Data-Independence of Read, Write, and Control Structures in PRAM computations*

Klaus-Jörn Lange and Rolf Niedermeier†
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
lange/niedermr@informatik.uni-tuebingen.de

*A preliminary version of parts of this work appeared in the proceedings of the Thirteenth Conference on Foundations of Software Technology and Theoretical Computer Science (Springer *LNCS* 761, pages 104–113) held in Bombay, India, December 15–17, 1993 under the title "Data-independences of parallel random access machines."

†Part of this research was done while both authors were at the Fakultät für Informatik, Technische Universität München. This research was supported by the Deutsche Forschungsgemeinschaft, SFB 342, Teilprojekt A4 "KLARA" and by the DFG Project La 618/3-1 "KOMET." The work of the first author was also supported by the International Computer Science Institute. The work of the second author was also supported by a Feodor Lynen fellowship of the Alexander von Humboldt-Stiftung, Bonn, and the Center for Discrete Mathematics, Theoretical Computer Science, and Applications (DIMATIA), Prague.

Abstract

We introduce the notions of control and communication structures in PRAM computations and relate them to the concept of data-independence. Our main result is to characterize differences between unbounded fan-in parallelism AC^k, bounded fan-in parallelism NC^k, and the sequential classes $DSPACE(\log n)$ and $LOGDCFL$ in terms of a PRAM's communication structure and instruction set. Our findings give a concrete indication that in parallel computations writing is more powerful than reading. Further characterizations are given for parallel pointer machines and the semi-unbounded fan-in circuit classes SAC^k. In particular, we obtain the first characterizations of NC^k and $DSPACE(\log n)$ in terms of PRAM's. Finally, we introduce Index-PRAM's which in some sense have "built-in data-independence." We propose Index-PRAM's as a tool for the development of data-independent parallel algorithms. Index-PRAM's serve for studying the essential differences between the above mentioned complexity classes with respect to the underlying instruction set used.

1 Introduction

Parallel random access machines (PRAM's) are *the* favorite model for design and analysis of parallel algorithms. This favoritism has led to the desire for efficient general-purpose simulations of PRAM's on real machines, as opposed to special implementations of certain algorithms on certain machine architectures. The direct, general approach, using techniques such as hashing and slackness (see [1, 51, 61, 62]), is limited by problems such as hardware cost, (non-)scalability, and interconnect length (see [9, 19, 31, 45, 61, 62, 67] for discussion). The main alternative has been to base algorithm design on restricted forms of PRAM's (see [12, 32]) or on models that try to build in more "computational realism" [2, 3, 19, 21, 33, 62, 61]. Our approach is to stay with the generality and simplicity of the basic PRAM models, but without using the classifications offered by parallel complexity theory in terms of P-completeness and NC-membership. Since this dichotomy apparently is not appropriate when trying to speed-up running times by using rather weak parallel machines or networks of workstations, we tried to abstract out general features of algorithms that lead to efficient implementations. The features we emphasize are *data-independence* of the read, write, and control structures of the algorithm.

Many of the computation-intensive tasks targeted by the "Grand Challenges" [44, 57] seem to be simpler than what is needed for a general PRAM simulation. Their communication and control structures are simple enough that an efficient implementation on existing architectures, working with distributed memories and message passing mechanisms, is possible. For example, tasks like FFT, parallel prefix sums ("scans"), and matrix operations are data-independent on all counts. For operations on graphs as pointer jumping (list ranking) this may not be the case. In general, parallel graph algorithms depend on how graphs are represented. There are two simple representations of graphs: adjacency matrices and edge lists. Algorithms working on adjacency matrices usually show data-independent behavior (e.g., Warshall algorithm), whereas algorithms using edge lists (e.g., pointer jumping or list ranking problem) show inherent dependence of the communication structure in the underlying data—the addresses of global memory cells used strongly depend on the list structure. The importance of data-independent reads/writes/control in distributed memory computing has been pointed out in several papers, e.g., [29, 39, 58, 66].

The main contributions of this paper are as follows:

1. We formally introduce the notions of data-independent reads, writes, and control: Data-independence of control means that the statement executed by a processor of a PRAM depends only on time, processor identification number (PIN for short), and length of the input, but not on the input itself. Data-independence of communication structure means that in global read accesses (resp., the receipt of messages) or write accesses (resp., the sending of messages) the addresses of shared memory cells depend only on time, PIN, and input length.

2. From the formal notion we obtain surprising results—these restrictions lead to complexity classes whose original definitions had seemingly nothing to do with PRAM's: Whereas unbounded fan-in parallelism, represented by the classes AC^k, is characterized by a data-dependent control or write structure in combination with a data-independent read structure, bounded fan-in parallelism, represented by the classes NC^k, is characterized by computations where all three structures have to be data-independent. The remaining case, where we have a data-dependent read structure but data-independent control and write structures, leads to characterizations

of the sequential classes $DSPACE(\log n)$, $LOGDCFL$, and, as an intermediate class defined by a parallel device, of Cook's parallel pointer machines [15, 18] operating in logarithmic time. Eventually, we discuss the power of Akl's concurrent write OR-feature for PRAM's [4] and obtain a characterization of $LOGCFL$ and, more generally, of Venkateswaran's semi-unbounded fan-in circuit classes SAC^k [64] by monotonic, fully data-independent OR-PRAM's.

3. The formal analysis of different PRAM instructions and their data-(in)dependence finally leads to a computation model (so-called *Index-PRAM*) intended to fulfill the desire for a more realistic and nevertheless simple PRAM model: The basic idea is to consider PRAM's where the indexing of global memory cells is only possible through special local index registers. The value of index registers only depends on time, PIN, and input length, but not the input data. Within the framework of Index-PRAM's, it is possible to study the differences between various complexity classes with respect to the instruction set used by the underlying Index-PRAM. For example, it will be shown that the fundamental difference between $DSPACE(\log n)$ and parallel pointer machines [18] operating in logarithmic time is that for the first only a restricted form of conditional assignments may be used—the condition may depend only on the value of a bit of the input word and must not depend on a result of a previous computation.

The paper is organized as follows. In the next section, we provide basic definitions and concepts relevant for our work. In the third section, we present simple PRAM's and the notions of data-independence of communication and control. In the fourth section, we present the characterizations of various complexity classes within the unified framework of data-independence as discussed above. We define Index-PRAM's in the fifth section We give characterizations by Index-PRAM's that parallel those of the fourth section. Finally, we conclude this paper with a summarizing table, a discussion of the main benefits of our work with respect to a more realistic parallel complexity theory, and some directions for future research.

2 Preliminaries

We assume familiarity with the basic concepts and notations of computational complexity theory [7, 8, 10, 35, 37, 49, 69]. By $DSPACE(\log n)$, ($DTIME(\log n)$, and $ATIME(\log n)$) we denote the class of languages accepted by deterministic (resp. alternating) Turing machines whose working space (resp. running time) is bounded by $\log n$. Augmenting a $DSPACE(\log n)$ Turing machine with an unbounded auxiliary push-down store yields a so-called auxiliary push-down automaton [14]. We refer to the class of languages logspace many-one reducible to context-free languages (deterministic context-free languages) as $LOGCFL$ ($LOGDCFL$). In the following, we briefly review some concepts and facts of parallel complexity theory. For more details, we refer to the literature [16, 27, 36, 38, 50, 53].

2.1 Uniform circuits

A *boolean circuit* C is a finite, acyclic, directed graph. Nodes of in-degree (out-degree) zero are *inputs (outputs)*. Inner nodes with non-zero in-degree are labeled by boolean functions, throughout this paper by negations, disjunctions, and conjunctions. We call the inner nodes *gates* and the edges *wires*. Given an assignment of boolean values to all inputs, each gate evaluates to either *true* (or 1) or *false* (or 0), according to the interconnection structure of C. If C has just one output, we use C to recognize binary languages, defining $L(C)$ to be the set of assignments to the inputs which let the output evaluate to *true*. The *size* of C is the number of its gates, not counting the inputs. The *depth* of C is the length of the longest path connecting an input node with an output node.

A *circuit family* C is a sequence $C = (C_n)_{n \geq 1}$ of circuits, where C_n has exactly n inputs. Family C has *polynomial size* if for some polynomial p, the size of each C_n is bounded by $p(n)$. Similarly, the depth of C is *bounded by* $O(\log^k n)$ if for some constant $c > 0$ the depth of each C_n is less than $c \log^k n$. If for some constant m (usually $m = 2$) the in-degree of each gate in each C_n is bounded by m, then C is of *bounded fan-in*. If there is no bound on the in-degrees, then C is of *unbounded fan-in*.

In order to relate classes of languages defined by circuits with standard complexity classes, it is necessary to consider *uniform* circuit families by requiring that the members of a circuit family are "sufficiently similar" to each other. There are several uniformity conditions which have fortunately turned out to be equivalent in most cases [55].

Throughout the paper we use the notion of $DTIME(\log n)$-uniformity for circuits [55, 16, 11]. A circuit family of bounded fan-in of size $z(n)$ and depth $t(n)$ is $DTIME(\log n)$-uniform if there is a deterministic Turing machine recognizing the extended connection language L_{EC} in time $O(\log n)$. Here, $L_{EC} = \{\langle 1^n, g, \tau\rangle \mid \text{gate } g \text{ has type } \tau\} \cup \{\langle 1^n, g, p, g'\rangle \mid g' \text{ is predecessor of } g \text{ via path } p\}$ where $2n < g, g' < z(n), \tau \in \{\vee, \wedge\}$, and $p \in \{0,1\}^*, |p| \leq \log(z(n))$. By convention, gates 1 to n contain the input bits and gates $n+1$ to $2n$ their negations. A circuit family of unbounded fan-in of size $z(n)$ and depth $t(n)$ is called $DTIME(\log n)$-uniform if there is a deterministic Turing machine recognizing the direct connection language L_{DC} in time $O(\log n)$. Here, $L_{DC} = \{\langle 1^n, g, \tau\rangle \mid \text{gate } g \text{ has type } \tau\} \cup \{\langle 1^n, g, g'\rangle \mid g' \text{ is direct predecessor of } g\ \}$ where $2n < g \leq z(n)$, $1 \leq g' \leq z(n)$, and $\tau \in \{\vee, \wedge\}$. If we speak about the direct connection language for circuits of bounded fan-in, this is defined to be $L_{DC} = \{\langle 1^n, g, \tau\rangle \mid \text{gate } g \text{ has type } \tau\} \cup \{\langle 1^n, g, p, g'\rangle \in L_{EC} \mid |p| = 1\}$. Of main importance here is that the uniformity machine is provided only with the length of the input word, and not the input word itself. Thus for fixed input length n, one particular circuit is always constructed.

Clearly, each $DTIME(\log n)$-uniform circuit is also $DSPACE(\log n)$-uniform, because $DTIME(\log n) \subseteq DSPACE(\log n)$. The classes NC^k (AC^k, respectively) denote the families of languages recognizable by $DTIME(\log n)$-uniform, polynomial size, $O(\log^k n)$-depth bounded circuit families of bounded (unbounded, respectively) fan-in. Recently, Venkateswaran [64] introduced the classes SAC^k of languages recognized by $ATIME(\log n)$-uniform, polynomial sized, $O(\log^k n)$ depth bounded circuits of semi-unbounded fan-in. That is, only OR-gates may have unbounded fan-in and negations are forbidden except for the input gates. The inclusions

$$NC^k \subseteq SAC^k \subseteq AC^k \subseteq NC^{k+1}$$

and

$$NC^1 \subseteq DSPACE(\log n) \subseteq SAC^1$$

are well-known [37, 38, 64].

At the end of this section, we rephrase some normal forms concerning uniformity by Ruzzo and by Damm *et al.*

Lemma 1 **(a)** [55] *For $k \geq 2$ the following uniformity conditions for NC^k-circuit families $C = (C_n)$ are equivalent:*

1. $L_{DC} \in DTIME(\log n)$ *(U_D-uniformity).*
2. $L_{DC} \in ATIME(\log n)$.
3. $L_{EC} \in DTIME(\log n)$ *(U_E-uniformity).*
4. $L_{DC} \in ATIME(\log n)$ *(U_{E^*}-uniformity).*
5. *The description of C_n is deterministically computable out of 1^n in logarithmic space (U_{BC}-uniformity).*

(b)[20] *For $k \geq 1$, NC^k is equal to the class of languages recognized by bounded fan circuits of polynomial size and depth $O(\log^k n)$ such that $L_{EC} \in NC^k$.*
(c) *For $k \geq 2$, NC^k is equal to the class of languages recognized by bounded fan circuits of polynomial size and depth $O(\log^k n)$ such that $L_{DC} \in NC^k$.*

Proof. "(b)": Damm *et al.* [20, Lemma 6] prove this result only for the case $k = 1$. An essential part of their construction consists in transforming an NC^1-circuit into a balanced complete binary tree of alternating layers of $\vee$ and $\wedge$ gates. In an NC^k-circuit, this must be done by decomposing it in layers of depth $\log n$ and transforming each layer into a disjoint family of balanced complete binary trees. After this step, the rest of their construction goes through.
"(c)": Follows from part (b), since Ruzzo showed for $k \geq 2$ that $L_{DC} \in NC^k$ implies $L_{EC} \in NC^k$ [55, Lemma 2]. □

A corresponding result holds for circuits of unbounded and of semi-unbounded fan-in. To save space we only sketch the constructions without filling in all the details.

Lemma 2 **(a)***: For $k \geq 1$, AC^k is equal to the class of languages recognized by unbounded fan-in circuits of polynomial size and depth $O(\log^k n)$ such that $L_{DC} \in AC^k$.* **(a)***: For $k \geq 1$, SAC^k is equal to the class of languages recognized by semi-unbounded fan-in circuits of polynomial size and depth $O(\log^k n)$ such that $L_{DC} \in SAC^k$.*

Proof. "**(a)**": Let L be recognized by an unbounded fan-in circuit family $C = (C_n)_{n \geq 1}$ of polynomial size and depth $O(\log^k n)$ such that the direct connection language L_{DC} of C is in AC^k. We indicate how to construct a circuit family C' recognizing L with a direct connection language in $DTIME(\log n)$. Since L_{DC} of C is in AC^k, we know that there are $DTIME(\log n)$-uniform circuits answering the questions "$\langle 1^n, i, \exists \rangle \in L_{DC}$" (*is gate i an OR-gate?*),"$\langle 1^n, i, \forall \rangle \in L_{DC}$" (*is gate i an AND-gate?*), and "$\langle 1^n, i, j \rangle \in L_{DC}$" (*is gate j a predecessor of gate i?*). The circuit family C' is now constructed by replacing in each C_n each gate i for $i \geq 2n$ by the following circuitry:

$$\langle i \rangle := \left[\langle 1^n, i, \exists \rangle \in L_{DC} \wedge \bigvee_{j=1}^{n} \left(\langle j \rangle \wedge \langle 1^n, i, j \rangle \in L_{DC} \right) \right] \vee$$

$$\left[\langle 1^n, i, \forall \rangle \in L_{DC} \wedge \bigwedge_{j=1}^{n} \left(\langle j \rangle \vee \langle 1^n, i, j \rangle \notin L_{DC} \right) \right].$$

In this way, we end up in a $DTIME(\log n)$-uniform AC^k circuit recognizing L.
"**(b)**": Let L be recognized by a semi-unbounded fan-in circuit family $C = (C_n)_{n\geq 1}$ of polynomial size and depth $O(\log^k n)$ such that the direct connection language L_{DC} of C is in SAC^k. We indicate how to construct a circuit family C' recognizing L with a direct connection language in $DTIME(\log n)$. The direct connection language L_{DC} of C consists in elements of the form $\langle 1^n, i, j\rangle$, indicating that gate i is an $\exists$-gate and that j is a predecessor of j, and of those of the form $\langle 1^n, i, 0 \text{ or } 1, j\rangle$, indicating that gate i is an $\wedge$-gate and that j is the left or right predecessor of i. Since L_{DC} of C is in SAC^k, we know that there are $DTIME(\log n)$-uniform circuits answering the questions "$\langle 1^n, i, j\rangle \in L_{DC}$" (*is gate i an OR-gate with predecessor j?*), and "$\langle 1^n, i, 0 \text{ or } 1, j\rangle \in L_{DC}$" (*is gate i an AND-gate and j is a predecessor of i?*). The circuit family C' is now constructed by replacing in each C_n each gate i for $i \geq 2n$ by the following circuitry:

$$\langle i\rangle := \left[\bigvee_{j=1}^{n} (\langle j\rangle \wedge \langle 1^n, i, j\rangle \in L_{DC})\right] \vee$$

$$\left[\left(\bigvee_{j=1}^{n} (\langle j\rangle \wedge \langle 1^n, i, 0, j\rangle \in L_{DC})\right) \wedge \left(\bigvee_{j=1}^{n} (\langle j\rangle \wedge \langle 1^n, i, 1, j\rangle \in L_{DC})\right)\right].$$

In this way, we end up in a $DTIME(\log n)$-uniform SAC^k circuit recognizing L. □

Throughout the paper we will denote the class of languages recognized by bounded fan circuits of polynomial size and logarithmic depth that fulfill $L_{DC} \in NC^1$ by *weakly uniform* NC^1.

2.2 Parallel Random Access Machines

A Parallel Random Access Machine (PRAM) is a set of random access machines, called *processors*, that work *synchronously* and communicate via a *global shared memory*. Each PRAM computation step takes one time unit regardless of whether it performs a local or global (i.e., remote) operation. We assume the standard definition of PRAM's [36, 38]. All processors execute in parallel the same sequence of statements $S_1, S_2, \ldots, S_k$, which is independent of the input. In fact, allowing conditional jumps for PRAM's only guarantees a single program, multiple data mode instead of the single instruction, multiple data mode [5, pages 142–143], which we are assuming here. However, due to the constant program size of the PRAM it is easy to always achieve the single instruction, multiple data mode. For the ease of presentation, we assume throughout the paper that each processor has a constant number of local memory cells. This is no restriction, since we can use global memory instead. Hence, our model of a PRAM has no indirect addressing of local memory. Let each processor have a constant amount of local memory cells L_1, $L_2, \ldots, L_D$, and let $G_1, G_2, \ldots, G_{q(n)}$ be the cells of global memory, where n is the length of the input and q is some polynomial. The input is given bitwise in $G_1, \ldots, G_n$. A usual instruction set is shown below. We do not fix the instructions yet, but stress that it is always of finite size. Subsequently, a, b, and c denote some constants, *Length* denotes the length of the input n, *PIN* yields the uniquely determined Processor Identification Number, and *NoOp* means "No Operation." Since the usual comparison of numbers is not an NC^0 operation, we use the relation $\gtrdot$ that asks whether the last bit is set to 1 or to 0.

Constants: $L_a := \langle constant\rangle$, $L_a := Length$, or $L_a := PIN$,

Global Write: $G_{L_a} := L_b$,
Global Read: $L_a := G_{L_b}$,
Local Assignment: $L_a := L_b$,
Conditional Assignment: *if* $L_a > 0$ *then* $\langle$*assignment*$\rangle$,
Binary Operations: $L_a := L_b \circ L_c$,
Jumps: *goto* S_a or *if* $L_a > 0$ *then goto* S_b,
Others: *if* $L_a > 0$ *then Halt* or *if* $L_a > 0$ *then NoOp.*

A PRAM A is determined or described in form of its program which is a fixed sequence $S_1, \cdots, S_K$ of instructions. We will use this instruction set freely. For instance, we will use abbreviations like "perform the following loop $O(\log^k n)$ times" and leave it to the reader to realize this on a PRAM.

All PRAM's in this paper do not use more than a *polynomial number of processors.* In order to get a reasonable hardware cost measure for PRAM's, we require that they have (as usual) *logarithmically bounded word length.* This means that a PRAM working on inputs of length n, generates and uses only numbers of size polynomial in n. For the sake of simplicity of presentation, we use PRAM's only to accept languages and not to compute functions. The contents of global memory cell G_1 determines acceptance or rejection at the end of the computation.

We consider mainly two types of write access to global memory. A machine with *Concurrent Write* access allows simultaneous writing of several processors into the same memory cell. We assume that the value of a writer with highest *priority* is actually stored (Priority-CRCW-PRAM). A machine with *Owner Write* access is more restricted by assigning to each cell of global memory a processor, called *write-owner*, that is the only one allowed to write into this memory cell [24]. More common than the owner concept in formulating algorithms is exclusive access, where we only require that for each point of time there is at most one processor writing into a cell. Exclusive write PRAM's are intermediate in computational power between owner write and concurrent write PRAM's and the same holds for read access. While the owner and the concurrent concept are closely related to determinism and nondeterminism [24, 42, 59], the concept of exclusiveness corresponds to unambiguity [41, 42, 48], which explains the inconstructive features of this concept. Correspondingly, we get two ways to manage read access: *Concurrent Read* and *Owner Read*. In this way, we get four versions of PRAM's, denoted as XRYW-PRAM's with $X, Y \in \{O, C\}$, XR specifying the type of read access, and YW that of the write access.

Denote the class of languages recognizable in time $O(f(n))$ by *XRYW*-PRAM's with a polynomial number of processors by *XRYW-TIME*$(f(n))$. For *XRYW-TIME*$(\log^k n)$, we use the abbreviation $XRYW^k$. We know the relationships (for $k \geq 1$)

$$CRCW^k = AC^k \text{ [59]},$$

$$NC^k \subseteq OROW^k \subseteq CROW^k \subseteq SAC^k \text{ [54, 64]},$$

$$CROW^1 = LOGDCFL \text{ [24], and}$$

$$DSPACE(\log n) \subseteq OROW^1 \text{ [54]}.$$

In CRCW-PRAM's, global memory behaves like a shared memory, since each processor can access each cell of global memory. In the most restricted model, the OROW-PRAM, however, the global memory is deteriorated to a set of one-directional channels between pairs of processors. Thus an OROW-PRAM is something like a completely connected synchronous network. Although this model seems to be much more restricted than CRCW-PRAM's, the relation

$$NC^k \subseteq OROW^k \subseteq CROW^k \subseteq SAC^k \subseteq CRCW^k = AC^k \subseteq NC^{k+1}$$

indicates that it is a model "as parallel as" a CRCW-PRAM. With respect to the implementation of algorithms on existing parallel machines, results of this work demonstrate that even OROW-PRAM's are a parallel model that in some sense are still too powerful. That means algorithms efficiently realizable on OROW-PRAM's still lack some of the features (that is, restrictions) that are necessary for an efficient implementation on distributed memory machines.

2.3 Parallel Pointer Machines

Parallel pointer machines (PPM's) were introduced by Cook [15] and are studied in several papers [18, 22, 23, 34, 40]. In earlier papers [15, 22, 34, 40] the PPM is called *hardware modification machine* (HMM). A PPM consists of a finite collection of finite state transducers, which are called *units*. Each unit is connected to a constant number of other units (points to other units). The units operate synchronously. Each unit receives a constant number of input symbols from other units via the pointer connection, produces according to the inputs read and the current state a constant number of outputs, and changes its state according to the transition function, which is the same for all units. In each step, a unit may modify its pointers to other units. That is, it may change its pointers to show to units that are reachable via pointers by paths of length at most two. Initially, a single unit U_0 is the only one active, starting in some state q_0. At each time step an active unit may activate another one. An input word w is accepted by a PPM if U_0 enters an accepting state. A PPM accesses an input word w in the following way. The starting unit U_0 points to the root of a fixed, complete binary tree of special units that do not count for the hardware costs of the PPM. The input word w is stored from left to right in the leaf nodes of the tree. Leaf nodes point to their neighboring leaf nodes and to a parent node. Each inner node of the tree has pointers to its parent and its two children nodes.

An essential property of PPM's according to Lam and Ruzzo [40], is that they formally capture the notion of parallel computation by pointer manipulation. In addition, PPM's, in contrast to e.g. PRAM's, have the advantage that the unit of hardware is a finite state transducer of constant size [18, 22]. So, PPM's are a parallel model less powerful than PRAM's and have the potential to be a more realistic model for existing parallel computers than PRAM's are. By $PPM\text{-}TIME(t(n))$, we denote the class of languages recognizable in time $O(t(n))$ by PPM's using a polynomial amount of hardware, i.e., a polynomial number of units. We write PPM^k for $PPM\text{-}TIME(\log^k n)$.

Lam and Ruzzo [40] showed that PPM's and an arithmetically restricted form of CROW-PRAM's (*rCROW-PRAM's* for short) are equivalent. More precisely, arithmetic restriction means that the arithmetic capabilities of the CROW-PRAM are limited to incrementation ("+1") and doubling ("∗2"). Recently, Dymond *et al.* [23] demonstrated that any step-by-step simulation of a full n-processor CROW-PRAM by a PPM using an arbitrary number of processors requires time $\Theta(\log\log n)$ per step. This strongly suggests a separation between CROW-PRAM's and PPM's and thus of $LOGDCFL$ and $DSPACE(\log n)$, because $CROW\text{-}TIME(\log n) = LOGDCFL$ [24] and $DSPACE(\log n) \subseteq PPM^1$ [18, 22].

3 Data-Independence and Simple PRAM's

Motivated by the problem to characterize the class of problems that are efficiently implementable on existing, asynchronous parallel machines with distributed memory, the criterion of *data-independence* has been considered in an informal way [29, 39, 58]. The underlying idea is the fact that an algorithm with simple, data-independent communication pattern can be easier partitioned and desynchronized at compile time than one with a more dynamic behavior. Vishkin and Wigderson [66] studied the prospects of data-independence in the context of reducing the size of global memory used during a PRAM algorithm. Cook, Dwork, and Reischuk [17] considered oblivious (i.e., data-independent) and semi-oblivious CREW-PRAM's in order to prove lower bounds for various simple functions, including sorting n keys and finding the logical "OR" of n bits.

In order to formally introduce data-independence, it is first of all necessary to formalize notions like communication pattern or dynamic behavior. We distinguish between three aspects of dynamic, input-dependent behavior:

i) flow of control,

ii) read access (or the receipt of messages), and

iii) write access (or the sending of messages).

Data-independence of control means that the statement executed by a processor of a PRAM depends on the time, the processor identification number, and the length of the input, only, but not on the input itself. If we knew the control flow of each processor in advance, we could determine every direct read and every direct write. In order to determine indirect reads and writes, we need to know the content of the participating indexing register. That is why we are mainly interested in indirect reads and indirect writes.

Before we come to the formal definitions of these three aspects, we have to separate the control aspect from the communication aspect. Consider the following conditional assignment.

$$(*) \qquad \begin{array}{ll} S_\mu : & \textit{if } L_a > 0 \textit{ then } G_{L_b} := L_c; \\ S_{\mu+1} : & \ldots \end{array}$$

It is possible to simulate the conditional assignment S_μ with the help of conditional jumps. The following sequence of instructions has the same effect as $(*)$.

$$(**) \qquad \begin{array}{ll} S_\mu : & \textit{if } L_a > 0 \textit{ then goto } S_{\mu+2/3}; \\ S_{\mu+1/3} : & \textit{goto } S_{\mu+1}; \\ S_{\mu+2/3} : & G_{L_b} := L_c; \\ S_{\mu+1} : & \ldots \end{array}$$

In $(**)$ the problem whether the indirect write takes place is a question whether the control structure, that is, the index of a statement executed at a certain point of time, is data-dependent. It depends on the value of L_a whether the PRAM executes $S_{\mu+1/3}$ or $S_{\mu+2/3}$. On the other hand, $(*)$ has a data-independent control structure. Thus $(*)$ transfers this question into the communication structure. In order to clearly separate communication and control aspects, we will handle those cases always in the manner of $(*)$ and not in that of $(**)$. So, we always get data-independent control structures in the following.

Definition 3 Let A be a $T(n)$ time bounded PRAM with $p(n)$ processors and a program of length k. For any input w of length n, we consider the following sets, where $1 \leq i, j \leq p(n)$, $1 \leq t \leq T(n)$, and $1 \leq \mu \leq K$:

a) By the *control structure* CS_A and the *execution structure* ES_A, we refer to the flow of control of A:

$$\begin{aligned} CS_A(w) :=\ & \{\langle 1^n, t, i, \mu\rangle \mid \text{ in step } t \text{ processor } i \text{ executes statement } \mu\}, \\ ES_A(w) :=\ & \{\langle w, t, i, \mu, v\rangle \mid \text{ in step } t \text{ processor } i \text{ executes statement } \mu \\ & \text{and if } \mu \text{ is a conditional assignment, then } v \text{ contains the} \\ & \text{truth value of the condition, and contains } \mathit{true}, \text{ otherwise}\}. \end{aligned}$$

b) By the *read structure* RS_A, the *write structure* WS_A, and the *semi-write structure* SWS_A, we refer to the communication structure of A:

$$\begin{aligned} RS_A(w) :=\ & \{\langle 1^n, t, i, j\rangle \mid \text{ in step } t \text{ processor } i \text{ executes a (conditional)} \\ & \text{indirect read assignment of the form ``}(\mathit{if}\ L_c > 0\ \mathit{then}) \\ & L_a := G_{L_b}\text{'' } (L_c > 0 \text{ is } \mathit{true}) \text{ and } L_b \text{ contains value } j\}, \\ WS_A(w) :=\ & \{\langle 1^n, t, i, j\rangle \mid \text{ in step } t \text{ processor } i \text{ executes a (conditional)} \\ & \text{indirect write assignment of the form ``}(\mathit{if}\ L_c > 0\ \mathit{then}) \\ & G_{L_a} := L_b\text{'' } (L_c > 0 \text{ is } \mathit{true}) \text{ and } L_a \text{ contains value } j\}, \\ SWS_A(w) :=\ & \{\langle 1^n, t, i, j\rangle \mid \text{ in step } t \text{ processor } i \text{ executes a (conditional)} \\ & \text{indirect write assignment of the form ``}(\mathit{if}\ L_c > 0\ \mathit{then}) \\ & G_{L_a} := L_b\text{'' and } L_a \text{ contains value } j\}. \end{aligned}$$

c) A structure XS_A, $X \in \{C, R, W, SW\}$, is called *data-independent* if for all input words w and w' of same length, $XS_A(w)$ and $XS_A(w')$ coincide. In this case, we set

$$XS_A := \bigcup_{w \in \Sigma^*} XS_A(w).$$

d) Independently of any data-dependencies, we define

$$ES_A := \bigcup_{w \in \Sigma^*} ES_A(w).$$

Observe that the elements of ES_A contain the complete input while the other structures only have information on the length of this input.

When we speak of *communication structure*, we address both the read and the write structure. Note that the only difference between semi-write and write structure is that in the latter we know whether the *if*-part of a conditional assignment evaluates to *true*. Formally, we have the set inclusions $WS_A(w) \subseteq SWS_A(w)$. There is a close connection between semi-write structures and what Cook, Dwork, and Reischuk [17] call semi-oblivious PRAM's. For semi-oblivious PRAM's, whether or not a processor writes into a cell may also only depend on the input. We close this subsection with a fundamental problem of parallel algorithmics and exemplify herein the notions of Definition 3.

Example 1 (*Pointer Jumping, List Ranking*) Let's have two arrays $S[1 \ldots n]$ of *successor* and $P[1 \ldots n]$ of *predecessor* nodes describing a set of acyclic chains for nodes in $\{1, \ldots, n\}$. Assume that each node is a member of a chain beginning in some starting node that is marked by $P[i] = i$, and ending in some final node that is marked by $S[j] = j$. That is, we have that if $k \neq l$, then $S[k] = l \Leftrightarrow P[l] = k$. The task is to determine for each node both the final node in the chain of its successors and the first node in the chain of its predecessors. There are intricate algorithms that solve this problem in optimal $O(\log n)$ steps on a PRAM with $O(n/\log n)$ processors [6, 13]. To illustrate the notion of data-independence, we sketch two simple algorithms that use $O(n)$ processors:

a) Assign to each index $1 \leq i \leq n$ two processors Q_i^S and Q_i^P that execute $\log n$ times $S[i] := S[S[i]]$ resp. $P[i] := P[P[i]]$. Both the control structure and the write structure of this algorithm are data-independent. On the other hand, we use the inputs $S[i]$ and $P[i]$ as index values, i.e., addresses, and thus the read structure is data-dependent.

b) Another possibility is to use Rossmanith's OROW-algorithm [54]. Its underlying idea is that now Q_i^S and Q_i^P execute $\log n$ times the statements $S[P[i]] := S[i]$ resp. $P[S[i]] := P[i]$. Here both the control structure and the read structure are data-independent, whereas the write structure is data-dependent.

We solved the pointer jumping problem either with a data-independent read or with a data-independent write structure. To give a logarithmic time algorithm with data-independent read *and* write structure would mean a major breakthrough in complexity theory, because (as will be proved in the next section) as a consequence we would have $NC^1 = DSPACE(\log n)$, and thus $ATIME(n) = DSPACE(n)$.

If we assume, however, that the input for the list ranking problem is given in a different way, namely in form of an adjacency matrix instead of a pointer list, we can obtain a logarithmic time algorithm that has data-independent control, read, *and* semi-write structure:

Example 1 *(continued)* Now assume that global memory cell $G_{\langle i,j\rangle}$ initially contains the value *true* if there is a connection from node i to node j within the list, and contains value *false*, otherwise. We use the repeated squaring technique. The processor whose PIN is $\langle i,k,j\rangle$ mainly repeats a logarithmic number of times an instruction

$$\textit{if } G_{\langle i,k\rangle} \wedge G_{\langle k,j\rangle} \textit{ then } G_{\langle i,j\rangle} := \textit{true}.$$

After that, for each node we may easily determine whether there are connections to the starting or the final node of the list.

We get, however, the additional data-independence of the semi-write structure at the expense of a cubic instead of a linear number of processors. On the other hand, the above algorithm is *monotonic* in the sense that a value *true* of a global cell is never overwritten by a value *false*. If we allow that a value *true* is overwritten by a *false*, then, together with the feature of conditional writes where only the value of the condition is data-dependent, we can already simulate AC^k-circuits. This will be important when we consider PRAM's with OR write conflict resolution [4] instead of CRCW-PRAM's with priority write conflict resolution later on.

Finally, for technical reasons we have to introduce two PRAM properties that restrict the power of PRAM's, leading to so-called simple PRAM's. Investigating Stockmeyer and Vishkin's [59] proof of equivalence between CRCW-PRAM's and circuits of unbounded fan-in, we show that each language in AC^k can be accepted by such simple CRCW-PRAM's in time $O(\log^k n)$. This result is important for considerations in the next section.

Definition 4 We call a PRAM *simple* if its instruction set fulfills two restrictions:

M: All operations f that modify data are *monadic,* i.e., of the form "$L_a := f(L_b)$."

S: All operations are computable by an $DTIME(\log n)$-uniform, bounded fan-in circuit of constant depth. We call these operations NC^0-computable or *simple.*

4 Characterizing complexity classes by data-independence

In this section we will develop characterizations of the complexity classes AC^k, NC^k, $DSPACE(\log n)$, $LOGDCFL$, PPM^k, and SAC^k in terms of the "structural sets" introduced in Definition 3.

Theorem 5 *For $k \geq 1$, the following statements are equivalent:*

(a) $L \in CRCW\text{-}TIME(\log^k n)$,

(b) $L \in AC^k$,

(c) *L can be recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, A has data-independent control, read, and semi-write structures, and CS_A, RS_A, and SWS_A are in $DTIME(\log n)$.*

(d) *L can be recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, the control, read, and semi-write structure of which all are data-independent.*

Proof. The implication from (a) to (b) is due to Stockmeyer and Vishkin [59]. Observe that the constructed AC^k-circuits are $DTIME(\log n)$-uniform. The implications from (c) to (d) and from (d) to (a) are trivial. Thus it remains to prove that (b) implies (c). Assume that L is recognized by an AC^k-circuit. The basic idea of the proof is to look at Stockmeyer and Vishkin's simulation of an AC^k-circuit by a CRCW-PRAM [59]. We first refer to the simulation and then show that the simulation can be performed by a simple CRCW-PRAM A having the desired properties.

For the actual simulation of an AC^k-circuit, we deal with two parts. First, assume that we are given a pointer structure in global memory representing the interconnection structure of the circuit. The pointer structure permits the usual simulation of a circuit of unbounded fan-in [59]. With each wire of C, there is associated a processor P. Processor P gets the addresses of two global memory cells representing the source and the sink of a wire corresponding to P. The gist of the simulation of C is that each P asks $O(\log^k n)$-times the value of its source and correspondingly updates the value of its sink. This can be done alone with global reads and writes and a conditional assignment.

How can a simple CRCW-PRAM construct a description of the circuit in global memory? We simulate the uniformity machine locally in each processor of the PRAM such that after the simulation the local memory registers of the processors contain the addresses of the source and the sink gate of the interconnection wire represented by the processor. Since the uniformity machine only works on the input length n as its input and since a processor of a simple PRAM can simulate in logarithmic time a $DTIME(\log n)$-TM making use of its constantly many local registers of logarithmic word length, this first point follows. Observe that the successors of TM-configurations (represented by the PIN's of processors) can be computed with NC^0-operations [8, Volume I, pages 110–115].

The above shows that the simulation can be done by a *simple* CRCW-PRAM. It remains to be shown that the simulating simple CRCW-PRAM fulfills the claimed structural restrictions with respect to data-independence:

The simulation of the uniformity machine of the circuit only depends on length n of input word w. The actual simulation of an AC^k-circuit by a simple CRCW-PRAM essentially consists of repeating $O(\log^k n)$ times an instruction of the form "*if* $G_i > 0$ *then* $G_j := 1$," which can be simulated by "$L_a := G_i$; *if* $L_a > 0$ *then* $G_j := 1$". So, the control flow of the simulating PRAM does not depend on the input word w except for its length n. The indirect reading of G_i in the above instruction is also done independent of w—address i does not depend on w. The only thing that depends on w is whether the write on G_j takes place.

The above observations show that the simulation of an AC^k-circuit can be done with data-independent control and read structures. The only thing depending on the concrete input word is whether the *if*-part of a conditional global write instruction evaluated to *true* or *false*. Thus we also get a data-independent semi-write structure. Due to the simplicity of the actual circuit simulation, we immediately have $CS_A, RS_A, SWS_A \in DTIME(\log n)$ for this part of the simulation: The global read and write addresses i and j above are determined by the uniformity machine, which is a $DTIME(\log n)$-machine. The flow of control can be computed by a $DTIME(\log n)$-machine because it consists mainly of a loop repeated $O(\log^k n)$ times. Thus given $\langle 1^n, t, i, l\rangle$, looking at a constant number of the last bits of the binary representation of t completely determines the statement number currently executed.

So, it remains to consider the construction of the circuit in global memory of PRAM A, that is, the simulation of the circuit's uniformity machine by A. Because the simulated uniformity machine is a $DTIME(\log n)$-machine and because of similar considerations as before, we also have $CS_A, RS_A, SWS_A \in DTIME(\log n)$ for the construction phase. The data-independence of the control, read, and semi-write structures for this phase follows from the "data-independent definition" of a uniformity machine. □

The next theorem yields the first characterization of NC^k in terms of PRAM's. Recently, Regan [52] gave another characterization of NC^k by a parallel vector model, using a quite different approach.

Theorem 6 *For $k \geq 2$, the following statements are equivalent:*

(a) $L \in NC^k$,

(b) *L can be recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, A has data-independent control, read, and write structures, and CS_A, RS_A, and WS_A are in $DTIME(\log n)$.*

(c) *L can be recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, A has data-independent control, read, and write structures, and CS_A, RS_A, and WS_A are in NC^k.*

Proof. "(a) implies (b)": As in the proof of Theorem 5 we simulate a circuit C on an input w of length n. Since C is $DTIME(\log n)$-uniform, we know that its extended connection language $L_{EC} \in DTIME(\log n)$. Assume C to have $p(n)$ gates, where gates numbered $1, \ldots, 2n$ contain the input w and its bitwise complement.

In a first phase, for each $\tau \in \{\vee, \wedge\}$ and all $2n < g, g', g'' \leq p(n)$, it is checked whether τ is the type of gate g, g' is the left predecessor of g, and g'' is the right predecessor of g, i.e.: whether $\langle 1^n, g, \tau\rangle, \langle 1^n, g, 0, g'\rangle$, and $\langle 1^n, g, 1, g''\rangle$ are elements of L_{EC}. This can be done locally without any communication deterministically in logarithmic time. In a second phase, the results of the first phase are collected and combined such that for $2n < g \leq p(n)$ processor P_g locally knows the type of gate g and the indices g' and g'' of the two predecessors of gate g. These two phases use n, the length of the input w, but don't read the input and hence are data-independent. Furthermore, they can be done in a very regular way, for instance using a complete binary tree of depth $O(\log(p(n))) = O(\log n)$ to collect and compare the data, such that CS_A, RS_A, and WS_A are elements of $DTIME(\log n)$.

In a third phase, for each $2n < g \leq p(n)$, let processor P_g for some constant c execute $c \log^k n$ times the following three instructions.

1. $L_a := G_{g'}$;
2. $L_b := G_{g''}$;
3. $G_g := L_a \; op \; L_b$;

where op depends on the type of gate g. Then CS_A of this phase is obviously in $DTIME(\log n)$. Also $WS_A \in DTIME(\log n)$ since $\langle 1^n, t, i, j\rangle \in WS_A$ if and only if $2n < i = j \leq p(n)$ and t is divisible by 3. Finally, $\langle 1^n, t, i, j\rangle \in RS_A$ if either $t \equiv 1 \mod 3$ and $\langle 1^n, i, 0, j\rangle \in L_{EC}$, i.e., j is the left predecessor of i, or $t \equiv 2 \mod 3$ and $\langle 1^n, i, 1, j\rangle \in L_{EC}$, i.e., j is the right predecessor of i. Since $L_{EC} \in DTIME(\log n)$, we get $RS_A \in DTIME(\log n)$.

"(b) implies (c)" : trivial

"(c) implies (a)": We will now construct a circuit family $C = (C_n)_{n \geq 1}$ recognizing $L(A)$. The uniformity of this construction will be done by a circuit, as well. Hence, in the following we will speak about the uniformity circuit U and the (main) circuit C. To simulate a PRAM by a circuit, we work with recursive constructions similar to those in several other papers [24, 25, 26, 28, 41, 48]. We consider functions *Global* and $Local_a$, stating

i) $Global(t, i) = j \Leftrightarrow$ global memory cell i contains after step t value j, and

ii) $Local_a(t, p) = j \Leftrightarrow$ local memory cell a of processor p after step t contains value j.

To each such function we assign in C a bunch of logarithmically many gates that represent its value. The main work is hidden in the interconnection structure of the circuit and is done by the uniformity circuit.

We go through the instructions of the PRAM (see Subsection 2.2 for the underlying instruction set) and show how to compute $Local_a$ and *Global* for all possible cases. Because the central ideas apply to several similar contexts, we only present the typical and most difficult cases.

Computation of $Global(t, i)$:

To compute the interconnection structure for $Global(t, i)$-gates, the uniformity circuit U determines in depth $O(\log^k n)$ for each processor p whether $\langle 1^n, t, p, i\rangle \in WS_A$ in parallel. If no such p is found by U, then each bit of $Global(t, i)$ is connected with $Global(t-1, i)$.

Otherwise, U finds with additional depth $O(\log n)$ and thus in total depth $O(\log^k n + \log n) = O(\log^k n)$ the p which is minimal among all p with $\langle 1^n, t, p, i\rangle \in WS_A$. In this way, we simulate the *priority* way of solving write conflicts. After that, U determines the unique μ such that $\langle 1^n, t, p, \mu\rangle \in CS_A$. Now U knows that statement S_μ executed by processor p at time t is either "$G_{L_a} := L_b$" or "*if* $L_c > 0$ *then* $G_{L_a} := L_b$," where $L_c > 0$ is *true*. In either case we connect $Global(t, i)$ with the gates representing $Local_b(t-1, p)$.

Computation of $Local_a(t,p)$:

To compute $Local_a(t, p)$, we first let U determine the uniquely existing μ such that $\langle 1^n, t, p, \mu\rangle \in CS_A$. Let S_μ be the statement executed at time t by processor p. We must consider several cases.

1. $S_\mu \equiv$ "$L_a := G_{L_b}$": Here U searches for the uniquely existing index j such that $\langle 1^n, t, p, j\rangle \in RS_A$. Then U connects $Local_a(t, p)$ with $Global(t-1, j)$.[1]

2. $S_\mu \equiv$ "*if* $L_c > 0$ *then* $L_a := L_b$": The uniformity circuit builds two intermediate bunches of gates. Each gate of $Local_b(t-1, p)$ is conjoined with the last bit of $Local_c(t-1, p)$, giving the bunch $Local_b^{+c}(t-1, p)$. This coincides with $Local_b(t-1, p)$, if $Local_c(t-1, p) > 0$, and is $\langle 0, \ldots, 0\rangle$, otherwise. Correspondingly, U conjoins each gate of $Local_a(t-1, p)$ with the negation of the last bit of $Local_c(t-1, p)$, yielding the bunch $LOCAL_a^{-c}(t-1, p)$. This coincides with $Local_a(t-1, p)$, if not $Local_a(t-1, p) > 0$, and is $\langle 0, \ldots, 0\rangle$, otherwise. Eventually, U connects $Local_a(t, p)$ with the bitwise disjunction of $Local_a^{-c}(t-1, p)$ and $Local_b^{+c}(t-1, p)$.

3. $S_\mu \equiv$ "$L_a := f(L_b)$": We know that function f is computable in NC^0. Thus U connects $Local_b(t-1, p)$ with the input gates of the NC^0-circuit for f and lets the outputs of this circuit be $Local_a(t, p)$.

4. *Other cases*: If, for example, $S_\mu \equiv$ "$L_a := PIN$", then U connects $Local_a(t, p)$ to the binary coding of p. The other cases "$L_a := Length$," "$L_a := \langle constant\rangle$," "$L_a := L_b$" are also simple. If S_μ is a statement where we have no assignment to L_a, then U connects $Local_a(t, p)$ to $Local_a(t-1, p)$.

In total, we end up with a circuit family $C = (C_n)_{n\geq 1}$ of depth $O(\log^k n)$. Obviously, the direct connection language $L_{DC}(C)$ is an element of NC^k. By a result of Ruzzo [55, Lemma 2], this implies $L_{EC}(C) \in NC^k$. Applying Lemma 1, we finally get $L(A)$ is in NC^k. □

Remark 7 Observe that in the construction of the PRAM simulating the circuit C we only made queries concerning the direct predecessors of gates. Thus, for $k = 1$ the constructions show (a) how to simulate circuits of logarithmic depth with $L_{DC} \in DTIME(\log n)$ by fully data-independent PRAM's and (b) how to simulate fully data-independent PRAM's by circuits of logarithmic depth with $L_{DC} \in NC^1$, i.e., by weakly uniform NC^1. We remark that it is possible to come down to $DTIME(\log n)$ if we would work with *functional* uniformities where we demand on the one hand that for a gate g its left and right predecessor can be computed deterministically in logarithmic time and, on

[1] We simulate the conditional global read statement *if* $L_c > 0$ *then* $L_a := G_{L_b}$ by the two instructions $L_d = G_{L_b}$; *if* $L_c > 0$ *then* $L_a := L_d$. This trick does not work for conditional global write statements.

the other hand, for the communication structures that for index i and time t the processor p writing into i (if existent) can be computed in $DTIME(\log n)$ and for time t and processor p the cell i from which p reads (if existent) can be computed in $DTIME(\log n)$.

Theorem 6 naturally leads to the question of whether it is at all necessary that, besides being data-independent, the complexity of the sets CS_A, RS_A, WS_A has to be restricted in the given way. Clearly, without any such assumptions, we have that CS_A, RS_A, WS_A are contained in AC^k, because the running time of the considered PRAM A is $O(\log^k n)$. Thus by the construction of "(c) implies (a)" given in the above proof, we would only get that L belongs to AC^k-uniform NC^k where AC^k-uniform means that the extended connection language of the circuit family is in AC^k. However, the construction of "(a) implies (b)", together with the constructions of Damm *et al.* [20], give the converse. Hence, we have:

Corollary 8 *$L \in AC^k$-uniform NC^k if and only if L can be recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, such that A has data-independent control, read, and write structures.*

The last result shows how essential the complexity assumptions of the control and communication structures in Theorem 6 are. For example, by padding arguments the equality of AC^1-uniform NC^1 and NC^1 would imply $ATIME(n) = DSPACE(n)$, which would be a major surprise. In summary, the above consideration shows that the assumptions in Theorem 6, i.e., CS_A, RS_A, WS_A belong to NC^k (or, equivalently (cf. Lemma 1), to $DTIME(\log n)$), appear to be necessary and cannot be omitted.

Let us shortly recapitulate the fundamental difference between the characterizations of AC^k and NC^k. For NC^k, we had to "fix everything." Neither the read nor the write structure are allowed to be data-dependent. By way of contrast, for AC^k "everything is free." Both read and write structure may be data-dependent, but it is not necessary to allow so much in order to get AC^k. As we saw in Theorem 5, we can even demand for data-independent read and semi-write structures.

Now it's only natural to ask what happens if we do not allow data-dependent writes, but data-dependent reads instead. Does that also suffice to get AC^k? No. Data-dependent reads are only enough for a characterization of $DSPACE(\log n)$. This is a concrete indication that in parallel computations, writing is more powerful than reading.

Theorem 9 *$L \in DSPACE(\log n)$ if and only if L is recognized by a simple CRCW-PRAM A in $O(\log n)$ time, A has data-independent control and write structures, a data-dependent execution structure, and CS_A, WS_A, and ES_A are in $DTIME(\log n)$.*

Proof. "if": Again the simulation of PRAM A works with recursive constructions. We use functions $Global(t,i)$ and $Local_a(t,p)$, where $Global(t,i) = j$ if global memory cell i contains after step t value j, and $Local_a(t,p) = j$ if local memory cell a of processor p contains after step t value j. The main idea is to compute the values of $Global$ and $Local_a$ by a recursion of logarithmic depth that stacks only items of constant length. Thus, we can keep the stack on the logarithmically bounded working tape. The working tape of the simulating, logarithmically space bounded Turing machine M is organized as follows: First, M has a stack of logarithmic depth that stores statement numbers and certain markings concerning the progress of the simulation. The number of statements is bounded by a constant, and thus the stack fits onto the working tape. Then, M has space to store the parameters (step number, cell number, processor number) and the intermediate result of the last recursive call. We proceed in the same way as in the proof of Theorem 6. We begin with the computation of $Global(t, i)$. Remember that a cell of global memory can only be affected by indirect writes.

Computation of $Global(t, i)$:

To find out the value of $Global(t, i)$ in logarithmic space, M exhibits whether there exists a p such that $\langle 1^n, t, p, i\rangle \in WS_A$. This is possible due to the obvious inclusion $DTIME(\log n) \subseteq DSPACE(\log n)$. If there is no such p, then M knows that no processor tried to write into G_i and M recursively computes $Global(t-1, i)$ by stacking a symbol "no write." Otherwise, M computes the unique statement number μ such that $\langle 1^n, t, p, \mu\rangle \in CS_A$. Statement S_μ must be either of the form "$G_{L_a} := L_b$" or "*if* $L_c \gtrdot 0$ *then* $G_{L_a} := L_b$", and in the latter case we have $L_c \gtrdot 0$. So, M recursively computes $Local_b(t-1, p)$, stacking the statement number μ.

Computation of $Local_a(t, p)$:

To compute $Local_a(t, p)$, M first determines an index μ such that $\langle w, t, p, \mu, v\rangle \in ES_A$. The recursion is now guided by the type of statement S_μ.

1. $S_\mu \equiv$ "$L_a := G_{L_b}$": The simulating machine M goes into the recursion by computing $Local_b(t-1, p)$ and stacking index μ, which is marked as "undone." When M returns from the recursion with a result $j = Local_b(t-1, p)$, it recognizes the stack entry μ to denote an indirect read. Thus M transfers j on the parameter place. Then M continues with the computation of $Global(t-1, j)$ and μ is unmarked. Should M later on return from a higher level of recursion, it will pass this level and simply hand through the result, popping entry μ.

2. $S_\mu \equiv$ "*if* $L_c \gtrdot 0$ *then* $L_a := L_b$": Since ES_A also provides the value v (*true* or *false*) of the *if*-part, M simply does the following: If v evaluates to *true*, then go into the recursion $Local_b(t-1, p)$, and go into the recursion $Local_a(t-1, p)$, otherwise.

3. All the other cases can be led back to the above two or are handled similarly as in the proof of Theorem 6.

Observe that it was for statements of form "*if* $L_c \gtrdot 0$ *then* $L_a := L_b$" where we made decisive use of $ES_A \in DTIME(\log n)$. By this we could overcome the problem that this kind of instruction is not monadic and would cause a branching of the recursion, otherwise.

"only if": This inclusion follows along the lines of the proof of Theorem 5: The outline of the simulation of a $DSPACE(\log n)$-machine M is as follows: First, the PRAM A generates all possible configurations of M, second, it computes the successors of all configurations (thus computing the configuration graph), and third it does pointer jumping to find out whether M accepts or rejects the input. Note that the configuration graph of M has outdegree one and by the standard "clocking trick" we may assume that it is acyclic.

As in the proof of Theorem 5, the first step of the above outline is done by interpreting the PIN of a processor as a configuration of M. We assume A to be equipped with the following three NC^0-computable operations: $Headpos(i)$ which computes out of a configuration i the position of the input head, $Next_0(i)$ which computes the successor configuration of i if the input bit under the input head is a 0, and $Next_1(i)$ which computes the successor configuration of i if the input bit under the input head is a 1. The program of A looks as follows:

1. $L_a := Headpos(PIN)$;
2. $L_b := Next_0(PIN)$;
3. *if* $G_{L_a} > 0$ *then* $L_b := Next_1(PIN)$;
4. $G_{PIN} := L_b$;
   After that A performs $c \log n$ times the loop:
5. $L_b := G_{L_b}$;
6. $G_{PIN} := L_b$;

The control structure of this program is $CS_A = \{\langle 1^n, t, i, t\rangle \mid 1 \le t \le 5\} \cup \{\langle 1^n, t, i, \mu\rangle \mid 5 < t \le 4 + c\log n, \mu = 5 + (t-5)i \bmod 2\}$. The write structure is even simpler: $WS_A = \{\langle 1^n, 4+2j, i, i\rangle \mid 1 \le j \le c\log n/2\}$. Finally, the execution structure of A is $ES_A = \{\langle w, t, i, \mu, true\rangle \mid \langle 1^{|w|}, t, i, \mu\rangle \in CS_A, t \ne 3\} \cup \{\langle w, 3, i, 3, v\rangle \mid v = true \Leftrightarrow$ the jth bit of w is $1\}$, where in the last expression $j := Headpos(i)$. Obviously, these three sets are in $DTIME(\log n)$. □

Remark 10 **a)** Theorem 9 is only stated for a logarithmic running time and not for the general cases of the form $O(\log^k n)$. The reason is that we know of no analogue of $DSPACE(\log n)$ in the higher levels of the NC-hierarchy. We can interpret the Theorem as offering us this missing link in form of the class of all languages recognized by PRAM's in time $O(\log^k n)$ which have CS, WS, and ES in $DTIME(\log n)$.

b) Corresponding to Theorem 6 we could afford to let the sets CS_A, WS_A, and ES_A to be members of $DSPACE(\log n)$ without strengthening the computational power.

c) Theorem 9 implies that if there was a completely data-independent algorithm for the $DSPACE(\log n)$-complete list ranking problem, then we had the equality of NC^1 and $DSPACE(\log n)$ and hence of $ATIME(n)$ and $DSPACE(n)$.

Reviewing the fundamental properties of the characterizations of AC^k, NC^k, and $DSPACE(\log n)$, it appears that the essential differences lie in the distinct communication possibilities with respect to data-(in)dependence. Unbounded fan-in parallelism allows for data-dependent writes, bounded fan-in parallelism restricts reads and writes to be data-independent, and $DSPACE(\log n)$ allows for data-dependent reads, but demands for data-independent writes.

In the above proof it is possible to drop the requirement that the operations of the CRCW-PRAM be NC^0-computable. We can allow operations computable in logarithmic space. On the other hand, it is essential that all operations are monadic, because this leads to the linear recursion structure. If we drop the requirement for monadic NC^0-computable operators and further on do not require $ES_A \in DTIME(\log n)$, then, among others, we get $LOGDCFL$, the class of languages logspace many-one reducible to deterministic context-free languages [60].

Theorem 11 *For each $k \ge 1$ the following statements are equivalent:*

(a) *L is recognized in time $O(2^{\log^k n})$ by a deterministic auxiliary push-down automaton equipped with a work tape of size $O(\log n)$.*

(b) *L is recognized by a CROW-PRAM in time $O(\log^k n)$.*

(c) *L is recognized by a CRCW-PRAM A with standard PRAM operation set*[2] *in $O(\log^k n)$ time, A has data-independent control and write structures, and CS_A and WS_A are in $DTIME(\log n)$.*

Proof. "(a) implies (b)": This was shown by Dymond and Ruzzo for the case $k = 1$ in [24]. Their algorithm was extended by Fernau *et al.* to arbitrary k [25].

"(b) implies (c)": The constructions that simulate deterministic push-down automata by CROW-PRAM's lead to algorithms with an extremely simple control and write structure [24, 25]. A simple loop has to be executed $O(\log^k n)$ times in which each processor p executes conditional assignments to local registers or unconditional assignments to global cells which are *write-owned* by p, i.e.: only p is allowed to write into these cells. Given an index i in global memory, the write owner of i is easily computable. CS_A, $WS_A \in DTIME(\log n)$ is easily checked for these algorithms.

"(c) implies (a)": We use the same recursion as in Theorem 9. But now we augment the $DSPACE(\log n)$ Turing machine with an auxiliary push-down store (thus yielding a so-called auxiliary push-down automaton [14]), since the recursion is no longer linear. More precisely, the data-flow of the recursion is no longer linear. Since each recursive predicate *Global* and $Local_a$ is of constant "branching-width," the total amount of recursion calls is bounded exponentially in the time of the PRAM. Observe that now the use of conditional assignments of the form "*if* $L_c \cdot\!> 0$ *then* $L_a := L_b$" does not require any more $ES_A \in DTIME(\log n)$, because a branching of the recursion no longer needs to be avoided. □

Remark 12 **(a)** For $k = 1$ we get a characterization of the complexity of deterministic context-free languages since $LOGDCFL$ is precisely the class of languages accepted in polynomial time by deterministic auxiliary push-down automata equipped with logarithmic work tapes [60].
(b) In the above proof it is crucial that global writes of the PRAM are data-independent. Data-dependent write instructions in the case $k = 1$ would lead to a recursion requiring running time $n^{O(\log n)}$ for the simulating AuxPDA. However, such an AuxPDA can already simulate AC^1-circuits [55].

Next, we come to the characterization of parallel pointer machines or, equivalently, rCROW-PRAM's [40]. Cook and Dymond [18, 22] showed the inclusion $DSPACE(\log n) \subseteq PPM\text{-}TIME(\log n)$. Dymond *et al.* [23] recently proved that any step-by-step simulation of CROW-PRAM's by PPM's needs time $\Theta(\log\log n)$ per step. In our setting the difference appears in the requirement for simple PRAM's (similar to Lam and Ruzzo [40]) in the case of PPM's, whereas for the CROW-PRAM's the operation set is unrestricted (compare Theorem 11 with Theorem 13).

Theorem 13 *For $k \geq 1$, we have $L \in PPM\text{-}TIME(\log^k n)$ if and only if L is recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, A has data-independent control and write structures, and CS_A and WS_A are in $DTIME(\log n)$.*

[2]More precisely, an operation set as used for CROW-PRAM's by Dymond and Ruzzo [24] suffices.

Proof. "if": For this direction we make decisive use of Lam and Ruzzo's [40] equality $PPM\text{-}TIME(\log^k n) = rCROW\text{-}TIME(\log^k n)$. We perform a simulation of the simple CRCW-PRAM A, which is data-independent in the above required way, by an rCROW-PRAM. The only subtle point herein is the question how to convert the concurrent write of A into the owner write of the rCROW-PRAM. Here we make use of the restricted write structure of A.

We proceed in two main steps. First, we demonstrate how a concurrent write can be simulated by an rCROW-PRAM with time-dependent owner function. Second, we explain how to convert the latter into a time-independent one.

Let us turn to the first step. By the help of $WS_A \in DTIME(\log n)$, for each point of time t, for each processor p, and for each global memory cell i of the CRCW-PRAM, the simulating rCROW-PRAM finds out whether p writes into i at time t. Afterwards, the rCROW-PRAM determines for each t and each i some p writing into i at t. This p then is the write-owner of i at time t. This information is stored in a look-up table. An rCROW-PRAM does all these computations in time $O(\log n)$, using no more than a polynomial number of processors.

Now it remains to explain how to convert an rCROW-PRAM with time-dependent write owners into one with time-independent ones. The basic idea is to replace the various time-dependent write-owners by only one fixed write-owner that communicates with the respective (time-dependent) write-owners and then writes by itself the value the previous write-owner wanted to write. To do that, this particular write-owner has to know for each point of time the original write-owner. Here the look-up table generated before comes into play. Thus the new, fixed write-owner may look up the current write-owner, communicate the value to be written and, eventually, writes the value by itself.

"only if": For the reverse direction we simulate a parallel pointer machine by a PRAM in the usual way [40]. Each processor simulates one PPM unit, using a block of constantly many cells of global memory to hold the state, output and taps of the simulated unit plus some additional housekeeping information. The simulation works as follows. Each PRAM processor reads the outputs of the neighbors of the unit it is simulating and updates the state, output, and pointers stored in its block according to the PPM's transition function. The case whenever a PPM unit spawns new units requires some care (especially a "cleanup" phase every $\log n$ steps to re-balance the tree of processors simulating the active PPM units is necessary: using a prefix sums computation, which can be done in NC^1 and thus (by Theorem 6) in a data-independent way by a simple PRAM), but is basically straightforward. Further details can be found in Lam and Ruzzo's work [40].

From the above simulation of a PPM by a PRAM, it is easy to conclude that the control structure of the simulating PRAM, which essentially consists of one main loop, is data-independent and contained in $DTIME(\log n)$. To see that the simulating PRAM also has a data-independent write structure contained in $DTIME(\log n)$, observe that for the above described simulation of a PPM we may w.l.o.g. lay down that each PRAM processor always (unconditionally) writes in a fixed order into the block of constantly many global memory cells it is responsible for. Furthermore, the updating of the pointer, state, and output information can be done in each processor's local memory, thus avoiding any conditional global writes. The computation of the transition function of the PPM is done by the help of the conditional assignment "*if* $L_a > 0$ *then* $L_b := L_c$" in a basically straightforward manner. Lam and Ruzzo [40] use incrementation to address cells within the global memory blocks. We can avoid the use of the incrementation operation and stick to NC^0-computable ones. If we lay down that the addressing within memory blocks works with with a base address where only the least significant bits have to be modified to address a cell within a block, then this can be done by NC^0-operations without the need for incrementation. Obviously this addressing scheme can be used without loss of generality. Thus, NC^0-operations suffice. □

Theorem 9 and Theorem 13 show that the essential difference between $DSPACE(\log n)$ and $PPM\text{-}TIME(\log n)$ is that for the latter we need not require a data-dependent execution structure contained in $DTIME(\log n)$. To the authors' best knowledge, only $DSPACE(\log n) \subseteq PPM\text{-}TIME(\log n) \subseteq DSPACE(\log^2 n)$ is known [18, 22].

Until now, all our characterizations have worked with CRCW-PRAM's using the Priority resolution protocol for write conflicts. Let us shortly consider an enhanced CRCW-PRAM model, Akl's OR-PRAM [4]. The OR-PRAM resolves write conflicts by writing the bitwise OR of all data to be written. This seemingly slight revision of the underlying CRCW-PRAM model has drastic consequences for our "data-(in)dependent world." A fully data-independent OR-PRAM suffices to get AC^k: In the characterization of AC^k (Theorem 5), the decisive, data-dependent write instruction was "$\mathit{if}\, G_i > 0\, \mathit{then}\, G_j := 1$," where the value of the *if*-part depended heavily on the input, but the indexing values i and j were data-independent. In an OR-PRAM this instruction can be replaced by an instruction "$G_j := G_i$", using only the last bit of G_i. So we get data-independent control, read *and* write structures for the simulation of AC^k-circuits. Remember that in our standard model of simple CRCW-PRAM's, this is a very strong restriction decreasing the computational power from AC^k to NC^k.

An essential property of the above, fully data-independent simulation of AC^k-circuits by OR-PRAM's is the need for non-monotonic operations. The OR-feature for concurrent writes directly applies only to unbounded OR-gates. For AND-gates, we make use of de Morgan's law by $x \wedge y = \overline{(\overline{x} \vee \overline{y})}$. As a consequence, ones may be overwritten by zeroes.

By way of contrast, in a *monotonic PRAM,* global memory cells shall only contain values 0 or 1 and a 1 is never overwritten by a 0. Observe that assuming the input bits are given in non-negated *and* negated form, by applying de Morgan's laws to NC-circuits these can be made monotonic. Thus, in fact, Theorem 6 can be tightened to simple, *monotonic* PRAM's. Monotonicity of a PRAM algorithm can be an important criterion concerning implementation on asynchronous machines. A monotonic algorithm may tolerate processors that make different progress in the course of the computation. If the slower processor needs data from the faster one, with monotonic algorithms we avoid storing (old) data of the faster processor that have the time stamp the slower processor has currently reached. The slower processor can simply use the newest data delivered from the faster one, it can work with "data from the future." This avoids synchronization overhead. For example, consider the (parallel version of the) Warshall algorithm computing the transitive closure of graphs given as adjacency matrices. Here an existing 1, signaling the existence of a path between two nodes, is never overwritten by a 0—the algorithm is monotonic. It does no harm to the Warshall algorithm that one processor works with matrix entries that are produced by another one that is ahead.

We get AC^k if we require for monotonic OR-PRAM's, but allow data-dependent writes: It is clear how to simulate OR-gates, but what about the AND-gates? The trick is to simulate AND-gates just like OR-gates by interpreting an input 1 as a 0 and vice versa. Then output 1 of such computed AND-function, in fact, means 0 and 0 means 1. Data-dependent conditional write instructions are necessary to realize such opposite interpretations of values for AND-gates.

What if we require a fully data-independent OR-PRAM to be monotonic? We get SAC^k, the classes of languages recognized by semi-unbounded fan-in circuits of polynomial size and $O(\log^k n)$ depth [64]. Venkateswaran [64] proved that SAC^1 is equal to $LOGCFL$, the class of languages logspace many-one reducible to context-free languages [60]. Observe that for the subsequent theorem we make use of the fact that, given that input word bits etc. are stored in negated and unnegated form, we can assume w.l.o.g. the NC^0-operations to be restricted to AND's and OR's (no negations).

Theorem 14 *For $k \geq 1$, we have $L \in SAC^k$ if and only if L is recognized by a simple, monotonic OR-PRAM A in time $O(\log^k n)$, A has data-independent control, read, and write structures, and CS_A, RS_A, and WS_A are in $DTIME(\log n)$.*

Proof. "if": We again make use of the recursive functions *Global* and $Local_a$ in order to construct a semi-unbounded fan-in circuit for the simulation of PRAM A. The construction works analogous to the proof of Theorem 6, so we will only describe the comparative differences.

The computation of $Local_a(t,p)$ is the same as in Theorem 6. It is decisive here that A uses only monotonic operations, because in SAC^k-circuits negating gates are not admissible. The computation of $Global(t,i)$ is the same as in Theorem 6 except for the following. For each bit of cell i, we have an unbounded fan-in OR-gate. The inputs of these OR-gates are the respective bits of $Local_b(t-1,p)$, where p stands for all processors writing into i by an instruction "$G_{L_a} := L_b$" or "*if* $L_c > 0$ *then* $G_{L_a} := L_b$". This reflects the OR concurrent write feature of A. If no processor writes, we connect $Global(t,i)$ with $Global(t-1,i)$.

"only if": For the reverse direction, we refer to Theorem 5 and Theorem 6. Again we just state the main changes we have to observe here. For the evaluation of bounded fan-in OR-gates proceed as in Theorem 6—sequentially read all inputs of the gate. For the evaluation of unbounded fan-in gates, A makes use of its OR feature in order to avoid the necessity for data-dependent conditional writes. The simulation of the circuit's uniformity machine works as in Theorem 6. Note that Venkateswaran's SAC^k-circuits [64] are even $DTIME(\log n)$-uniform. Altogether, in each case monotonic instructions are sufficient. Data-independence follows in the same way as in Theorem 6. □

5 Index-PRAM's

In the previous section we obtained our results by requiring several structural restrictions for CRCW-PRAM's. By way of contrast, Index-PRAM's in some sense possess "built-in data-independence". There are several additional features to the basis model of an Index-PRAM, which are to be chosen by the programmer. The rough idea behind them is that the fewer the deviations from the basis model, the easier the implementation on real distributed memory machines will be.

In the first subsection we introduce our basis model and a basic lemma. In the second subsection we provide results analogous to the structural characterizations of the preceding section.

5.1 The model and its features

The central point in the definition of Index-PRAM's is the introduction of *index registers*. Index registers are exclusively used for addressing global memory cells. Consequently, we distinguish between three kinds of registers for PRAM's. By G we refer to global registers, by L to local data registers, and by I to local index registers. In general, local data registers are not used any longer to index global memory cells, but for this purpose are replaced by index registers. We still have, however, a constant number of local data and index registers per processor. We allow index registers only to access the length of the input and the processor identification number, but *not* to depend on the input word. A usual instruction set for Index-PRAM's is shown below. Compare it to that of (general) PRAM's given in Subsection 2.2.

Constants: $L_a := \langle constant \rangle$, $L_a := Length$, or $L_a := PIN$,
$I_a := \langle constant \rangle$, $I_a := Length$, or $I_a := PIN$,

Global Write: $G_{I_a} := L_b$,

Global Read: $L_a := G_{I_b}$,

Local Assignment: $L_a := L_b$, $L_a := I_b$, *or* $I_a := I_b$,

Conditional (Local) Assignments: *if* $\langle$*Input-Bit*$(I_c)\rangle$ *then* $L_a := L_b$,
if $I_c > 0$ *then* $L_a := L_b$, or *if* $I_c > 0$ *then* $I_a := I_b$,

Monadic Operations: $L_a := f(L_b)$,

Monadic and Binary Operations: $I_a := f(I_b)$, *or* $I_a := I_b \circ I_c$,

Jumps: *goto* S_a or *if* $I_a > 0$ *then goto* S_b,

Others: *if* $I_c > 0$ *then Halt* or *if* $I_c > 0$ *then NoOp* .

The condition "$\langle$*Input-Bit*$(I_c)\rangle$" in the above input conditional local assignment means that here by I_c we give the position i of a bit in the input word w, where $1 \leq i \leq |w|$, and the condition is *true* iff the bit is 1 and is *false*, otherwise.

Our *basis model* of an Index-PRAM is as follows.

1. As can be seen in the given instruction set, binary operations are only allowed for index registers, otherwise monadic operations are obligatory.

2. Only NC^0-computable operations are admissible for monadic as well as binary operations.

Obviously, the control and communication structure of an Index-PRAM are data-independent. The following result quantifies the complexity of the control and communication structures and is used several times in proving the results of the following subsection.

Lemma 15 *Let A be an Index-PRAM operating in time $O(\log^k n)$. Then for $k \geq 2$, the control and communication structures CS_A, RS_A, and WS_A are recognizable in NC^k. For $k = 1$, these sets are in weakly uniform NC^1.*

Proof. Let $S_1, S_2, \ldots, S_K$ be the program of A and $c = O(\log n)$ be the word-length of A, that is, each register of A can hold c bits. First we describe a circuit U that will compute the statements executed by the processors and the contents of their index registers. For each time step t, each processor p, each local index register a, and each statement μ there are gates named $State(t, p, \mu)$ and $Ibit_a(t, p, j)$ with the following meanings:

1. Gate $State(t, p, \mu)$ evaluates to *true* if and only if in time step t processor p executes statement S_μ, and

2. gate $Ibit_a(t, p, j)$ evaluates to *true* if and only if the jth bit of index register I_a of processor p after the tth time step is set to one.

We now give the interconnection structure between these gates which is influenced by a construction in [41, Theorem 4.9].

Computation of $State(t, p, \mu)$:

Gate $State(t, p, \mu)$ is the disjunction of the following expressions:

$State(t-1, p, \mu')$ for all μ' such that $S_{\mu'}$ is of the form *goto* S_μ,

$State(t-1, p, \mu') \wedge Ibit_a(t-1, p, 0)$ for all μ' and a such that $S_{\mu'}$ is of the form *if* $I_a \dot{>} 0$ *then goto* S_μ,

$State(t-1, p, \mu-1) \wedge \neg Ibit_a(t-1, p, 0)$ for all μ' and a such that $S_{\mu-1}$ is of the form *if* $I_a \dot{>} 0$ *then goto* $S_{\mu'}$,

$State(t-1, p, \mu)$ if S_μ is the *Halt* statement, and

$State(t-1, p, \mu-1)$ for all μ such that $S_{\mu-1}$ is neither a *goto* statement, nor a conditional *goto* statement nor a *Halt* statement.

Computation of $Ibit_a(t, p, j)$:

Gate $Ibit_a(t, p, j)$ is the disjunction of the following expressions:

$\langle constant \rangle_j \wedge State(t, p, \mu)$ for all μ such that S_μ is of the form $I_a := \langle constant \rangle$, where $\langle constant \rangle_j$ denotes the jth bit of the constant,

$p_j \wedge State(t, p, \mu)$ for all μ such that S_μ is of the form $I_a := PIN$, where p_j denotes the jth bit of p,

$n_j \wedge State(t, p, \mu)$ for all μ such that S_μ is of the form $I_a := Length$, where n_j denotes the jth bit of n,

$Ibit_b(t-1, p, j) \wedge State(t, p, \mu)$ for all μ such that S_μ is of the form $I_a := I_b$,

$(Ibit_b(t-1, p, j) \wedge Ibit_c(t-1, p, 0) \ \vee \ Ibit_a(t-1, p, j) \wedge \neg Ibit_c(t-1, p, 0)) \wedge State(t, p, \mu)$ for all μ such that S_μ is of the form *if* $I_c \dot{>} 0$ *then* $I_a := I_b$,

$output_j \wedge State(t, p, \mu)$ for all μ such that S_μ is of the form $I_a := I_b \circ I_c$, where $output_j$ is the jth output bit of an NC^0 circuit computing the operation $\circ$ with inputs $Ibit_b(t-1, p, .)$ and $Ibit_c(t-1, p, .)$, and

$Ibit_a(t-1, p, j) \wedge State(t, p, \mu)$ for all μ such that S_μ is neither an assignment nor an conditional assignment to index register I_a.

The width of the disjunctions for $State(t, p, \mu)$ and $Ibit_a(t, p, j)$ is bounded by K, the length of A's program. Since K is constant, the circuit U consists of $O(\log^k n)$ layers of constant depth and hence has depth $O(\log^k n)$ in total. Furthermore, it should be clear that its direct connection language is in $DTIME(\log n)$. By Lemma 1 U fulfills the stated uniformity conditions.

Using U the statement now follows from the following relations:
a) $\langle 1^n, t, i, \mu\rangle \in CS_A$ if and only if gate $State(t,i,\mu)$ evaluates to *true*.
b) $\langle 1^n, t, i, j\rangle \in RS_A$ if and only if

$$\bigvee_{S_\mu \text{ is } L_a := G_{I_b}} State(t,i,\mu) \wedge \bigwedge_{1\le m\le c} (j_m \equiv Ibit_b(t-1,i,m)),$$

where j_m is the mth bit of j.
c) $\langle 1^n, t, i, j\rangle \in WS_A$ if and only if

$$\bigvee_{S_\mu \text{ is } G_{I_a} := L_b} State(t,i,\mu) \wedge \bigwedge_{1\le m\le c} (j_m \equiv Ibit_a(t-1,i,m)).$$

The last two expression can be built in additional depth $O(\log c) = O(\log\log n)$. The total depth stays $O(\log^k n)$. □

We see that the following questions can be answered with an additional depth of $O(\log n)$ not increasing the total depth of $O(\log^k n)$:

- Is $I_a = i$ in processor p after step t?
- Does p write into global cell i in step t?
- Is p the minimal processor writing into cell i at time t?
- Does p read from global cell i in step t?
- Does some processor write into global cell i in step t, and if so, which one?

The interested reader might wonder that we do not get any AC^k-hard problems in determining the control structure as we did in the characterization results with data-independence. The reason for this is that our model of an Index-PRAM doesn't have global index registers. Hence any communication has to be treated as data and cannot affect the control flow.

If in subsequent characterizations the Index-PRAM has to be enhanced by removing one or another restriction or by allowing some additional feature, we shall always explicitly indicate the deviations from the basis model. These deviations will affect data-independence of the communication structure, while the control structure CS_a will always stay data-independent with a word problem in NC^k.

5.2 Characterization results

As in the previous section, we start with a characterization of AC^k.

Theorem 16 *For $k \ge 1$, we have $L \in AC^k$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log^k n)$ that is additionally equipped with the instruction "if $L_c > 0$ then $G_{I_a} := L_b$."*

Proof. "if": This direction is clear, because a CRCW-PRAM can trivially simulate an Index-CRCW-PRAM. The characterization of AC^k by CRCW-PRAM's [59] now yields the desired result.

"only if": Due to Stockmeyer and Vishkin [59] we can assume that the AC^k-circuit to be simulated is $DTIME(\log n)$-uniform.

Two things have to be done. First, by simulating the uniformity machine we set up a pointer structure in global memory representing the circuit to be simulated. To do so, each participating processor interprets its PIN as a pair (g, g') of gates and simulates in logarithmic time the uniformity machine to check whether g' is a direct predecessor of gate g and to determine the type of gate g. Observe that the successors of TM-configurations (represented by the PIN's of processors) can be computed with NC^0-operations [8, Volume I, pages 110–115].

Second, we simulate the circuit making use of the pointer structure. The simulation of the AC^k-circuit represented by a pointer structure now works in the well-known way [59]. Each wire between two gates gets a processor. The processor $\log^k n$ times reads the value from the source gate and executes a conditional write depending on the value read and the type of the sink gate. □

Analogously, a characterization of NC^k by Index-PRAM's can be given.

Theorem 17 *For $k \geq 2$, we have $L \in NC^k$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log^k n)$ that additionally is equipped with the* data-conditional assignment *"if $L_c > 0$ then $L_a := L_b$."*

Proof. "if": As in the proof of Theorem 6, we will construct a circuit family $C = (C_n)_{n \geq 1}$ recognizing $L(A)$. Again the uniformity of the construction will be done by a uniformity circuit family U of depth $O(\log^k n)$ and again the simulation of the PRAM by an NC-circuit works with the recursive functions *Global* and $Local_a$. We assign to each of these functions a bunch of logarithmically many gates that represent the values of $Global(t, i)$ and $Local_a(t, p)$. The main work is done by the uniformity circuit U in computing the interconnection structure between gates. Observe that the possibility to use the data-conditional assignment doesn't have consequences for the control or communication structure since goto statements and memory access are controlled by index registers. Hence Lemma 15 still applies. With the help of the information contained in CS_A, RS_A, and WS_A, the structure of C becomes trivial:

Computation of $Global(t, i)$:

First, uniformity circuit U checks whether some processor writes into cell i in step t. If not, $Global(t, i)$ is connected to $Global(t-1, i)$. Otherwise, U determines the processor p which writes into i and determines the statement S_μ executed by p in step t. Statement S_μ must be of the form "$G_{I_a} := L_b$" (cf. instruction set of Index-PRAM's). In this case, U connects $Global(t, i)$ with $Local_b(t-1, p)$.

Computation of $Local_a(t, p)$:

First, U determines the statement S_μ executed by p in step t. We have to distinguish the following cases:

1. $S_\mu \equiv$ "$L_a := G_{I_b}$": Here U determines the value of $I_b = j$ and connects $Local_a(t, p)$ with $Global(t-1, j)$.

2. S_μ is neither a conditional nor an unconditional assignment to local cell L_a. In this case, U connects $Local_a(t, p)$ with $Local_a(t-1, p)$.

3. All the other cases, i.e. (conditional) assignments to L_a with the exception of a global read, are completely analogous to their corresponding counterparts in Theorem 6.

In total, we get an NC^k-circuit family $C = (C_n)_{n\geq 1}$ with a direct connection language in NC^k. Applying Lemma 1 we get $L(A) \in NC^k$.

"only if": We will now describe a PRAM A simulating a circuit family $C = (C_n)_{n\geq 1}$ of size $p(n)$. The idea of A is of course to repeat $O(\log^k n)$ times a loop in which for each gate g a processor takes the values of the left and of the right predecessor of g, combines these values according to the type of g, and writes the result in global cell G_g. The problem is that these values are data-dependent and that we have only monadic operations for the manipulation of data. Here, and only here, we use the conditional assignments. If L_b and L_c hold boolean values, then the assignment $L_a := L_b \vee L_c$ can be expressed by the two statements $L_a := L_b$; *if* $L_c > 0$ *then* $L_a :=$ *true* and, dually, the statement $L_a := L_b \wedge L_c$ is equivalent to $L_a :=$ *false*; *if* $L_c > 0$ *then* $L_a := L_b$.

The structure of A is now as follows: in parallel, each processor P_p interprets and decodes its PIN as a triplet of three gate names $1 \leq g, g', g'' \leq p(n)$ and puts these in its registers I_a, I_b, and I_c. Then, P_p simulates the uniformity machine of C and computes in logarithmic time the type of gate g. If g is an unnegated input, P_p stops. If it is a negated input, it writes $1-$ the content of G_g into G_{2g}. If g is an inner gate, P_p puts the type information in index register I_d, where $I_d = 0$ stands for an $\vee$-gate and $I_d = 1$ for an $\wedge$-gate. Then P_p checks in time $O(\log n)$ whether $\langle 1^n, g, 0, g'\rangle$ and $\langle 1^n, g, 1, g''\rangle$ are members of $L_{EC}(C)$. In case at least one of these questions is answered negatively, P_p stops. Otherwise, we know that g' and g'' are the two predecessors of gate g. After these local steps without communication, the "surviving" processors perform $O(\log^k n)$ times the following loop:

1. $L_b := G_{I_b}$;
2. $L_c := G_{I_c}$;
3. $L_a := L_b \vee L_c$;
4. *if* $I_d > 0$ *then* $L_a := L_b \wedge L_c$;
5. $G_{I_a} := L_a$

After the execution of this program cell $G_{p(n)}$ contains the bit output by C_n. $\square$

Remark 18 For the case $k = 1$, these constructions give us that Index-PRAM's augmented with a data-conditional assignment in logarithmic time recognize *all* languages in NC^1. On the other hand, Index-PRAM's recognize only languages in weakly uniform NC^1, i.e.: they can be simulated by NC^1-circuits with direct connection language in NC^1

Compare Theorem 16 to Theorem 17. The only difference between AC^k and NC^k within the framework of Index-PRAM's is that for AC^k we are allowed to use *conditional global* write instructions of the form "*if* $L_c > 0$ *then* $G_{I_a} := L_b$," whereas for NC^k we only have "*if* $L_c > 0$ *then* $L_a := L_b$."

We proceed with a characterization of $DSPACE(\log n)$. In order to do this, it is necessary to relax the fundamental concept of Index-CRCW-PRAM's. Up to now, no data registers were allowed as indexing registers. Now we loosen this by admitting that global reads may be data-dependent, that is we additionally have an instruction of the form "$L_a := G_{L_b}$" instead of only "$L_a := G_{I_b}$."

Theorem 19 *$L \in DSPACE(\log n)$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log n)$ that additionally is equipped with the instruction "$L_a := G_{L_b}$."*

Proof. Let A denote the Index-PRAM and M the $DSPACE(\log n)$-TM.

"if": The possibility to use local registers to control global reads affects neither the control nor the write structure. That is, Lemma 15 still gives us that CS_A and WS_A are in weakly uniform NC^1 and hence in $DSPACE(\log n)$. Also the value of the index registers are computable in $DSPACE(\log n)$. Since we have no data-conditional assignment of the form "*if* $L_c \cdot\!> \ 0$ *then* $L_a := L_b$," all conditional statements are controlled by index registers. That is also why the execution structure ES_A is in $DSPACE(\log n)$. Using Remark 10b) we can apply Theorem 9 which gives $L(A) \in DSPACE(\log n)$.

"only if": This direction is nearly identical to the corresponding part of Theorem 9. The point is to be careful to distinguish between local registers and index registers. The only subtlety is to simulate the conditional assignment "*if* $G_{L_a} \cdot\!> \ 0$ *then* $L_b := Next_1(PIN)$" which was used to set up in dependence of the input the correct successor of a configuration. To do this, we compute the two possible successors into local registers L_a and L_b and put the index of the input bit in index register I_c. Then the statement "*if* $\langle Input\text{-}Bit(I_c)\rangle$ *then* $L_a := L_b$" does the job. □

Relaxing some of the restrictions in the characterization of $DSPACE(\log n)$, we get a result corresponding to Theorem 11. For $k = 1$, it yields a characterization of $LOGDCFL$ in terms of Index-PRAM's. We state this result without proof, which is just an "index version" of the proof of Theorem 11.

Theorem 20 *L is accepted by a CROW-PRAM in time $O(\log^k n)$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log^k n)$ that additionally is equipped with the instructions "$L_a := G_{L_b}$" and "if $L_c \cdot\!> \ 0$ then $L_a := L_b$" and a standard PRAM operation set with binary operations.*

Again the difference between CROW-PRAM's and PPM's lies in the operation set used, as Lam and Ruzzo [40] already demonstrated. Again we state this result without proof which works basically in the same way as the proof of Theorem 13.

Theorem 21 *For $k \geq 1$, we have $L \in PPM\text{-}TIME(\log^k n)$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log^k n)$ that additionally is equipped with the instructions "$L_a := G_{L_b}$" and "if $L_c \gtrdot \ 0$ then $L_a := L_b$."*

Remark 22 We see, the only difference between $PPM\text{-}TIME(\log n)$ and $DSPACE(\log n)$ with respect to the Index-PRAM characterization is that for the first we may use conditional instructions of the form "*if* $L_c \gtrdot \ 0$ *then* $L_a := L_b$," whereas for the latter we may only use "*if* $\langle Input\text{-}Bit\rangle$ *then* $L_a := L_b$."

We conclude this section with an Index-PRAM characterization of the semi-unbounded fan-in circuit class SAC^k. This parallels the structural characterization given in Theorem 14. Again we use monotonicity and Akl's OR-PRAM's [4], resulting in a natural way in monotonic Index-OR-PRAM's.

Theorem 23 *For $k \geq 1$, we have $L \in SAC^k$ if and only if L is recognized by a monotonic Index-OR-PRAM in time $O(\log^k n)$ that additionally is equipped with the instruction "if $L_c \gtrdot \ 0$ then $L_a := L_b$."*

The proof of Theorem 23 is a straightforward combination of the arguments in the proofs of Theorem 14 and Theorem 17 and is therefore omitted.

6 Conclusion

We investigated the concept of data-independence from the complexity theoretic viewpoint. This notion provides a unifying framework relating many sequential and parallel models. It allows the grouping of various complexity classes into three levels:

Full data-independence characterizes parallel models of bounded fan-in.

Data-dependent read access joins several sequential and parallel classes.

Data-dependent write access characterizes parallel models of unbounded fan-in.

Apparently, the class OROW-TIME($\log^k n$) is missing among the second level. Both OROW-PRAM's and PPM's are restrictions of CROW-PRAM's. The first results are achieved by forbidding concurrent read access, the second by forbidding binary operations. Aside from the search for an Index-PRAM characterizing OROW-PRAM's, this also poses the question whether OROW-PRAM's without binary operations characterize $DSPACE(\log n)$.

One might ask for the place of the exclusive write concept in this picture. Results in [41, 47] showed closest relations to unambiguous computations. In particular, classes defined by exclusive write PRAM's can be regarded as *promise classes*, i.e.: the exclusive write access can only be guaranteed if the input fulfills some property which cannot be checked by the PRAM itself without breaking the exclusive write. A consequence is the apparent lack of complete problems for exclusive write classes. Thus it is no surprise that the concept of exclusive write has not been captured in the framework of data-independence.

Data-independence of parallel algorithms appears to be a fundamental prerequisite for an efficient implementation on existing distributed memory machines which may be closer to parallelism of bounded fan-in. Data-independence of communication and control gives the opportunity to optimize parallel algorithms with respect to their communication pattern at compile time. Let us close with a discussion of the complexity-theoretical relevance of data-(in)dependence. As a parallel analogue of the fruitful notion of NP-completeness and its opposition to P-membership, parallel complexity theory offers the opposition of P-completeness to NC-membership. The former as a demonstration of a problem being inherently sequential, and the latter as proof of a problem being efficiently parallelizable. But in reality not all NC-algorithms are efficient [39] and there are P-complete problems that are in a very intuitive sense *efficiently* parallelizable [68].

The main reason for this problem lies in the fact that all notions of reducibility used so far allow for a polynomial growth of the output [39]. Hence, the resulting complexity classes are closed under "polynomially bounded padding." But in order to be able to distinguish, for example, between a quadratic and a cubic resource bound or to work with an appropriate notion of speedup and work load, we would need reducibilities that are based on a linear or quasilinear growth of the output length [30, 46, 56]. We remark here that nearly all results obtained in this paper rely heavily on this freedom for polynomial growth.

A very different question is that of choosing an appropriate machine model. Our results give further evidence that in current parallel complexity theory both the machine model to define classes (e.g., PRAM's and circuits of unbounded fan-in) and the machine model to define reducibilities (e.g., space-bounded Turing machines) are not appropriate. The comparison of Theorem 6 with Theorem 9 specifically shows that $DSPACE(\log n)$ reductions spoil the communication structure: The current notions of reducibility are based on sequential machines and thus by Theorem 9 are burdened with a data-dependent read structure. Hence they cannot distinguish between data-dependence and data-independence of communication structures. In particular, it is possible to reduce a data-dependent computation to a data-independent one. This defect doesn't matter when working with PRAM's or circuits of unbounded fan-in, but should bother when working with more realistic models like distributed memory machines. That underpins for the field of efficient parallel computation the importance of the development of reducibility notions that are finer than $DSPACE(\log n)$ reductions. As a consequence of our work, these reducibilities should not only have (quasi)linear output length, but should, in addition, be based on Index-PRAM's or circuits of bounded fan-in.

Our results also shed some light on the classification of PRAM's according to their read and write access to global memory. In Subsection 2.2 we gave the current classification scheme for PRAM's and presented the OROW-PRAM as the weakest model. Following the same argumentation as before, Rossmanith's inclusion $DSPACE(\log n) \subseteq$ $OROW\text{-}TIME(\log n)$ [54] expresses the inadequacy of the XRYW classification scheme of PRAM's as a criterion with respect to implementability on existing parallel machines. This is due to the observation that even $DSPACE(\log n)$ seems to be too powerful for a simulation by fully data-independent PRAM's in logarithmic time. With the presence of a concrete machine model like the Index-PRAM, the possibility arises to develop algorithms that are efficiently implementable on existing and foreseeable parallel machines.

In Table 1 we summarize the main results of our work. The purpose of this table is to highlight the main differences between various complexity classes within our framework of data-independence and Index-PRAM's.

One direction for further research emerging from our work is to investigate far more combinations of the restrictions applicable to PRAM's. It would be interesting to find further classification criteria other than data-independence and monotonicity that play an important rôle for the transfer of PRAM algorithms to realistic parallel machines. Among the many ideas in this direction we refer the reader to the papers [2, 3, 12, 19, 32, 43, 58, 65] and many others. A matter of special interest could be to analyze and classify from a complexity theoretic point of view various degrees of synchronization that are necessary to implement parallel algorithms in a distributed environment, i.e., in a concurrent system without a global clock as it is still present in the Index-PRAM.

Acknowledgment Thanks to Carsten Damm and Markus Holzer for stimulating discussions and helpful comments. In particular, we are grateful to Peter Rossmanith for useful suggestions improving presentation and results and to a referee of *Journal of Computer and System Sciences* for insightful remarks and proposals concerning the organization of the introduction and the treatment of uniformity issues in the paper.

Table 1: Structural characterizations: The PRAM type refers to the definition of simple PRAM's in Section 2; we always assume a data-independent control structure contained in $DTIME(\log n)$; a letter "I" means that the corresponding structure is data-independent and contained in $DTIME(\log n)$; in the other case, "D" stands for data-dependence.

Class	Structural Characterizations PRAM type	RS	WS	Index-PRAM characterizations with deviations from the basis model
NC^k	simple	I	I	"*if* $L_c > 0$ *then* $L_a := L_b$"
$DSPACE(\log n)$	simple[a]	D	I	"$L_a := G_{L_b}$"
PPM^k	simple	D	I	"*if* $L_c > 0$ *then* $L_a := L_b$" and "$L_a := G_{L_b}$"
$LOGDCFL$	full	D	I	"*if* $L_c > 0$ *then* $L_a := L_b$," "$L_a := G_{L_b}$," and binary $DSPACE(\log n)$-operations
SAC^k	monotonic + OR-write	I	I	"*if* $L_c > 0$ *then* $L_a := L_b$," monotonic, and OR-write
AC^k	simple	I	D	"*if* $L_c > 0$ *then* $G_{I_a} := L_b$"

[a]Here, in contrast to all other cases, we also have to require that the execution structure ES is contained in $DTIME(\log n)$.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the physical design of PRAMs. *The Computer Journal*, 36(8):756–762, 1993.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Comput. Sci.*, 71:3–28, 1990.

[4] S. G. Akl. On the power of concurrent memory access. *Computing and Information*, pages 49–55, 1989.

[5] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 2d edition, 1994.

[6] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991.

[7] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory II*. Springer, 1990.

[8] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory I*. Springer, 2d edition, 1995.

[9] G. Bilardi and F. P. Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27:172–182, 1995.

[10] D. P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, 1994.

[11] S. R. Buss. The boolean formula problem is in ALOGTIME. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 123–131, 1987.

[12] A. Chin. Complexity models for all-purpose parallel computation. In Gibbons and Spirakis [27], chapter 14, pages 393–404.

[13] R. Cole and U. Vishkin. Approximate parallel scheduling. part i: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17(1):128–142, 1988.

[14] S. A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18:4–18, 1971.

[15] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *Enseign. Math.*, 27:99–124, 1981.

[16] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[17] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1):87–97, 1986.

[18] S. A. Cook and P. W. Dymond. Parallel pointer machines. *Computational Complexity*, 3:19–30, 1993.

[19] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[20] C. Damm, M. Holzer, and P. Rossmanith. Expressing uniformity via oracles. *Theory of Computing Systems*, 30:355–366, 1997.

[21] P. de la Torre and C. P. Kruskal. Towards a single model of efficient computation in real parallel machines. *Future Generation Computer Systems*, 8:395–408, 1992.

[22] P. W. Dymond and S. A. Cook. Hardware complexity and parallel computation. In *Proceedings of the 21st IEEE Conference on Foundations of Computer Science*, pages 360–372, 1980.

[23] P. W. Dymond, F. E. Fich, N. Nishimura, P. Ragde, and W. L. Ruzzo. Pointers versus arithmetic in PRAMs. *J. Comput. Syst. Sci.*, 53:218–232, 1996.

[24] P. W. Dymond and W. L. Ruzzo. Parallel RAMs with owned global memory and deterministic language recognition. In *Proceedings of the 13th International Conference on Automata, Languages, and Programming*, number 226 in Lecture Notes in Computer Science, pages 95–104. Springer, 1986.

[25] H. Fernau, K.-J. Lange, and K. Reinhardt. Advocating ownership. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1180 in Lecture Notes in Computer Science, pages 286–297, India, Dec. 1996. Springer.

[26] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[27] A. Gibbons and P. Spirakis, editors. *Lectures on parallel computation.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1993.

[28] L. M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 89–94. ACM, 1978.

[29] D. Gomm, M. Heckner, K.-J. Lange, and G. Riedle. On the design of parallel programs for machines with distributed memory. In A. Bode, editor, *Proceedings of the 2d European Conference on Distributed Memory Computing*, number 487 in Lecture Notes in Computer Science, pages 381–391, Munich, Federal Republic of Germany, Apr. 1991. Springer.

[30] E. Grandjean. Linear time algorithms and NP-complete problems. In *6th Workshop on Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 248–273. Springer, 1992.

[31] T. J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–206, 1994.

[32] T. Heywood and C. Leopold. Models of parallelism. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, chapter 1, pages 1–16. Oxford University Press, 1995.

[33] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation: I. The model. *Journal of Parallel and Distributed Computing*, 16:212–232, November 1992.

[34] J.-W. Hong. On similarity and duality of computation. *Information and Control*, 62:109–128, 1984.

[35] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[36] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[37] D. S. Johnson. A catalog of complexity classes. In van Leeuwen [63], chapter 2, pages 67–161.

[38] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In van Leeuwen [63], chapter 17, pages 869–941.

[39] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Comput. Sci.*, 71:95–132, 1990.

[40] T. W. Lam and W. L. Ruzzo. The power of parallel pointer manipulation. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 92–102, 1989.

[41] K.-J. Lange. Unambiguity of circuits. *Theoretical Comput. Sci.*, 107:77–94, 1993.

[42] K.-J. Lange and P. Rossmanith. Unambiguous polynomial hierarchies and exponential size. In *Proceedings of the 9th IEEE Symposium on Structure in Complexity*, pages 106–115, 1994.

[43] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[44] T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, 1992.

[45] W. F. McColl. General purpose parallel computing. In Gibbons and Spirakis [27], chapter 13, pages 337–391.

[46] A. V. Naik, K. W. Regan, and D. Sivakumar. Quasilinear time complexity theory. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings of the 11th Symposium on Theoretical Aspects of Computer Science*, number 775 in Lecture Notes in Computer Science, pages 97–108. Springer, 1994.

[47] R. Niedermeier and P. Rossmanith. Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits. *Information and Computation*, 118(2):227–245, May 1995.

[48] I. Niepel and P. Rossmanith. Uniform circuits and exclusive read PRAMs. In S. Biswas and K. V. Nori, editors, *Proceedings of the 11th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 560 in Lecture Notes in Computer Science, pages 307–318, New Delhi, India, Dec. 1991. Springer.

[49] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[50] I. Parberry. *Parallel Complexity Theory.* Pitman, 1987.

[51] A. G. Ranade. How to emulate shared memory. *J. Comput. Syst. Sci.*, 42:307–326, 1991.

[52] K. W. Regan. A new parallel vector model, with exact characterization of NC^k. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings of the 11th Symposium on Theoretical Aspects of Computer Science*, number 775 in Lecture Notes in Computer Science, pages 289–300. Springer, 1994.

[53] J. H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[54] P. Rossmanith. The owner concept for PRAMs. In C. Choffrut and M. Jantzen, editors, *Proceedings of the 8th Symposium on Theoretical Aspects of Computer Science*, number 480 in Lecture Notes in Computer Science, pages 172–183, Hamburg, Federal Republic of Germany, Feb. 1991. Springer.

[55] W. L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22:365–383, 1981.

[56] C. P. Schnorr. Satisfiability is quasilinear complete in NQL. *J. ACM*, 25:136–145, 1978.

[57] H. J. Siegel, S. Abraham, W. L. Bain, K. E. Batcher, T. L. Casavant, D. DeGroot, J. B. Dennis, D. C. Douglas, T.-Y. Feng, J. R. Goodman, A. Huang, H. F. Jordan, J. R. Jump, Y. N. Patt, A. J. Smith, J. E. Smith, L. Snyder, H. S. Stone, R. Tuck, and B. W. Wah. Report on the Purdue Workshop on Grand Challenges in computer architecture for the support of high performance computing. *J. Parallel Distrib. Comput.*, 16:199–211, 1992.

[58] M. Snir. Scalable parallel computers and scalable parallel codes: From theory to practice. In F. Meyer auf der Heide, B. Monien, and A. L. Rosenberg, editors, *Proceedings of the 1st Heinz Nixdorf Symposium on Parallel Architectures and Their Efficient Use*, number 678 in Lecture Notes in Computer Science, pages 176–184, Paderborn, Federal Republic of Germany, 1993. Springer.

[59] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, May 1984.

[60] I. H. Sudborough. On the tape complexity of deterministic context-free languages. *J. ACM*, 25:405–414, 1978.

[61] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[62] L. G. Valiant. General purpose parallel architectures. In van Leeuwen [63], chapter 18, pages 943–971.

[63] J. van Leeuwen, editor. *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[64] H. Venkateswaran. Properties that characterize LOGCFL. *J. Comput. Syst. Sci.*, 43:380–404, 1991.

[65] U. Vishkin. Workshop on "Suggesting computer science agenda(s) for high-performance computing" (Preliminary announcement). Announced via electronic mail on "TheoryNet", January 1994.

[66] U. Vishkin and A. Wigderson. Dynamic parallel memories. *Information and Control*, 56:174–182, 1983.

[67] P. Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM J. Comput.*, 17(4):659–672, August 1988.

[68] J. S. Vitter and R. A. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Trans. Comp.*, C-35(5):403–418, 1986.

[69] K. W. Wagner and G. Wechsung. *Computational complexity*. Reidel Verlag, Dordrecht and VEB Deutscher Verlag der Wissenschaften, Berlin, 1986.