On relations between arrays of processing elements of different dimensionality

Alf-Christian Achilles*
achilles@ira.uka.de

Martin Kutrib†
kutrib@informatik.uni-giessen.de

Thomas Worsch*
worsch@ira.uka.de

4th February, 1996

Abstract

We are examining the power of d-dimensional arrays of processing elements in view of a special kind of structural complexity. In particular simulation techniques are shown, which allow to reduce the dimension at an increased cost of time only. Conversely, it is not possible to regain the speed by increasing the dimension. Moreover, we demonstrate that increasing the computation time (just by a constant factor) can have a more favorable effect than increasing the dimension (arbitrarily).

1 Introduction

We can regard d-dimensional arrays of processing elements as models for massively parallel computers. A usual step towards a formal model is to treat the single processing elements as finite-state machines. Various types of such devices have been studied under manifold aspects for a long time (see e.g. [2, 4, 6, 7, 8, 9, 10, 12, 14, 15, 16, 17, 18, 19]).

Mainly, the types differ in how the single machines are interconnected and in how the input is supplied. Here we are investigating d-dimensional arrays with a very simple interconnection pattern. Each node is connected to its $2 \cdot d$ immediate neighbors only. They are usually called cellular arrays (or cellular automata) (CA) if the input is supplied in parallel and iterative arrays (IA) in case of a sequential input manner to a designated cell.

*Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler, Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe

†AG Informatik, Universität Gießen, Arndtstraße 2, D-35392 Gießen

We also investigate a stack-augmented variant of IAs. Instead of finite-state machines deterministic pushdown state machines are used as processing elements. The corresponding model is called iterative pushdown array (IPDA).

Our special attention is focussed on structural complexity issues concerning with the dimension of arrays. Prior work in this field dealt with reducing the interconnection patterns or the state set cardinality and speeding-up the computation (see e.g. [4] for IAs, [14] for CAs).

Investigating the relationships between arrays of different dimension is done by studying their power as language acceptors, since cellular programming techniques for language recognition are similar to those used in numerical algorithms.

The aim of the present paper is to establish a number of relationships between $(d+1)$- and d-dimensional arrays, including techniques for dimension reduction. To this end we define the complexity measures "time", "space", "volume" and, if we require all edges of the volume to have identical lengths "cube".

In the following section the definitions and basic notions are reviewed. Results concerning the reduction of dimensions are obtained in section 3. In section 4 we investigate in some sense the reverse problem, namely, the increasing of dimensions.

2 Basic notions

A *d-dimensional iterative array* (IA) is a (infinite) d-dimensional array of identical finite-state machines, sometimes called *cells*. We can identify each machine by its coordinates in $\mathbb{Z}^d$. The input symbols are fed serially to the finite-state machines at the origin. Each cell is connected to its $2 \cdot d$ immediate neighbors. All cells work synchronously at discrete time steps. With an eye towards language recognition more formally an iterative array is a system $(S, A, \sigma, F, s_0, a_0)$, where S is the finite nonempty *set of states*, $F \subseteq S$ is the *set of accepting states*, and A is the finite nonempty set of *input symbols* containing a_0, the *end-of-input symbol*.

The local transition function σ maps from $S^{2d+1} \cup (S^{2d+1} \times A)$ (depending on whether the cell is the origin or not) to S. It satisfies $\sigma(s_0, \ldots, s_0) = s_0$. Due to this property s_0 is called the *quiescent state*. We assume that at the end of the input the symbol a_0 is not consumed.

In iterative arrays initially all cells are in the quiescent state. We call the time a cell leaves the quiescent state for the first time its *activation*. If for input w cell $(0, \ldots, 0)$ eventually enters an accepting state, then w is accepted. The *accepted language* is $L = \{w \mid w \text{ is accepted}\}$.

If we exchange the finite-state machines by deterministic pushdown store machines the resulting device is an *iterative pushdown array*. The definitions

are straightforward and omitted here. The reader interested in details is referred to [9].

For convenience each finite-state machine of an array may be thought of as a finite sequence of finite-state registers. Therefore, in presenting algorithms we may restrict the CAs and IAs to $\mathbb{N}^d$ instead of $\mathbb{Z}^d$ since we can map each cell $(i_1, \ldots, i_d)$ to cell $(|i_1|, \ldots, |i_d|)$ which again has to consist of 2^d internal registers. Obviously, the original computation can be simulated without any slow-down.

The ith registers of all cells together are called ith *track*. Another well-known presentation technique is the concept of propagating pulses or signals [15].

Definition 2.1 Let M be a d-dimensional CA, IA or IPDA accepting a formal language $L \subseteq A^+$. For all $n \in \mathbb{N}$:

a) the *time complexity* $t(n)$ of M is the minimum number of time steps needed to accept each input word from $L \cap A^n$,

b) the *space complexity* $s(n)$ is the maximum number of cells which have been activated, i.e. which have left the quiescent state, during the computations for an input $w \in L \cap A^n$

c) similarly, the *volume complexity* $v(n)$ is the size of the smallest rectangular array comprising all cells which have been activated,

d) and we say, that a d-dimensional system M has *cube complexity* $c(n)$ if and only if for each input of length n the smallest cube comprising all cells which have been activated has volume $c(n)$ (and hence side length $\sqrt[d]{c}$).

For 1-dimensional systems space volume and cube complexity obviously coincide. But for d-dimensional systems, $d \geq 2$, the latter may be much larger. In general $v \in \mathrm{O}(s^d)$ and of course $s \in \mathrm{O}(v)$.

For example $\mathscr{L}^d_{\mathrm{IA-TIME-SPACE}}(n^2, n)$ denotes the family of all formal languages which are acceptable by a d-dimensional IA having time resp. space complexity of n^2 resp. n.

3 Reducing the dimension in iterative arrays

If we are going to reduce the dimension of iterative arrays, one of the problems we are confronted with is the possibly unknown space complexity. Of course, the space complexity can be bounded by the time complexity which is known in most cases. But if we know something about the time complexity, e.g. it is linear-time, then we might know too little, e.g. the linear-time constant for two-dimensional arrays is not computable [1, 8].

Definition 3.1 A mapping $t : \mathbb{N} \longrightarrow \mathbb{N}$ is IA-time-constructible if there exists an iterative array the origin cell of which enters for all $n \in \mathbb{N}$ on all inputs of length n a designated state exactly at time step $t(n)$.

It should be mentioned here that the family of IA-time-constructible functions is very rich, includes the "usual" time complexity functions and is closed under various operations. For a detailed discussion of this topic refer to e.g. [3, 15].

For d-dimensional IAs we can consider its expansion in dimensions which depends on the length of the input as the space complexity does. We denote the expansion in dimension i, $1 \leq i \leq d$, by $k_i(n)$. Trivially, the space complexity can be bounded by $s(n) \leq v(n) = k_1(n) \cdot \ldots \cdot k_d(n)$.

In the following we assume that at least one of the k_i is known and that it is IA-time-constructible. As stated above the second assumption is a weak one. In order to prove our next result, we need a technical lemma. We need to construct an (one-dimensional) iterative array which behaves as follows (see figure 1):

In a first phase it marks $p(n)$ cells to the left and $p(n)$ cells to the right of the origin, where $p : \mathbb{N} \longrightarrow \mathbb{N}$. In a second phase every $p(n)$ cells are synchronized in such a manner that they are able to recognize periods of $p(n)+1$ time steps, continually. The third phase repeats the first one to the left and right of the already marked areas and starts immediately after the first one. Between each two marked areas there is a cell in the state @. The fourth phase repeats the second one. Further phases are straightforward. We call such iterative arrays $p(n)$-*self-partitioning*.

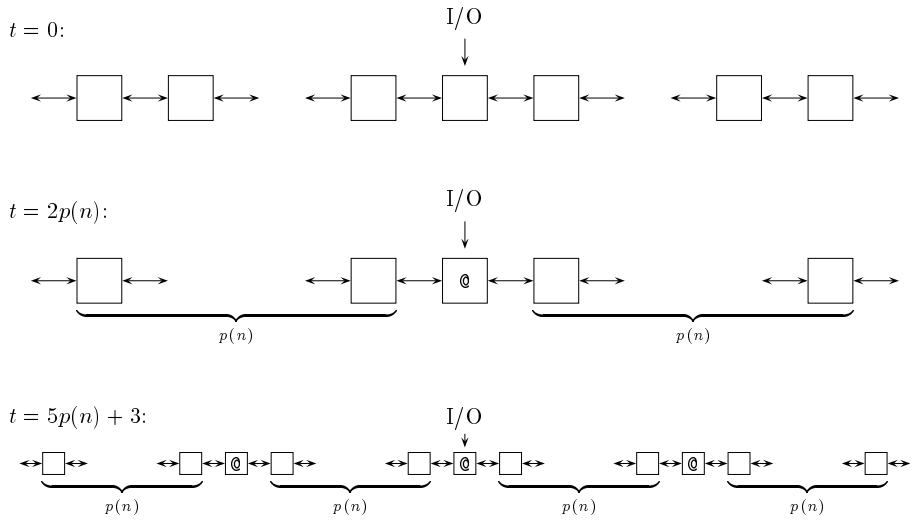

Figure 1: Excerpt of a computation of a self-partitioning IA.

Lemma 3.1 Let $p(n)$ be an IA-time-constructible mapping. Then there exists a $p(n)$-self-partitioning iterative array.

Proof. We give a somewhat informal construction since details are tedious and hard to read.

The first phase is realized by simulating a $p(n)$ time constructing iterative array in a separate register. During the simulation at each time step signals are sent to the left and right which mark the first unmarked cells on both sides. The last signal additionally initiates a (nearly) time optimal FSSP [11, 17] algorithm to synchronize the chain of $p(n)$ cells and causes the next unmarked cell to enter state @. The consumed input is stored on a separate track for reuse.

The FSSP needs $2 \cdot p(n) + 3$ time steps to synchronize the chains for the first time, which occurs, consequently, at time step $3 \cdot p(n) + 3$. Subsequently modified FSSPs with generals at both borders [5] are continually performed each of which take $p(n) + 1$ time steps only.

After the first synchronization all $p(n)$ cells send signals simultaneously which will mark another $p(n)$ cells after crossing the @-cell. Again, the last of them will initiate a FSSP. Altogether the second chains are synchronized $3 \cdot p(n) + 3$ time steps after the first ones for the first time. □

Except for the first $3 \cdot p(n) + 3$ time steps we can speed up the partitioning by the usual technique of grouping. Grouping three cells to one we achieve a behavior which adds partitions at a rate of $p(n) + 1$ time steps.

For a moment we deal with the lucky case of knowing a little bit about the space complexity.

Theorem 3.1 Let the expansion k_i be an IA-time-constructible mapping for an appropriated i. Then

$$\mathscr{L}^{d+1}_{\text{IA-TIME-VOL}}(t, v) \subseteq \mathscr{L}^{d}_{\text{IA-TIME-VOL}}(\mathrm{O}(k_i \cdot t), \mathrm{O}(v))$$

Proof. Let $j \in \{1, \ldots, i-1, i+1, \ldots, d+1\}$. Imagine, the $(d+1)$-dimensional rectangle $k_1 \times k_2 \times \ldots \times k_{d+1}$ is divided into k_j d-dimensional rectangles the expansion in direction i of which has length k_i. The idea is to chain up the sub-rectangles along dimension i (see figure 2, 3). Since in the initial state all cells are quiescent this can initially be done by the IA which is $k_i(n)$-self-partitioning. Creating the first two partitions is the first phase of the algorithm. From now on adding partitions and doing further computations are performed in parallel. The further computation consists of continually simulating state transitions of the $(d+1)$-dimensional IA.

In order to simulate one time step we have to supply to each cell the state information of its original neighbors. Except for the neighboring cells in neighboring sub-rectangles (i.e. the neighbors along direction j) this information is available immediately. The missing information is moved on

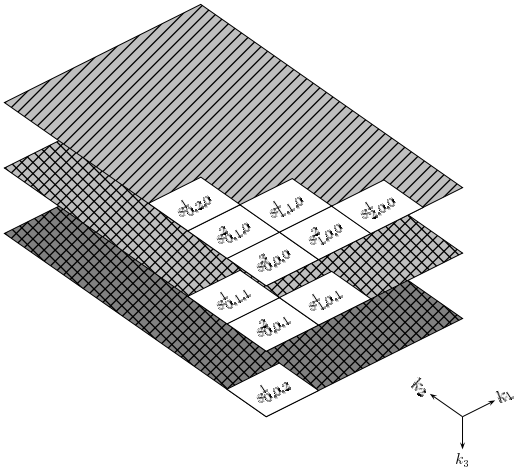

Figure 2: A 3-dimensional IA configuration at $t = 3$. $s^i_{j,k,l}$ denotes the ith state of cell (j, k, l) after its activation.

@	$s^3_{0,0,0}$	$s^2_{1,0,0}$	$s^1_{2,0,0}$	@	$s^2_{0,0,1}$	$s^1_{1,0,1}$	s_0	@	$s^1_{0,0,2}$	s_0	s_0	@
@	$s^2_{0,1,0}$	$s^1_{1,1,0}$	s_0	@	$s^1_{0,1,1}$	s_0	s_0	@	s_0	s_0	s_0	@
@	$s^1_{0,2,0}$	s_0	s_0	@	s_0	s_0	s_0	@	s_0	s_0	s_0	@

Figure 3: A 2-dimensional embedding of a 3-dimensional IA configuration for $k_j = k_3$ and known $k_i = k_1 = 3$ at $t = 3$. s_0 denotes the quiescent state.

separate tracks in $k_i(n)$ time steps to the cells (see figure 4). The movement is controlled by the clock realized by the FSSPs. Subsequently the transition is simulated and the whole cycle repeats.

The first phase of the algorithm takes $3 \cdot k_i(n)+3$ time steps. Subsequent simulation cycles need $k_i(n)+1$ time steps respectively. Remember that we have grouped the cells by 3. Therefore, the time complexity is $3 \cdot k_i(n) + 3 + \frac{t(n)\cdot(k_i(n)+1)}{3}$, which is less than $t(n) \cdot k_i(n)$ for $k_i > 1$ and $t > 9$. Since our grouping and because partitions are added with a rate of $k_i(n)+1$ time steps the volume complexity is less than $v = k_1 \cdot \ldots \cdot k_{d+1}$. □

Now we turn to the question what happens if almost nothing about the time and space complexity is known. In this case we cannot use the technique of pre-partitioning since we do not know anything about the size of the partition. Instead, we can start with 2 partitions of size 1 and before

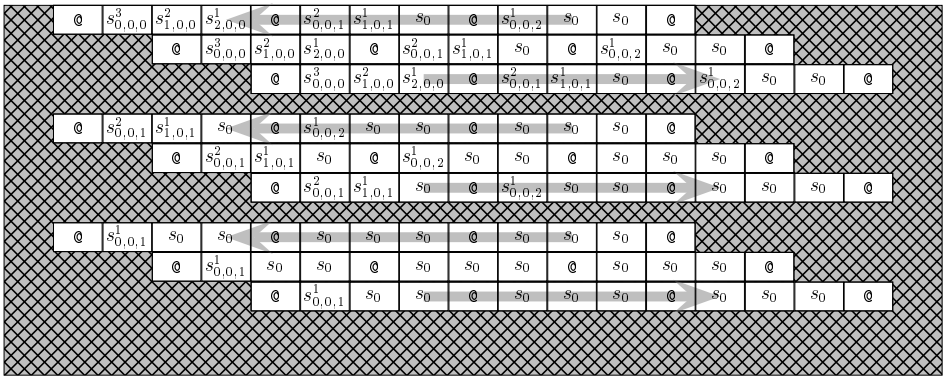

Figure 4: Movement of a 2-dimensional embedding of a 3-dimensional IA configuration after 2 shifts.

the movement and simulation phase perform a phase of enlarging and adding partitions.

Theorem 3.2

$$\mathscr{L}^{d+1}_{\text{IA}-\text{TIME}}(t) \subseteq \mathscr{L}^{d}_{\text{IA}-\text{TIME}}(t^2)$$

Proof. Since we have no further information about the space complexity we can only use the fact $k_i \leq t$, $i \in \{1, \ldots, d+1\}$, which leads to the cube complexity $c = t^{d+1}$. But unfortunately we do not know anything about t.

The main idea of the algorithm is based on the previous theorem. But instead of static partitioning we have to partition dynamically. The division into sub-rectangles of dimension d is done according to k_1. Assume there is initially 2 partitions of size 1. The computation is divided in 3 phases which are continually performed. These are "enlarging the partitions", "adding partitions" and "simulate 1 transition step".

For the first phase assume that there are $p+1$ partitions of size p in direction $d+1$. Obviously there are neither more nonempty partitions nor partitions of longer size after p simulated time steps. Before the next simulation we need $p+2$ partitions of size $p+1$. Observe that the partition $0 \leq q \leq p$ contains at least q cells in quiescent state (in each direction) since up to now these cells cannot be activated in the original computation. Controlled by the clock realized by the FSSPs now the states of these cells propagating towards the origin along direction 1 whereby the states of crossed cells are "pushed" in the other direction. After pushing a state @ a signal is cancelled (see figure 5).

The result consists of p partitions of length $p+1$ and one empty partition. Phase 1 takes p time steps.

During phase 2 the empty partition is enlarged to size $p+1$ and an additional one is generated. This can be done in less than $4(p+1)$ further

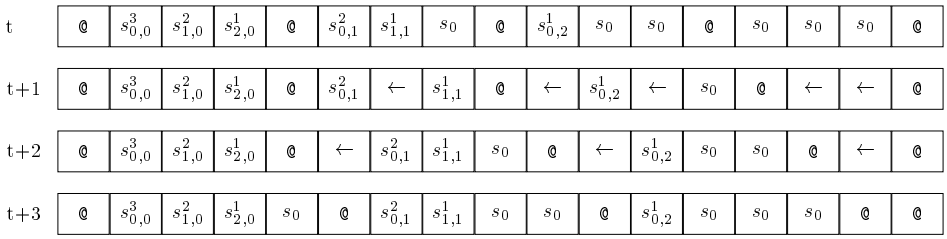

Figure 5: Enlarging partitions. Excerpt of one line.

time steps.

The third phase is as the simulation phase of the previous theorem with the exception that the controlling FSSPs are started at the end of phase 1 and synchronize after $4(p+1)$ time steps (due to the new partition) and after $5(p+1)$ time steps, respectively. Therefore phase 3 needs further $(p+1)+1$ time steps.

Altogether the simulation of one original transition takes $p+4(p+1)+(p+1)+1 = 6(p+1)$ time steps. If we use the technique of grouping we can group 6 cells to 1 and achieve a solution in $p+1$ time. Therefore, the time complexity of the whole algorithm is $\sum_{i=1}^{t} i+1 = t+\frac{t^2+t}{2} = \frac{3}{2}+\frac{1}{2}t^2$ which is less than t^2 for $t \geq 3$. □

The difference between the both previous theorems depends on the "quality" of the knowledge necessary for the first one. If the known k_i is large against the other k_j its favorable effect is gone. On the other hand we can safe a lot of time if it is small compared with t.

Now we focus our interest on iterative pushdown arrays. The main problem arises from the fact that we cannot simulate two stacks by just one, otherwise the context-free languages would be closed under e.g. intersection. Therefore the technique of grouping is not applicable. Furthermore if a cell should switch to the behavior of a neighboring one it has to fetch the whole stack content, too. For that reason the technique of dynamic partitioning seems to be fairly unrealistic. On the other hand in the order of magnitude theorem 3.1 holds for IPDAs, too.

Theorem 3.3 Let for an appropriated i k_i be an IA-time-constructible mapping. Then

$$\mathscr{L}^{d+1}_{\text{IPDA-TIME-VOL}}(t,v) \subseteq \mathscr{L}^{d}_{\text{IPDA-TIME-VOL}}(\mathrm{O}(k_i \cdot t), \mathrm{O}(v))$$

Proof. The proof is only a slight modification of the proof of theorem 3.1 and left to the reader. □

4 Increasing the dimension in iterative arrays

We now turn to questions concerned with the increasing of dimensions. Unfortunately the converse of the results in the previous section does not hold, from which follows that we cannot regain the speed lost by reducing the dimension.

Theorem 4.1 There is a language in $\mathscr{L}^1_{\text{IPDA-TIME}}(2 \cdot id)$ which for any $d \in \mathbb{N}$ does not belong to $\mathscr{L}^d_{\text{IPDA-TIME}}(id)$.

Proof. $L = \{vv' \mid v, v' \in \{0,1\}^* \wedge v' = v'^R \wedge |v'| \geq 3\}$ was shown to be a real-time one-way CA language [6]. Consequently it is a real-time CA and a two times real-time IA language. For structural reasons then it belongs to $\mathscr{L}^1_{\text{IPDA-TIME}}(2 \cdot id)$.

On the other hand it has been shown that L cannot be accepted by any d-dimensional IPDA in real-time [9]. □

The following corollary has been shown in [4].

Corollary 4.1 There is a language in $\mathscr{L}^1_{\text{IA-TIME}}(2 \cdot id)$ which for any $d \in \mathbb{N}$ does not belong to $\mathscr{L}^d_{\text{IA-TIME}}(id)$.

From the previous we derive the fact that increasing the computation time (just by a constant factor) can have more favorable effect than increasing the dimension (arbitrarily). Now the question arises whether increasing the dimension leads to different complexity classes at all. Regarding real-time language families the answer is yes.

The following theorem was shown in [4].

Theorem 4.2 For all $d \in \mathbb{N}$:

$$\mathscr{L}^{d+1}_{\text{IA-TIME}}(id) \supsetneq \mathscr{L}^d_{\text{IA-TIME}}(id)$$

A similar result concerning nondeterministic iterative arrays and linear-time was proved in [13].

Theorem 4.2 can easily be extended to IPDAs.

Theorem 4.3 For all $d \in \mathbb{N}$:

$$\mathscr{L}^{d+2}_{\text{IPDA-TIME}}(id) \supsetneq \mathscr{L}^d_{\text{IPDA-TIME}}(id)$$

Proof. It is sufficient to show that the inclusion is a proper one. We may regard a d-dimensional IPDA as a restricted $(d+1)$-dimensional IA as follows. The finite control of a cell $(i_1, \ldots, i_d)$ of the IPDA is simulated by cell $(i_1, \ldots, i_d, 0)$ of the IA. For all $i \in \mathbb{N}$ the IA cells $(i_1, \ldots, i_d, j)$, $j < 0$,

are idle and the IA cells $(i_1, \ldots, i_d, j)$, $j \geq 0$, are simulating the stack of IPDA cell $(i_1, \ldots, i_d)$.

From Cole's result and for structural reasons we obtain $\forall d \in \mathbb{N}$:
$\mathscr{L}^d_{\text{IPDA-TIME}}(id) \subseteq \mathscr{L}^{d+1}_{\text{IA-TIME}}(id) \subsetneqq \mathscr{L}^{d+2}_{\text{IA-TIME}}(id) \subseteq \mathscr{L}^{d+2}_{\text{IPDA-TIME}}(id)$,
from which the assertion follows. □

It should be mentioned that we can also prove a stronger version of the previous theorem by adapting Cole's argumentation directly to IPDAs.

The previous theorems of this section were concerned with the special case of real time. We now return to the case of an arbitrary time complexity t. Contrasting theorem 3.2 it will be shown that, though none of the time loss can be regained, at least the dimension can be increased without any time loss, even if the cube complexity is fixed.

Theorem 4.4 Let $t(n)$ be an arbitrary function and let $c(n)$ be a function which can be computed in time $t(n)$ and within a 3-dimensional cube of volume $c(n)$. Then it holds:

$$\mathscr{L}^2_{\text{IA-TIME-CUBE}}(t, c) \subseteq \mathscr{L}^3_{\text{IA-TIME-CUBE}}(\mathrm{O}(t), c)$$

Proof. Let n be an arbitrary input size and consider a square of area $c = c(n)$ and side length $l = \sqrt{c}$.

In part a) below we first describe how the cells of the square can be rearranged into a cube of side length $k = \sqrt[3]{c}$. Afterwards it is shown in part b), that it is possible to simulate a 2-dimensional cellular automaton working on the square by another one working on the cube without loss of time.

a) First cover the square with stripes of length l and width $k = \sqrt[3]{c}$ as depicted in figure 6.

   As can be seen, there are $2\frac{l}{k} - 1$ stripes which overlap by $\frac{k}{2}$. Let the stripes be numbered as follows: The $q := \frac{l}{k}$ stripes drawn with dashed lines in the figure and partitioning the whole square are assigned the even numbers $0, 2, \ldots, 2q-2$ from left to right. The other stripes are assigned the odd numbers $1, 3, \ldots, 2q-3$ from left to right.

   Each stripe is folded such that the parts folded on top of each other are squares of side length k. This results in a 3-dimensional rectangular array of size $k \cdot k \cdot q$. Now all $2q-1$ of these rectangular arrays are put on top of each other in the same order in which they are numbered. The resulting rectangular array is of size $k \cdot k \cdot (q(2q-1)) \in \Theta(l^2) = \Theta(c)$. By storing two simulated cells in one simulating cell, the whole can be made to fit into a cube of side length $k = \sqrt[3]{c}$.

b) Now call the cells in the middle part of width $\frac{k}{2}$ and height l of a stripe its kernel (see the right part of figure 6). Observe that the kernels of the

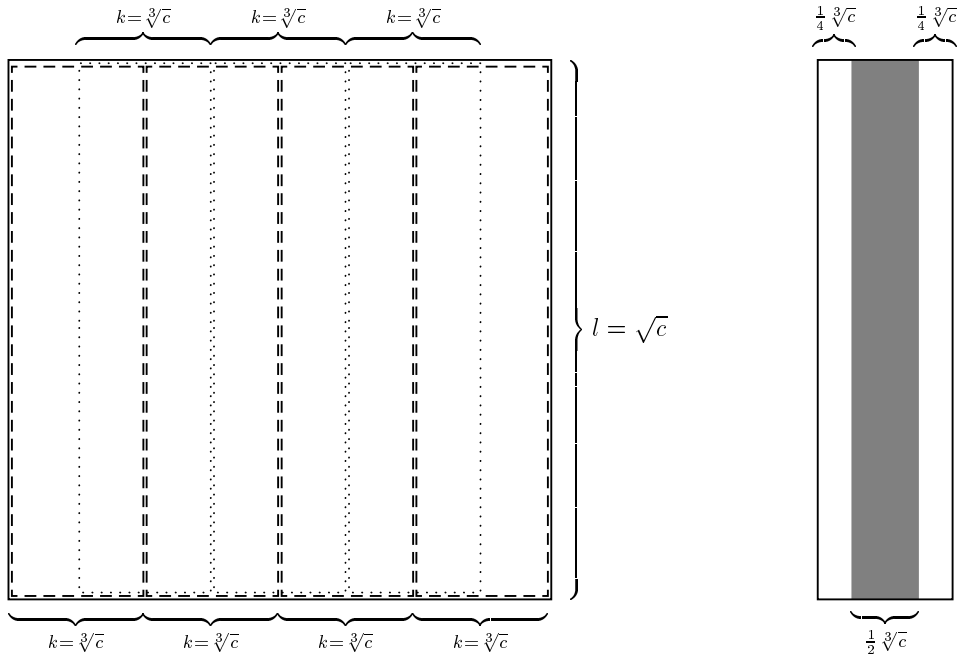

Figure 6: Left: Covering of a square with $2q-1$ stripes. Right: a stripe and its kernel (gray).

stripes still (almost) cover the whole square[1]. The simulation is done in alternating phases: first a number of steps of the original cellular automaton is simulated (i), then the overlapping parts are updated (ii).

(i) Because of its whole width, $\frac{k}{4}$ steps of the kernel of a stripe can be simulated in real-time without the need to refer to cells of other overlapping stripes.

(ii) Afterwards in each stripe the two non-kernel parts have to be updated by transferring states from the kernels of neighboring stripes. The time needed for this task is bounded by the height q of the rectangular arrays into which the stripes have been folded. Hence it is smaller than the simulation time.

Therefore the whole time needed on average for the simulation of $\frac{k}{4}$ steps is proportional to k.

□

[1]The left and right borders are special cases. Since their proper treatment is straightforward, we ignore it in the following.

References

[1] Beyer, W. T. *Recognition of topological invariants by iterative arrays.* Technical Report MAC TR-66, Massachusetts Institute of Technology, 1969.

[2] Bleck, B. and Kröger, H. *Cellular algorithms.* In Evans, D. (ed.), *Advances in Parallel Computing Volume 2.* JAI Press, London, 1992, pp. 115–143.

[3] Buchholz, Th. and Kutrib, M. *On time constructibility of functions in one-way cellular automata.* Report 9502, Arbeitsgruppe Informatik, Universität Gießen, Gießen, 1995.

[4] Cole, S. N. *Real-time computation by n-dimensional iterative arrays of finite-state machines.* IEEE Conference Record of 7^{th} Annual Symposium on Switching and Automata Theory, 1966, pp. 53–77.

[5] Čulik II, K. *Variations of the firing squad problem and applications.* Inform. Process. Lett. 30 (1989), 153–157.

[6] Dyer, C. R. *One-way bounded cellular automata.* Inform. Control 44 (1980), 261–281.

[7] Ibarra, O. H., Kim, S. M., and Moran, S. *Sequential machine characterizations of trellis and cellular automata and applications.* SIAM J. Comput. 14 (1985), 426–447.

[8] Kosaraju, S. R. *On some open problems in the theory of cellular automata.* IEEE Trans. Comput. C-23 (1974), 561–565.

[9] Kutrib, M. *On stack-augmented polyautomata.* Report 9501, Arbeitsgruppe Informatik, Universität Gießen, Gießen, 1995.

[10] Kutrib, M. and Worsch, Th. *Investigation of different input modes for cellular automata.* Proc. Parcella '94, Akademie Verlag, 1994, pp. 141–150.

[11] Mazoyer, J. *A six-state minimal time solution to the firing squad synchronization problem.* Theoret. Comput. Sci. 50 (1987), 183–238.

[12] Seidel, S. R. *Language recognition and the synchronization of cellular automata.* Technical Report 79-02, Department of Computer Science, University of Iowa, Iowa City, 1979.

[13] Seiferas, J. I. *Linear-time computation by nondeterministic multidimensional iterative arrays.* SIAM J. Comput. 6 (1977), 487–504.

[14] Smith III, A. R. *Cellular automata complexity trade-offs*. Inform. Control 18 (1971), 466–482.

[15] Terrier, V. *Signals in linear cellular automata*. Proc. Workshop on Cellular Automata, Centre for Scientific Computing, Espoo Finland, 1991.

[16] Vollmar, R. *Algorithmen in Zellularautomaten*. Teubner, Stuttgart, 1979.

[17] Waksman, A. *An optimum solution to the firing squad synchronization problem*. Inform. Control 9 (1966), 66–78.

[18] Worsch, Th. *Komplexitätstheoretische Untersuchungen an myopischen Polyautomaten*. Dissertation, Technische Universität Braunschweig, 1991.

[19] Yamada, H. and Amoroso, S. *Tessellation automata*. Inform. Control 14 (1969), 299–317.