of w). In other words, we can take a look at the path of the multi-head on an input word a^n, for any $a \in \Sigma$; it will be the same for w. If all heads are around the middle of the input, they will read only a for a long time and hence their movement will become periodic until one of them reaches an end-marker. That implies that the path of the multi-head is very simple as long as it is not near the *outer edge* $\mathcal{O} = \{p \in \mathcal{W} \mid \|p - \frac{n+1}{2}(1, \ldots, 1)\|_\infty = \frac{n+1}{2}\}$. To be more accurate, we can be sure the path has already become periodic precisely when the multi-head enters the central part of

$\mathcal{W}$ by crossing the *inner edge* $\mathcal{I} = \{p \in \mathcal{W} \mid \|p - \frac{n+1}{2}(1,\dots,1)\|_\infty = \frac{n+1}{2} - |Q| - 1\}$ and it remains periodic until reaching the outer edge $\mathcal{O}$ (cf. Fig. 1).

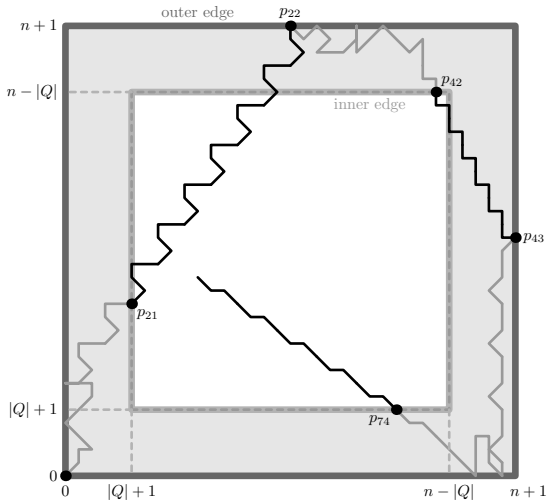

Figure 1 A representation of $\mathcal{W}$ for $k = 2$. The (beginning of the) path of the multi-head is drawn with the periodic sections (jumps) crossing the central white square (delimited by the inner edge $\mathcal{I}$) in black. For each jump a period shape is indicated in bold. The first five key points p_i that begin or end a jump are displayed as black dots.

The simulation of a 2DIDFA $\mathcal{F}$ on an input word w will involve the storing of specific configurations occurring in the run of $\mathcal{F}$ over the input word $a^{|w|}$ (see Fig. 1). We hence define the sequence of *key points* $(p_i, e_i, t_i)_i \in (\mathcal{W} \times Q \times \mathbb{N})^{\mathbb{N}}$ where (p_i, e_i) is the configuration at instant t_i of $\mathcal{F}$ over $a^{|w|}$, by $p_0 = (0,\dots,0) \in \mathcal{O}$, $e_0 = q_0$, $t_0 = 0$ and for all $i > 0$,

- if $p_i \in \mathcal{I}$, p_{i+1} is the next position of $\mathcal{O}$ the multi-head encounters, at a certain time t_{i+1} with $\mathcal{F}$ in state e_{i+1}; this point is called a *jump* (across a periodic section);
- otherwise $t_{i+1} = t_i + 1$ and (p_{i+1}, e_{i+1}) is the next configuration of $\mathcal{F}$; this point is called a *step*.

Since the automaton is deterministic, any non-looping (accepting or rejecting) path cannot go through twice the same position in the same state. Thus, we can bound the number of jumps by $|Q||\mathcal{I}| \sim 2|Q|kn^{k-1}$. In the same way, as steps are located between the outer and inner edges, their number is bounded by $|Q|((n+2)^k - (n-2|Q|)^k) \sim 2|Q|(|Q|+1)kn^{k-1}$. The number of key points of a non-looping path is thus in $O(n^{k-1})$. In particular, it is linear when $k = 2$.

2.2 Basic techniques on cellular automata

A given computation of a CA can be easily represented by drawing successive configurations each one above its predecessor, forming a *space-time diagram*.

We will often have to perform several rather independent computations at the same time; this can easily be done by a "product" automaton which works with a finite number of *layers* supporting each a specific computation. Although rather independent, the layers can communicate between one another to exchange information, since any cell can see all layers. Typically, cells may be waiting for a firing squad to end on a layer before changing their behavior on another layer.

In the remainder of this article, we will have to handle computations involving coordinates. It is classical on CA to write integers in binary on segments of cells (one bit per cell) and execute basic operations with them. Here, all integers will be spelled backward on cells, the lowest bit being the leftmost one. For binary operators, we consider two integers superimposed on the same segment of cells.

It is easy to see that arithmetical operation such as *addition*, *subtraction* or *comparison* can be done in space and time linear in the number of bits of the operands. The same goes for *division by a fixed constant* where we consider the result to be both the quotient and the remainder.

Using the power of parallelism, it is also possible to achieve *multiplication* in linear space and time (see [1]). Moreover, we shall also use implicitly the fact that it is possible to synchronize any interval of cells in space and time linear in the size of this interval. This problem is often referred to as the *firing squad synchronization problem* (see [3, 8]).

In our construction, we shall use another basic operation that we call *selection*. The principle is the following: we fix an interval of cells of size n. Given an integer i (written in binary) between 0 and l positioned at the left of our interval, we want to select the i-th element of the interval and bring it at the right end of the interval (see Fig. 2).

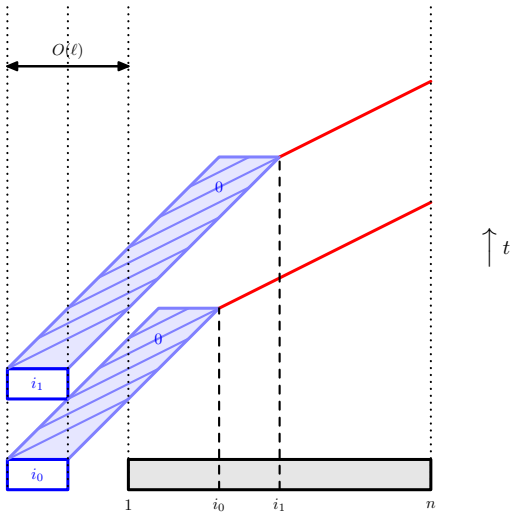

Figure 2 Schematic space-time diagram of a selection. The time scale of the picture has been compressed by factor 2, so that a signal of speed 1 (for instance, in red) seems to be of speed 2.

The basic principle of this operation is quite simple, we shift the integer i along the interval and at each shift decrease its value by one. When reaching 0, we take the content of the corresponding cell inside the interval and send it at maximal speed toward the end of the interval [5]. Thus, the shift speed can be constant and the whole selection achieved in linear time using as workspace only the constant logarithmic space that is required to write the value l.

In addition, this operation can be pipelined. That is, if we suppose that for a fixed interval, we have m integers written as ℓ-bit strings at the beginning of the interval every $\ell' = O(\ell)$ time steps, then the total time for achieving all the selections is $m\ell' + n$.

3 Simulation

▶ **Theorem 4.** *Given $k > 1$, for any 2DIDFA(k) $\mathcal{F}$ recognizing a language $\mathcal{L}$, there exists a CA $\mathcal{C}$ recognizing $\mathcal{L}$ in time $O(n^{k-1}\log(n))$ and space $O(n^{k-1})$, where n is the size of the input word.*

The rest of this paper will be devoted to the proof of this theorem. We will only consider the case $k = 2$ to alleviate the descriptions, but it should be straightforward to generalize this proof.

We assume now that we have a 2DIDFA(2) $\mathcal{F} = (\Sigma,\ Q, \triangleright, \triangleleft, q_0, q_a, q_r, 2, \delta)$. Let us take an arbitrary input word $w \in \Sigma^n$, given an integer $n \geq 2|Q|$. We will define a CA $\mathcal{C} = (\Sigma, Q', \#, q'_a, q'_r, \delta')$ fulfilling the requirements of the theorem. Instead of giving the full description of its state set and transition function, we will describe its behavior when given w as input (the finite number of words that are too short are treated by the CA as a particular case, we can hence forget them).

The execution of $\mathcal{C}$ on w involves coordinates. They all range from 0 to $n+1$, hence we need $\ell(n) = \lceil \log_2(n+2) \rceil$ bits to be able to write any of them. We have to compute it, thus the first thing $\mathcal{C}$ will do is to write $n+1$ in binary on cells $[\![1, \ell(n)]\!]$. This can be done in time $O(n)$.

3.1 Computation of the sequence of key points

We have, as defined previously, $\mathcal{W} = [\![0, n+1]\!]^2$, $\mathcal{O} = (\{0, n+1\} \times [\![0, n+1]\!]) \cup ([\![0, n+1]\!] \times \{0, n+1\})$, $\mathcal{I} = (\{|Q|+1, n-|Q|\} \times [\![|Q|+1, n-|Q|]\!]) \cup ([\![|Q|+1, n-|Q|]\!] \times \{|Q|+1, n-|Q|\})$ and the sequence of key points $(p_i, e_i, t_i)_i$ summing up the computation over a^n. What we want to do now is to output these key points in order on the CA. Note that the coordinates x_i, y_i of position p_i, time t_i, and index i are polynomial in n; thus, they can be encoded in logarithmic space, while state e_i lies in a finite set, only requiring a single cell. Each such point will be written as superimposed $\ell(n)$-bit strings (for x_i, y_i, t_i, and i) together with e_i.

We will compute the sequence of key points iteratively:

1. The procedure is initiated from the first key point $((0,0), q_0, 0)$.
2. At the start of iteration i, key point $(p_i = (x_i, y_i), e_i, t_i)$ is given. First we determine if $p_i \in \mathcal{I}$. It consists in checking whether at least one coordinate of p_i is equal to $|Q|+1$ or $n-|Q|$.
   - *if* $p_i \notin \mathcal{I}$ (case of a step). We check for each head whether it lies on the outer edge (*i.e.*, whether $x_i, y_i \in \{0, n+1\}$). This indicates which letter is read ($\triangleright$,$\triangleleft$ or a). According to this information, we mimic a single transition of the automaton. Namely, we compute the coordinates of p_{i+1} and t_{i+1} by means of some increment or decrement, along with the next state e_{i+1}.
   - *if* $p_i \in \mathcal{I}$ (case of a jump). Between p_i and p_{i+1}, the automaton follows a periodic behavior that only depends on the current state e_i. Such behavior can be specified by finite parameters, namely its period $r_i = (u_i, v_i) \in [\![-|Q|, |Q|]\!]^2$ and its shape s_i (*i.e.*, the head's sequence of moves in $\{-1, 0, 1\}^2$, of length at most $|Q|$). Since the number of states and so the number of distinct periodic behaviors are finite, we can assume that r_i and s_i are available in due time. Thus, we perform operations $(n-x_i)/u_i$ and $(n-y_i)/v_i$ to get the respective quotients a_i and b_i and remainders h_i and k_i. We select the smaller quotient $c_i \in \{a_i, b_i\}$ and then compute $a'_i = u_i \times c_i$ and $b'_i = v_i \times c_i$. The next key point is $p_{i+1} = (x_i + a'_i, y_i + b'_i)$. To be accurate, we have to add the remainder of the period depending on h_i (if $c_i = a_i$) or k_i (if $c_i = b_i$) and on the particular

shape s_i of the period. Although this remainder can have negative coordinates in some particular cases, their absolute values are always bounded by $|Q|$.

3. The procedure stops if either $e_{i+1} \in \{q_a, q_r\}$ or a loop is detected. In this last case, we check the number of key points already computed and the elapsed time. If $i+1 = |Q|((n+2)^2 - (n-2|Q|-2)^2)$ or $t_{i+1} > |Q|(n+2)^2$, we know that the automaton has entered a loop and we can thus definitely stop the simulation.

What is the cost of the whole iteration procedure? Each iteration performs only a finite number of linear operations over integers of size $\ell(n) = O(\log(n))$ and thus is done in space and time $O(\log(n))$. Since the number of key points for a non-looping computation is in $O(n)$, the whole procedure takes $O(n \log(n))$ time steps. One can notice it is conducted very slowly, with $O(\log(n))$ time steps to get only one move of the multi-head in case of step points. But we save a lot of time with every jump across a periodic section, computing $O(n)$ moves of the multi-head within $O(\log(n))$ time steps of the CA.

3.2 Computation of the states

For the moment, we have computed the sequence of key positions $(p_i)_i$ that the multi-head would follow on input word w, but we still do not know its successive states (we have seen only those corresponding to input word a^n). A fortiori we do not know whether w should be accepted. We are now about to get past this lack. What we want to do is to compute, for all key positions p_i, the function $\delta_i \in Q^Q$ such that for all $q \in Q$, if at some time step t the DFA is in state q (for input word w) with the multi-head on key position p_i, then at time $t + t_{i+1} - t_i$ it is in state $\delta_i(q)$ with the multi-head on key position p_{i+1}. One can notice that this way $\delta_i(q)$ may be undefined if state q does not lead to the actual path between p_i and p_{i+1}. If so, it is no problem, we just set $\delta_i(q) = \bullet \notin Q$.

To compute these functions, we have two cases according to whether key position p_i is in $\mathcal{I}$ or not. If it is a step ($p_i \notin \mathcal{I}$), its associated function δ_i only performs a single transition of the DFA and so can be simply computed from the letter lying at this position. In case of a jump ($p_i \in \mathcal{I}$), the problem is more complex, since the associated function mimics all successive transitions performed from p_i to p_{i+1}. But, making use of the regularities of oblivious computation, we can compute simultaneously all these jump functions in linear time.

Pre-computation of all feasible jump functions

Each cell $c \in \{|Q|+1, n-|Q|\}$ will compute the jump function associated to potential key position $p = (c, |Q|+1)$ (and at the same time $(c, n-|Q|)$ and $(|Q|+1, c)$ and $(n-|Q|, c)$). First we will assume that the DFA is in some state q and that the forthcoming periodic trajectory is of shape s.

Cell c plays the role of the multi-head. Once every two time steps, it will update the state of the DFA according to the letters it reads and send two signals at speed 1, one toward the left and one toward the right, to tell the rest of the cells supporting the input word what is the next move of the multi-head. Two copies of the input word are shifted according to the signals received and hence this cell has access to the letters encountered by the multi-head. Because of the duration of the transmission of the order to shift, we need to have parts of the copies provisionally compressed (two letters may lie on the same cell) or dilated, some cells being empty (cf. Fig 3). This process (for particular values q and s) is conducted until

an end-marker is fed to cell c, or until it realizes state q does not induce a period of shape s. In both cases, if $p = p_i$ for some i, we have got $\delta_i(q)$ after $O(n)$ time steps.

What we just described grants us state $\delta_i(q)$ only for the key positions at the four corners of the inner edge. Anyway, there is a nice way to get all the other functions simultaneously. Suppose p_i is at a corner of $\mathcal{I}$; how do we compute, for instance, $\delta_j(q)$ for $p_j = (x_i+h, y_i) \in \mathcal{I}$? The two points share their first coordinate (every $p \in \mathcal{I}$ shares a coordinate with a corner), thus for a same periodic behavior the two trajectories of the multi-head are identical, except for the first head, which is shifted by h letters. $\delta_i(q)$ is computed on cell x_i from some time step $t \in \mathbb{N}$; this cell will send each second-head letter it reads at speed 1 to the other cells. Thus, from time step $t + h$, cell $x_i + h$ will receive the correct letters of the second head, while the copy of the tape for the first head is shifted on this cell with the same delay of h time steps (cf. Fig. 3). $\delta_j(q)$ can in this way be computed on cell $x_i + h$.

This previous process is actually achieved simultaneously for every $q \in Q$. Finally, using one layer for each possible shape, we can have all feasible jump functions written on segment $[\![|Q| + 1, n - |Q|]\!]$ in $O(n)$ time steps.

Selecting the transition function associated to a key point

Suppose we are given all the feasible jump functions written on segment $[\![|Q| + 1, n - |Q|]\!]$, a copy of the input written on cells $[\![1, n]\!]$ and a key point initially written in position $[\![-2\ell(n) + 1, -\ell(n)]\!]$. Using those data, we want to retrieve the function associated to the key point. Two cases are considered depending on whether the function performs a single transition or a jump transition of the DFA.

- Case of step points: according to the coordinates of the step position, we collect the input letters read by each head of the DFA and then deduce the proper function. This is done in time and space $O(n)$.
- Case of jump points: making use of the operation of selection described in section 2.2, we select the jump function written on cell c and layer l, where c is specified by the position and l (depending on the period) by the state of the jump point. This is done in time and space $O(n)$.

3.3 Final stage

Compiling the previous procedures, the cellular automaton $\mathcal{C}$ works the following way:

- First, $\mathcal{C}$ pre-computes all the feasible jump functions and writes them on cells $[\![|Q| + 1, n - |Q|]\!]$. This is done in both space and time $O(n)$.
- Then, $\mathcal{C}$ generates the list of key points. This is done in time $O(n \log(n))$ and space $O(\log(n))$.
- As soon as one key point is written, $\mathcal{C}$ selects its associated function. Each selection is done in time and space $O(n)$.
- Once a new selection is over, $\mathcal{C}$ updates the current state of the automaton $\mathcal{F}$ according to the proper function. In case it is the accepting or the rejecting state, or if the time of the last key point reaches the maximal number of moves the multi-head is supposed to perform, $\mathcal{C}$ terminates its computation.

These methods obviously work in space $O(n)$. In regard to the time, since the selections are pipelined, the bound corresponds to the time required to generate all the key points plus the time of the last selection. Hence the total running time is in $O(n \log(n)) + O(n)$, *i.e.* in $O(n \log(n))$, leading to the theorem. ◀

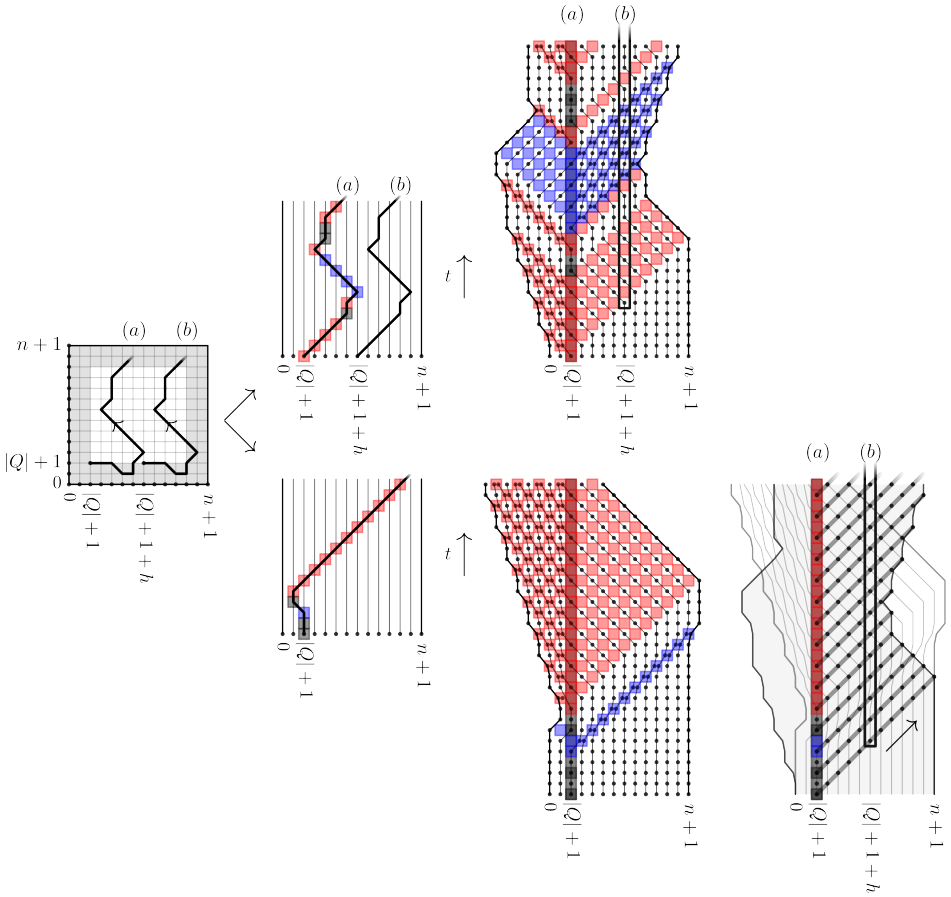

Figure 3 Simulation of the multi-head from a corner of the inner edge. On the left we have two portions of the path (a) and (b) that are translated copies of each other by h letters. Portion (a) starts at corner $(|Q|+1, |Q|+1)$ and is simulated by cell $|Q|+1$ (the darkened one) by the CA (on the right) with two layers, one by head. On each layer, according to the state of the DFA, the dark cell may send signals telling the other cells to shift their letter to the left (blue signal) or to the right (red signal). If they see no signal, the cells keep their letter, which are symbolized by dots, linked to indicate where they are moved at each time step – end-markers are also represented by dots, linked in black. As the shifted tapes on each layer cross the dark cell, the latter gets the appropriate letters within two time steps to deduce the next move and state of the DFA. Cell $|Q|+1+h$ simulates portion (b) h time steps later. It sees directly the correct letters on the first layer while those for the second head are sent by the dark cell.

Conclusion

We have presented an efficient construction that simulates oblivious k-head finite automata on cellular automata in time $O(n^{k-1}\log(n))$ and space $O(n^{k-1})$. Such simulations achieving parallel speed-up are scarce. The performance gain is in $O(n/\log(n))$ as regards the naïve simulation without speed-up, which processes the same task in time $O(n^k)$.

Our result fully exploits the oblivious feature of the sequential computation. Now, it is another challenge to achieve parallel speed-up for multi-head finite automata without the constraint of data-independence.

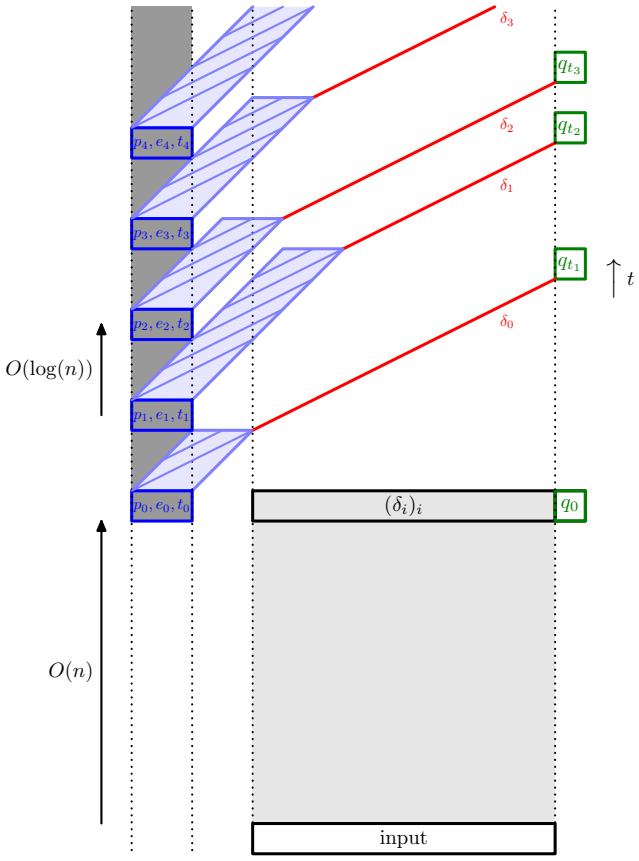

Figure 4 Global simulation. Light grey corresponds to the pre-computation of jumps functions, dark grey to the computation of the sequence of key points, and the rest is the selection (here again, the time scale is compressed by factor 2).

References

1 A. J. Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Transactions on Electronic Computers*, 14(1):394–399, 1965.

2 S. N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, 18(4):349–365, 1969.

3 K. Čulík II. Variations of the firing squad problem and applications. *Information Processing Letters*, 30(3):152–157, 1989.

4 M. Delorme. An introduction to cellular automata. In M. Delorme and J. Mazoyer, editors, *Cellular Automata: A Parallel Model*, pages 5–50. Kluwer Academic Publishers, 1997.

5 M. Delorme and J. Mazoyer. Algorithmic tools on cellular automata. In G. Rozenberg, T. Baeck, and J. Kok, editors, *Handbook of Natural Computing*. Springer, Berlin, to appear.

6 M. Holzer. Multi-head finite automata: Data-independent versus data-dependent computations. *Theoretical Computer Science*, 286(1):97–116, 2002.

7 M. Holzer, M. Kutrib, and A. Malcher. Multi-head finite automata: Characterizations, concepts and open problems. In T. Neary, D. Woods, A. K. Seda, and N. Murphy, editors, *The Complexity of Simple Programs (CSP'08)*, EPTCS, pages 93–107, 2008.

8 J. Mazoyer. On optimal solutions to the firing squad synchronization problem. *Theoretical Computer Science*, 168(2):367–404, 1996.

9 A. R. Smith III. Simple computation-universal cellular spaces. *Journal of the ACM*, 18(3):339–353, 1971.

10 K. Wagner and G. Wechsung. *Computational Complexity*. D. Reidel, Dordrecht, 1986.