Data-Independences of Parallel Random Access Machines[*][†]

Klaus-Jörn Lange and Rolf Niedermeier

Fakultät für Informatik, Technische Universität München,
Arcisstr. 21, D-80290 München, Fed. Rep. of Germany

April 29, 1994

[*]This research was supported by the Deutsche Forschungsgemeinschaft, SFB 342, Teilprojekt A4 "KLARA." The work of the first author was also supported by the International Computer Science Institute.

[†]An extended abstract 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.

Proposed running title: Data-Independences of PRAM's

Address for correspondence:

Rolf Niedermeier
Technische Universität München
Fakultät für Informatik
Arcisstr. 21
D-80290 München
Fed. Rep. of Germany

Email: niedermr@informatik.tu-muenchen.de

Abstract

We introduce the notions of control and communication structures for PRAM's 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. 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 — in some sense they 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. Until recently the PRAM has been viewed as a purely theoretical model far from realization. To bridge the seemingly large gap between theory (PRAM's) and practice (bounded degree networks with distributed memory) there are basically two main approaches. One deals with the realization of full general purpose machines [26, 40] on bounded degree networks, using techniques like hashing and slackness [1, 2, 45, 54], having to fight against problems like hardware costs, scalableness, and interconnect length [17, 59]. The other approach is to restrict the PRAM models in use [10, 27], that is, to consider restricted forms of PRAM's [3, 4, 28] or to propagate even weaker models [17]. The idea to consider restricted models is additionally supported by the observation that many of the current tasks requiring a large amount of computational power (e.g., most of the Grand Challenges [39, 50]) seem to be of a simple nature without a need for the general purpose overhead that is necessary to simulate a full PRAM. 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.

This paper is devoted to the investigation of restricting concepts relevant for an efficient realization of PRAM algorithms on existing distributed memory machines (thus, we follow the second approach described above) *and* to convert them into a formal notion. So we study the concept of data-independence of communication and control and make use of it to obtain *sub*classes of conventional PRAM complexity classes. We do not do this by introducing new models and classes, but by applying our formal notion to existing PRAM classes. The surprising fact is that these restrictions again lead to well-known classes of complexity theory. In particular, this enables the comparison of these classes with respect to various PRAM restrictions to be applied.

A communication and control structure *independent of the input* (other authors, especially in automata theory, also use the term *oblivious*) is an important criterion for efficient realization on distributed memory machines [24, 34, 51, 58]. Here by data-independence we mean independence of the concrete input word except for its length. Let us consider graph problems as an illustration: There are two simple representations of graphs. One are adjacency matrices and the other one are 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.

We study data-independence from a complexity theoretical point of view. We distinguish between the two aspects reading and writing of a communication structure and introduce data-independence for control flow such that we can separate three aspects of dynamic, but input-independent behavior: Data-independence of control means that the statement executed by a processor of a PRAM only depends on time, processor identification number (PID 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 only depend on time, PID, and input length.

The main result of this work is the characterization of several complexity classes within the unified framework of data-independence: 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 [13, 16] operating in logarithmic time. Eventually, we discuss the power of Akl's concurrent write OR-feature for PRAM's [5] and obtain a characterization of $LOGCFL$ and, more generally, of Venkateswaran's semi-unbounded fan-in circuit classes SAC^k [56] by monotonic, fully data-independent OR-PRAM's.

To summarize, the gist of our work is that with respect to accessing global shared memory of PRAM's,

1. unbounded fan-in parallelism corresponds to data-dependent writes,
2. bounded fan-in parallelism corresponds to data-independent reads and writes, and
3. sequential computations correspond to data-dependent reads and data-independent writes.

We obtain the above results not only for PRAM's with structurally restricted control and communication structures, but also for some new type of PRAM's that are "data-independent by construction." The basic idea is to consider PRAM's where the indexing of global memory cells only is possible through special local index registers. The value of index registers only depends on time, PID, and input length, but not the actual input data. In such a way, we present a concrete machine model, called *Index-PRAM,* that offers the prospect of developing algorithms that are efficiently implementable on existing distributed memory machines. In addition, within the framework of Index-PRAM's it is possible to study the essential differences between various complexity classes with respect to the instruction set used by the underlying Index-PRAM. For example, it will turn out that the fundamental difference between $DSPACE(\log n)$ and parallel pointer machines [16] operating in logarithmic time is that for the first only a restricted form of conditional assignments may be used — the condition may only depend 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 the notion of structural data-independence of communication and control. This enables the characterizations discussed above. In the fourth section we define Index-PRAM's. We give characterizations by Index-PRAM's that parallel those of the third section. Finally, we conclude this paper with a discussion of the main benefits of our work with respect to a more practical parallel complexity theory and some directions for future research.

2 Preliminaries

We assume familiarity with the basic concepts and notations of computational complexity theory [8, 30, 32, 43, 61]. By $DSPACE(\log n)$ ($DTIME(\log n)$) we denote the class of languages accepted by deterministic Turing machines whose working space (resp. running time) is bounded by $\log n$. 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 shortly review some concepts and facts of parallel complexity theory. For more details we refer to the literature [14, 22, 31, 33, 44].

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 set $\{C_n | n \geq 0\}$ of circuits, where C_n has exactly n inputs. Family C has *polynomial size* if for some polynomial $p(\cdot)$, the size of each C_n is bounded by $p(n)$. Similarly, the depth of C is *bounded by* $log^k n$ if for some constant $c > 0$ the depth of each C_n is less than $c \cdot log^k n$. If for some constant integer 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 fortunately turned out to be equivalent in most cases [48].

For the uniformity notion used in this paper we have to shortly introduce *alternating Turing machines* (ATM's) [9]. ATM's are a generalization of nondeterministic TM's: states are partitioned into "existential" and "universal" states. *All* computation paths starting in a universal configuration paths have to lead to an accepting configuration. Existential configurations are the same as we know for NTM's.

Throughout the paper we use the notion of U_{E^*}-uniformity for circuits [48]. A circuit family of size $z(n)$ and depth $t(n)$ is called U_{E^*}-uniform if there is an ATM M recognizing the extended connection language L_{EC} in time $O(t(n))$ and space $\log(z(n))$. Herein, M is called the *uniformity machine* and L_{EC} essentially describes the interconnection structure of two gates of the circuit each time. The details, which are unimportant for this paper, can be found in Ruzzo's work [48]. If not stated otherwise, we always assume that L_{EC} can even be recognized by a logarithmically time bounded ATM, that is, in $ATIME(\log n)$. The main thing of importance for us is that the uniformity machine only is provided with the length of the input word, and not the concrete input word itself. Thus for fixed input length n always one particular circuit is constructed.

Clearly, each $ATIME(\log n)$-uniform circuit is also $DSPACE(\log n)$-uniform, because $ATIME(\log n) \subseteq DSPACE(\log n)$. The classes NC^k (AC^k, respectively) denote the families of languages recognizable by $ATIME(\log n)$-uniform, polynomial sized, $O(\log^k n)$-depth bounded circuit families of bounded (unbounded, respectively) fan-in. Recently, Venkateswaran [56] 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 [32, 33, 56].

2.2 Parallel Random Access Machines

A 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 whether it performs a local or a global (i.e., remote) operation. We assume the standard definition of PRAM's [31, 33]. All processors execute in parallel the same sequence of statements S_1, S_2, ... , S_k, which is independent of the input. In fact, allowing conditional jumps for PRAM's only guarantees a single program, multiple data modus instead of the single instruction, multiple data modus [6, pages 111–112,466] we are assuming here. But due to the constant program size of the PRAM it is easy to always achieve the single instruction, multiple data modus. For the ease of presentation, we assume throughout the paper that each processor only 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, ..., L_D, and let G_1, G_2, ..., $G_{q(n)}$ 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, and *NOOP* means "no operation.")

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*.

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 demand that they to 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 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 an *arbitrary* writer is actually stored (ARBITRARY-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 [20]. More common than the owner concept in formulating algorithms is exclusive access, where we only demand 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 [20, 37, 52], the concept of exclusiveness corresponds to unambiguity [36, 37, 42], which explains the unconstructive 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.

We denote the class of languages recognizable in time $O(f)$ by $XRYW$-PRAM's with a polynomial number of processors by $XRYW\text{-}TIME(f(n))$. For $XRYW\text{-}TIME(\log^k n)$ we shortly write $XRYW^k$. We know the relationships (for $k \geq 1$)

$$CRCW^k = AC^k \text{ [52]},$$

$$NC^k \subseteq OROW^k \subseteq CROW^k \subseteq SAC^k \text{ [47, 56]},$$
$$CROW^1 = LOGDCFL \text{ [20], and}$$
$$DSPACE(\log n) \subseteq OROW^1 \text{ [47].}$$

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 is 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 [13] and are studied in several papers [16, 18, 19, 29, 35]. In earlier papers [13, 18, 29, 35] 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 [35] is that they capture formally 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 [16, 18]. So PPM's are a parallel model less powerful than PRAM's and have the flavor to be a more realistic model for existing parallel computers than PRAM's are. By $\mathit{PPM\text{-}TIME}(t(n))$ 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 $\mathit{PPM\text{-}TIME}(\log^k n)$.

Lam and Ruzzo [35] showed that PPM's and an arithmetically restricted form of CROW-PRAM's (*rCROW'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, Fich, Nishimura, Ragde, and Ruzzo [19] demonstrated that any step-by-step simulation of a full n-processor CROW-PRAM by a PPM requires time $\Theta(\log\log n)$ per step. This strongly suggests a separation between CROW-PRAM's and PPM's and thus of $\mathit{LOGDCFL}$ and $\mathit{DSPACE}(\log n)$, because $\mathit{CROW\text{-}TIME}(\log n) = \mathit{LOGDCFL}$ [20] and $\mathit{DSPACE}(\log n) \subseteq \mathit{PPM}^1$ [16, 18].

2.4 Simple PRAM's

In this subsection we introduce two PRAM properties that restrict the power of PRAM's, leading to so-called simple PRAM's. Investigating Stockmeyer and Vishkin's [52] 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 a base for considerations in the next section.

Definition 1 We call a PRAM *simple* if its instruction set fulfills two restrictions:

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

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

A predicate "$L_a > 0$" in the previous instruction set is not NC^0-computable, since we would have to compute the AND of all bits of L_a. We overcome this problem by using hence-forward the predicate "L_a is odd", i.e., look whether the last bit of L_a is 1. We write this predicate "$L_a \gtrdot\ 0$". In essence, this restriction is only important for the characterization of NC^k ($k \geq 0$). In other cases as e.g. the characterization of $\mathit{DSPACE}(\log n)$ this requirement plays no rôle. Although the above restrictions are apparently very severe — neither addition nor incrementation could be done directly by such a simple PRAM in constant time — nevertheless they do not restrict decisively the power of CRCW-PRAM's with at least logarithmic running time.

Proposition 2 *For $k \geq 1$, we have $L \in AC^k$ if and only if L can be accepted by a simple CRCW-PRAM in time $O(\log^k n)$.*

Proof. The inclusion from right to left is obvious, since every simple CRCW-PRAM can trivially be simulated by a (non-simple) CRCW-PRAM. Thus the characterization of AC^k by CRCW-PRAM's [52] yields the desired result.

The idea of the proof for the reverse inclusion is to look at Stockmeyer and Vishkin's [52] simulation of an AC^k-circuit by a CRCW-PRAM. For this purpose, we have to take care of 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 [52]. 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 updates correspondingly the value of its sink. This can be done alone with global reads and writes and a conditional assignment.

It remains to be shown how a simple CRCW-PRAM can construct a description of the circuit in global memory. In order to set up a pointer structure representing an AC^k-circuit C, the simple CRCW-PRAM A has to simulate the uniformity machine B of C in time $O(\log^k n)$. This is done by interpreting the *PIN* (processor identification number) of a processor as a configuration of B. Then one makes use of the fact that the one-step-successors of a configuration of an alternating Turing machine can be computed by an NC^0-circuit [8, Volume I, pages 104–109] Thus it is possible to determine the successor configurations of a given configuration (i.e., the *PIN*'s of constantly many PRAM processors) with the monadic NC^0-operations a simple PRAM is equipped with. This enables A to compute for each configuration the uniquely determined processors representing the successor configurations. In this way, A constructs the configuration forest of B in constant time. Then A simply evaluates the configuration forest of B in logarithmic time making use of the evaluation technique used for circuits of bounded fan-in [33]: Associate processors with configurations. A processor reads sequentially in constant time all "inputs" of its configuration (like e.g. predecessor configurations or bits of the input word of the circuit).

In a messy but straightforward way, the simulation of a language recognizing ATM by the simple CRCW-PRAM can be generalized to the simulation of the uniformity machine. This eventually yields a pointer structure in global memory representing the AC^k-circuit. $\square$

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* $G_i \cdot\!\!> 0$ *then* $G_j := 1$", which can be simulated by "$L_a := G_i$; *if* $L_a \cdot\!\!> 0$ *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 also is 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. Altogether, this will lead to notions of data-(in)dependent control, read, and write structures, respectively. In the next section we will formally introduce these notions.

3 Data-Independences of control and communication structures

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 [24, 34, 51]. 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 [58] 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 [15] considered oblivious (i.e., data-independent) and semi-oblivious PRAM's in order to prove lower bounds.

3.1 Structural definition of data-independence

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.

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

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 $(*)$.

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

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 $1 \leq l \leq k$:

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

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

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:

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

c) A structure XS_A, $X \in \{C, E, R, W, SW\}$ 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

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

When we speak of *communication structure* we address to 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*. There is a close connection between semi-write structures and what Cook, Dwork, and Reischuk [15] call semi-oblivious PRAM's. For semi-oblivious PRAM's also only whether or not a processor writes into a cell may 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 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 "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 [7, 11]. 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 a variation of Rossmanith's OROW-algorithm [47]. 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.

Above 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 subsection) as a consequence we had $NC^1 = \mathit{DSPACE}(\log n)$ and thus $\mathit{ATIME}(n) = \mathit{DSPACE}(n)$.

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 *PID* 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 already can simulate AC^k-circuits. This will become important when we later on consider PRAM's with OR write conflict resolution [5] instead of CRCW-PRAM's with ARBITRARY write conflict resolution.

3.2 Characterizing complexity classes by PRAM's with data-independent communication structures

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

Proposition 2 in Subsection 2.4 gives a characterization of AC^k by simple CRCW-PRAM's. Its proof reviews the inclusions $CRCW - TIME(\log^k n) \subseteq AC^k \subseteq CRCW - TIME(\log^k n)$ of Stockmeyer and Vishkin [52]. We now can easily classify AC^k making use of the new notions given in Definition 3.

Theorem 4 *For $k \geq 1$, we have $L \in AC^k$ if and only if L is 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, and CS_A, RS_A, and SWS_A are in $ATIME(\log n)$.*

Proof. In Proposition 2 we already showed that each language in AC^k can be recognized by a simple CRCW-PRAM in time $O(\log^k n)$. In addition, in the remarks following Proposition 2 we pointed out 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 was 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 circuit simulation we immediately have $CS_A, RS_A, SWS_A \in ATIME(\log n)$ for this part of the simulation. 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 an $ATIME(\log n)$-machine, we also have $CS_A, RS_A, SWS_A \in ATIME(\log n)$ 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. $\Box$

The next theorem yields the first characterization of NC^k in terms of PRAM's. Recently, Regan [46] gave another characterization of NC^k by a parallel vector model, using a quite different approach.

Theorem 5 *For $k \geq 1$,we have $L \in NC^k$ if and only if L is recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, the control, read, and write structures of which all are data-independent, and CS_A, RS_A, and WS_A are in $ATIME(\log n)$.*

Proof. "if": To simulate a PRAM by a circuit, we work with recursion constructions similar to those in several other papers [20, 21, 23, 36, 42]. 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.

In the circuit we will construct to each such function value we assign a bunch of logarithmically many gates that represent the value. The main work is hidden in the interconnection structure of the circuit and is done by the uniformity machine.

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 $\mathit{GLOBAL}(t,i)$:

To compute the interconnection structure for $\mathit{GLOBAL}(t,i)$-gates, the uniformity machine U determines in alternating, logarithmic time whether there is an element $\langle n,t,p,i\rangle$ in WS_A. This is done in the following way. First, U existentially guesses in logarithmic time the logarithmically many bits of a polynomially bounded value p. Since n, t, and i are already known to U, the question whether $\langle n,t,p,i\rangle \in WS_A$ holds can subsequently be answered by U in alternating, logarithmic time. If $\langle n,t,p,i\rangle \notin WS_A$, then each bit of $\mathit{GLOBAL}(t,i)$ is connected with $\mathit{GLOBAL}(t-1,i)$. If $\langle n,t,p,i\rangle \in WS_A$, then U determines the uniquely existing μ such that $\langle n,t,p,\mu\rangle \in CS_A$. Now U knows that statement S_μ executed by processor p at time t is either "$G_{L_a} := L_b$" or "*if* $L_c \dot{>} 0$ *then* $G_{L_a} := L_b$", where $L_c \dot{>} 0$ is *true*. In either case we connect $\mathit{GLOBAL}(t,i)$ with the gates representing $\mathit{LOCAL}_b(t-1,p)$.

Computation of $\mathit{LOCAL}_a(t,p)$:

To compute $\mathit{LOCAL}_a(t,p)$, we first let the uniformity machine U determine the uniquely existing μ such that $\langle n,t,p,\mu\rangle \in CS_A$. Let S_μ be the statement executed at time t by processor p. We have to consider several cases.

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

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

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

4. *Other cases*: If, for example, $S_\mu \equiv$ "$L_a := \mathit{PID}$," then U connects $\mathit{LOCAL}_a(t,p)$ to the binary coding of p. The other cases "$L_a := \mathit{LENGTH}$," "$L_a := \langle \mathit{constant}\rangle$," "$L_a := L_b$" are of similar simplicity. If S_μ is a statement where we have no assignment to L_a, then U connects $\mathit{LOCAL}_a(t,p)$ to $\mathit{LOCAL}_a(t-1,p)$.

So U constructs an $\mathit{ATIME}(\log n)$-uniform circuit of bounded fan-in of depth $O(\log^k n)$. Hence $L(A)$ is in NC^k.
"only if": Consider the proofs of Proposition 2 and Theorem 4. If the simulated circuit is of bounded fan-in, we can replace the statement *if* $G_i \dot{>} 0$ *then* $G_j := 1$ used there by a program part executed by a single processor attached with each gate, asking sequentially all inputs. This additionally results in a data-independent write structure. Observe that

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

uniformity machines (and thus their simulations) work data-independent. The inclusion of CS_A, RS_A, WS_A in $ATIME(\log n)$ works in the same way as in the proof of Theorem 4. $\square$

Let us shortly recapitulate what made 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 4, we can even demand for data-independent read and semi-write structures. For write instructions this meant that we used conditional assignments of the form "*if* $L_c > 0$ *then* $G_j := L_b$," where everything was data-independent except for the value of the condition. On the contrary, we can get by on without any conditional assignment if we have statements of the form $G_{L_a} := L_b$, where now the indexing value L_a is data-dependent: Assume that L_c only contains values 0 or 1. The basic idea is to simulate

$$\textit{if } L_c > 0 \textit{ then } G_j := L_b$$

by the two instructions

$$L_a := j + L_c - 1; G_{L_a} := L_b.$$

Note that if $L_c > 0$ is false, then a write into G_{j-1} occurs. This requires that G_{j-1} is a cell without importance for the computation.

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 only are enough for a characterization of $DSPACE(\log n)$. This may indicate that in parallel computations writing is more powerful than reading.

Theorem 6 *$L \in DSPACE(\log n)$ 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 $ES_A(w)$ (on input word w) are in $ATIME(\log n)$.*

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 5. 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 $\mathit{GLOBAL}(t, i)$:

To find out the value of $\mathit{GLOBAL}(t, i)$ in logarithmic space, M exhibits whether there exists a p such that $\langle n, t, p, i\rangle \in \mathit{WS}_A$. This is possible due to the well-known inclusion $\mathit{ATIME}(\log n) \subseteq \mathit{DSPACE}(\log n)$. If there is no such p, then M knows that no processor tried to write into G_i and M recursively computes $\mathit{GLOBAL}(t-1, i)$ by stacking a symbol "no write." Otherwise M computes the unique statement number μ such that $\langle n, t, p, \mu, v\rangle \in \mathit{ES}_A(w)$. 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 $L_c \gtrdot\ 0$ (resp. $v = \mathit{true}$). So M recursively computes $\mathit{LOCAL}_b(t-1, p)$, stacking the statement number μ.

Computation of $\mathit{LOCAL}_a(t, p)$:

To compute $\mathit{LOCAL}_a(t, p)$, M first determines an index μ such that $\langle n, t, p, \mu, v\rangle \in \mathit{ES}_A(w)$. 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 $\mathit{LOCAL}_b(t-1, p)$ and stacking index μ, which is marked as "undone." When M returns from the recursion with a result $j = \mathit{LOCAL}_b(t-1, p)$, 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 $\mathit{GLOBAL}(t-1, j)$ 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 $\mathit{ES}_A(w)$ 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 $\mathit{LOCAL}_b(t-1, p)$ and go into the recursion $\mathit{LOCAL}_a(t-1, p)$, otherwise. Thus no branching of the recursion occurs.

3. All the other cases can be led back to the above two ones or are handled similar easily as in the proof of Theorem 5.

"only if": This inclusion follows along the lines of the proof of Proposition 2 and Theorem 4: The configuration graph of a $\mathit{DSPACE}(\log n)$-machine is a forest of polynomial size. The reachability problem can be solved by pointer jumping — we could directly apply Example 1, part a), if we consider the Euler-tours generated by the configuration forest. The claim that we have data-independent control and write structures contained in $\mathit{ATIME}(\log n)$ follows in a way analogous to previous proofs. The essential point why the execution structure $\mathit{ES}_A(w)$ has to be data-dependent, but is in $\mathit{ATIME}(\log n)$, can be seen as follows. Since only monadic operations are allowed in the construction of the computation tree of a $\mathit{DSPACE}(\log n)$-machine, we need to have "input-conditional" assignments in order to compute the successor of a configuration. So we have a data-dependent execution structure. On the other hand, we still have $\mathit{ES}_A(w) \in \mathit{ATIME}(\log n)$ because of the simplicity of input-conditional local assignments. Note that these input conditional assignments are the only conditional assignments needed (also cf. Theorem 13 in Subsection 4.2). $\square$

Theorem 6 implies that if there was a completely data-independent algorithm for the $\mathit{DSPACE}(\log n)$-complete list ranking problem, then we had the equality of NC^1 and $\mathit{DSPACE}(\log n)$ and hence of $\mathit{ATIME}(n)$ and $\mathit{DSPACE}(n)$.

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 request that the operations of the CRCW-PRAM have to 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 led to the linear recursion structure. If we drop the request for monadic NC^0-computable operators and further on do not require $ES_A(w) \in ATIME(\log n)$, then we get $LOGDCFL$, the class of languages logspace many-one reducible to deterministic context-free languages [53].

Theorem 7 *$L \in LOGDCFL$ if and only if L is recognized by a CRCW-PRAM A with standard PRAM operation set*[2] *in $O(\log n)$ time, A has data-independent control and write structures, and CS_A and WS_A are in $ATIME(\log n)$.*

Proof. "if": We use the same recursion as in Theorem 6. But now we have to augment the $DSPACE(\log n)$ Turing machine with an auxiliary push-down store (thus yielding a so-called auxiliary pushdown automaton [12]), 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 polynomially bounded. By results of Sudborough [53] we get $L \in LOGDCFL$. Observe that now also the use of conditional assignments of the form "*if* $L_c > 0$ *then* $L_a := L_b$" does not require any more $ES_A(w) \in ATIME(\log n)$, because a branching of the recursion no longer needs to be avoided.
"only if": This direction follows from the equation $LOGDCFL = CROW-TIME(\log n)$ of Dymond and Ruzzo [20]. Since CROW-PRAM's w.l.o.g. have logarithmic word length, $DSPACE(\log n)$-computable operations, and write-owner functions restricted to be the identity function (that is, write-owner(i,n) = i), it remains to be shown that CS_A, $WS_A \in ATIME(\log n)$. An inspection of Dymond and Ruzzo's simulation of deterministic auxiliary pushdown automata by CROW-PRAM's reveals that the CROW-PRAM can be assumed to have a very regular flow of control. To see $WS_A \in ATIME(\log n)$, notice that CROW-PRAM instructions of the form "*if* $L_c > 0$ *then* $G_{L_a} := L_b$" can be replaced by the equivalent sequence of instructions "$L_d := G_{L_a}$; *if* $L_c > 0$ *then* $L_d := L_b$; $G_{L_a} := L_d$" avoiding any conditional writes to global memory cells. The sequence really is equivalent to the conditional global write due to the owner write restriction. The simplicity of the write-owner function and $CS_A \in ATIME(\log n)$ now yield $WS_A \in ATIME(\log n)$ in a straightforward way. □

Next, we come to the characterization of parallel pointer machines or, equivalently, rCROW-PRAM's [35]. Cook and Dymond [16, 18] showed the inclusion $DSPACE(\log n) \subseteq PPM\text{-}TIME(\log n)$. Dymond, Fich, Nishimura, Ragde, and Ruzzo [19] recently proved that any step-by-step simulation of CROW-PRAM's by PPM's needs time $\Theta(\log\log n)$. Thus we have some evidence for a separation of the corresponding complexity classes. In our setting the difference appears in the requirement for simple PRAM's (similar to Lam and Ruzzo [35]) in the case of PPM's, whereas for the CROW-PRAM's the operation set is unrestricted (compare Theorem 7 with Theorem 8).

[2]More precisely, an operation set as used for CROW-PRAM's by Dymond and Ruzzo [20] suffices.

Theorem 8 *For $k \geq 1$, we have $L \in$ PPM-TIME$(\log^k n)$ if and only if L is recognized by a simple CRCW-PRAM A in time $O(\log^k n)$, the control and write structures are data-independent, and CS_A and WS_A are in ATIME$(\log n)$.*

Proof. "if": For this direction we make decisive use of Lam and Ruzzo's [35] equality *PPM-TIME*$(\log^k n) =$ *rCROW-TIME*$(\log^k n)$. So 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 $WS_A \in$ *ATIME*$(\log n)$, 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 [35]. Each processor simulates one PPM unit, using a block of constant 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), but is basically straightforward. Further details can be found in Lam and Ruzzo's work [35].

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 *ATIME*$(\log n)$. To see that the simulating PRAM also has a data-independent write structure contained in *ATIME*$(\log n)$, 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 constant 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* $L_a \cdot\!> \ 0$ *then* $L_b := L_c$" in a basically straightforward manner. Lam and Ruzzo [35] 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. $\Box$

Theorem 6 and Theorem 8 show that the essential difference between $DSPACE(\log n)$ and $PPM\text{-}TIME(\log n)$ is that for the latter we need not demand for a data-dependent execution structure contained in $ATIME(\log n)$. Up to now only $DSPACE(\log n) \subseteq PPM\text{-}TIME(\log n) \subseteq DSPACE(\log^2 n)$ is known [16, 18].

Until now all our characterizations worked with CRCW-PRAM's using the ARBITRARY resolution protocol for write conflicts. Let us shortly consider an enhanced CRCW-PRAM model, Akl's OR-PRAM [5]. 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 4), the decisive, data-dependent write instruction was "*if* $G_i \dot{>} 0$ *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 an 1 is never overwritten by a 0. 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, in a monotonic algorithm it can be avoided to store data of the faster processor that have the time stamp the slower processor has currently reached. The slower processor simply can 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, never is 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 demand for monotonic OR-PRAM's, but allow data-dependent writes: It is clear how to simulate OR-gates, but what's 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 happens if we demand for 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 [56]. Venkateswaran [56] proved $SAC^1 = LOGCFL$, the class of languages logspace many-one reducible to context-free languages [53].

Theorem 9 *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 $O(\log^k n)$, the control, read, and write structures of which are data-independent and CS_A, RS_A, and WS_A are in $ATIME(\log n)$.*

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 Theorem 5, so we will only describe the changes compared with there.

The computation of $LOCAL_a(t,p)$ is the same as in Theorem 5. It is decisive here that A only uses monotonic operations, because in SAC^k-circuits negating gates are not admissible. The computation of $GLOBAL(t,i)$ is the same as in Theorem 5 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 4 and Theorem 5. 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 5 — 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 5. Note that Venkateswaran's SAC^k-circuits [56] are even $DTIME(\log n)$-uniform. Altogether, in each case monotonic instructions are sufficient. Data-independence follows in the same way as in Theorem 5. $\square$

4 Index-PRAM's

In the previous section we obtained our results by demanding 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 is that the lesser deviations from the basis model are necessary, the easier the implementation on parallel machines with distributed memory will be.

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

4.1 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 concrete input word. A usual instruction set for Index-PRAM's is shown below. Compare this one 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* $\langle INPUT\text{-}BIT \rangle$ *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: $L_a := f(L_b)$,

Binary Operations: $I_a := I_b \circ I_c$,

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 "$\langle INPUT\text{-}BIT \rangle$" in the above input conditional local assignment means that here we specify a position i in the input word w, where $1 \leq i \leq |w|$, and the condition is *true* iff the bit has a given value.

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

- As can be seen in the given instruction set, binary operations only are allowed for index registers, otherwise monadic operations are obligatory.
- Only NC^0-computable operations are admissible.
- The flow of control is regular: The statement executed at time t by all PRAM processors can be determined by a data-independent $ATIME(\log n)$-computation.
- Each processor only reads from and writes into a constant number of global memory cells.

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.

4.2 Characterization results

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

Theorem 10 *For $k \geq 1$, 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 additionally is 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 [52] now yields the desired result.
"only if": 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. Second,

we simulate the actual circuit making use of the pointer structure. Due to Stockmeyer and Vishkin [52], whose circuit construction for the simulation of CRCW-PRAM's is $\mathit{DTIME}(\log n)$-uniform, we can assume that the AC^k-circuit is $\mathit{DTIME}(\log n)$-uniform.

We simulate the uniformity machine locally in each processor of the PRAM such that after the simulation the index 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 an Index-PRAM can simulate in logarithmic time a $\mathit{DTIME}(\log n)$-TM making use of its constant many index registers of logarithmic word length, this first point follows. Observe that the successors of TM-configurations (represented by the PID's of processors) can be computed with NC^0-operations [8, Volume I, pages 104–109].

The simulation of the AC^k-circuit represented by a pointer structure now works in the well-known way [52]. 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.

It remains to be shown that we keep the restrictions required for Index-PRAM's. Except for the regular flow of control all other restrictions follow immediately. Observe that the actual circuit simulation only consists of the above described loop repeated $\log^k n$ times. The simulation of the uniformity machine in essence also just requires a simple loop repeated $\log n$ times. Thus the the static flow of control is obvious. $\square$

For the proof of the subsequent theorem the following result of Ruzzo [48] is necessary.

Proposition 11 [48] *For $k \geq 1$, $\mathit{DTIME}(\log n)$-uniform NC^k is equal to NC^k-uniform NC^k, where the latter is defined as $\mathit{ATIME}, \mathit{SPACE}(\log^k n, \log n)$-uniform NC^k.*

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

Theorem 12 *For $k \geq 1$, 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 instruction "if $L_c > 0$ then $L_a := L_b$."*

Proof. Let A denote the Index-PRAM, C the NC^k-circuit, and U its "uniformity circuit."
"if": As in the proof of Theorem 5, the simulation of the PRAM by a NC-circuit works with the recursive functions GLOBAL and LOCAL_a. We additionally have recursive functions INDEX_a with $\mathit{INDEX}_a(t,p) = j$ iff the local index register a of processor p after step t contains value j. We assign to each of these functions a bunch of logarithmically many gates that represent the values of $\mathit{GLOBAL}(t,i)$, $\mathit{LOCAL}_a(t,p)$, and $\mathit{INDEX}_a(t,p)$, respectively. The main work is done by the uniformity circuit U in computing the interconnection structure between gates. In what follows, we construct an NC^k-uniform NC^k-circuit. Application of Proposition 11 then provides the if-direction of the proof. To do the construction, we go through all possible instructions of our Index-CRCW-PRAM and show how to compute GLOBAL, LOCAL_a, and INDEX_a. Before we come to the details, we first describe a precomputation common to all three cases: For arbitrary point of time t the uniformity circuit U finds out the statement A is executing at time t. This is possible due to the regular control flow of the Index-PRAM. Let S_μ denote the statement executed at time t in all PRAM processors.

Computation of $\mathit{GLOBAL}(t, i)$:

The only way G_i may be affected at time t is when $S_\mu \equiv$ "$G_{I_a} := L_b$" (cf. instruction set of Index-PRAM's). In all other cases $\mathit{GLOBAL}(t, i)$ is connected to $\mathit{GLOBAL}(t-1, i)$. If $S_\mu \equiv$ "$G_{I_a} := L_b$," then $\mathit{GLOBAL}(t, i)$ is an OR-gate connected to two conjunctions. In the first conjunction, $\mathit{LOCAL}_b(t-1, p)$ is conjoined with a gate computing $\mathit{INDEX}_a(t-1, p) \overset{?}{=} i$. For this purpose, U computes $\mathit{INDEX}_a(t-1, p)$ and then has a circuit of depth $O(\log\log n)$ to do the comparison with i. In the second conjunction, U conjoins $\mathit{GLOBAL}(t-1, i)$ with the negation of $\mathit{INDEX}_a(t-1, p) \overset{?}{=} i$. It remains open how to determine the processor p writing into G_i. The NC^k uniformity circuit U computes for each processor p the value of $\mathit{INDEX}_a(t-1, p)$ and ascertains some (e.g, the smallest) p such that $\mathit{INDEX}_a(t-1, p) = i$. Due to the way of using index registers in Index-PRAM's, the value of all $\mathit{INDEX}_a(t-1, p)$ functions can be ascertained without making use of any LOCAL_a or GLOBAL values. Therefore, the computation of p can be done by U independent of the input word except for its length.

Computation of $\mathit{LOCAL}_a(t, p)$:

We have to distinguish between two main cases.

1. $S_\mu \equiv$ "$L_a := G_{I_b}$": Here U computes $\mathit{INDEX}_b(t-1, p) = j$ and uses the resulting value j to connect $\mathit{LOCAL}_a(t, p)$ with $\mathit{GLOBAL}(t-1, j)$.

2. All the other cases for $\mathit{LOCAL}_a(t, p)$ are completely analogous to their corresponding counterparts in Theorem 5.

Computation of $\mathit{INDEX}_a(t, p)$:

The uniformity circuit U does the whole computation of $\mathit{INDEX}_a(t, p)$ values. The different cases are analogous to the ones considered for the computation of LOCAL_a in Theorem 5. In the computation of $\mathit{INDEX}_a(t, p)$ the processor number p always remains the same because only processor local computations determine the values of index registers.
"only if": Due to Proposition 11 it suffices to prove the inclusion of $\mathit{DTIME}(\log n)$-uniform NC^k in *Index-CRCW-TIME*$(\log^k n)$. As usual, we have to deal with two parts, that is the construction of a pointer structure in global memory representing the NC^k-circuit C and the actual circuit simulation. The first point is completely analogous to the simulation of the uniformity machine in Theorem 10. But for the circuit simulation we now associate with each gate of C a processor and a cell in global memory. Each processor sequentially reads the values of the global memory cells representing the inputs of the gate and accordingly updates the value of its memory cell. Here we use the instruction "*if* $L_c \cdot\!> 0$ *then* $L_a := L_b$." To check that all the other Index-PRAM restrictions are fulfilled is done in a way akin to Theorem 10. $\square$

Compare Theorem 10 to Theorem 12. 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* $L_c \cdot\!> 0$ *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 $\mathit{DSPACE}(\log n)$. 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 13 *$L \in DSPACE(\log n)$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log n)$ that additionally is equipped with the instruction "$L_a := G_{L_b}$" and that allows the processors to read from a non-constant number of global memory cells.*

Proof. Let A denote the Index-PRAM and M the $DSPACE(\log n)$-TM.
"if": The simulation of A by M follows the ideas in the proofs of Theorem 6 and Theorem 12. We again use the recursive functions $GLOBAL$, $LOCAL_a$, and $INDEX_a$. In any case, M first finds out the statement S_μ PRAM A is executing at time t making use of the regular control flow of A.

Computation of $GLOBAL(t, i)$:

The only case of interest is when $S_\mu \equiv$ "$G_{I_a} := L_b$." We do this in a manner analogous to the proof of the preceding Theorem 12. The main difference in comparison with there is that the things that are done there by the NC^k uniformity circuit are now done by M itself.

Computation of $LOCAL_a(t, p)$:

We consider three cases.

1. $S_\mu \equiv$ "$L_a := G_{L_b}$": First M computes the value j of $LOCAL_b(t-1, p)$ and then j is fed into the recursive call $GLOBAL(t-1, j)$. Clearly, we have no branching of the recursion at this point.

2. $S_\mu \equiv$ "*if* $\langle INPUT\text{-}BIT\rangle$ *then* $L_a := L_b$": First M takes a look at the considered input bit. Then depending on its value M performs a recursive call either to $LOCAL_a(t-1, p)$ or to $LOCAL_b(t-1, p)$ and any branching of the recursion is avoided.

3. All the other cases are straightforward (also cf. proof of Theorem 6) and are omitted, therefore.

Computation of $INDEX_a(t, p)$:

In a recursive call $INDEX_a(t, p)$ the parameter p always remains the same in the subsequent recursive calls that are necessary to compute $INDEX_a(t, p)$. For the straightforward details only simple modifications to the proofs of Theorem 6 resp. Theorem 12 are required.
"only if ": To simulate TM M by Index-PRAM A, processor identification numbers (PID's) of processors of A are interpreted as configurations of M and each processor locally computes in its index registers the respective successor configuration. Herein, A makes use of input conditional local assignments "*if* $\langle INPUT\text{-}BIT\rangle$ *then* $L_a := L_b$." Identifying global memory cells (resp. their numbers) with PID's now enables A to build a pointer structure in global memory representing the computation graph of M. The reachability problem can be solved by pointer jumping of type a) in Example 1 of Section 2, where we make essential use of operations of the form "$L_a := G_{L_b}$" and "$G_{I_a} := L_b$." The global write instruction in this pointer jumping algorithm is data-independent.

The regular control flow derives from the constant time construction of the computation graph of M and the simplicity of the pointer jumping algorithm (which mainly consists of one loop, cf. Example 1). The remaining Index-PRAM properties are obviously fulfilled. □

Relaxing some of the restrictions in the characterization of $DSPACE(\log n)$, we gain a characterization of $LOGDCFL$ in terms of Index-PRAM's.

Theorem 14 *$L \in LOGDCFL$ if and only if L is recognized by an Index-CRCW-PRAM in time $O(\log n)$ that additionally is equipped with the instructions "$L_a := G_{L_b}$" and "if $L_c \dot{>} 0$ then $L_a := L_b$," a standard PRAM operation set with binary operations, and allows the processors to read from a non-constant number of global memory cells.*

Proof. "if": We use the same recursion as in Theorem 13, simulating the given Index-PRAM by polynomially time and logarithmically space bounded deterministic auxiliary push-down automata. Due to Sudborough's [53] equivalence between those deterministic auxiliary push-down automata and $LOGDCFL$ the claim follows. Again (cf. Theorem 7) we need the additional push-down store in order to deal with the branching of the recursion.
"only if": Dymond and Ruzzo [20] showed $LOGDCFL = CROW\text{-}TIME(\log n)$. They pointed out that this can even be done by CROW-PRAM's where the write-owner function is restricted to be identity, that is, write-owner(i,n) $= i$ (write owner of global memory cell i is processor i). That means that the indexing values for global writes in the simulating Index-PRAM can be computed by each processor simply by first performing "$I_a := PID$" and replacing all global writes "$G_{L_a} := L_b$" of a CROW-PRAM by "$G_{I_a} := L_b$" (L_a has to have value PID). The global reads of a CROW-PRAM transfer directly. Finally, analogously to Theorem 7 a careful inspection of the simulation of $LOGDCFL$ by CROW-PRAM's shows the regularity of the control flow. Altogether, this enables the simulation of a logarithmically time bounded CROW-PRAM by an Index-PRAM in the required way. □

In the above proof it is crucial that global writes "$G_{I_a} := L_b$" of the Index-PRAM are data-independent. The data-dependent instruction "$G_{L_a} := L_b$" would lead to a recursion requiring running time $n^{O(\log n)}$ for the simulating AuxPDA. But such an AuxPDA can already simulate AC^1-circuits [48].

Again the difference between CROW-PRAM's and PPM's lies in the operation set used, as Lam and Ruzzo [35] already demonstrated.

Theorem 15 *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 $L_c \dot{>} 0$ then $L_a := L_b$" and that allows the processors to read from a non-constant number of global memory cells.*

Proof. The proof works basically in the same way as the proof of Theorem 8 and is omitted, therefore. □

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* $L_c \dot{>} 0$ *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 9. Again we use monotonicity and Akl's OR-PRAM's [5], resulting in a natural way in monotonic Index-OR-PRAM's.

Theorem 16 *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 $L_c > 0$ then $L_a := L_b$."*

Proof. The proof is a straightforward combination of the arguments in the proofs of Theorem 9 and Theorem 12. □

5 Conclusion

Data-independence of parallel algorithms appears to be a fundamental prerequisite for an efficient implementation on existing distributed memory machines. Data-independence of communication and control gives the opportunity to optimize parallel algorithms with respect to their communication pattern at compile time. It is an application-oriented concept that nevertheless fits into the groundwork of parallel complexity theory. Unbounded fan-in parallelism, bounded fan-in parallelism, and sequential computations correspond to various degrees of data-(in)dependence.

At this point let us discuss the practical applicability of current parallel complexity theory. 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 [34] and there are P-complete problems that are in a very intuitive sense *efficiently* parallelizable [60].

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 [34]. 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 [25, 41, 49]. Eclipsed by this problem of polynomial growth is the question of choosing an appropriate machine model. Our result gives 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.

So the comparison of Theorem 5 with Theorem 6 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 6 are burdened with a data-dependent communication 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. 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 have (quasi)linear output length and be based on Index-PRAM's or circuits of bounded fan-in.

But 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 $DSPACE(\log n) \subseteq OROW\text{-}TIME(\log n)$ [47] 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 besides 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 [3, 4, 10, 17, 27, 38, 51, 57] 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.

Acknowledgement Thanks to Carsten Damm, Markus Holzer, and Peter Rossmanith for stimulating discussions and helpful comments.

Table 1: Structural characterizations: The PRAM type refers to the definition of simple PRAM's in Section 2; we always presuppose a data-independent control structure contained in $ATIME(\log n)$; a letter "I" means that the corresponding structure is data-independent and contained in $ATIME(\log n)$; in the other case "D" stands for data-dependence. Index-PRAM characterizations: The term "unbounded reads" shall mean that processors may read from a non-constant number of global memory cells.

Class	Structural Characterizations			Index-PRAM characterizations with deviations from the basis model
	PRAM type	RS	WS	
NC^k	simple	I	I	"*if* $L_c > 0$ *then* $L_a := L_b$"
$DSPACE(\log n)$	simple[a]	D	I	"$L_a := G_{L_b}$" and unbounded reads
PPM^k	simple	D	I	"*if* $L_c > 0$ *then* $L_a := L_b$," "$L_a := G_{L_b}$," and unbounded reads
$LOGDCFL$	full	D	I	"*if* $L_c > 0$ *then* $L_a := L_b$," "$L_a := G_{L_b}$," unbounded reads, 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 demand that the execution structure $ES(w)$ is contained in $ATIME(\log n)$.

References

[1] F. Abolhassan, J. Keller, and W. Paul. On the cost-effectiveness of PRAMs. In *Proceedings of the 3d Symposium on Parallel and Distributed Processing*, pages 2–9, Dec. 1991.

[2] F. Abolhassan, J. Keller, and W. J. Paul. On physical realizations of the theoretical PRAM model. Technical report, Univ. Saarbrücken, 1991.

[3] 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.

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

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

[6] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, 1989.

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

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

[9] A. K. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28:114–133, 1981.

[10] A. Chin. Complexity models for all-purpose parallel computation. In Gibbons and Spirakis [22], chapter 14, pages 393–404.

[11] 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.

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

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

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

[15] 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.

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

[17] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[18] 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.

[19] P. W. Dymond, F. E. Fich, N. Nishimura, P. Ragde, and W. L. Ruzzo. Pointers versus arithmetic in PRAMs. In *Proceedings of the 8th IEEE Symposium on Structure in Complexity*, pages 239–252, 1993.

[20] 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.

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

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

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

[24] 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.

[25] 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.

[26] T. J. Harris. A survey of PRAM simulation techniques. Technical Report CSR-23-92, The University of Edinburgh, Department of Computer Science, October 1992.

[27] T. Heywood and C. Leopold. Models of parallelism. Technical Report CSR-28-93, The University of Edinburgh, Department of Computer Science, July 1993.

[28] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation: I. The model. *J. Parallel Distrib.Comput.*, 16:212–232, November 1992.

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

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

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

[32] D. S. Johnson. A catalog of complexity classes. In van Leeuwen [55], chapter 2, pages 67–161.

[33] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In van Leeuwen [55], chapter 17, pages 869–932.

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

[35] 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.

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

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

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

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

[40] W. F. McColl. General purpose parallel computing. In Gibbons and Spirakis [22], chapter 13, pages 337–391.

[41] 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*, Lecture Notes in Computer Science, pages 97–108. Springer, 1994.

[42] 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.

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

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

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

[46] 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*, Lecture Notes in Computer Science, pages 289–300. Springer, 1994.

[47] 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.

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

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

[50] 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.

[51] 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.

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

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

[54] L. G. Valiant. General purpose parallel architectures. In van Leeuwen [55], chapter 18, pages 943–971.

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

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

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

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

[59] P. Vitányi. Locality, Communication, and Interconnect Length in Multicomputers. *SIAM J. Comput.*, 17(4):659–672, August 1988.

[60] 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.

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