Theoretical Computer Science 37 (1985) 183–216
North-Holland

TOPOLOGICAL TRANSFORMATIONS AS A TOOL IN THE DESIGN OF SYSTOLIC NETWORKS*

Karel CULIK, II

Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Ivan FRIS**

Department of Computer Science, University of New England, Armidale, N.S.W. 2351, Australia

Communicated by A. Salomaa
Received May 1984
Revised December 1984

Abstract. We introduce the notion of computational network (CN) which is a general model of an arbitrary (finite or infinite) system of parallel synchronized processors (systolic network). Our basic and very useful tools are topological transformations of the space-time diagrams (unrollings) of computations on CN. We show that the topological transformations on unrollings can be used to design systolic networks, to give simple proofs of their correctness, and to demonstrate the equivalence of different networks. For example, we use the transformation technique to give a concise proof of a strengthened version of Leiserson's and Saxe's Retiming Lemma and Systolic Conversion Theorem. As a practical application we show the correctness of a simple algorithm for distributed sorting on a systolic ring. Many other examples are given.

Contents

Introduction 184
1. Preliminaries 186
2. Computational schemas and networks 186
3. Retiming and systolic conversion 190
4. Simulation of networks 195
5. Further examples 203
References 215

* This research was supported by the Natural Sciences and Engineering Research Council of Canada under Grant No. A-7403.

** This work was done during the second author's visit at the Department of Computer Science, University of Waterloo.

Introduction

Systolic systems are arrays of synchronized processors which process data in parallel by passing them from one processor to neighboring ones in a regular rhythmical pattern. Most systolic systems use only a few different types of processors arranged in a regular pattern. The principal idea is to perform the required computations with minimal input-output communication. Systolic algorithms were explicitly introduced by Kung and Leiserson [19, 24] but many algorithms of this type were designed earlier, see for example [5, 11, 16].

Recently, systolic systems have been studied extensively; see [17] for a list of references. Most of this work has been devoted to the design of individual algorithms for many different areas, but the efficient layout of systolic systems (and VLSI in general) has also been well studied, see for example [20, 21]. Other topics that have been studied are: the development of general programming (design) techniques for systolic algorithms [3, 18, 22, 25, 28, 30], and the systematic study of the power and limitations of certain types of networks from an automata-theoretic point of view [1, 4–7, 9–11, 13, 14, 26, 29].

Our approach is also automata-theoretic, but rather than the study of a specific class of language recognizers or transducers we introduce precise notions to the study of arbitrary systolic networks. For example, we want to make precise the statement that the square grid and hexagonal grid are essentially equivalent, or that the bidirectional linear array and the unidirectional ring are essentially equivalent. Our main goal is to give a general framework for the study of arbitrary systolic systems and computations on them.

We introduce the notion of a 'computational network' which in general is an arbitrary finite or infinite (synchronous) network of processors connected by communication lines with arbitrary integer (even negative) 'delays' and no queuing capability. We are mainly interested in homogeneous (identical processors) and regular networks, but our definitions do not make any such assumption.

Our main tool is the space-time diagram, called the unrolling, of a computation on a computational network in [7]. The unrolling of a computational network is a simple form of a data-flow diagram; it motivates our definition of a 'computational diagram'. We consider two networks to be equivalent when they have isomorphic unrollings. This is a much stronger equivalence than equivalence based on the same input-output function. Two different networks can have isomorphic unrollings, that is, the unrolling of one network can be topologically transformed to the unrolling of the other. Such a topolotical transformation is a useful tool when designing new networks and in proving their correctness. Topological transformations as such are not new, see [4], but we introduce a general model of a computational network and a computational diagram which allow concise proofs using the transformation technique for a broad class of parallel networks. We demonstrate that most of the known techniques for systolic system design, such as systolic conversion [18, 22]

folding [3], and speed up [26], are special cases of topological transformations on unrollings. This also holds for the wavefront technique [30] and geometric transformations [2] but lack of space prevents their discussion here.

A particularly simple type of a computational network is the pure computational network in which for each node n the paths from all inputs to node n have the same delay. In practice such a network always allows pipelining of inputs with pipelining of period one. We show that a pure computational network and its unrolling are isomorphic.

In Section 3 we study the semisystolic and systolic networks introduced in [22] and generalize the Retiming Lemma and Systolic Conversion Theorem from [22]. Our proofs using unrolling are simpler than the original proofs and at the same time they are more general, since we do not restrict ourselves to a 'single host' and throughout this paper we study not only finite but also infinite networks.

The systolic conversion preserves not only the structure, that is the underlying graph, of a computational network but also the functions performed by the individual processors. Only the timing, that is the initiation of the processors, and the delays between them are changed. Thus we obtain a very strong equivalent network. In our definition of equivalent networks we require that the networks have the same unrolling, that is they perform step by step identical computations but not necessarily in pairwise matching processors. In Section 4 we consider networks which are not equivalent in this sense but which still perform essentially the same computations. We introduce the notion of one network being (m, k)-simulated by another network. Intuitively this means that the simulating network performs identical computations using processors each of which simulates m original processors and requiring k steps on the simulating network to simulate one step of the original one.

The notion of simulation allows us to compare precisely the power of various well-known networks. It is generally known, even though not stated explicitly in the literature, that the square grid is equivalent to the hexagonal grid. We make this comparison precise and give a number of further practical examples. One of them generalizes a result from [29] and shows that any network on a bidirectional linear array can be transformed into a unidirectional ring of the same size which is half as fast. Similarly, a bidirectional two-dimensional array can be converted into a unidirectional toroid, and similarly for higher dimensions.

We close Section 4 by describing a simple efficient algorithm for distributed sorting on a unidirectional ring of processors. This algorithm is obtained and proved correct by transforming the well-known odd–even transposition sort algorithm from the linear bidirectional array to the unidirectional ring.

In the last section we give further applications of the transformation technique. For instance, we give a simple proof of the result from [25] that global control does not increase the power of m-dimensional iterative arrays. As a new result we demonstrate that the same result also holds for m-dimensional cellular automata.

1. Preliminaries

Given a possibly infinite set V, the set of all finite sequences of elements from V (words) is denoted by V^*. For $x \in V^*$ the length of the sequence x is denoted by $|x|$.

For a finite set of M the cardinality of M is denoted by $|M|$. We use $\mathbb{Z}$ to denote the set $\{\ldots, -1, 0, 1, \ldots\}$ and $\mathbb{N}$ to denote the set $\{0, 1, 2, \ldots\}$, $\emptyset$ denotes the empty set.

Generally we omit double parentheses in cases where f is a function whose argument is a pair (x, y), i.e., we write $f(x, y)$ rather than $f((x, y))$. Similarly for functions returning functions, we prefer to use simpler $\phi(v)(x, y)$ to more precise $(\phi(v))(x, y)$. An *ordered digraph* is a structure $H = \langle V, \pi \rangle$ where V is a (finite or infinite) set of nodes and $\pi: V \to V^*$ is a function. If $\pi(v) = v_1 \ldots v_k$, for v_i in V, then $(v_1, v), \ldots, (v_k, v)$ are (directed) edges (in that order) from nodes v_i to the node v, for $i = 1, \ldots, k$. To stress that the edges are directed we will often use the notation $v_i \to v$ rather than (v_i, v), or even $e: v_i \to v$, if we want to name the edge.

We denote the set of all edges of H by E, and will often talk about the *underlying digraph* $\langle V, E \rangle$. Note that parallel edges and self-loops are allowed in H.

The meaning of terms such as (directed) path, cycle, indegree, outdegree, start node, end node (of a path) will be applied to H meaning the corresponding terms for $\langle V, E \rangle$. Thus, for example the indegree indeg(v) is $|\pi(v)|$, i.e., the length of the word $\pi(v)$. A path $p = (v_0 \to v_1, v_1 \to v_2, \ldots, v_{k-1} \to v_k)$ $(k > 0)$ will be written as $p: v_0 \to v_1 \to \cdots \to v_k$ or, simply as $p: v_0 \to^+ v_k$. The length of p will be denoted by $|p|$. If $u \to v$ we also call u the (immediate) parent of v, and if $u \to^+ v$ we call v the descendant of u.

For the ordered digraph $H = \langle V, \pi \rangle$ according to our definition, the indegree of every node is finite. Thus if we define $V_i = \{v \in V \mid |\pi(v)| = i\}$ for $i = 0, 1, \ldots$, then $(V_0, V_1, \ldots)$ is a partion of V. Note that there is no requirement on the finiteness of an outdegree.

2. Computational schemas and networks

Let Q be a set (finite or infinite). A *pre-computational diagram* (preCD) S is a structure $S = \langle V, \pi, Q, \phi \rangle$ where $\langle V, \pi \rangle$ is an ordered digraph and ϕ is a map (a collection of maps) $\phi: V_k \to |Q^k \to Q|$ for $k > 0$. A preCD is called a *computational diagram* (CD) if the length of any path (of its underlying digraph) with a given end-point is bounded. Formally, for every v in V, there is $b \geq 0$ such that if p is a path with $\text{end}(p) = v$, then $|p| \leq b$. Clearly, there cannot be cycles in a CD.

A *computation* α on a preCD is a map $\alpha: V \to Q$ such that for each $v \in V$ with $\pi(v) = v_1 \ldots v_k$ $(v_i \in V)$, $k > 0$, we have

$$\alpha(v) = \phi(v)(\alpha(v_1), \ldots, \alpha(v_k)), \tag{1}$$

i.e., the 'value' $\alpha(v)$ in Q is computed from the values at all nodes $w \in V$ for which

there is an edge (w, v) in E. Note that (1) imposes no condition on nodes $v \in V_0$; we may call these nodes the *inputs of the* preCD.

2.1. Example. Let us consider the CD $A = \langle V, \pi, \phi \rangle$ where $V = \{a, b, c, d, e\}$, $\pi(a) = \pi(b) = \pi(c) = 1$, $\pi(d) = ab$, $\pi(e) = dc$, $\phi(d)(x, y) = x + y$, $\phi(e)(x, y) = x \cdot y$. CD A is shown in Fig. 1. In the other examples we will show the names of the processors inside the circles, since usually no specific functions will be considered. This very simple CD computes the value $X_c(X_a + X_b)$ in node e from the inputs X_a, X_b and X_c given in nodes a, b, and c, respectively.

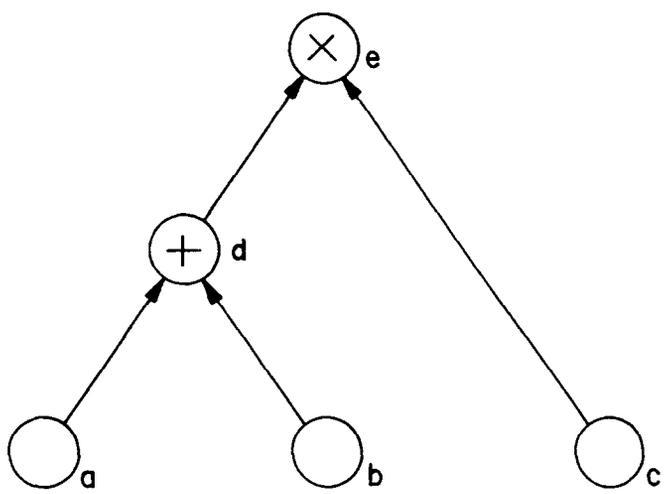


Fig. 1.

2.2. Example. (*Signal processing*). Adopted from [18] is an example of an infinite computation. This is a CD that given w_1, w_2, w_3, w_4 and inputs $x_1, x_2, x_3, \ldots$, computes $y_1, y_2, y_3, \ldots$ where $y_i = w_1 x_i + w_2 x_{i+1} + w_3 x_{i+2} + w_4 x_{i+3}$ (see Fig. 2).

There are five columns; if we number them 0 to 4 (left-to-right), then, in column i for $i = 1, 2, 3$, $\phi(v)$ is a pair (g_1^i, g_2^i) where $g_1^i(S, x) = S + w_{5-i}x$, $g_2^i(s, x) = x$ and, in column 0, $\phi(v)(y) = y$, where S and x are the left and the right component, resp., of the values (pairs) at the previous nodes. Finally, in column 4, $\phi(v)(x) = w_1 x$.

In this example there are five kinds of nodes, one in each column. It is easy to design a CD for the same computation in which all the functions are the same (homogeneous CD).

The important property of CD is that given values at inputs there is a unique computation. Formally, we have the following theorem.

2.3. Theorem. *If $S = \langle V, \pi, Q, \phi \rangle$ is a CD, then for every $\alpha_0 : V_0 \to Q$ there is a unique computation α on S which extends α_0, i.e., for which $\alpha(v) = \alpha_0(v)$ for all v in V_0.*

Proof. For $v \in V$ put $|v| = \max\{|p| \mid p \text{ is a path with the end node } v\}$. By the assumption of boundedness of paths in S, $|v|$ is well defined for each v in V. There is $|v| = 0$ if and only if $v \in V_0$. Eq. (1) defines α recursively. To see this we define partial

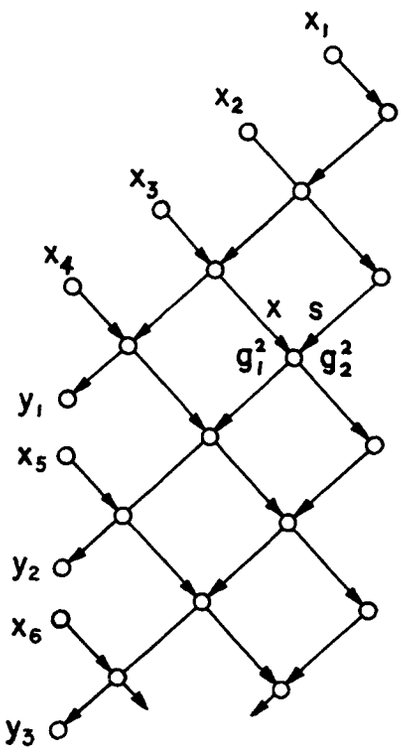


Fig. 2.

maps $\alpha_t: V \to Q$ as follows:

$$\alpha_t(v) = \phi(v)(\alpha_{t-1}(v_1), \ldots, \alpha_{t-1}(v_k)) \quad \text{for } t = |v|$$

and

$$\alpha_t(v) = \alpha_{t-1}(v) \qquad \text{for } t > |v|$$

($\alpha_t(v)$ is undefined if $t < |v|$). Finally, $\alpha(v)$ is defined as $\alpha_{|v|}(v)$. □

Actually, we have proved a somewhat stronger result, namely that given a preCD and a map $\alpha_0: V_0 \to Q$, map α_0 can be extended to all nodes in V for which $|v| < \infty$.

Note that alternatively we could have defined a CD with ϕ defined everywhere on V, that is also on V_0, and interpret (1) for $v \in V_0$ as $\alpha(v) = \phi(v)$. Then there would had been exactly one computation on every CD.

A *computation network* (CN) is a structure $N = \langle V, \pi, Q, \phi, \lambda, r \rangle$ where $H = \langle V, \pi, Q, \phi \rangle$ is a preCD with underlying digraph $\langle V, E \rangle$, λ is a map $E \to \mathbb{Z}$ labeling each edge with an integer $\lambda(e)$. We interpret $\lambda(e)$ as the *delay* (in some time units—clock cycles). Finally, $r: V \to \mathbb{Z}$ is a partial function which determines when a computation (see below) begins at a node v in V.

Note: The interpretation of $\lambda(e)$ as time-delay makes sense only when $\lambda(e) \geq 0$; however, we are not assuming this in general because our results are valid also when λ is possibly negative. In many examples of CN's the functions λ and r will be defined by $\lambda(e) = 1$ for each $e \in E$ and $r(v) = 0$ for each $v \in V$, that is, each edge involves a unit delay, and initial conditions are specified for each processor (node) at time 0 (and thus the computation starts everywhere at time 1).

We shall often omit the functions λ and/or r from the description of a network. Such an omission means that the 'default' functions $\lambda(e)=1$ and $r(v)=0$ are considered. In most examples Q and ϕ are left unspecified, in that case it is understood that arbitrary Q and ϕ are considered.

To define computations on CN, we associate with $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ the ordered graph $G_N=\langle V\times\mathbb{Z}, \pi'\rangle$ where π' is defined as follows. If $\pi(v)=v_1v_2\ldots v_k$, and $d_i=\lambda(v_i, v)$ $(v_i\in V, 1\leqslant i\leqslant k)$ then $\pi'(v, t)=(v_1, t-d_1)(v_2, t-d_2)\ldots(v_k, t-d_k)$.

$S=\langle V\times\mathbb{Z}, \pi', Q, \phi'\rangle$, where $\phi'(v, t)=\phi(v)$, is a preCD which is meant to describe computations on N spread in time. The problem is that S is a proper preCD with all paths of infinite length, and thus there are no computations in our sense on S. We are interested in computations starting at a particular time and continue from that time on. This is why we have the function τ.

Let us consider the following three subsets of $V\times\mathbb{Z}$:

(i) $\boldsymbol{S}$, the starting set is $\{(v, \tau(v))\,|\,v\in\mathrm{dom}(\tau)\}$,

(ii) $\boldsymbol{D}$, the *descendants* of S, i.e., the set $\{(v, t)\,|\,(v, t)\notin\boldsymbol{S}$ and there is $(s, t_0)\in\boldsymbol{S}$ such that $(s, t_0)\rightarrow^+(v, t)\}$,

(iii) $\boldsymbol{P}$, the *parents of* $\boldsymbol{D}$ is the set $\{(v, t)\,|\,(v, t)\rightarrow(u, t')$ for some $(u, t')\in\boldsymbol{D}\}$.

In both cases $\rightarrow^+$ and $\rightarrow$ are paths and edges in G_N. Denote by $\hat{V}=\boldsymbol{S}\cup\boldsymbol{D}\cup\boldsymbol{P}$. A *computation* α on N is a function $\alpha\colon\hat{V}\rightarrow Q$ such that, for all $(v, t)\in\boldsymbol{D}$,

$$\alpha(v, t)=\phi(v)(\alpha(v_1, t-d_1), \ldots, \alpha(v_k, t-d_k)). \tag{2}$$

Note that it is often convenient to consider α as a partial function $V\times\mathbb{Z}\rightarrow Q$. As this cannot lead to ambiguity we shall do so, when convenient.

Intuitively a computation on a network N proceeds as follows. Arbitrary 'initial' values are chosen at the processors in $\mathrm{dom}(\tau)$. More specifically, an initial value from Q is chosen for each processor v in $\mathrm{dom}(\tau)$ at the time $\tau(v)$. Then new values are successively computed according to (2). During this computation arbitrary 'input' values are supplied, when needed, at the nodes without parents.

We are not concerned with formalizing how outputs are produced. In any particular network outputs can be taken from suitable processors at suitable times to realize a desired input-output function. This is the same situation as for gate networks where for many considerations it is not relevant whether the output of a gate is external or not.

We are mainly interested in computational networks. The computational diagrams are auxiliary constructs, they are important because for every CN N there is a CD H, called the unrolling of N, such that each computation α on N has an 'isomorphic' computation on H.

Let $N=\langle V, \pi, Q, \phi, \tau\rangle$ be a CN. The *unrolling* $\hat{N}$ of N is the preCD $\hat{N}=\langle\hat{V}, \hat{\pi}, Q, \hat{\phi}\rangle$ where $\hat{\pi}$ is the restriction of π' to $\hat{V}$ and $\hat{\phi}(v, t)=\phi(v)$ for (v, t) in $\hat{V}$ ($\hat{V}$ and π' are defined above). It is easy to see that $\hat{N}$ is not always a CD; for example, $\hat{N}$ might contain cycles, and thus computations need not exist on every $\hat{N}$. On the other hand, it is obvious that $\alpha\colon V\times\mathbb{Z}\rightarrow Q$ is a computation on a CN N if and only if it is a computation on its unrolling $\hat{N}$.

Given a network $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$, let $\langle V, E\rangle$ be the underlying graph. We may extend λ from edges to paths by putting, for $p: v_0 \to v_1 \to \cdots \to v_k$, $\lambda(p) = \sum_{i=1}^{k} \lambda(v_{i-1}, v_i)$.

For each network N there is a preCD, namely $\hat{N}$ which has (by definition) the same computations. It is easy to show that conversely, given an arbitrary CD $S=\langle V, \pi, Q, \phi\rangle$ we can construct a CN with the same computations as S. To do so, we use the function $|v|: v \to \mathbb{N}$ defined in the proof of Theorem 2.3, and define $\lambda(u, v) = |v| - |u|$ for all $u \to v$. If we also define τ by $\tau(v)=0$ for $v \in V_0$, then $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ is a network with the same computations as S.

The network N has a useful property, it generalizes the notion of pure network of [6]. We say that a network N is *pure* if:

(i) $\mathrm{dom}(\tau) = V_0$, the set of nodes with indegree 0, and $\tau(v)=0$ for $v \in V_0$,

(ii) for each node $v \in V - V_0$ there is a path $p: v_0 \to^+ v$, $v_0 \in V_0$, and

(iii) all such paths $p: v_0 \to^+ v$ where $v_0 \in V_0$ and $v \in V - V_0$ have the same weight $\lambda(p)$, i.e., $\lambda(p)$ depends only on the end node of p.

It is easy to see, that more generally, for every two paths $p_1: u \to^+ v$ and $p_2: u \to^+ v$ there is always $\lambda(p_1) = \lambda(p_2)$. Now we show that for a pure network $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ the computations on N are the same as on the preCD $\langle V, \pi, Q, \phi\rangle$.

2.4. Theorem. *Let $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ be a pure CN. Then its unrolling $\hat{N} = (\hat{V}, \hat{\pi}, Q, \hat{\phi})$ is isomorphic to preCD $\langle V, \pi, Q, \phi\rangle$.*

Proof. Let $d(v)$ be the uniquely defined distance of a node v in V from V_0 (nodes with indegree 0). Clearly, $\hat{V} = \{(v, d(v)) \mid v \in V\}$ and thus the mapping $v \to (v, d(v))$ establishes the isomorphism. □

3. Retiming and systolic conversion

An important property of computations on CN is that if they exist they are uniquely defined given some 'initial values', i.e. an analog of Theorem 2.3 holds for CN's.

First, we observe that in general there might not exist any computation on a CN. This can happen on networks with zero or negative 'delays' in $\lambda(u, v)$. Following [22] we define: A CN $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ is *semisystolic* if $\lambda(v, w) \geq 0$ for each edge $v \to w$. It is *systolic* if $\lambda(v, w) > 0$ for each $v \to w$ in E.

Note: In [22] a semisystolic network (simpler version) was called a synchronous system.

It is easy to see that there always exist computations on a systolic network on which $\tau(v)$ is bounded from below. However, we will not restrict ourselves to this case and will consider even networks which are not systolic.

3.1. Example. Consider $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ where $V=\{1, 2, \ldots\}$, $\pi(k)=k+1$ for $k\geqslant 1$, Q and ϕ are arbitrary, $\lambda(e)=0$ for all $e\in E$ and $\tau(k)=k$ for all $k\in V$ (see Fig. 3).

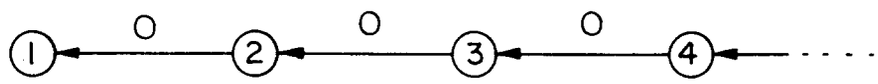


Fig. 3.

Clearly, there is no computation on the related preCD $S=\langle V, \pi, Q, \phi\rangle$; however, for the τ defined above (and suitable ϕ) there are computations on N.

Now, we want to find conditions under which a general CN can be transformed to a semisystolic CN or even a systolic CN with the same computations. First, we have to make precise the notion of equivalence for CN. We say that two CN's are *equivalent* if their unrollings are isomorphic as preCD's.

3.2. Lemma. (Retiming Lemma of [22]). *Given a CN* $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ *and a function* δ, *called lag,* $\delta: V\to\mathbb{Z}$, *define* $\lambda_\delta: E\to\mathbb{Z}$ *by*

$$\lambda_\delta(u, v)=\lambda(u, v)-\delta(u)+\delta(v) \tag{3}$$

for each $e=u\to v$ *in* E, *and* $\tau_\delta: V\to\mathbb{Z}$ *by* $\tau_\delta(v)=\tau(v)+\delta(v)$. *Consider CN* $N_\delta=\langle V, \pi, Q, \phi, \lambda_\delta, \tau_\delta\rangle$. *If* $\alpha: \hat{V}\to Q$ *is a computation on* N, *and* $\alpha_\delta: \hat{V}_\delta\to Q$ *is defined by* $\alpha_\delta(v, t)=\alpha(v, t-\delta(v))$ *for all* $(v, t)\in \hat{V}_\delta$, *then* α_δ *is a computation on* N_δ.

Proof. Clearly, α and α_δ are identical computations when considered as computations on unrollings $\hat{N}$ and $\hat{N}_\delta$. □

3.3. Corollary. *Networks* N *and* N_δ *are equivalent.*

The following two theorems are generalizations of the Systolic Conversion Theorem from [22]. They give the necessary and sufficient conditions for the existence of a lag-function δ which converts a CN network to an equivalent semisystolic (systolic) network. Given a CN $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ we define

$$\rho: V\times V\to\mathbb{Z}\cup\{-\infty, \infty\} \quad \text{by } \rho(u, v)=\inf\{\lambda(p)\,|\,p: u\to^+ v\}$$

for all u, v in V. (By definition, $\inf(\emptyset)=\infty$.)

3.4. Theorem. *Let* $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ *be a CN. There exists a lag-function* $\delta: V\to\mathbb{Z}$ *such that CN* $N_\delta=\langle V, \pi, Q, \phi, \lambda_\delta, \tau_\delta\rangle$ *(equivalent to* N*) is semisystolic if and only if*

$$\rho(u, v)>-\infty \quad \textit{for all } u, v \textit{ in } V. \tag{4}$$

Proof. Assume (4) holds. First we show three properties of ρ following from (4).

3.5. Claim.

(i) $\rho(u, v)+\rho(v, w) \geq \rho(u, w)$ *for all* u, v, w *in* V.

(ii) $\rho(u, u) \geq 0$ *for each* u *in* V.

(iii) $\rho(u, v)+\rho(v, u) \geq 0$ *for all* u, v *in* V.

Proof. (i) If $u \to^+ v$ and $v \to^+ w$, i.e. $\rho(u, v)$ and $\rho(v, w)$ are finite, then $u \to^+ w$, thus $\rho(u, w)$ is finite and the path $e: u \to^+ w$ is considered when calculating the infimum defining $\rho(u, w)$. Thus $\rho(u, w) \leq \rho(u, v)+\rho(v, w)$. If there is no path $u \to^+ v$ or $v \to^+ w$, then the right-hand side of (i) is ∞ and (i) holds trivially.

(ii) Let $d=\rho(u, u)$ and $d<0$. This means that there is a path $p: u \to^+ u$ with $\lambda(p)=d$, but then p followed by p is also a path $u \to^+ u$ and $\lambda(p \cdot p)=2d<d$. This contradicts the assumption that

$$\rho(u, u)=\inf\{\lambda(p) \mid p: u \to^+ u\}.$$

(iii) The last inequality immediately follows from (i) and (ii). This completes the proof of the claim. □

Proof of Theorem 3.4 (*continued*). Now we show that it is possible to define a lag-function $\delta: V \to \mathbb{Z}$ so that $\lambda_\delta(u, v) \geq 0$ for each edge $u \to v$, and where λ_δ is defined by (3.) Let U be a maximal subset of V on which $\delta: U \to \mathbb{Z}$ can be defined so that

$$\delta(u)+\rho(v, u) \geq \delta(v) \geq \delta(u)-\rho(u, v) \tag{5}$$

holds for all u, v in U. If $U \neq V$, then consider $w \in V-U$. By claim 3.5 (iii) we have $\rho(w, u) \geq -\rho(u, w)$ for each u in U. Therefore $\delta(u)+\rho(w, u) \geq \delta(u)-\rho(u, w)$, and we can choose $\delta(w)$ so that

$$\delta(u)+\rho(w, u) \geq \delta(w) \geq \delta(u)-\rho(u, w).$$

From this we immediately have $\delta(w)+\rho(u, w) \geq \delta(u)$ and $\delta(u) \geq \delta(w)-\rho(w, u)$ which contradicts the fact, that U is a maximal subset of V for which (5) holds. Thus $U=V$, and δ can be defined on the whole V. By the Retiming Lemma, N_δ and N are equivalent and, by (5),

$$\lambda_\delta(u, v)=\lambda(u, v)-\delta(u)+\delta(v) \geq \lambda(u, v)-\rho(u, v).$$

For every edge $u \to v$, $\rho(u, v) \leq \lambda(u, v)$ by the definition of ρ, therefore $\lambda_\delta(u, v) \geq 0$, which means that N_δ is semisystolic.

To prove the converse, assume that there is a δ such that N_δ is semisystolic. That means $\lambda_\delta(p) \geq 0$ for each path $p: u \to^+ v$ in N_δ and, by (3), $\lambda(p) \geq \delta(u)-\delta(v)$. This gives a lower bound for $\rho(u, v) \geq \delta(u)-\delta(v)>-\infty$, hence (4) holds. □

Given a CN $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$, let $[N-1]$ be the CN $[N-1]=\langle V, \pi, Q, \phi, [\lambda-1], \tau\rangle$ where $[\lambda-1]: E \to \mathbb{Z}$ is given by $[\lambda-1](e)=\lambda(e)-1$. More generally, we can define $[f(N)]$for every function $f: \mathbb{Z} \to \mathbb{Z}$; however, we shall need only $[N-1]$, $[N+1]$, and $[kN]$ for $k=2, 3, \ldots$. Clearly, N is systolic if and only

if $[N-1]$ is semisystolic. For CN $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ we define $\sigma: V\times V\to \mathbb{Z}\cup\{\infty, -\infty\}$ by $\sigma(u, v)=\inf\{\lambda(p)-|p|\,|p: u\to v\}$ for all u, v in V. Now we are ready for the following result.

3.6. Theorem. (Systolic Conversion Theorem). *Let $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ be a CN. There exists a lag-function $\delta: V\to\mathbb{Z}$ such that CN $N_\delta=\langle V, \pi, Q, \phi, \lambda_\delta, \tau_\delta\rangle$ (equivalent to N) is systolic if and only if*

$$\sigma(u, v)>-\infty \quad \textit{for all } u, v \textit{ in } V. \tag{6}$$

Proof. We have already noted that N_δ is systolic if and only if $[N_\delta-1]$ is semisystolic. Clearly $[N_\delta-1]=[N-1]_\delta$. By Theorem 3.4 there exists a δ such that $[N-1]_\delta$ is semisystolic if and only if $\rho_{[N-1]}(u, v)\to-\infty$ for all u, v in V. However, $\rho_{[N-1]}(u, v)=\sigma_N(u, v)$ for all u, v in V. Thus there exists a δ such that N_δ is systolic if and only if (6) holds. □

3.7. Corollary. *Let $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ be a semisystolic CN in which paths of weight zero are bounded. Then there exist $k>0$ and a lag-function $\delta: V\to\mathbb{Z}$ such that $[kN]_\delta$ is systolic.*

3.8. Corollary. *Let $N=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ be a finite semisystolic CN. Then there exist $k>0$ and a lag-function $\delta: V\to\mathbb{Z}$ such that $[kN]_\delta$ is systolic if and only if there are no cycles of weight zero in N.*

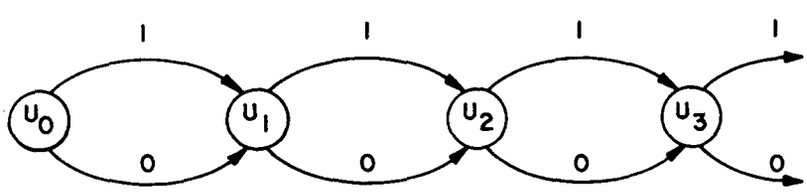


Fig. 4.

3.9. Example. Consider CN N given by the diagram in Fig. 4. N is an example of a semisystolic network which can be transformed in a systolic one without any change of time (speed). The resulting network is shown in Fig. 5.

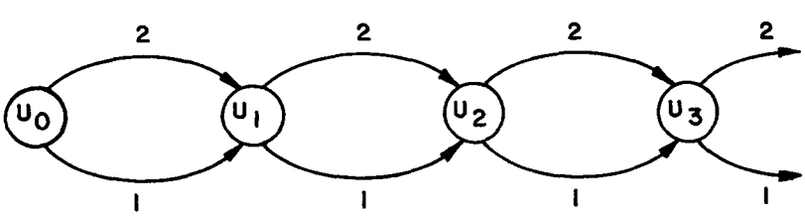


Fig. 5.

In the next example the network M itself cannot be converted; however, the network $[2M]$ can be.

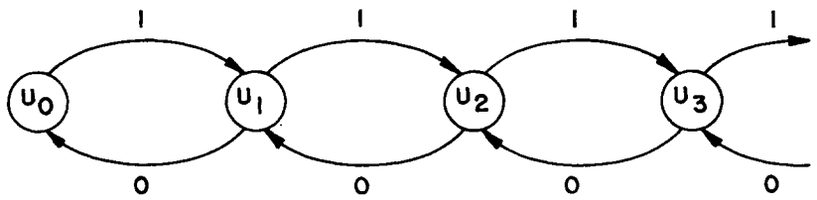


Fig. 6.

3.10. Example. Consider network M given in Fig. 6. The network $[2M]$ which may be interpreted as M running twice slower, can be converted to an equivalent network $[2M]_\delta$, for $\delta(u_k) = -k$, $k = 0, 1, 2, \ldots$, shown in Fig. 7.

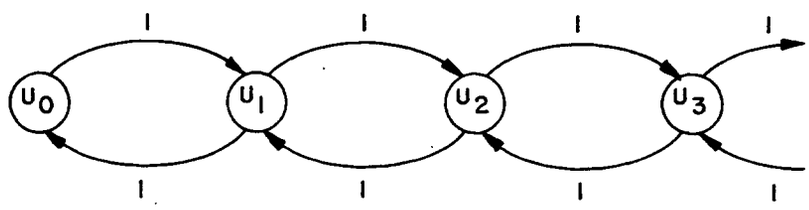


Fig. 7.

The next example shows that arbitrary large k (slowdown) may be needed in Corollary 3.8.

3.11. Example. Consider the CN UR_k with k nodes given in Fig. 8, called unidirectional ring. Here the network $[iUR_k]_\delta$ for $i \geq k$ is systolic for suitable δ; however, it cannot be converted to a systolic network for $i < k$.

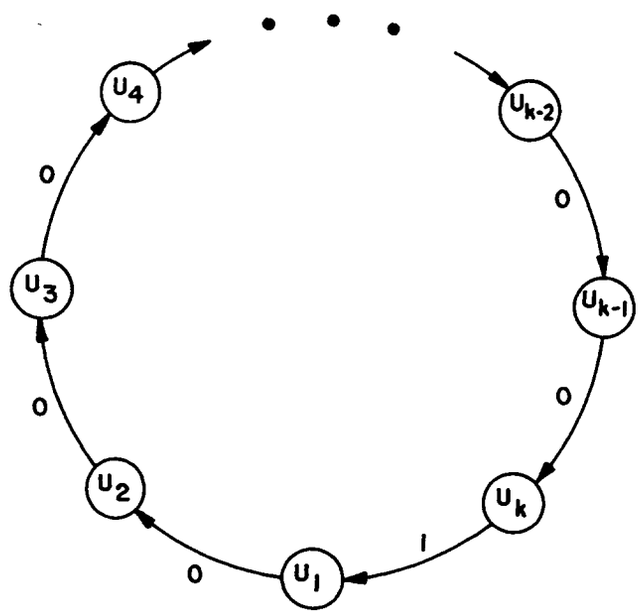


Fig. 8.

Finally, the following example shows that for an infinite network a conversion from semisystolic to systolic is not always possible.

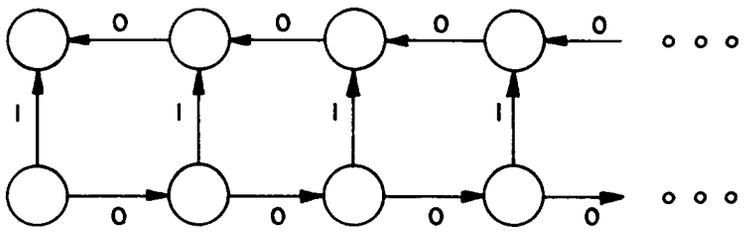


Fig. 9.

3.12. Example. Consider the network N given in Fig. 9. It is easy to verify that for no k the network $[kN]$ satisfies condition (6) from Theorem 3.6.

4. Simulation of networks

In this section we shall investigate networks performing essentially identical computations, however not necessarily equivalent in the sense of the previous section.

Given two CD's $G_i = \langle V_i, \pi_i, Q_i, \phi_i \rangle$ $(i = 1, 2)$ we say that G_2 *simulates* G_1 or that G_1 *is simulated on* G_2 if there are two maps $\rho\colon V_1 \to V_2$ and $\psi\colon V_1 \times Q_2 \to Q_1$ such that for every computation α_1 on G_1 there exists a computation α_2 on G_2 for which

$$\alpha_1(v) = \psi(v, \alpha_2(\rho(v))).$$

We say that the pair of maps ρ, ψ establishes the simulation, or that G_2 simulates G_1 through ρ, ψ.

The notion of simulation is too general; for example, every finite CD (i.e., V and Q finite) can be simulated on a finite automaton. Therefore we restrict ρ by requiring the existence of a bound m on the number of nodes that can be merged by ρ, i.e., $|\rho^{-1}(u)| \leq m$ for all $u \in V_2$. If G_2 simulates G_1 through ρ, ψ where ρ meets this restriction then we say that G_2 *m-simulates* G_1.

Now, given two networks N_1, N_2 we extend the definition of simulation to CN by saying that a network N_2 simulates N_1 if and only if the unrolling of N_2 simulates the unrolling of N_1. In more details let $N_i = \langle V_i, \pi_i, Q_i, \phi_i, \lambda_i, \tau_i \rangle$, $i = 1, 2$, be two networks. If N_2 m-simulates N_1 through ρ, ψ we can relate time of the computation on N_2 to the time of the computation on N_1. Thus if $\rho\colon (v, t) \mapsto (v', t')$ we denote the second component of ρ by ρ_t, i.e. $\rho_t(v, t) \mapsto t'$.

If

$$\sup \left\{ \frac{\rho_t(v_1, t_1) - \rho_t(v_2, t_2)}{t_1 - t_2} \,\middle|\, (v_1, t_1), (v_2, t_2) \in \hat{N}_1 \right\} = k,$$

we say that N_2 (m, k)-simulates N_1.

Intuitively, k corresponds to the slowdown of N_1 in order to enable N_2 to simulate it.

4.1. Lemma. *Let $N_1 = \langle V_1, \pi_1, Q_1, \phi_1, \lambda_1, \tau_1 \rangle$ be a systolic CN and $G = \langle V_2, E_2, \lambda_2 \rangle$ be a labeled digraph with $\lambda_2(e) > 0$ for all $e \in E_2$. Let there be a map $\gamma: V_1 \to V_2$ such that:*

(i) *There is a fixed $k > 0$ such that for every edge $e: u \to v \in E_1$ there is a path $p(e): \gamma(u) \to^+ \gamma(v)$ in G of weight $\lambda_2(p(e)) = k\lambda_1(e)$.*

(ii) *For all $u, v \in V_1$, if $\gamma(u) = \gamma(v)$, then either $\tau(u) = \tau(v)$ or τ is undefined for both u and v.*

(iii) *There is m such that $|\gamma^{-1}(v)| \leq m$ for all $v \in V_2$.*

Then we can construct a CN $N_2 = \langle V_2, \pi_2, Q_2, \phi_2, \lambda_2, \tau_2 \rangle$ such that the underlying graph of N_2 is $\langle V_2, E_2 \rangle$ and N_2 (m, k)-simulates N_1. Moreover, if the indegree of $\langle V_1, E_1 \rangle$ is finite, then finite Q_1 implies finite Q_2.

Proof. Since the idea of the proof is fairly straightforward we will omit the lengthy technical details and give only the outline of the construction of N_2.

First, π_2 is chosen arbitrarily so that $\langle V_2, E_2 \rangle$ is the underlying graph of N_2. Next, given this π_2, we construct Q_2 (tuples of elements of Q_1) and ϕ_2 as follows. Each processor of N_2, i.e. $\phi_2(v)$ for each $v \in V_2$, performs two kinds of tasks. It simulates concurrently all the processors in $\gamma^{-1}(v)$ (at most m) and if v is a node other than the end-node of any path $p(e)$, $e \in E_1$, then v passes values (from Q_1) along that path. If a node lies on several paths (not end-node), then it passes, generally different, values along each path.

τ_2 is defined as follows. If $u \in \mathrm{dom}(\tau_1)$, then $\gamma(u) \in \mathrm{dom}(\tau_2)$ and $\tau_2(\gamma(u)) = \tau_1(u)$. Condition (ii) assures that τ_2 is well defined. Using the conditions (i) and (iii) it is easy to verify that CN N_2 (m, k)-simulates CN N_1.

Finally, if the indegree of $\langle V_1, E_1 \rangle$ is finite (as is the case in all practical applications), then the number of all paths $p(e)$ going through any fixed node is uniformly bounded, so the length of the tuples in Q_2 is bounded too implying the finiteness of Q_2. □

Note that if we are interested in networks with processors of limited complexity, that is, functions ϕ restricted to a certain class of functions, then we notice that the construction in the above proof preserves those classes of functions that contains all projections and are closed under composition.

Two special cases of Lemma 4.1 often occur.

4.2. Corollary. *Let $G_1 = \langle V_1, E_1 \rangle$ be a graph and let $\{U_j | j \in V_2\}$ be a partition of V_1. We let $G_2 = \langle V_2, E_2 \rangle$ where $(u, v) \in E_2$ if and only if $u \in U_a$, $v \in U_b$ and there is an edge in E_1 from some node in U_a to a mode in U_b. For every network N_1 on G_1 there is a network N_2 on G_2 such that N_1 is $(m, 1)$-simulated on N_2. Here again $m = \max_{j \in V_2} |U_j|$.*

4.3. Corollary. *Let G_1 be a subgraph of G_2, i.e., $G_i = \langle V_i, E_i \rangle$, $i = 1, 2$, and $V_1 \subset V_2$ and $E_1 \subset E_2$. Then every network N_1 on G_1 can be $(1, 1)$-simulated by a network N_2 on G_2.*

Finally, the following lemma is easy to prove.

4.4. Lemma. *Let N_1, N_2, N_3 be networks such that $N_2(i_1, j_1)$-simulates N_1 and N_3 (i_2, j_2)-simulates N_2. Then N_3 $(i_1 i_2, j_1 j_2)$-simulates N_1.*

The following lemma is useful in proving for given two networks, that one cannot simulate the other.

4.5. Lemma. *Let $N_i = \langle V_i, \pi_i, Q_i, \lambda_i, \tau_i \rangle$, $i = 1, 2$, be two networks. If N_2 (m, n)-simulates N_1 through ρ and Ψ, then for every path $p: v_0 \to v_1 \to \cdots \to v_k$ in the underlying graph of N_1 $\rho(p): \rho(v_0) \to^+ \rho(v_1) \to^+ \cdots \to^+ \rho(v_k)$ is a path of N_2. If, moreover, $\lambda_1 = \lambda_2 = 1$, then $|\rho(p)|/|p| \leq n$.*

In the examples throughout the paper we will frequently use bidirectional communications between processors (nodes), and processors (nodes) with selfloops. We introduce the notational abbreviations for these cases as shown in Fig. 10. Another abbreviation also shown in Fig. 10 is the omission of 0-degree nodes (input processors), only the 'half edges' entering the other nodes are shown.

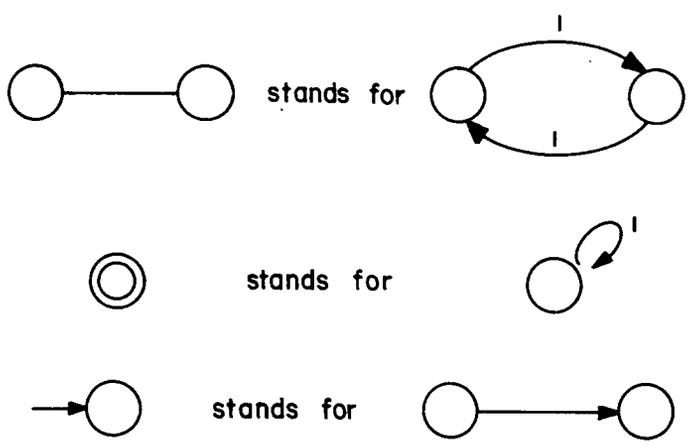
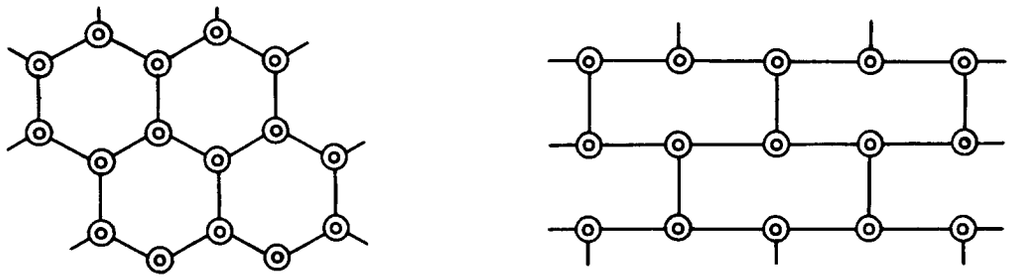


Fig. 10.

Finally, we would like to recall the following conventions. Omitted edge-label implies the value of λ for this edge is one. When τ is not explicitly given, we assume that τ is defined for all the nodes and is zero everywhere.

4.6. Example (*Simulations between hexagonal and square grid networks*). A hexagonal grid network, i.e. a grid as in Fig. 11(a) can be drawn as in Fig. 11(b); that is, as a square grid network with some connections omitted. Therefore, a hexagonal grid network can be (1, 1)-simulated by a square grid network.

To show the converse, consider any two nodes connected by an edge of the square grid. Clearly, there is a path between them in the hexagonal grid of the length at most 3, and because of the presence of self-loops there is a also a path of length exactly 3. Thus every square grid network can be (1, 3)-simulated by a hexagonal grid network. Note that if a pair of nodes can be connected by a path of length 2



(a) (b)

Fig. 11.

on the square grid, then the corresponding pair of nodes on the hexagonal grid can be connected by a path of length 4. Thus a more elaborate construction would allow slowdown of only 2 rather than 3 as in our construction above.

4.7. Example (*Simulations between square and triangular grid networks*). When we draw the triangular grid as shown in Fig. 12, we see that similar considerations as in Example 4.6 show that a square grid network can be (1, 1)-simulated by a triangular grid network and that (1, 2)-simulation is possible in the reverse direction.

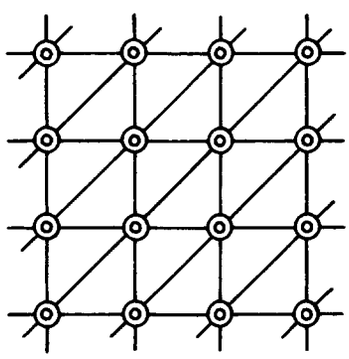


Fig. 12.

4.8. Example (*Simulations between bidirectional ring and bidirectional linear array* (each consisting of n nodes)). Let us denote these networks BR_n and BA_n, respectively. (1, 1)-simulation of BA_n on BR_n is trivial, since BR_n is obtained from BA_n by omitting one edge.

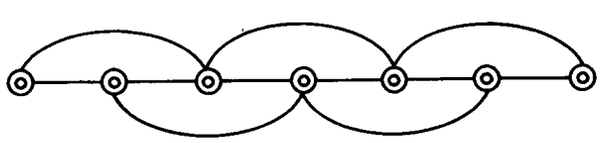


Fig. 13.

To prove the converse, we first note that it follows by Lemma 4.1 that the network M_n given (for $n=7$) in Fig. 13 can be (1, 2)-simulated on BA_n. Now BR_n can be (1, 1)-simulated on M_n since it is obtained from M_n by removing all but two 'short' edges, which is demonstrated by drawing BR_7 as in Fig. 14.

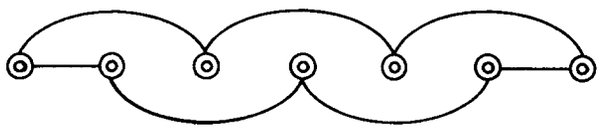


Fig. 14.

4.9. Example (*Simulations between homogeneous* BR_n *and the homogeneous unidirectional ring with n nodes* (UR_n)). Again BR_n trivially simulates UR_n. To show the converse we must assume that BR_n is homogeneous, i.e., all its processors perform identical functions ($\phi(u) = \phi(v)$ for u, v in V). We show the simulation in two steps.

Let $BR_n = \langle V, \pi, Q, \phi \rangle$ where $V = \{v_1, \ldots, v_n\}$ (BR_n has the 'default' λ, r). Consider network $C_n = \langle V, \pi_c, Q, \phi \rangle$ where $\pi_c: V \to V^*$ is defined by $\pi_c(v_i) = v_i v_{i\oplus 1} v_{i\oplus 2}$, where $\oplus$ means addition mod n. C_7 is shown in Fig. 15. Clearly, since the networks BR_n and C_n are homogeneous $\rho: (v_i, t) \mapsto (v_{i\oplus t}, t)$, for $i = 1, \ldots, n$ and $t \geq 0$, establishes the isomorphism of the unrollings of BR_n and C_n. Thus BR_n and C_n are equivalent, and also each (1, 1)-simulates the other. By Lemma 4.1 C_n can be (1, 2)-simulated on UR_n and therefore also BR_n can be (1, 2)-simulated on UR_n.

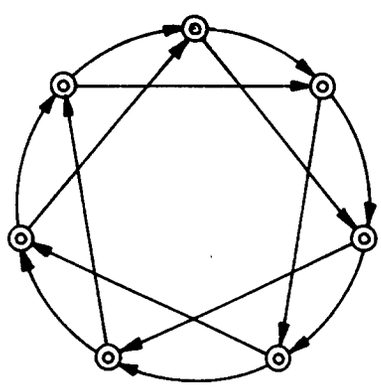


Fig. 15.

4.10. Example (*Simulations between bidirectional linear array* (BA_n) *and bidirectional linear array without selfloops* (W_n)). Network W_7 is shown in Fig. 16. Trivially, BA_n (1, 1)-simulates W_n. We show that BA_n can be (1, 2)-simulated on W_{2n-1} and W_{2n-1} can be (2, 1)-simulated on BA_n. Let $V_n = \{1, \ldots, n\}$ be the nodes of both BA_n and W_n. The map $\gamma: i \mapsto 2i - 1$, $i = 1, \ldots, n$, maps V_n into V_{2n-1}. Clearly, for every edge of BA_n there is a path of length 2 in W_n connecting the corresponding nodes. Thus W_{2n-1} simulates BA_n by Lemma 4.1. The converse is easy to see by mapping pairs of nodes $2i-1$, $2i$ of W_n, into one node i of BA_n (except the last node n).

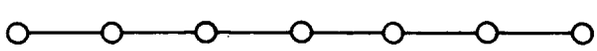


Fig. 16.

4.11. Example. Note that the unrolling of W_n has two disjoint components, only one of them is used in our (1, 2)-simulation of BA_n on W_{2n-1}. Here we assumed the default definition of function τ, i.e., τ defined everywhere and equal to zero. If

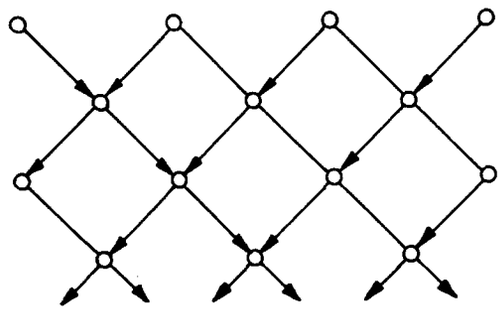


Fig. 17.

we restrict τ only to the 'old' nodes of W_{2n-1} then the unrolling of this modified $\tilde{W}_{2n-1}$ has only one component. The unrolling of $\tilde{W}_7$ is shown in Fig. 17. We can also consider pure homogeneous network $T_{2n-1} = \langle V, \pi, Q, \phi, \lambda, \tau \rangle$ where V, π, Q, ϕ are as in the unrolling of $\tilde{W}_{2n-1}$, $\lambda = 1$ for all edges and τ is defined (and equal to zero) only on the top row. Clearly, the unrollings of T_{2n-1} and $\tilde{W}_{2n-1}$ are isomorphic and therefore T_{2n-1} and $\tilde{W}_{2n-1}$ are equivalent.

We can summarize the results demonstrated in Examples 4.8 to 4.11 in the following Theorem 4.12. This theorem generalizes similar results for various types of cellular, iterative or trellis automata defined on structures like unidirectional linear arrays, bidirectional arrays, and rings.

4.12. Theorem. *Any of the homogeneous CN of the following type can be* (i, j)*-**simulated on any other with** $i, j \leq 2$.

(i) *bidirectional ring* BR_n,
(ii) *unidirectional ring* UR_n,
(iii) *bidirectional linear array* BA_n,
(iv) *bidirectional linear array without selfloops (memory)* W_{2n-1},
(v) *trellis* T_{2n-1}.

In more details, Table 1 shows the values of (i, j). A pair i, j in the intersection of a row and column means that a network named in the row (i, j)-simulates the network named in the column. The values not shown in Examples 4.8 to 4.11 follow from transitivity of simulation. The exception is the pair 2, 2. From the transitivity we get 1, 4, but the value 2, 2 can be shown easily. This result means that a solution

Table 1.

	BA_n	BR_n	UR_n	W_{2n-1}	T_{2n-1}
BA_n	1, 1	1, 2	1, 2	2, 1	2, 1
BR_n	1, 1	1, 1	1, 1	2, 1	2, 1
UR_n	1, 2	1, 2	1, 1	1, 1	1, 1
W_{2n-1}	1, 2	2, 2	2, 2	1, 1	1, 1
T_{2n-1}	1, 2	2, 2	2, 2	1, 1	1, 1

(systolic algorithm) for a problem for a network of one of the above types can be easily converted into a network (systolic algorithm) of any of the other type. Also note that the simulations between networks of type (iii)-(v) are also valid for corresponding (potentially) infinite networks.

Note that we are implicitly assuming that the 'cost' of computing a new value $\phi(v)(q)$ is constant. In some applications this is not the case. For example, if a processor with selfloop is implemented as a device with memory then it is typically much cheaper to update one location than to rewrite the whole memory. Thus if a single update means to change the value q to q', then this can be easily done in one clock step inside one processor but might be impossible to do in our step in the neighboring processor, since this could require to communicate the whole contents of the memory in one step.

Theorem 4.12 can be used to design systolic algorithms and prove their correctness. For example, the unidirectional (systolic) ring has been implemented by Ostlund and used specifically for computations in molecular dynamics (see [23]). The algorithms described in [23] were designed directly for the unidirectional ring. However, having Theorem 4.12, it is typically easier to program such algorithms, and in particular to prove their correctness, for bidirectional linear arrays and then transform them to unidirectional rings. The algorithm for distributed sorting is designed using this method in the following example.

4.13. Example. A distributed sorting algorithm is designed and proved correct for the unidirectional ring of microprocessors each of which can store the same number of records (numbers). The well-known odd-even transportation sort for $2n$ records [15, p. 241] can be easily implemented on the network T_{2n-1} in time $2n-1$. Each processor sorts two records (numbers), with suitable modification for the 'end-processors'. Now we use the result that every sorting network also works for multisets when we start with sorted multisets and replace the operation of sorting two elements by merging two multisets, see [15, p. 241]. By Theorem 2.4 the unrolling of $\tilde{W}_{2n-1}$ is isomorphic to $\tilde{W}_{2n-1}$ itself. Thus both $\tilde{W}_7$ and its unrolling are shown in Fig. 17. We compare it with the unrolling of UR_4 shown in Fig. 18. We see that by omitting

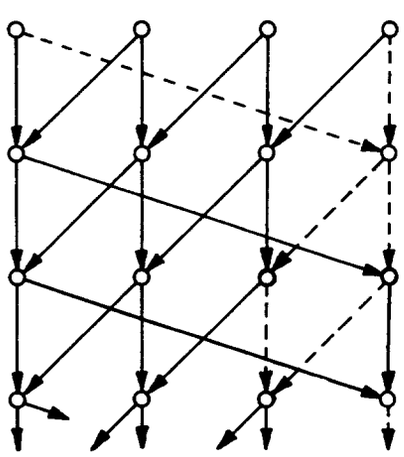


Fig. 18.

one node in each even row (and therefore also the dotted edges) in Fig. 18 we obtain a subgraph isomorphic to the one shown in Fig. 17.

It is now easy to verify that the following is a correct algorithm for $(2n-1)$-step sorting on UR_n. We assume that initially there is a sorted multiset of at most $2k$ elements in each processor (if necessary local sorting is performed first). Then in each step each processor performs the following. It sends the 'left half' of the multiset, i.e., the k smallest elements (or all of them if there are less than k of them) to the left neighbor and merges the remaining elements with those coming from the right neighbor. The only exception is that no sorting is done across 'the fence'. The fence is originally between the processors v_n and v_1. The fence is sent at 'half speed' through the processor, i.e., it is in the 'middle' of processor v_{n-t} at the time $2t-1$ and that processor is 'inactive'. After $2n-1$ steps the 'fence' returns to its original position and the sorting is completed as shown in the following example of 3 processors, each containing 4 numbers. The bar represents the fence.

|	8	6	3	4	5	6	3	1	9	1	4	6
	5	6	6	8	1	1	3	4	6	9 |	3	4
	1	1	6	8	3	4	6	9 |	3	4	5	6
	3	4	6	8	6	9 |	3	4	1	1	5	6
	6	6	8	9 |	1	1	3	4	3	4	5	6
	8	9 |	1	1	3	3	4	4	5	6	6	6
|	1	1	3	3	4	4	5	6	6	6	8	9

Note that if some processor is not 'full', i.e. contains less than $2n$ elements, it still sends n elements to the left, which means that it pretends that it contains additional dummy elements considered larger than all the other, and therefore these dummies are always retained. Thus the algorithm is correct also in this case. Alternatively we can retain the n largest elements and send the rest to the left thus treating the nonexistent elements as the smallest.

Except for the pure network T_{2n-1}, our Theorem 4.12 deals with one-dimensional structures. This result can be generalized to m-dimensional structures (arrays and 'toroidal structures'). We formulate it for the most interesting case of the two-dimensional structures, the other cases are left for the reader.

4.14. Example (*Simulations between homogeneous bidirectional two-dimensional array* ($BA_{m,n}$) *and homogeneous unidirectional two-dimensional toroid* ($UT_{m,n}$)). $BA_{4,3}$ is shown in Fig. 19 and $UT_{4,3}$ in Fig. 20. A straightforward generalization of the technique used in Example 4.9 (rotation of the nodes both horizontally and vertically) shows that the homogeneous bidirectional two-dimensional toroid $BT_{m,n}$ can be $(1, 2)$-simulated on $UT_{m,n}$. Since $BA_{m,n}$ can trivially be $(1, 1)$-simulated on $BT_{m,n}$ we conclude that $BA_{m,n}$ can be $(1, 2)$-simulated on $UT_{m,n}$. To show the converse is easier. Using the argument as in Example 4.9 we conclude that not only $BA_{m,n}$ but

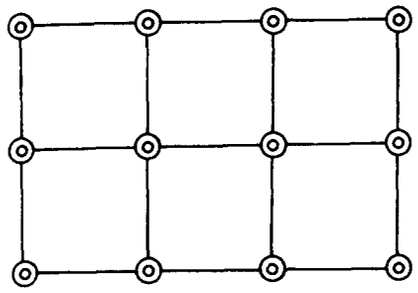


Fig. 19.

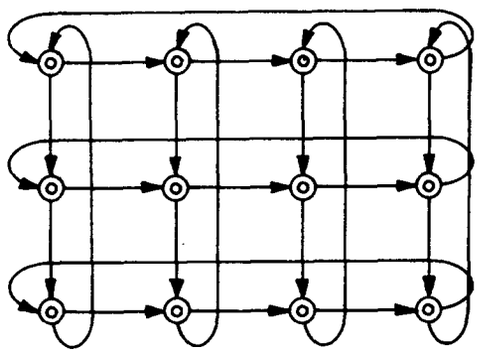


Fig. 20.

even homogeneous bidirectional two-dimensional toroid $BT_{m,n}$ can be $(1, 2)$-simulated on $UT_{m,n}$.

Therefore, we have another useful design tool; namely, every algorithm for bidirectional two-dimensional array (mesh-connected processors) can be easily modified to run at half speed on a unidirectional two-dimensional toroid of the same size.

5. Further examples

We start this section by examining a relation between networks M and kM as far as simulation is concerned. Obviously M can be $(1, k)$-simulated on kM. To see this, it is enough to take $\rho(v, t) = (v, kt)$ and $\psi(v, q) = q$. If we modify the definition of (j, k)-simulation so that k can be a rational number, then conversely kM can be $(1, 1/k)$-simulated by M. As was already mentioned, the intuitive interpretation of 'N_2 (m, k)-simulates N_1' is 'N_2 can do what N_1 does, but k-times more slowly'. The concept of time is particularly important when considering how networks receive their inputs. Informally, a network working k-times more slowly needs to get its input k-times more slowly to do the same work.

In Example 5.1 below we confirm this interpretation by considering a well-known example of a linear iterative array which recognizes palindromes.

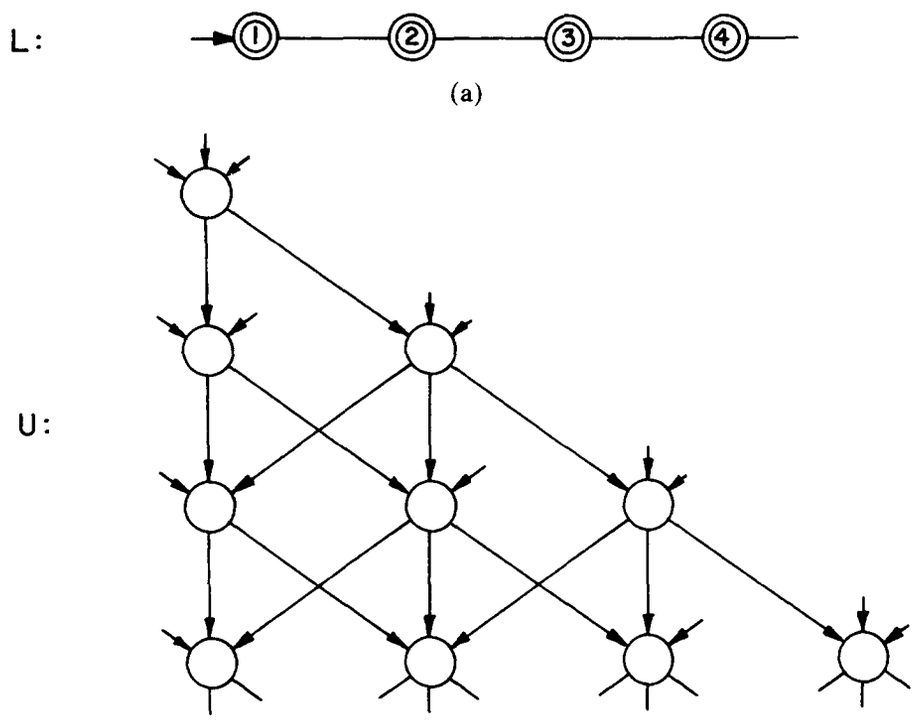
First we describe an iterative array (see [5]) as CN.

Consider the network L in Fig. 21(a), with τ defined for node 1 only ($\tau(1) = 1$). The unrolling of L is the CD U in Fig. 21(b). In order to define a computation on U (and thus on L) initial values must be given in processors (input nodes) with indegree 0. These are the nodes

(a) $(0, t)$ for $t = 0, 1, 2, \ldots$, which, because they are given to the same processor at different times, are called serial inputs, and

(b) $(j+1, j)$ and $(j+2, j)$ for $j = 0, 1, 2, \ldots$, which are called parallel inputs.

Now, L is an iterative array if Q is a finite set which contains a special element, say, $\#$, and the parallel inputs are 'quiescent', i.e., only those computations α on L are considered for which $\alpha(j+1, j) = \alpha(j+2, j) = \#$ for all $j = 0, 1, \ldots$.



(a)

(b)

Fig. 21.

5.1. Example (*Speed up*). Consider the example of a linear iterative array of finite-state machines recognizing a palindrome. Such an iterative array was given in [5]. In [22], Cole's result was reproved using the Systolic Conversion Theorem. It is easy to construct a semi-systolic linear array P_0 recognizing palindromes. Its underlying graph is shown in Fig. 22, τ is defined (as zero) in the leftmost node only. The description of ϕ can be found in [22], but is unimportant for the following discussion.

To convert this semisystolic network to a systolic network, we need to 'slow-down' the network, i.e., $[2P_0]$ rather than P_0 is equivalent to some linear iterative array P. Unfortunately, P running at half speed, as explained above, needs its input coming

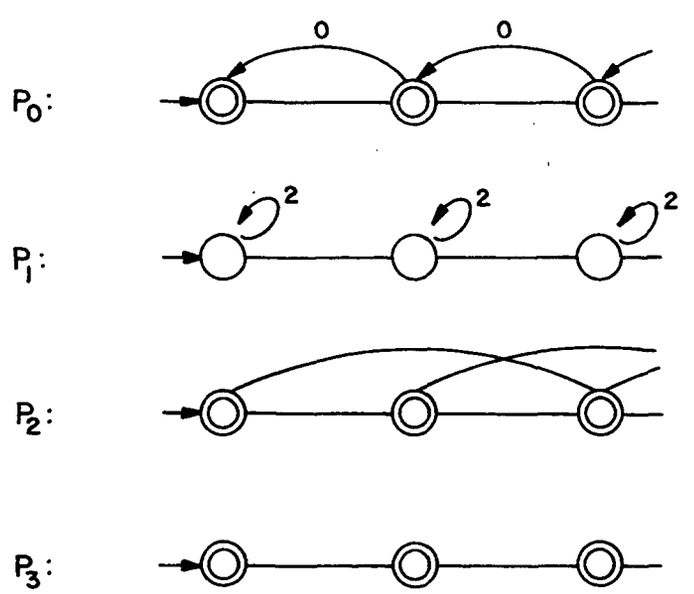


Fig. 22.

at half speed—otherwise, under strict interpretation of what iterative arrays recognize, as stated in [22], P recognizes a palindrome 'in 2 clock ticks per character'. Thus P does not recognize palindromes. Instead, given a string $x_1 \ldots x_n$, P will recognize whether $x_1 x_3 x_5 \ldots$ is a palindrome or not. In this particular case, the small gap in the proof of [22] can be filled easily. It can be shown that in general for an n-dimensional iterative array, if N_2 $(1, k)$-simulates N_1, then there is an N_3 which $(k, 1/k)$-simulates N_2, and thus N_3 $(k, 1)$-simulates N_1. We shall show this only for the example of a one-dimensional palindrome recognizer ($k = 1$), but the generalization is obvious.

In Fig. 22 P_0 represents a semisystolic network—a recognizer of palindromes. Network P_1 is obtained by the Conversion Theorem (edges with delay 3 are omitted because they are redundant), and P_1 is equivalent to $[2P_0]$. Note that $\langle V \times \mathbb{Z}, E \rangle$, the graph underlying the network P_1, has two components, one of which is the unrolling of P_1, the other component absorbs the odd-numbered inputs, as discussed above. P_1 is not a linear iterative array, since the selfloops have delay 2. However, the function ϕ can be easily modified to obtain a network with the delay 1 on each selfloop of a linear array. Calling this modified network P_1', we see that P_1' is a linear iterative array, and also that any computation on P_1 can be done on P_1'. However, P_1' still does not recognize palindromes (because P_1 does not).

Consider now the network P_2. Informally, we may say that two steps of a computation on P_1 correspond to one step on P_2. Formally, we may only say that $P_1'(1, 2)$-simulates P_2, this follows from Lemma 4.1. We did not define $(1, \frac{1}{2})$-simulation, but according to the discussion above, there is some justification in saying that P_2 $(1, \frac{1}{2})$-simulates P_1'. Regardless of whether $(1, \frac{1}{2})$-simulation is defined or not, comparing computations on P_1 and P_2 we see that P_2 is now a proper palindrome recognizer, however, P_2 fails to be a linear iterative array. Thus, one more step is needed. By Lemma 4.1, P_3 $(2, 1)$-simulates P_2—this is seen by taking the function ρ from the lemma as $\rho: i \to [(i+1)/2]$, where the processors (nodes) in both P_2, P_3 are numbered $0, 1, \ldots$ from left to right. We can conclude that P_3 is a linear iterative array which does recognize palindromes.

Note that it would be possible to develop an alternative theory for retiming. Instead of using networks like $2N$ we could have allowed retiming by $\frac{1}{2}, 1, \frac{3}{2}, \ldots$—however, as it does not seem to give any significant advantages, this route has not been taken.

Also note, that it was essential in this example that we consider iterative (linear or more general n-dimensional) arrays. The 'speed up' step from P_2 to P_3 cannot be done in a general case. In particular [10, Theorem 6.4] shows that speed up is not possible for iterative tree automata.

It is shown in [25] that for d-dimensional iterative arrays the power of a system is not increased by allowing Direct Central Control (or Global Control). In the following example we give another proof of this result using the Systolic Conversion Theorem. For simplicity we consider the case $d = 2$ only.

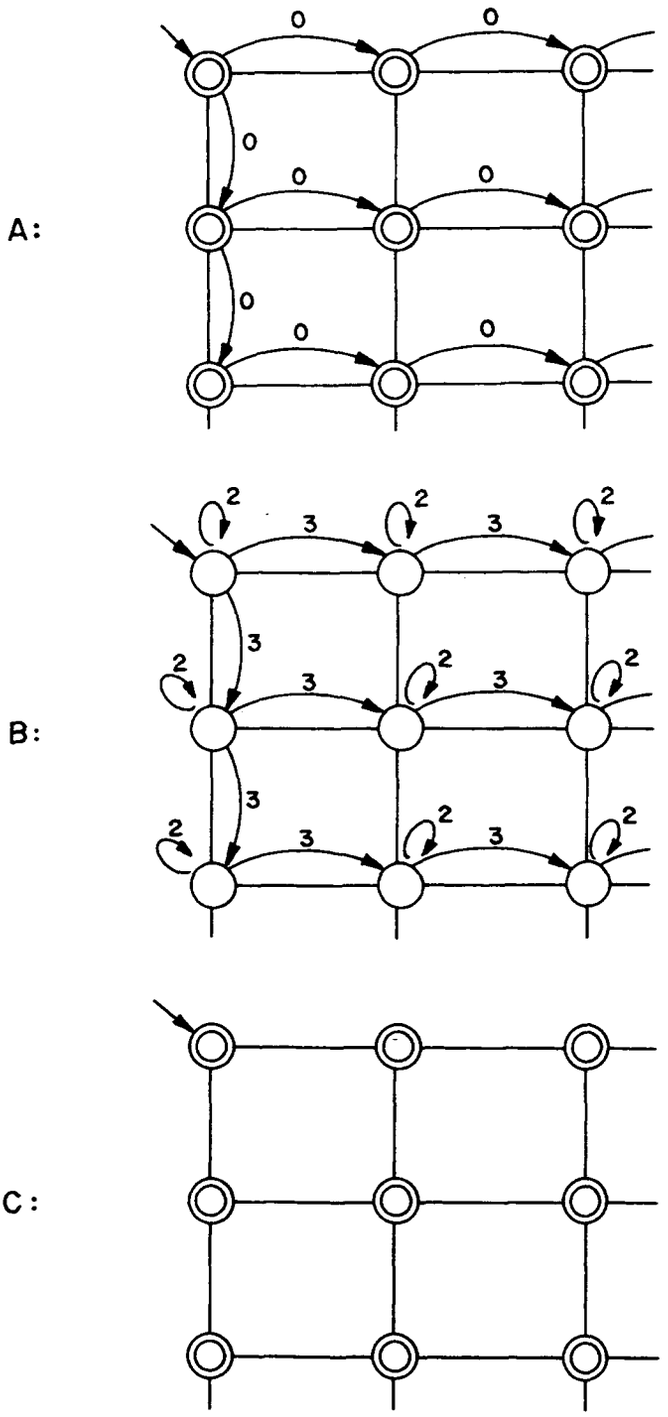


Fig. 23.

5.2. Example. The network A in Fig. 23 is a square grid network with the usual connections with delay 1, and with additional connections with delays 0. Clearly, these additional connections can be used to accomplish the global control. Note that the 0-connections are quite arbitrary as long as they connect the origin with each other node, and no 0-loops are introduced.

The network $[2A]$ can be converted, just as in the previous example, into the systolic network B (Fig. 23). It is useful to compare A in Fig. 23 with P_0 in Fig. 22. The difference, apart from P_0 being a one-dimensional array while A is two-

dimensional is that the 0-connections are oriented in opposite directions. In Fig. 23 they lead from a fixed node to every node, while in Fig. 22 they lead from every node to a fixed node. Despite of this difference our initial transformations are the same.

Since between every pair of nodes connected by an edge in B there is a path of weight 3 we can apply Lemma 4.1. Thus, we conclude the network C in Fig. 23 $(1, 3)$-simulates network B. The initial function τ for all three networks A, B, C is defined (as zero) for the origin only and therefore not affected by the modifications.

The following example demonstrates the influence of input and output considerations on geometric transformations. Cellular automata of [26] are superficially similar to iterative arrays investigated in the previous example. C and U in Fig. 24 illustrate a network, and its unrolling, which corresponds to one-dimensional version of cellular automaton (CA). Some nodes and edges of U are drawn dotted—this indicates that we consider real-time computations on CA. The figure pictures the computation with four inputs and one output. Formally, the finite input is accommodated on an infinite network C by requiring again a fixed symbol, say # in Q, and extending any finite input by appending # # ... on the right.

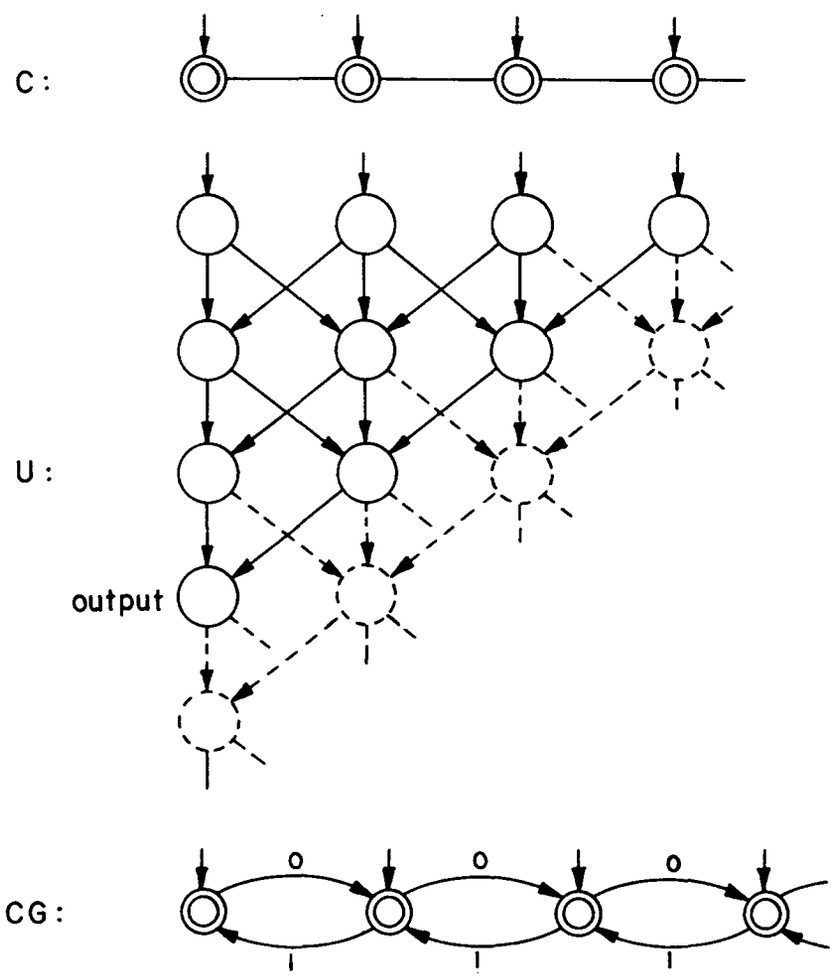


Fig. 24.

The network CG in Fig. 24 represents a possible definition of a (one-dimensional) CA with global (central) control. It could have been more natural to use also connection of delay 1 going in parallel with 0-delays, but, obviously these connections would be redundant.

5.3. Example. This example shows that any computation (in real time) on a CA with global control can be done (in real time) on the standard CA. Network C in Fig. 24 represents a CA, its unrolling is CD U in the same figure. The part of the unrolling which is irrelevant for real-time computations is drawn in dotted lines. Finally, CG in Fig. 24 represents a cellular automaton with additional 0-connections implementing the global control. Since CG has selfloops and 0-connections from left to right, the 1-connections from left to right become redundant and are omitted.

We proceeed initially in the same way as in Example 5.2 for iterative arrays, namely [2CG] is retimed. The resulting network C' and its unrolling U' are shown in Fig. 25.

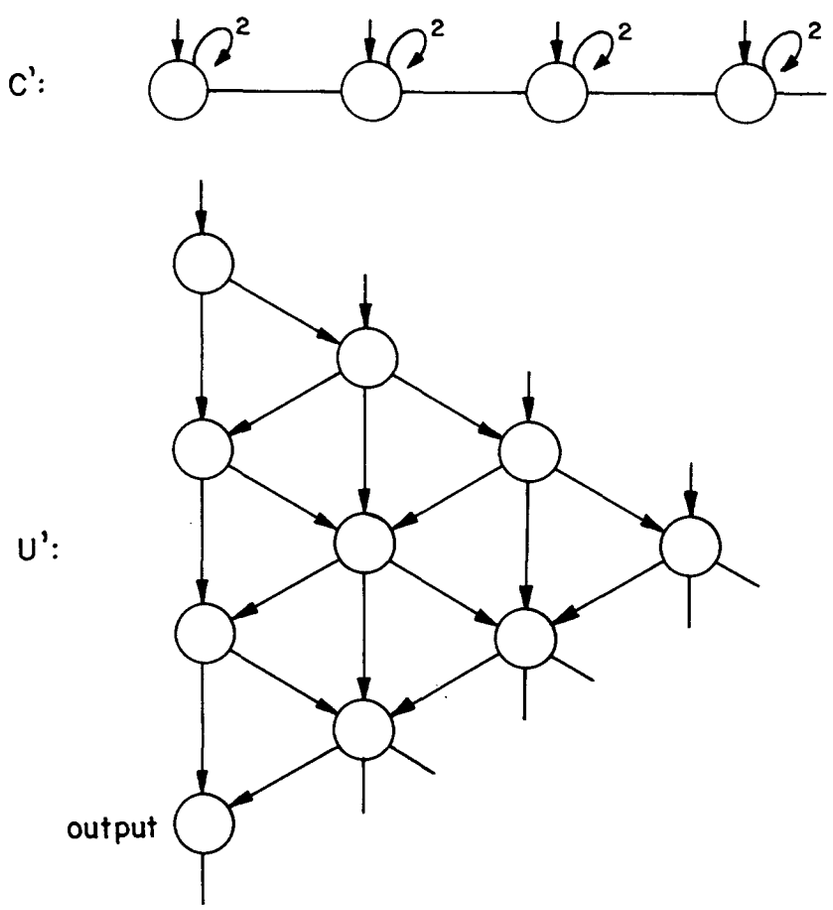


Fig. 25.

In the following simple steps we demonstrate that any computation α on U' can be simulated (in fact (2, 1)-simulated) on a CA.

(1) For the price of doubling the size of Q, the delays on selfloops in C' (Fig. 25) can be changed to 1.

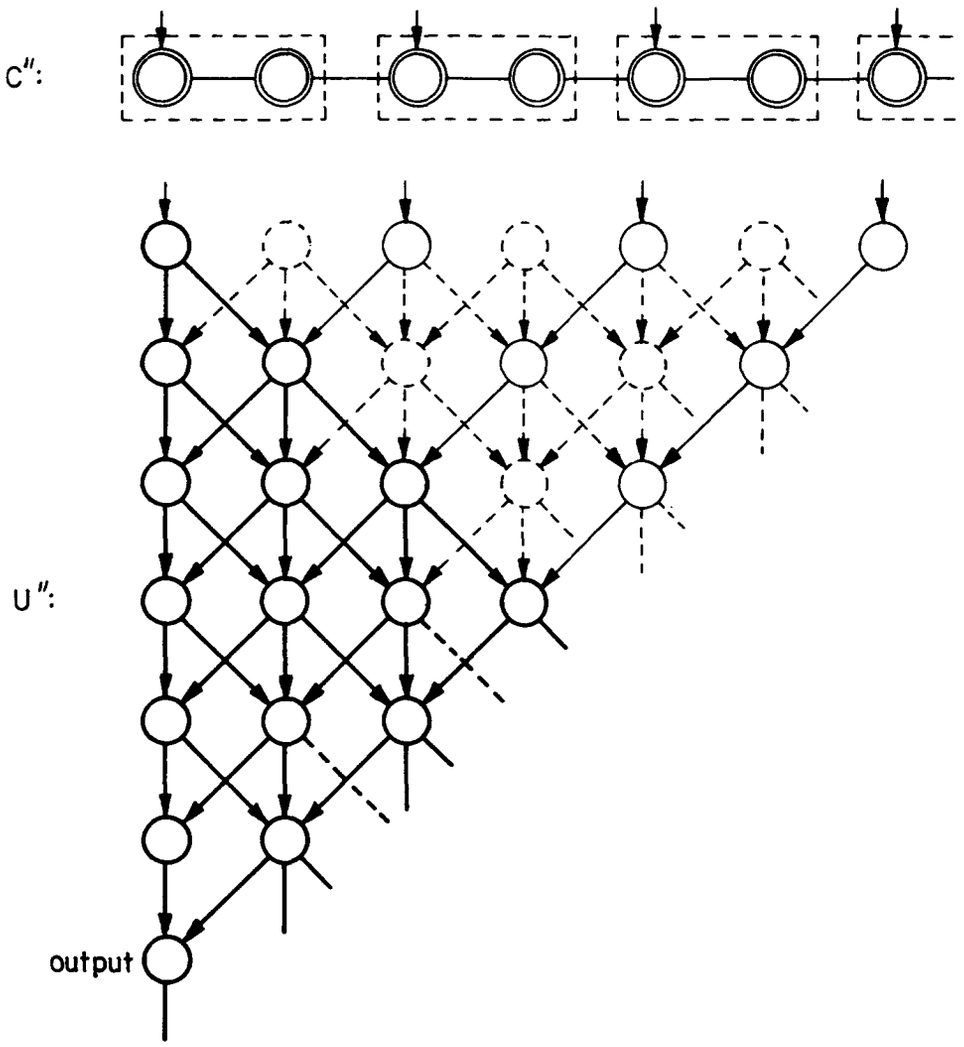


Fig. 26.

(2) The unrolling U' of C' after this modification is now a subgraph (more precisely a preCD) of the unrolling of a network C'' which is like C, but for input of double length. This is shown in Fig. 26 where the subgraph corresponding to U' is drawn in bold. It is easy to see that the network C'' whose unrolling is U'' can do the computation α (as modified already in (1)) as long as instead of the original input $i_1 i_2 i_3 i_4$ we use $i_i \# i_2 \# i_3 \# i_5$ (or similar). This is so because the diagonal path going from the top left node down right, the path on which the inputs of α are needed, can be 'computed' by sending a signal along it.

(3) The dashed boxes in the network C'' in Fig. 26 show which processors are mapped together in the final (2, 1)-simulation.

Note that a slightly simpler proof of the simulation of CG on CA would be possible had we not wanted to preserve the real-time. It is slightly simpler to show that CA (1, 2)-simulates C' than to show the (2, 1)-simulation; however, (i, j)-simulation preserves time only if $j = 1$.

We have just proved the following result for real time-CA (languange recognizer). This result can be easily generalized to n-dimensional cellular automata.

5.4. Theorem. *Given a real-time cellular automaton with central (global) control, there effectively exists a real time cellular automaton (without central control) recognizing the same language.*

5.5. Example. We now briefly discuss two more results about one-way (unidirectional) cellular automata. One-way cellular automaton allows communication between two nodes in only one direction. Here we consider two-dimensional triangular grid in which the communication is in three directions. Such a one-way cellular automaton M_n^a is shown in Fig. 27 for $n=3$. We are interested in the output produced at the node with outdegree zero (lower left corner in Fig. 27). Similarly, we have networks M_n^b, M_n^c, and M_n^d shown for $n=3$ in Fig. 27.

As a direct generalization of the result of [4] that one-dimensional one-way CA languages are closed under reversal we can show that networks M_n^a, M_n^b, M_n^c, and M_n^d are equivalent. Note that the input is presented in the same way to all four

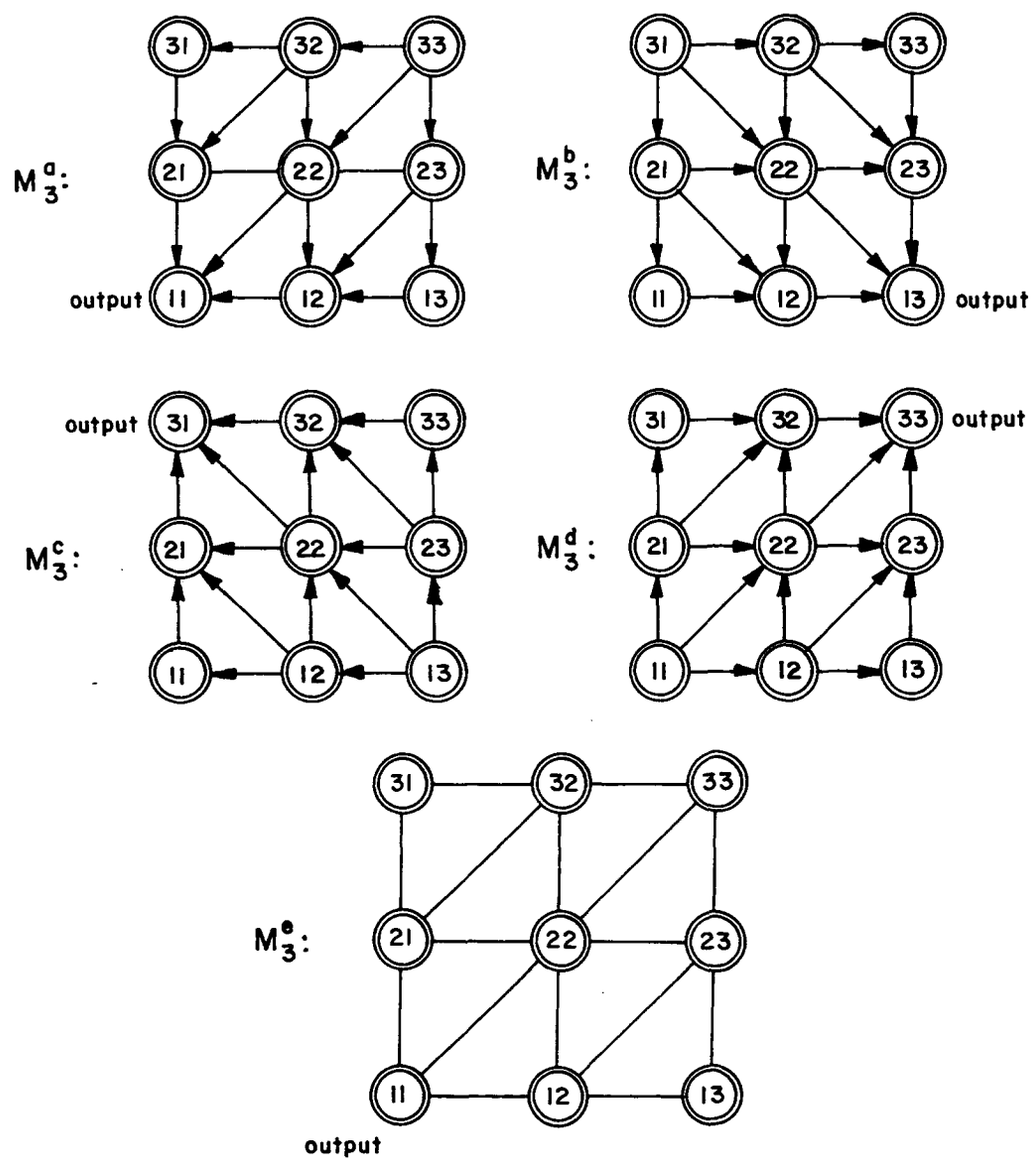


Fig. 27.

networks, that is, the input coming to node i, j in one network comes to node i, j in any other network. The following result from [4] can be generalized to higher dimensions. Real time bidirectional cellular automata (working in time n) are equivalent to one-way (unidirectional) CA working in time $2n$. One possible generalization of this is that the functions computable on network M_n^e (Fig. 27) in real time are the same as those computable on network M_n^d in time $2n$.

5.6. Example. It was shown in [9] that regular sets can be recognized by a parallel algorithm on an unidirectional binary tree network. To recognize a string of length n we need any tree (not necessarily balanced) with at least n nodes; thus to accept arbitrary long input, this algorithm requires a potentially infinite tree. If the tree is (almost) balanced the recognition is in logarithmic time. A network based on such a tree is shown in Fig. 28(b). Here, the initial function τ is defined (as zero) at all the input nodes. In [8] it has been shown that we can use finite tree-like network M_k illustrated in Fig. 28(a) for depth $k=2$. Here the initial function τ is defined (as zero) for the processors of the top row. On this network we recognize strings of length n, $n \leqslant 2^k$, in time k. Longer strings are cut into pieces of length 2^k and recognized in time $n/2^k + k$. The correctness of the modified algorithm for M_k follows easily from the fact that the unrolling of M_k is the infinite binary tree shown in Fig. 28(b).

5.7. Example. In Example 5.2 we considered a 'quarter plane' iterative array. Here we show that it is not important what regular infinite section of plane is used. We show this for the case of the 'full plane' iterative array A and the 'quarter plane' iterative array B (see Fig. 29).

The idea of showing two systolic systems equivalent by folding has been introduced in [3]. Clearly, the result of folding A twice is B. To illustrate here that folding is a special case of our technique we show that the two networks in Fig. 29 can simulate each other. $(1, 1)$-simulation of B by A is trivial (and follows from Corollary 4.3). Conversely, B can $(4, 1)$-simulate A. This is established by Corollary 4.2 with the help of the following partition of nodes $V_1 = \{(i, j) \mid i, j \in \mathbb{Z}\}$ of the network A: nodes (i_1, j_1), (i_2, j_2) belong to the same partition if and only if $|i_1| = |i_2|$ and $|j_1| = |j_2|$. Obviously, B is the network obtained by just described partition of A.

Clearly, this example generalizes to n-dimensional arrays. For less trivial examples of folding see [3].

5.8. Example. It is shown in [10] that an iterative tree network T_k (where $1+k$ is the number of nodes connected with each node) is more powerful than T_l for $k > l$. This paper considers T_k as a language recognizer and thus their result is stronger than what is shown in this example.

It immediately follows from Lemma 4.5 that T_l cannot $(1, 1)$-simulate T_k (for $l < k$); however, $(m, 1)$-simulation is possible. To simplify the notation, we show a

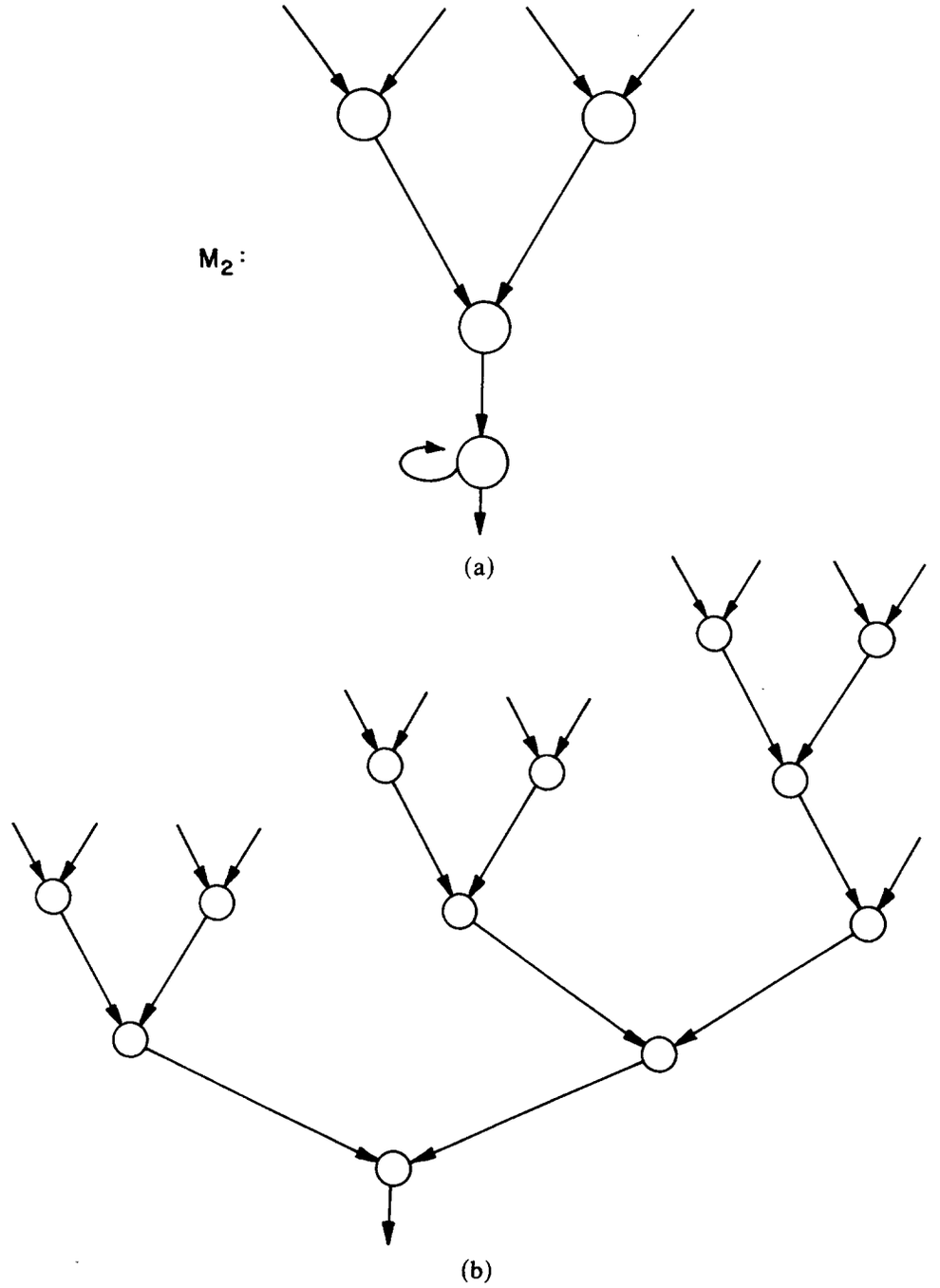


Fig. 28.

specific result, namely that T_2 can (3, 1)-simulate T_4; the generalization is straightforward.

Fig. 30(a) (ignoring the boxes) shows T_2. The boxes around nodes represent partition of nodes of this network. Corollary 4.2 shows that T_4 of Fig. 30(b) can be (3, 1)-simulated on T_2.

We will conclude this paper by showing the relation between various types of 'shuffle' networks. Perfect shuffle network has been introduced in [27]. It is a difficult network to layout but many algorithms can be efficiently implemented on it.

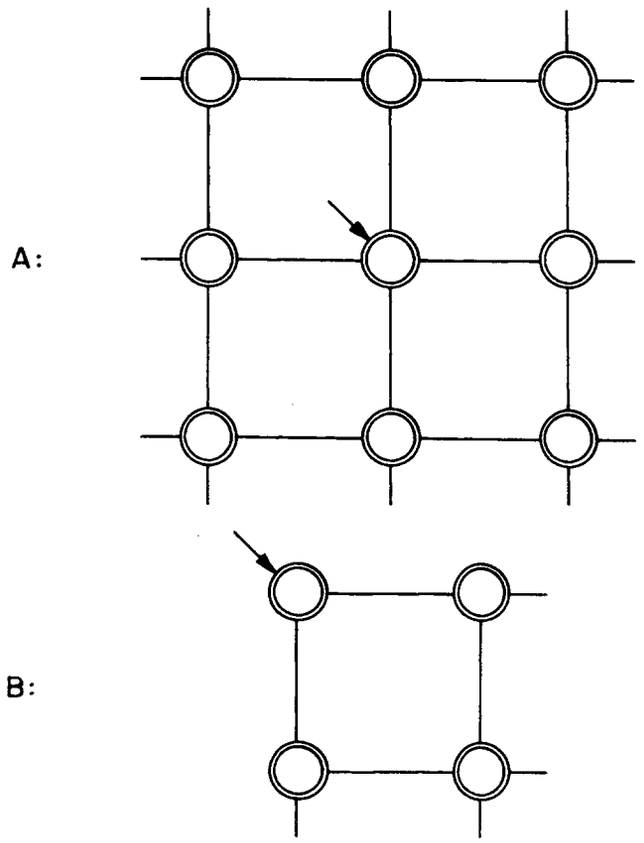
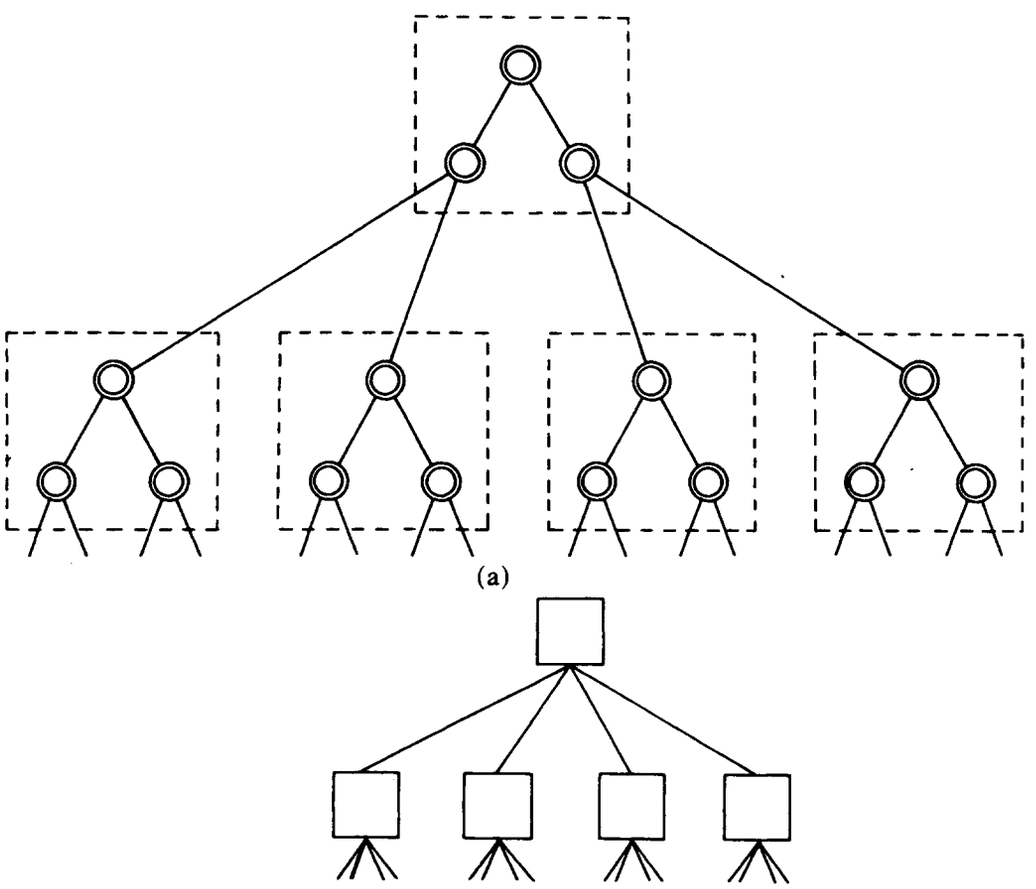


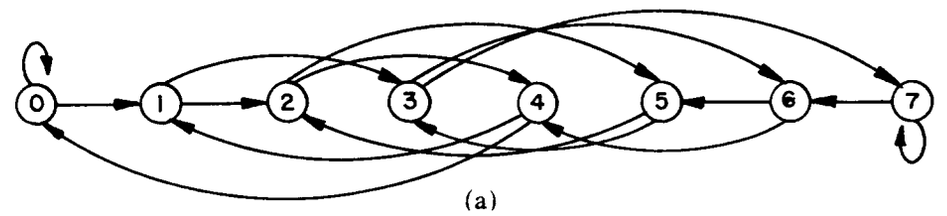
Fig. 29.



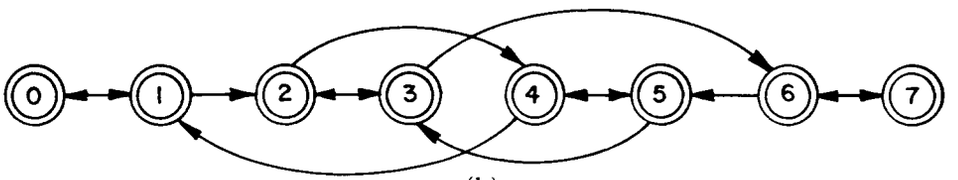
(a)

(b)

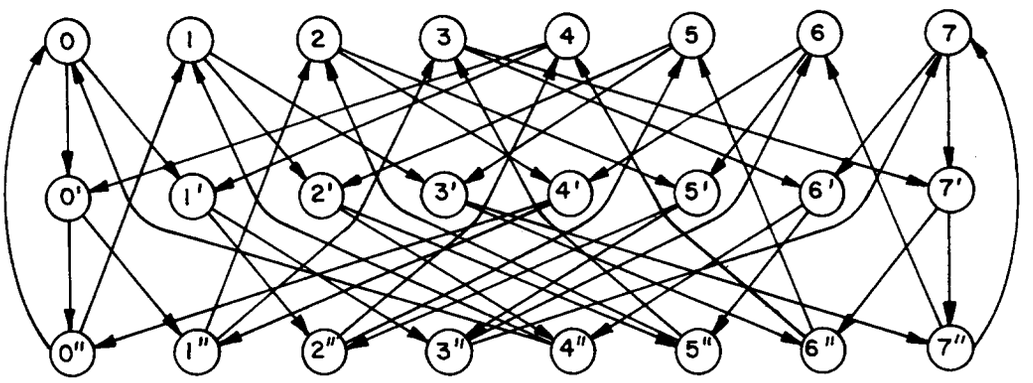
Fig. 30.


0
1
2
3
4
5
6
7

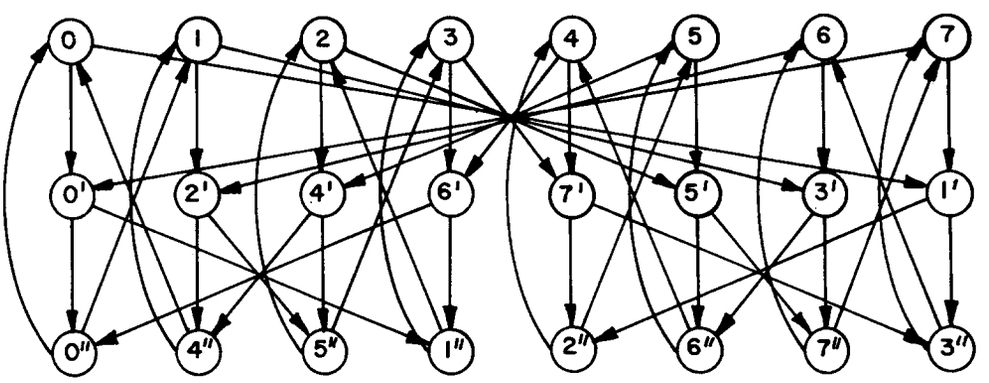
(a)


0
1
2
3
4
5
6
7

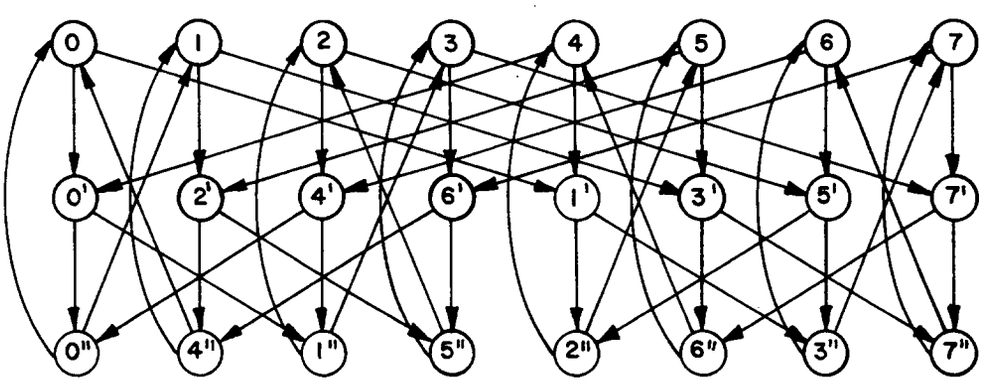
(b)


0
1
2
3
4
5
6
7
0'
1'
2'
3'
4'
5'
6'
7'
0''
1''
2''
3''
4''
5''
6''
7''

(c)


0
1
2
3
4
5
6
7
0'
2'
4'
6'
7'
5'
3'
1'
0''
4''
5''
1''
2''
6''
7''
3''

(d)


0
1
2
3
4
5
6
7
0'
2'
4'
6'
1'
3'
5'
7'
0''
4''
1''
5''
2''
6''
3''
7''

(e)

Fig. 31.

Perfect shuffle with 2^n processors, $n>1$, is the network $\mathrm{PS}_n=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ where $V=\{0,\ldots,2^n-1\}$, $\pi(2k)=\pi(2k+1)=k, 2^{n-1}+k$ for $k=0,\ldots,2^{n-1}-1$ and Q, ϕ depend on particular algorithm (with default λ, τ). PS_3 is shown in Fig. 31(a).

Shuffle exchange (cf. [28]) with 2^n processors in the network $\mathrm{SE}_n=\langle V, \pi, Q, \phi, \lambda, \tau\rangle$ where $V=\{0,\ldots,2^n-1\}$; $\pi(2k)=k, 2k, 2k+1$; $\pi(2k+1)=2k, 2k+1, 2^{n-1}+k$, for $k=0, 1,\ldots,2^{n-1}-1$. (Again default λ, τ are assumed.) SE_3 is shown in Fig. 31(b). Note that each processor has a selfloop (indicated by double circles in Fig. 31 (b)) which, in other words, means that the processors have memory.

It is known that algorithms for PS_n can be modified to run on SE_n and vice versa. Formally we have the following theorem.

5.9. Theorem. *For any $n>1$, PS_n can be $(1,2)$-simulated on SE_n.*

Proof. Since each processor of SE_n has memory (selfloop), we see that if $(u, v)\in E$ in PS_n than there is a path $u\to^+ v$ of length exactly 2 in SE_n. Thus the result follows by Lemma 4.1. □

Now, we will consider three networks obtained as 'partial unrolling' of perfect shuffle. Let networks N_n^c, N_n^d, and N_n^e be networks with $n\cdot 2^n$ processors connected as shown for $n=3$ in Fig. 31(c)-(e) with τ defined (as zero) in the top row. Note that network N_n^d is the unidirectional version of the butterfly network from [28]. The numbering of the nodes in Fig. 31(c)-(e) shows how the unrollings of the networks N_3^c, N_3^b, and N_3^e are isomorphic. We leave it for the reader to specify these networks for arbitrary n, and to verify that the unrollings are again isomorphic. Thus we have the following theorem.

5.10. Theorem. *For each $n>1$, the networks N_n^c, N_n^d, and N_n^e are equivalent.*

From Theorems 5.9 and 5.10 we have the following corollary, which in the case of N_n^d is the unidirectional version of [28, Theorem 6.3].

5.11. Corollary. *Any of networks N_n^c, N_n^d, N_n^e can be $(1, 2)$-simulated network on SE_n.*

Note that in the proof of [28, Theorem 6.3] it is also implicitly assumed that all the processors of the shuffle-exchange network have memory; however in [28, Fig. 6.1] the selfloops are shown for the leftmost and rightmost processors only. This is misleading since if these processors have memory, why the selfloops?

References

[1] W. Bucher and K. Culik II, On real time and linear time cellular automata, *RAIRO Inform. Théor.* **18** (1984) 307-325.

[2] P.R. Cappello and K. Steiglitz, Unifying VLSI design with geometric transformations, *Proc. IEEE 1983 Internat. Conf. on Parallel Processing* (1983) 448-457.
[3] C. Choffrut and K. Culik II, Folding of the plane and the design of systolic arrays, *Inform. Process. Letters* **17** (1983) 149-153.
[4] C. Choffrut and K. Culik II, On real-time cellular automata and trellis automata, *Acta Inform.* **21** (1984) 393-407.
[5] N. Cole Stephen, Real-time computation by n-dimensional iterative arrays of finite-state machine, *IEEE Trans. Comput.* **C-18** (1969) 349-365.
[6] K. Culik II, J. Gruska and A. Salomaa, Systolic trellis automata; Part I, *Internat. J. Comput. Math.* **15** (1984) 195-212; Part II, *ibid.* **16** (1984) 3-22.
[7] K. Culik II and J. Pachl, Folding and unrolling systolic arrays, *ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Ottawa (1982) 254-261.
[8] K. Culik II and H. Jürgensen, Programmable finite automata for VLSI, *Internat. J. Comput. Math.* **14** (1983) 259-275.
[9] K. Culik II, A. Salomaa and D. Wood, Systolic tree acceptors, *RAIRO Inform. Theoret.* **18** (1984) 53-69.
[10] K. Culik II and S. Yu, Iterative tree automata, *Theoret. Comput. Sci.* **32** (1984) 227-247.
[11] C.R. Dyer, One-way bounded cellular automata, *Inform. and Control* **44** (1980) 261-281.
[12] P.C. Fischer, Generation of primes by a one-dimensional real-time iterative array, *J. ACM* **12** (1965) 388-394
[13] O.H. Ibarra, S.M. Kim and S. Moran, Sequential machine characterizations of trellis and cellular automata and applications, Unpublished manuscript.
[14] O.H. Ibarra and S.M. Kim, A characterization of systolic binary tree automata and applications, Unpublished manuscript.
[15] D.E. Knuth, *The Art of Computer Programming, Vol. 3* (Addison-Wesley, Reading, MA, 1973).
[16] S.R. Kosaraju, Speed of recognition of context-free languages by array automata, *SIAM J. Comput.* **4** (1975) 331-340.
[17] H.T. Kung, Why systolic architectures?, *Comput. Magazine* (January 1982) 37-46.
[18] H.T. Kung and W.T. Lin, An algebra for VLSI algorithm,*Proc. Conf. on Elliptic Problems*, 1983.
[19] H.T. Kung and C.E. Leiserson, Systolic arrays (for VLSI), *Proc. Sparse Matrix Conf.*, 19878.
[20] F.T. Leighton, *Complexity Issues in VLSI* (MIT Press, Cambridge, MA, 1983).
[21] C.E. Leiserson, *Area-Efficient VLSI Computations* (MIT Press, Cambridge, MA, 1982).
[22] C.E. Leiserson and J.B. Saxe, Optimizing synchronous systems, *Proc 22nd Ann. FOCA Symp.* (1981) 23-36.
[23] N.S. Ostlund and R.A. Whiteside, A machine architecture for molecular dynamics: The systolic loop, *Conf. on Macromolecular Structure and Specificity: Computer Assisted Modeling* (New York Academy of Sciences, 1983).
[24] C.A. Mead and L. Conway, *Introduction to VLSI* (Addison-Wesley, Reading, MA, 1980).
[25] J.I. Seiferas, Iterative array with direct central control, *Acta Inform.* **8** (1977) 177-192.
[26] A.R. Smith III, Real-time language recognition by one-dimensional cellular automata, *J. ACM* **6** (1972) 233-253.
[27] H.S. Stone, Parallel processing with perfect shuffle, *IEEE Trans. Comput.* **C-20** (1971) 153-161.
[28] J.D. Ullman, *Computational Aspects of VLSI* (Computer Science Press, Rockville, MD, 1984).
[29] H. Umeo, K. Morita and K. Sugato, Deterministic one-way simulation of two-way real-time cellular automata, *Inform. Process. Letters* **14** (1982) 158-161.
[30] U. Weiser and A. Davis, A wavefront notation tool for VLSI array design, in: H.T. Kung, R.F. Sproull and G.L. Steele, Jr., eds., *VLSI Systems and Computing* (Carnegie-Mellon University, Computer Science Press, 1981) 226-234.