**Adaptive logspace reducibility
and parallel time**

Carme Àlvarez
José L. Balcázar
Birgit Jenner

Report LSI-91-48

ADAPTIVE LOGSPACE REDUCIBILITY AND PARALLEL TIME †

Carme Àlvarez*, José L. Balcázar*, and Birgit Jenner**

(Journal submission version, 15 November 1991)

Abstract

We discuss two notions of functional oracle for logarithmic space-bounded machines, which differ in whether there is only one oracle tape for both the query and the answer or a separate tape for the answer, which can still be read on while the next query is already being constructed. The first notion turns out to be basically non-adaptive, behaving like access to an oracle set. The second notion, on the other hand, is adaptive. By imposing appropriate bounds on the number of functional oracle queries made in this computation model, we obtain new characterizations of the NC and AC hierarchies; thus the number of oracle queries can be considered as a measure of parallel time. Using this characterization of parallel classes, we solve open questions of Wilson.

1. Introduction

Adaptiveness is an intuitive property of reducibilities quite relevant to complexity theory. It can be informally defined as the ability of a relativized computation to ask from its oracle set information that depends essentially on the answers obtained in previous queries.

Rooted, like many other complexity-theoretic concepts, in Recursive Function Theory, adaptiveness has a substantially different meaning when applied to resource-bounded reducibilities. Truth-table reducibility, as introduced in Recursive Function Theory, is non-adaptive by definition, as are all of its resource-bounded counterparts: in it, the relativized computation is explicitly requested to be able to produce ahead of time the list of queries that will be asked during the course of the computation. Unbounded Turing reducibility is adaptive, and this makes an enormous difference. Time-bounded reducibility looks like non-adaptive to the recursion-theorist, since the finiteness of the computation tree allows for an algorithm to collect all potential queries present there; but to the complexity-theorist, the exponentially large time bound required to perform this computation plays the role of the uncomputable: polynomial time Turing reducibility, for instance, is considered adaptive, and this particular characteristic allows for the distinction of polynomial-time Turing and truth-table reducibilities [19].

A quite different case arises for logarithmic space reducibility. The polynomial bound on the number of configurations allows one to perform, still within logarithmic space, an exhaustive search for all queries that the machine could ask in the course of the computation. This argument can be formalized to show that logarithmic space reducibility is non-adaptive, in the sense that Turing and truth-table reducibilities coincide under this resource bound [18]. One of the most important consequences of this fact is the possibility of disproving, under relativization, the inclusion $NC^1 \subseteq L$, by employing the adaptiveness abilities of the NC^1 reducibility [22].

We study here models for logarithmic space reducibilities between functions; thus here the oracle is a function instead of a set. We show that the most natural model is non-adaptive, and we introduce a second model trying to capture a definition of adaptive logarithmic space reducibility. We characterize it

† A preliminary version of this work was presented at STACS'90 (Hamburg).

* Department of Software (Llenguatges i Sistemes Informàtics), Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028 Barcelona, Spain. Work partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

** Institut für Informatik, Technische Universität München, Arcisstraße 21, 8000 München 2, Germany. Work done while visiting the Department of Software of UPC in Barcelona supported by Deutsche Forschungsgemeinschaft (DFG-Forschungstipendium Je 154/1-1). Revised version partially supported by DFG-SFB 342, subproject A4.

in various ways in terms of the hierarchies NC and AC, and in terms of the reducibilities based on these classes studied by Wilson [23].

It is proved in [23] that, when applied to classes in the NC or AC hierarchies, the reducibilities AC^k and NC^{k+1} coincide. It was left open whether they also coincide for other oracle classes; in particular, the classes L and NL. It is argued in [23] that answering such questions may be helpful in understanding the relationship between AC^k and NC^{k+1} .

As a side effect of introducing the mentioned model of adaptive logarithmic space reducibility, we will provide in the following (affirmative) answers to these open questions, indeed finding characterizations that explain in part these relationships for relativized classes. Moreover, our model allows us to build up both the AC hierarchy from AC^1 up, as well as the NC hierarchy from NC^2 up. Such a tight relationship is due to the fact that, if the oracles are taken from a function class $\mathcal{F}$ closed under log space Turing reductions, then the reduction class defined by this model corresponds to unbounded fan-in uniform circuits, with oracle nodes for functions in $\mathcal{F}$, whose depth is linearly related to the number of queries made by the logarithmic space adaptive machine.

2. Preliminaries

Throughout this paper $\log n$ means the function $\max(1, \lceil \log_2 n \rceil)$. We treat sets of words over a finite, fixed alphabet which when required we will identify with the set $\{0, 1\}$. Functions map $\{0, 1\}^*$ into $\{0, 1\}^*$, and they satisfy that all the values $f(x)$ have the same length for all x of the same length (a condition implicitly given if the function belongs to a circuit class).

Many of our complexity classes can be defined in a completely standard way by time-bounded or space-bounded multitape Turing machines; L denotes deterministic logarithmic space, NL denotes nondeterministic logarithmic space, P denotes deterministic polynomial time, PSPACE denotes deterministic polynomial space, and an F prefixing a complexity class name will correspond to functions computed by Turing machines with unbounded output tape within the corresponding resource bounds. See [2] for definitions and basic facts about these classes.

Oracle Turing machines usually query oracle sets. We also use Turing machines that query oracle functions. We will use two kinds of oracle devices: in the first one there is an unbounded oracle tape, which is used to write down the query and where in place of the query the oracle gives its answer; this answer is erased before writing a new query. In the second model there are separate tapes for queries and answers, so that the previous answer can be read on while writing the new query. Although these two models lead to the same reducibility in the case of polynomial time bounded Turing machines, we will see that these models are substantially different for logarithmic space bounded Turing machines.

There are many complexity classes below the class P. We concentrate on classes corresponding to very efficient parallel computation on a feasible amount of hardware: these are the NC and AC hierarchies. There are various characterizations of these classes; we will define them in terms of Boolean circuits. Basic facts about these classes can be found in [3]. Since we employ them to compute functions and allow oracle gates in them, we review concisely the model.

Circuits with bounded fan-in are finite directed acyclic graphs with nodes or gates up to indegree 2 with a certain label or type. Nodes of indegree zero are the input nodes $x_1, x_2, \dots, x_n$ or nodes labelled 0 or 1; nodes with indegree one are labelled $\neg$, and nodes with indegree two are labelled $\wedge$ or $\vee$. Some of the nodes are specified as output nodes $y_1, y_2, \dots, y_m$.

In circuits with unbounded fan-in there is no restriction on the indegree of the $\vee$ and $\wedge$ nodes. In this case these nodes are also called existential and universal, respectively.

A circuit family $\{\alpha_n\} := (\alpha_1, \alpha_2, \dots)$ computes a function f , if the output of α_n on input x of length n is the same as $f(x)$ for all x . If $f(x) \in \{0, 1\}$ for all x , i.e., there is only one output node in each circuit of the family, then we can also say that $\{\alpha_n\}$ accepts the set of all x for which $f(x) = 1$.

The size of a circuit is the number of nodes that it contains. The depth of a circuit is the length of the longest path from the input nodes to an output node. The NC and AC hierarchies contain all those functions that are computable by bounded fan-in, respectively unbounded fan-in circuits of polynomial size and polylogarithmic depth which satisfy a certain "uniformity" condition. Throughout this paper we will assume all circuits to be logspace uniform, i.e., the description of α_n can be constructed by a deterministic $O(\log n)$ space bounded Turing machine on input 1^n . Observe that this uniformity condition

already implies that the size of our circuits is polynomial. Each gate i of a circuit is described by a tuple $\langle i, t, p_1, p_2, \dots, p_m \rangle$ specifying the name i of the gate, the type t , and the name p_j of the j th input to gate i . This is valid for both NC and AC circuits, and also for circuits with oracle gates as described later in this section. This description is a natural extension of the direct connection languages proposed in [20], compare [11], for describing NC circuits; the extension just handles unbounded fan-in (or oracle) gates.

We have chosen this form of uniformity for simplicity. For all our circuits the weaker form of log time uniformity (see [4], [7]) would already be sufficient, but is somehow uninteresting in our context, since all of the oracles considered by us already have the computational power to compute functions in deterministic logarithmic space.

For $k \geq 0$ we denote by NC^k , respectively, AC^k , the class of functions computed by logspace uniform NC, respectively, AC circuits of depth $O(\log^k n)$.

Observe that some references use the notation NC and AC to denote classes of sets; these correspond in our definitions to the subclasses formed by 0-1 valued functions. But following e.g. [11] we extend the definition to arbitrary functions. The length of the output is anyway polynomially bounded due to the bound on the size of the circuit.

We will allow the NC and AC circuits to have access to oracle gates, which compute the value $f(x)$ of x for a functional oracle $f : \{0, 1\}^l \rightarrow \{0, 1\}^m$ (see [11]). Again, note that usually oracle nodes determine the membership of a string x in an oracle set; this corresponds in our approach to taking $m = 1$, i.e. using a 0-1 valued function f instead of the set $L_f := \{x \mid f(x) = 1\}$ (see e.g. [22], [23]).

For AC circuits, oracle nodes have depth 1. In the case of NC circuits with oracle nodes we want to charge in a fair manner the use of the unbounded fan-in oracle gate, in the same way as we should charge, say, an unbounded fan-in existential gate. Such a gate can only be simulated by an NC circuit of depth logarithmic in the number of its inputs. Thus, by definition, in a NC circuit an oracle gate with k inputs contributes $\log k$ to the depth of the circuit. This is the standard way of counting the depth of oracle nodes (see e.g. [23]). This decision furthermore is motivated by the fact that a charge of 1 for each oracle gate would yield reductions that are not appropriate for completeness proofs for the class FL or classes within the NC or AC hierarchies. Indeed, in general these classes are closed under NC^1 reducibility but are not known to be closed under such reductions.

Each functional oracle node for a function $g : \{0, 1\}^n \rightarrow \{0, 1\}^m$ for $m \geq 1$ can be replaced by a circuit family with the same depth that uses as oracle a set defining the individual bits of g (a construction often seen, e.g., in [11], [23]):

$$IB_g := \{x\$i\$b \mid |g(x)| \geq i, \text{ and the } i\text{th bit of } g(x) \text{ equals } b\},$$

and

$$IB_{\mathcal{F}} := \{IB_g \mid g \in \mathcal{F}\}.$$

Thus, for most function classes we can restrict ourselves to 0-1 oracle functions. This is also the reason why we do not charge extra depth for multiple outputs in functional oracle gates.

Let $\mathcal{F}$ be a function class. Then $NC^k(\mathcal{F})$ and $AC^k(\mathcal{F})$ denote the class of functions computed by logspace uniform NC, resp., AC circuits of depth $O(\log^k n)$ which contain functional oracle nodes for a function $f \in \mathcal{F}$. Note that a class of oracle sets $\mathcal{A}$ is equivalent to the class of oracle functions consisting of all the characteristic functions c_A for languages A contained in the language class $\mathcal{A}$.

As soon as we allow oracle gates to appear in the circuits, we can consider the relativized classes NC^k and AC^k as reducibilities. Furthermore, the non-adaptive versions of both NC^1 and AC^0 , denoted by NC_1^1 and AC_1^0 , are of special interest. They are defined as the class of functions computed by (uniform) NC_1^1 or, respectively, AC_1^0 circuits that have *at most* one oracle node on each path from an input node to an output node.

3. Non-adaptiveness

In this section we treat one of the two functional oracle computational models described in the preliminaries: the case in which the oracle answer is provided on the oracle tape, overwriting completely the query. Using that model, we define the class $FL(\mathcal{F})$ as the class of functions computable within logarithmic work space, using an oracle function from $\mathcal{F}$; we insist that the same tape is used for creating the oracle

queries and reading the oracle answers. Thus after a query, the answer can be read as many times as desired but must be completely erased before starting to write a new query. Neither the oracle tape nor the output tape are affected by the space bound, but an implicit polynomial bound is enforced by the logarithmic space bound of the work tapes unless nonterminating computations appear. The functions in a class $FL(\mathcal{F})$ can be considered reducible to functions in $\mathcal{F}$; we will refer to this kind of reducibility as *functional logspace Turing reducibility*.

This model will be shown to bear close resemblance to the set oracle model, and therefore we will use the same notation for the classes defined in both ways, the difference lying only in whether the class of oracles is a class of functions or a class of sets.

Note that this model yields the same class, whether the answer is read once (from left to right) or more often. This is because the base machine before going to the next query can ask each query q polynomially often by simply storing the configuration which "induced" q .

The fact that during the construction of a query the only information available is that of the work tapes implies a certain weakness of this model; more precisely, it is non-adaptive, in the sense that the answers obtained for previous queries cannot be relevant for the construction of forthcoming ones; this lack of adaptiveness makes this model weak. We will show this by proving that the queries might be asked simultaneously, and still get the same computational power. A fact analogous to this one arises when logarithmic space is used to compute Turing reductions between languages, since in this case the Turing and truth-table reducibilities coincide [18]. However, the analogy is not complete since the equivalence between truth-table reducibility and Turing reducibility with logarithmically many queries that sometimes holds for sets does not seem to hold for functions (see [1], [17]).

Define the class $FL_{||}(\mathcal{F})$ as follows: The oracle is called only once, but given an input x of length n , a polynomial number $p(n)$ of arguments $x_1, \dots, x_{p(n)}$ are passed to it simultaneously, and as answer a concatenation of the values of the function on each argument is obtained. The mode of operation can be formalized by using as functional oracle a specially structured function $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, and corresponds to the view of truth-table reducibility between languages as one round of queries made in parallel to an oracle set. This reducibility is clearly non-adaptive, since the answers of the first queries are not known when the next queries are prepared.

Note that like the model for $FL(\cdot)$, the model for $FL_{||}(\cdot)$ also yields the same class of functions, whether the answers (essentially given as a polynomially long 0-1 string) are read once or more often, because the sequence of queries can be replicated a polynomial number of times.

Another clearly non-adaptive restriction of functional logspace Turing reducibility $FL(\cdot)$ is given by restricting the number of queries to 1 only. This reducibility, denoted by $FL_1(\cdot)$, is similar to the "metric" reducibility defined for polynomial time in [17], since it holds:

$$f \in FL_1(g) \iff \exists h_1, h_2 \in FL \quad \text{such that} \quad f(x) = h_1(x, g(h_2(x))) \quad \text{for all } x.$$

Since no query depends of previous answers, the reducibilities $AC_1^0(\cdot)$ and $NC_1^1(\cdot)$ defined in the preliminaries are also non-adaptive.

The following result proves the non-adaptiveness of $FL(\cdot)$.

Proposition 1. Let $\mathcal{F}$ be a class of functions. Then the following statements are equivalent:

- (i) $f \in FL(\mathcal{F})$;
- (ii) $f \in FL_{||}(\mathcal{F})$;
- (iii) $f \in FL_1(AC_1^0(\mathcal{F}))$,
i.e., $f(x) = h_1(x, g(h_2(x)))$ for some $h_1, h_2 \in FL$, and $g \in AC_1^0(\mathcal{F})$;
- (iv) $f \in FL_1(NC_1^1(\mathcal{F}))$,
- (v) $f \in FL(FL(\mathcal{F}))$.

Proof. Let a "query inducing configuration" of a machine M computing a functional Turing reduction, be a configuration in which M starts to write the first symbol of a new query. W.l.o.g. we can assume that such configurations are specially marked, e.g. by requiring them to be in a special "start-query" state, and are thus easily generable within logarithmic space. Note that for a given query inducing configuration, its position in the lexicographic order of all the query inducing configurations can be computed with logarithmic space.

(i) $\implies$ (ii) Let $f \in \text{FL}(g)$ with $g \in \mathcal{F}$ be computed by the machine M . Then construct in lexicographic order all query inducing configurations and, for each one, the corresponding induced queries; then ask them all in parallel. Clearly, with the information of all answers to possible queries, the computation of M on x can now be completely simulated by referring to the i th answer if M queries a string w induced by the i th query configuration. Thus, $f \in \text{FL}_{\parallel}(g)$.

(ii) $\implies$ (iii) Let $f \in \text{FL}_{\parallel}(\mathcal{F})$ be computed by a machine M with parallel functional oracle $g \in \mathcal{F}$. We assume for the moment that each of the parallel queries has size $p(|x|)$ for a polynomial p and that there are exactly $p(|x|)$ many of them, given an input of length n . Then clearly, M is a machine which poses only 1 query, but to a functional oracle g' of the form

$$g' : (\{0, 1\}^{p(n)})^{p(n)} \rightarrow (\{0, 1\}^{q(n)})^{p(n)}$$

with $g'(w_1, w_2, \dots, w_{p(n)}) = g(w_1), g(w_2), \dots, g(w_{p(n)})$ such that each $g(w_i)$ has the same length $q(n)$ for a polynomial q .

Clearly, a logspace uniform AC_1^0 circuit that computes g' with functional oracle nodes for g can be constructed. The circuit is trivially structured; for an input of length n^2 each consecutive group i of n inputs are the inputs to oracle node i , $1 \leq i \leq n$, the m outputs of oracle node i are the outputs $m(i-1) + 1$ up to $m(i-1) + m = m \cdot i$ of the circuit. Thus, $g' \in \text{AC}_1^0(\mathcal{F})$. In the case of the parallel queries not having the assumed format, the base machine can "pad up" each query to length $p(n)$ and pose additional "dummy" queries to match the format. The circuit then provides in parallel oracle nodes for each length up to $p(n)$, filters out the relevant parts of the input, and attaches them to the right oracle gate. The output again will be padded up to the format.

By decomposing the computation of a $\text{FL}_1(\cdot)$ machine in three phases "computation before the oracle query", "oracle query", "computation after the oracle query", it can be shown generally that $f \in \text{FL}_1(\mathcal{G})$ for any function class $\mathcal{G}$, if and only if there exist $h_1, h_2 \in \text{FL}$, and $g \in \mathcal{G}$ such that $f(x) = h_1(x, g(h_2(x)))$. The functions h_1 and h_2 describe the computation of M from the start configuration up to the query (h_2), and from the query answer up to the end (h_1). Clearly, h_1 and h_2 are functions in FL .

(iii) $\implies$ (iv) Immediate.

(iv) $\implies$ (v) This holds since, for every function h , being $h \in \text{NC}_1^1(\mathcal{F})$ implies $h \in \text{FL}(\mathcal{F})$. To see this, consider a logspace uniform NC_1^1 circuit with oracle nodes for $f \in \mathcal{F}$, which satisfies that there is at most one oracle node on each path from an input node to an output node. Then, clearly, a deterministic logspace Turing machine M can evaluate each output gate of the circuit in a depth-first manner. Whenever an oracle gate, whose up to polynomially many input bits have to be evaluated, is encountered in the evaluation, M does so using its oracle tape to store these bits, and finally solves this gate by querying its oracle. Note that it seems critical here that there is at most one oracle gate on each path.

(v) $\implies$ (i) The idea of the proof is similar to that in the proof of the transitivity of logspace reductions. It consists of a (quasi) step-by-step simulation of a machine M_f computing a function $f \in \text{FL}(g)$ for $g \in \text{FL}(h)$ and $h \in \mathcal{F}$, by a machine M' which queries directly h .

Let the machine computing g with the help of the functional oracle be M_g . The simulating machine M' starts to simulate M_f step-by-step, so that each output of M_f will be output of M' . When M_f is not constructing a query, the simulation proceeds normally. But whenever M_f enters a query inducing configuration c , M' saves this configuration on its worktape, in order to be able to generate the coming query q_c arbitrarily many times, and goes on with the simulation. The induced query q_c is discarded.

The only moment in which the simulation cannot proceed is when M_f wants to read the i th bit of its oracle answer tape. Let us see how to find it: it is part of the answer of M_g on q_c , so M' starts simulating M_g . But q_c is never fully constructed; instead, whenever in the simulation M_g wants to read the j th bit of q_c , M' repeats again the relevant part of the simulation of M_f from the saved query inducing configuration c on, to find the j th bit of the induced query q_c . This process is iterated as many times as M_g needs an input bit before writing down its i th output bit. Then the computation of M_g can be aborted and the computation of M_f resumed, knowing which is the needed i th bit of the oracle answer. ■

Note that if $\mathcal{F}$ is taken to be the class of characteristic functions c_A for languages A in a language class $\mathcal{A}$, then Proposition 1 yields $\text{FL}(\mathcal{A}) = \text{FL}_{\parallel}(\mathcal{A})$. Therefore, these function classes with oracle sets behave in a manner similar to that of the analogous language classes.

From Proposition 1 it follows immediately that a function class is closed under logspace functional Turing reducibility if and only if it is closed under logspace many-one reducibility and $\text{AC}_1^0(\cdot)$ reducibility. Actually, in Section 6 we will point out that a weaker, more technical condition can be substituted for closure under $\text{AC}_1^0(\cdot)$ reducibility. These closure properties are known to hold for many function classes contained in FP that contain FL .

Corollary 2. The following function classes are closed under $\text{FL}(\cdot)$:

$$\text{FL}, \text{FNL}, \text{NC}^i \text{ for } i \geq 2, \text{ and } \text{AC}^j \text{ for } j \geq 1. \quad \blacksquare$$

This discussion allows us to trace an analogy between language classes and function classes, in the following sense. Assume that the language class $\mathcal{A}$ contains L , and take its closure under $\text{FL}(\cdot)$. The resulting class $\text{FL}(\mathcal{A})$ is in a sense “the” function class corresponding to the language class $\mathcal{A}$. We mean that endowing any of the classes mentioned in Corollary 2 with either oracle sets from $\mathcal{A}$ or with oracle functions from $\text{FL}(\mathcal{A})$ yields the same function class.

Note that in general for function classes $\mathcal{F}$ with polynomially bounded functions, it holds $\text{IB}_{\mathcal{F}} \subseteq L(\mathcal{F})$, and $\mathcal{F} \subseteq \text{FL}(\text{IB}_{\mathcal{F}})$, and thus $\text{FL}(\text{IB}_{\mathcal{F}}) = \text{FL}(\mathcal{F})$.

The closure of the class NL under this functional reducibility $\text{FL}(\cdot)$ yields a very interesting class of functions, the class $\text{FL}(\text{NL})$, which will play a very important role later on in this paper. To ease the reference to it we will use the shorthand FNL .

Indeed, this class is exactly the class of (single-valued) functions computable by nondeterministic logspace Turing transducers which have for each input x at least one accepting computation and compute on all accepting computations the same output; see also [15], where this concept is used to define a notion of NL -printability. This is an easy consequence of the closure of NL under complementation, which allows us to replace direct nondeterministic simulations for the oracle calls in $\text{FL}(\text{NL})$ computations.

An interesting point is that the class of single-valued functions computed by nondeterministic polynomial time transducers does not seem to be so well-behaved, and seems to lack an effective enumeration of nondeterministic polynomial time machines computing them. On the contrary, such an enumeration can be obtained for FNL using the characterization as $\text{FL}(\text{NL})$.

A second point in which FNL seems to differ from one of its natural polynomial time counterparts, namely $\text{FP}(\text{NP})$, is the adaptiveness property. $\text{FP}(\text{NP})$ is an adaptive class, but FNL is not, since in general the answers of the oracle cannot all be recorded (see Proposition 4 below).

The function class $\text{NL}^* = \text{NC}^1(\text{NL})$ (see [11]) has been suggested also as a function class closely related to NL . Indeed, its restriction to 0-1 functions equals the class NL , as stated e.g. in [6], because an NL machine can evaluate the NC^1 circuit by computing the reduction in a depth first manner from the output gate. On the other hand, as discussed above, we advocate $\text{FL}(\mathcal{A})$ as a kind of “functional analog” of a language class $\mathcal{A}$; in this sense, FNL is the generalization of the 0-1 functions in NL to arbitrary functions. There is no conflict, as shown by the following:

Proposition 3.

$$\text{FNL} = \text{FL}(\text{FNL}) = \text{FL}_{\parallel}(\text{NL}) = \text{AC}_1^0(\text{NL}) = \text{AC}^0(\text{NL}) = \text{NC}_1^1(\text{NL}) = \text{NC}^1(\text{NL}) = \text{NL}^*.$$

Proof. The inclusion $\text{FL}_{\parallel}(\text{NL}) \subseteq \text{AC}_1^0(\text{NL})$ follows from the fact that $\text{AC}_1^0(\text{FNL}) = \text{AC}_1^0(\text{NL})$, since we can compute the individual bits of a function $g \in \text{FNL}$ or $g \in \text{FL}_{\parallel}(\text{NL})$ in parallel using a set in NL . The inclusion $\text{NL}^* \subseteq \text{FNL}$ (resp., $\text{AC}^0(\text{NL}) \subseteq \text{FNL}$) follows from the same argument that shows that $\text{NL}^* \subseteq \text{NL}$ for 0-1 functions, by letting the FNL machine evaluate each of the output bits in sequence. The other inclusions are immediate or direct consequences of Proposition 1. $\blacksquare$

4. Adaptiveness

This section continues the study of functional oracles for space-bounded machines considering the second model presented in the preliminaries. In this model, the machine can create a query w on the query tape,

go into a query state, and receive the answer $f(w)$ on the oracle answer tape; the oracle query tape will be blank again. While the machine is reading the answer of its last query, it can now already create the following query. This query will thus not only depend on the work tape configuration of the transducer, but possibly also on the last answer; and hence using this model we obtain an adaptive reducibility, in which each query may depend essentially on the answer obtained from the previous query. We will be mainly interested in the classes FL, FNL, NC^k , AC^k for $k \geq 0$ as oracle classes, and also in the class $\{id\}$ containing just the identity function id .

Observe that in this model the logarithmic space bound on the working tape(s) no longer enforces any time bound of the machine nor a bound on the length of the queries. Without an explicit bound on the number of queries and without an explicit time bound or bound on the size of the queries any recursive enumerable function could be computed already with the identity function as functional oracle. Simply, because the oracle query and answer tape then function as an additional (unlimited) storage.

By bounding either the size of the oracle tapes or the number of queries (or time) of the machine by a polynomial in the length of the input, the power of this model with the identity function as oracle is reduced to FSPACE.

Proposition 4. The following are equivalent characterizations of the class FSPACE:

The class of functions computable by the logspace adaptive model with the identity function as oracle

- (i) with a polynomial bound on the *length* of the queries.
- (ii) with a polynomial bound on the *number* of queries.

Proof. The equivalence of FSPACE with (i) is immediate: to simulate the computation of the logspace machine with a polynomially bounded identity function as oracle clearly polynomial space is sufficient; viceversa such a machine computes any step of the polynomial space machine by applying the appropriate transition to a configuration stored on the oracle answer tape, writing the resulting configuration on the oracle query tape.

The equivalence of FSPACE with (ii) is more interesting, since the queries might reach exponential length (e.g., by doubling the length of the queried word at each query). Moreover, only polynomially many queries are available to simulate an exponentially long PSPACE computation. To prove that all of FSPACE can be computed in this way, let us describe first how to accept a PSPACE set. Consider a polynomial time alternating machine and start with its root configuration on the oracle query tape. Then, for polynomially many steps, transfer the oracle query to the answer tape by a query to the identity function, and then transfer it back substituting three configurations for each configuration in an odd position: add to the left one successor, copy the configuration, and add to the right the other successor. Configurations at even positions are simply copied. In this way the full inorder traversal of the alternating computation tree is constructed. Now, again for polynomially many steps, the configurations at odd positions can be evaluated and later deleted until evaluating the root. Generalizing this argument to computing functions requires repeating this algorithm simultaneously for exponentially many different alternating computations, each of them computing one single bit of the output (the sequence of roots must be set up by recursive doubling). See [3] for details regarding the alternating model of computation.

For the converse, naive considerations might suggest that it is possible for adaptive logspace machines to reach doubly exponential length of their queries, by iterating a process in which the length of the query is squared. Thus, we first argue that the maximal length of a query is $O(2^{p(n)})$ for some polynomial p . We do it inductively, by counting configurations between two consecutive queries: we have a factor of n for the positions of the input tape head, and a factor of n^k for configurations of worktapes, for some fixed k , and a factor of the length of the answer tape for the positions of the answer tape head. Together, these three factors bound the length of the query tape (since otherwise the machine is looping forever). Thus the length of the first query is at most n^{k+1} , the length of the i -th query is at most $n^{(i \times k + 1)}$, and the length of the last query is at most $n^{O(p(n))} = 2^{O(p'(n))}$ for polynomials p and p' . Note that any configuration can be represented with space polynomially in n .

Now consider the computation of the adaptive logspace machine as the composition of polynomially many logarithmic space transducer machines $M_1, \dots, M_{p(n)}$ for a given input of length n . Apply the technique of showing the transitivity for logarithmic space reductions, already used in the proof of one of the parts of proposition 1. An FSPACE machine can keep track of one configuration for each machine

M_i , excluding its currently output tape, but including a counter of output symbols already produced. The simulation starts with $M_{p(n)}$, until it requests a new input symbol; then this symbol is obtained from a simulation of $M_{p(n)-1}$, which may request again input symbols from its predecessor. Since there is not enough room to keep all the intermediate inputs (i.e. queries of the original machine), when some input head moves backwards all the preceding machines have to be restarted until recomputing the desired symbol. ■

A space bound on the oracle tape principally yields space classes unless further restrictions are imposed (see also Section 5), and a bound on the number of queries principally yields parallel time classes. In the following, we will suppose that the oracle tapes of the adaptive model are polynomially bounded in the length of the input. All our oracle classes will contain only functions whose answers are polynomially long in the queries.

In order to distinguish this model of reducibility from the functional logspace Turing reducibility, we introduce a different notation. Let $FL_G[\mathcal{F}]$ be the class of functions which are computable by log space machines with separate oracle query and answer tapes bounded in length by a polynomial in the length n of the input, and querying an oracle function from $\mathcal{F}$ with the number of queries bounded by $g(n)$ for a function $g \in \mathcal{G}$. Such restrictions on the number of functional queries then yield characterizations of “smaller” parallel classes than $FPSPACE$, as we shall see. More precisely the class $\mathcal{G}$ will be either the polylogarithmic (i.e., $O(\log^k n)$) or the polynomial functions.

With a polynomial number of queries, we can compute any polynomial time computable function, or equivalently, we can simulate any logspace uniform (and hence polynomially sized) circuit:

Proposition 5. $FL_{poly}[\{id\}] = FL_{poly}[FL] = FL_{poly}[FNL] = FL_{poly}[FP] = FP$.

Proof. The inclusion $FL_{poly}[FP] \subseteq FP$ can be proved by a straightforward simulation. By definition, the length of each new query remains polynomial in the length of the input. For the inclusion $FP \subseteq FL_{poly}[\{id\}]$ simulate a given FP transducer M step by step on a given input x with as many queries as the time bound of M . Each configuration of M of the computation on x can be constructed on the query tape given the last configuration on the answer tape. ■

More generally, a polynomial number of queries is equivalent to functional polynomial time Turing reducibility $FP(\cdot)$.

Proposition 6. Let $\mathcal{F}$ be a function class. Then it holds: $FL_{poly}[FL(\mathcal{F})] = FP(\mathcal{F})$.

In fact, the model turns out to be only interesting to distinguish functions classes *within* FP by looking at subpolynomial query bounds; for the case of polylogarithmic many queries we will get characterizations of circuit classes.

Several characterizations of parallel complexity classes by in a sense “sequential” models are based on the idea of *phases* of a computation: fragments of computations within which the operations are “nearly independent” of each other, but depend only on results obtained in previous phases. Frequently this idea of phase indicates parts of the computations that can be parallelized, so that the number of phases in a computation corresponds to the parallel time required to simulate this computation (and vice-versa). This idea appears as the driving force in a large number of simulations among parallel models of computation, as well as simulations between sequential and parallel models, which are the basis of the Similarity Principle proposed by Hong ([13], [14]). We should point out here that one of the tool models he uses, the Logspace Transform Machine (LSTM), is similar to our adaptive queries model, and it can be seen that it holds that $FL_r[\{id\}] = (r, poly)\text{-LSTM}$, where in the right hand side r stands for the number of phases and $poly$ for the width bound of the LSTM model. Thus, the main differences between the LSTM model and our adaptive reducibilities are the flexibility given by the choice of arbitrary oracle functions and the possibility of measuring polylogarithmic width (thus obtaining also characterizations of the class SC , as shown in Section 5).

Here we will show that also in our model the adaptive queries to functional oracles can be understood as dividing the computation process in a kind of phases, in the sense that an unbounded fan-in circuit can be evaluated with as many queries as its depth (which measures the parallel time). Conversely, to simulate a logspace machine with functional oracle, a circuit with oracle gates needs a depth equivalent to the number of oracle queries provided that the oracle class has a certain computational power.

Thus, here we extend the idea that number of phases corresponds to parallel time, so that it encompasses as well, for certain oracles, the “number of queries” resource. We obtain new characterizations of the AC and NC classes that are very well suited to the work with oracles. These characterizations allow us to complete the work of [23], where the techniques did not work for oracle classes other than circuit classes.

We prove now how the depth of unbounded fan-in circuits corresponds precisely to number of functional queries under this model.

Theorem 7. Let $\mathcal{F}$ be a function class. Then it holds:

$$AC^k(FL(\mathcal{F})) = FL_{\log^k}[\![FL(\mathcal{F})]\!] \quad \text{for all } k \geq 0.$$

Proof. For the inclusion from left to right, let the function h be computed by the logspace uniform family $\{\alpha_n\}$ of AC^k circuits with oracle nodes for a function $f \in FL(\mathcal{F})$. Then the circuit α_n for inputs of length n has depth $c \cdot \log^k n$ for a constant c . Recall that in the case of unbounded fan-in circuits, each oracle gate contributes 1 to the depth of the circuit.

We will show how α_n can be evaluated in exactly as many phases as its depth, using the oracle query and answer tapes of a logspace Turing machine M with functional oracle g as temporary storage means to keep provisional results, i.e. the partially evaluated circuit. Let the *depth* of a gate be its distance to the input gates. Then, in phase i all the gates of depth at most i will be evaluated.

For this, the functional oracle g is chosen such that it reproduces the complete description of the circuit α_n , supplemented by the values of those oracle gates which are directly evaluable with the information so far obtained, i.e., those oracle gates, whose direct predecessors are already evaluated.

Thus the oracle g is the following function:

Input: $\langle \alpha_n \rangle \$ \langle o_1, o_2, \dots, o_m \rangle \$ \langle x_1, x_2, \dots, x_m \rangle$, where $\langle \alpha_n \rangle$ is a description of a circuit α_n with oracle nodes in f , where all of the gates up to a certain depth are evaluated, i.e., have their value attached, $\langle o_1, o_2, \dots, o_m \rangle$ is a list of oracle gates of the circuit, and $x_1, x_2, \dots, x_m \in \{0, 1\}^*$ is a list of inputs to the oracle gates in the list, x_i being input for gate o_i ;

Output: $\langle \alpha_n \rangle \$ \langle o_1, o_2, \dots, o_m \rangle \$ \langle y_1, y_2, \dots, y_m \rangle$, where $y_j = f(x_j)$ for all $1 \leq j \leq m$.

Note that $g \in FL(\mathcal{F})$, since it has just to copy the first two parts and evaluate m many times the function f .

The base machine M does the following for a given input x of length n :

First, M constructs the complete circuit description $\langle \alpha_n \rangle$ on its oracle query tape by simulating the logspace machine that produces α_n . It thereby attaches to each input gate its respective value by reading it from the input tape. Then M repeats the following basic computation phases for $i := 1$ to $c \cdot \log^k n$:

- (i) It queries its functional oracle to provide on its answer tape the description of α_n partially evaluated up to depth $i - 1$, and the output of all oracle gates of depth i .
- (ii) It then produces a new description of α_n on the oracle query tape, where *all* gates of depth i are evaluated, and attaches a list of the oracle gates of depth $i + 1$, followed by a list of the inputs to these gates.

For (ii) M has to

- “update” the old circuit description with the values of the oracle gates of depth i , by appending this information at the end of the new description of the respective oracle gate;
- evaluate all “normal” gates of depth i , and append this information at the end of the new description of the respective gate;
- find those oracle gates whose inputs are now known, and create a list of all of them and a list of their inputs.

It is easy to see that logspace suffices to perform these computations, when the oracle answer tape provides the information of the circuit already evaluated up to depth $i - 1$ for all gates and up to depth i for the oracle gates. Note that the oracle answer tape has to be read more than once to obtain all the needed information.

The process of evaluation is repeated until $i = c \cdot \log^k n$. Then all gates are evaluated. The value $h(x)$ then can be produced on the output tape by simply reading out the values of the respective output gates in the appropriate order.

For the inclusion from right to left, let f be a function in $FL_{\log^k}[\![FL(\mathcal{F})]\!]$ computed by a logspace Turing transducer M with $c \cdot \log^k n$ queries to a functional oracle $g \in FL(\mathcal{F})$, for a constant c . Let the length of $f(x)$ for all x of length n be $q(n)$. W.l.o.g. M outputs nothing before the first oracle query.

Then a computation of M is (roughly speaking) composed of $c \cdot \log^k n$ computation phases between two oracle answers, which can be described by a function $h \in FL(g)$ with

$$h(\langle x, v_i, out_i, a_i \rangle) := \langle x, v_{i+1}, out_{i+1}, a_{i+1} \rangle$$

where

- x is the input of length n ;
- v_i and v_{i+1} are configurations of size $\log n$ (they include the work tape contents and input head position, but not the oracle tape contents or output tape contents); v_i is the configuration in which M is when receiving the i th oracle answer; v_0 is the start configuration, and $v_{c \cdot \log^k n + 1}$ is the (unique) halt configuration;
- out_i is the output of M up to configuration v_{i+1} (not v_i), (which is the same as the output up to the configuration M is in directly before query v_{i+1}), and out_{i+1} is the output produced by M up to configuration v_{i+2} , which has out_i as prefix; $out_0 = \lambda$; and
- a_i is the i th oracle answer.

Note that all of the parameters can be padded up such that they have a fixed length for each n , with the length of each out_i being always the length $q(n)$ of $f(x)$.

It is easily verified that $h \in FL(g)$; and thus $h \in FL(\mathcal{F})$.

Since the number of oracle queries of M is bounded by $c \cdot \log^k n$, we can construct an (essentially trivially structured) AC^k circuit consisting of a line of $c \cdot \log^k n$ functional oracle gates $o_1, o_2, \dots, o_{c \cdot \log^k n}$ for the function h . Here the circuit input gates are the input gates of o_1 , all the output gates of oracle gate o_i are input gates of oracle gate o_{i+1} , and some specified $p(n)$ outputs of the last oracle gate $o_{c \cdot \log^k n}$ are $f(x)$, the output of M in a computation on x . It is easy to see that the circuit is logspace uniform. ■

Observe that the proof shows that any function computable by an $AC^k(FL(\mathcal{F}))$ circuit of depth $c \cdot \log^k n$ for a constant c can be computed by a logspace machine with $c \cdot \log^k n$ queries to an oracle in $FL(\mathcal{F})$.

Theorem 7 implies that for any function class $\mathcal{F}$ closed under functional Turing reducibility $FL(\cdot)$ it holds: $AC^k(\mathcal{F}) = FL_{\log^k}[\![\mathcal{F}]\!]$. In particular, we get:

Corollary 8. For all $k \geq 0$ it holds:

- (i) $AC^k(FL) = FL_{\log^k}[\![FL]\!]$ = $FL_{\log^k}[\![\{id\}]\!]$;
- (ii) $AC^k(FNL) = FL_{\log^k}[\![FNL]\!]$;
- (iii) $AC^k(NC^i) = FL_{\log^k}[\![NC^i]\!]$ for $i \geq 2$;
- (iv) $AC^k(AC^j) = FL_{\log^k}[\![AC^j]\!]$ for $j \geq 1$.

Proof. The first equalities in (i) to (iv) all follow from Theorem 7 with Corollary 2. The second equality in (i) holds, since any of the oracle gates of a $AC^k(FL)$ circuit can be evaluated by the logspace base machine (recall the proof of Theorem 7 for the inclusion $AC^k(FL) \subseteq FL_{\log^k}[\![FL]\!]$). Thus, here we even have $AC^k(FL) \subseteq FL_{\log^k}[\![\{id\}]\!]$. ■

To complete the characterizations, we show now that we can also characterize parallel classes defined by bounded fan-in circuits. As a corollary we will obtain a positive solution to Wilson's open problems mentioned in the introduction.

Theorem 9. For any $k \geq 0$ it holds:

- (i) $NC^{k+1}(FL) \subseteq FL_{\log^k}[\![FL]\!]$;
- (ii) $NC^{k+1}(FNL) \subseteq FL_{\log^k}[\![FNL]\!]$.

Proof. We will show (i). The proof for (ii) can be obtained analogously.

Let $f \in NC^{k+1}(FL)$ be a function that is computed by the family $\{\alpha_n\}$ of circuits with oracle nodes for a function $g \in FL$. As said in the preliminaries, we can assume with no loss of generality that the oracle g is a 0-1 function in FL . Each α_n has depth $c \cdot (\log n)^{k+1}$ for a constant c , and is generated by the

logspace Turing machine M_α on input 1^n . Suppose that M_α outputs the specification of each gate i as $(i, t_i, p_{i_1}, p_{i_2}, \dots, p_{i_m})$, where t_i indicates the type of gate i , and the sequence $p_{i_1}, p_{i_2}, \dots, p_{i_m}$ specifies the inputs of gate i . For normal gates, this sequence contains two values, the left and right inputs; for oracle gates, this can be up to polynomially (in n) many inputs, say n^d . Thus for any gate, each predecessor gate of any gate i can be recorded with $d \cdot \log n$ bits by simply referring to its index in the specification of the gate i produced by M_α .

Recall that the contribution of an oracle node i to the depth of the circuit is logarithmic on the number m of its inputs $p_{i_1}, p_{i_2}, \dots, p_{i_m}$. Define the *weight* of a node i with m inputs to be $\log m$. Then any gate contributes to the depth of the circuit with its weight.

Define the *weighted depth* of a node i relative to a list of gates $l = (i_0, i_1, \dots, i_n)$ as the maximal sum of the weights of all nodes on a path from a node of l to i that does not pass through any other node of l , counting the weight of i but not that of the node of l . Then, clearly, the weighted depth relative to the list of input nodes of all nodes in α_n is smaller than or equal to $c \cdot \log^{k+1} n$. Note that any path from l to i can be recorded with as many bits as the weighted depth of i relative to l by simply recording descriptions of all its gates *relative* to the description of i (see also [5]). This fact is crucial for the proof.

We will construct a logspace bounded Turing transducer T with a functional oracle $h \in \text{FL}$ that evaluates the $\text{NC}^{k+1}(g)$ circuit in pieces of depth more or less logarithmic, taking into account the contribution of the oracle nodes to the total depth. The evaluation principle is similar to that of Theorem 7, where we described the evaluation of an unbounded fan-in circuit of depth $\log^k n$ with functional oracle nodes using as many phases as the depth. But now we are confronted with a bounded fan-in circuit of depth $\log^{k+1} n$. Thus we have to evaluate more than one "level" of gates in each phase.

T will evaluate α_n for a given input x using again its oracle query and answer tapes as temporary storage means. T will query $\log^k n$ times the functional oracle described below, always satisfying that after phase j all gates with weighted depth up to $j \cdot c \cdot \log n$ relative to the input gates are evaluated (and accessible as a partial result on the oracle answer tape).

The oracle query and answer tapes will contain during each phase a description of the partially evaluated circuit α_n , with gates of fan-in 1 and 2, and oracle nodes, which describes each gate i by $(i, t_i, p_{i_1}, p_{i_2}, \dots, p_{i_m}, v_i)$, where i, t_i , and $p_{i_1}, p_{i_2}, \dots, p_{i_m}$ are as above and the additional entry $v_i \in \{0, 1, +, \perp\}$ specifies a value 0 or 1 of the gate, marks the gate by $+$ or labels it "not yet evaluable" by means of $\perp$. Note that for an input gate i the list $p_{i_1}, p_{i_2}, \dots, p_{i_m}$ is empty, and v_i equals the i th bit x_i of the input $x = x_1 x_2 \dots x_n$ of the circuit.

The functional oracle h is the following:

Input: A description of the partially evaluated circuit α_n such that the gates marked with $v_i = +$ build a subcircuit (with oracle nodes) of depth maximally $c \cdot \log n$;

Output: Another description of α_n where, for all gates i that were marked with $v_i = +$ in the input description, now v_i has the correct value, and the other labels do not change.

Initially, T produces on the oracle query tape the circuit description of α_n by simulating M , integrating the values of the input nodes for the input x , and marking all other gates with $\perp$. Since no gates are marked $+$, a query to the oracle just transfers this description to the answer tape.

Then T performs $\log^k n$ computation phases, each consisting of the following two processes:

(i) marking process:

given the last partially evaluated circuit description, found on the oracle answer tape, T produces on the oracle query tape a new description of α_n in which a $+$ mark has been set labeling all the gates that have weighted depth $c \cdot \log n$ relative to the list of gates already evaluated;

(ii) evaluation:

a query to the oracle h yields another description of α_n , where all the marks $+$ set during the marking process are replaced by the appropriate values of the gates.

Thus, each query furnishes T with the values of an additional set of gates: those receiving a $+$ during the marking process, i.e. all gates with weighted depth up to $c \cdot \log n$ relative to the list of the gates that are already evaluated. Therefore, after j oracle queries, all gates with weighted depth up to $j \cdot c \cdot \log n$ relative to the input gates are evaluated in the description found on the oracle answer tape.

Hence the last description after $\log^k n$ phases contains the values of all gates, and T can now simply collect the values of the output gates to produce $f(x)$.

We claim the following:

1. The oracle function h belongs to FL. This can be verified with the same argument that proves that FL is closed under NC^1 reductions (see [11]).
2. The marking process can be performed within logarithmic space. This is shown by an argument that is implicit in the proof of $NC^1(FL) = FL$ in [11] (or [5]). For each gate i , it has to be checked whether i has weighted depth $c \cdot \log n$ relative to a list of gates l . A sufficient and necessary condition for this is that *all* of the paths that can be specified relative to i by at most $c \cdot \log n$ bits reach a gate on the list l . This can be checked by marking out $c \cdot \log n$ bits on the tape and constructing for each gate i marked i in the description each path upwards relative to i . If the stack overflows for one of the paths without reaching a gate in l then we leave the specification of i unchanged; if no overflow occurs, then we mark i substituting $+$ for i . ■

Observe that the statement of this theorem differs substantially from Theorem 7 in the level of generality. A natural question is whether other classes can be substituted for FL and FNL in Theorem 9. Most of the proof would remain valid for an arbitrary class, but some hypothesis on $\mathcal{F}$ must hold in order to guarantee that the function h employed in the proof remains in the class of oracle functions, and that the marking process can be achieved. Closure under $FL(\cdot)$ does not seem to suffice, since there may be several oracle gates of nonconstant fan-in linked together in the marked part of the circuit, and $FL(\cdot)$ would be unable to perform the evaluating process. Closure under $NC^1(\cdot)$ does not suffice since the marked part of the circuit has logarithmic depth but might not be uniformly generated in logspace. However, if h can be computed by a logspace uniform NC^1 family of circuits with oracle nodes for g , then Theorem 9 would hold for all classes $\mathcal{F}$ simultaneously closed under $FL(\cdot)$ and under $NC^1(\cdot)$. (Note that closure under $FL(\cdot)$ does not imply closure under NC^1 Turing reducibility as shown in [22].)

From Theorems 7 and 9 we get several interesting corollaries. The first one is the affirmative answer to Wilson questions: indeed the reducibilities AC^k and NC^{k+1} coincide on the oracle classes L and NL. Recall from the preliminaries that $AC^k(L) = AC^k(FL)$ and similarly for NC^{k+1} and/or FNL.

Corollary 10. Let id denote the identity function. Then it holds for all $k \geq 0$:

- (i) $AC^k(L) = AC^k(FL) = NC^{k+1}(FL) = NC^{k+1}(L) = FL_{\log^k} \llbracket FL \rrbracket = FL_{\log^k} \llbracket \{id\} \rrbracket$;
- (ii) $AC^k(NL) = AC^k(FL(NL)) = NC^{k+1}(FL(NL)) = NC^{k+1}(NL) = FL_{\log^k} \llbracket FNL \rrbracket$. ■

Note that for $k = 0$ we have:

$$AC^0(FL) = NC^1(FL) = \bigcup_c FL_c \llbracket FL \rrbracket = FL;$$

$$AC^0(FNL) = NC^1(FNL) = NL^* = \bigcup_c FL_c \llbracket FNL \rrbracket = FNL.$$

In a forthcoming work by different authors [9], certain closure conditions on a class will be shown to be sufficient for the reducibilities AC^k and NC^{k+1} to coincide on that class, thus encompassing results of [23] and Corollary 10.

An interesting consequence of the preceding results is that with the $FL_{\log^k} \llbracket \cdot \rrbracket$ operator both the NC and the AC hierarchy are built up in exactly the same manner. We obtain one or the other just depending on the starting class.

Corollary 11. For all $k \geq 0$ it holds:

$$NC^{k+2} = FL_{\log^k} \llbracket NC^2 \rrbracket = FL_{\log^i} \llbracket NC^j \rrbracket, \quad \text{with } j \geq 2, i + j = k + 2;$$

$$AC^{k+1} = FL_{\log^k} \llbracket AC^1 \rrbracket = FL_{\log^i} \llbracket AC^j \rrbracket, \quad \text{with } j \geq 1, i + j = k + 1;$$

and thus

$$NC = AC = \bigcup_k FL_{\log^k} \llbracket \{id\} \rrbracket.$$

In particular, we have

$$AC^{k+1} = FL_{\log^k} \llbracket AC^k \rrbracket \quad \text{for all } k \geq 1, \text{ and}$$

$$NC^{k+1} = FL_{\log^k} \llbracket NC^k \rrbracket \quad \text{for all } k \geq 2.$$

Proof. This follows with Theorem 7 and the results by Wilson [23] that for all $k \geq 0$ it holds $AC^{k+1} = AC^k(AC^1)$ and $NC^{k+2} = AC^k(NC^2)$. ■

Observe that although e.g., both the classes $NL^* = NC^1(NL)$ and $NC^2(NL)$ have a characterization in our model, constant versus logarithmic number of oracle queries, the class NC^2 does not.

5. Further characterizations via adaptive logspace reducibility

In the previous sections we have contrasted two notions of functional oracle for logarithmic space-bounded machines, giving rise to the reducibilities $FL(\cdot)$ and $FL_{\log^k}[\cdot]$. Among other things, we have shown that many of the function classes $\mathcal{F}$ contained in FP , like e.g. $\mathcal{F} \in \{FL, FL(NL), AC^1, NC^2, \dots\}$ are closed under the non-adaptive reducibility $FL(\cdot)$, whereas the closure of these classes under the adaptive reducibility $FL_{\log^k}[\cdot]$ gives us new characterizations of a variety of circuit classes. Thus in this model the number of functional oracle queries can be considered a measure of parallel time.

We have stated all theorems for the functional case. An inspection of the proof techniques shows that all theorems of the last section also hold for the respective language classes, $L_{\log^k}[\cdot]$. In this section, we present further complexity classes that can be characterized in terms of adaptive logspace Turing reducibility $L_{\log^k}[\cdot]$.

In the model $L_{\log^k}[\cdot]$ the oracle query and answer tapes were (explicitly) polynomial space bounded. Let us denote a bound $f(|x|)$ for $f \in \mathcal{F}$ on the *length* of the oracle query and answer tape in a $L_{\mathcal{G}}[\cdot]$ reducibility (i.e., a reducibility, where there exists a bound $g(|x|)$ on the number of queries for a function $g \in \mathcal{G}$) by $L_{\mathcal{G}}[\cdot]_f$, and let $L_{\mathcal{G}}[\cdot]_{\mathcal{F}} := \bigcup_{f \in \mathcal{F}} L_{\mathcal{G}}[\cdot]_f$.

Denoting by *poly* the class of all polynomials, we have shown e.g. that for all $k \geq 0$ it holds:

$$AC^k(FL) = NC^{k+1}(FL) = L_{\log^k}[FL]_{poly} = L_{\log^k}[\{id\}]_{poly};$$

and thus

$$AC = NC = \bigcup_k L_{\log^k}[FL]_{poly} = \bigcup_k L_{\log^k}[\{id\}]_{poly}.$$

By switching the bound $\log^k$ on the number of queries and the bound *poly* of the length of the oracle tapes, we get the classes $L_{poly}[FL]_{\log^k}$. These classes turn out to be exactly the classes $SC^k := DSPACE, DTIME(\log^k n, poly)$ of languages deterministically computable with $O(\log^k n)$ space and polynomial time; $SC := \bigcup_k SC^k$ (see e.g. [20]).

Theorem 12. For all $k \geq 1$ it holds

$$SC^k = L_{poly}[FL]_{\log^k} = L_{poly}[\{id\}]_{\log^k}; \quad \text{and thus} \quad SC = \bigcup_k L_{poly}[\{id\}]_{\log^k}.$$

Proof. For the inclusion $SC^k \subseteq L_{poly}[FL]_{\log^k}$, simulate an SC^k machine M on input x step by step using the identity function as oracle. The i th query posed then will be the i th configuration (input head position, work tape content, etc.) of size $\log^k |x|$ in M 's computation on x .

The inclusion $L_{poly}[FL]_{\log^k} \subseteq SC^k$ can be obtained by a straightforward simulation of the oracle machine, since there is the bound $O(\log^k n)$ on the length of the oracle query and answer tapes and a $O(\log n)$ space complexity bound on the oracle function. ■

Having shown how to characterize "small" parallel classes with the help of adaptive logspace reducibility, let us make some remarks about this reducibility, when applied to higher classes such as NP or corresponding function classes such as $FL_1(NP)$, $FL_{\log}(NP)$, and $FL(NP)$. These classes are the functions computable by a logspace transducer that queries an NP oracle once, logarithmically often, or without bound on the number of queries, respectively.

Let $P_{\log}(NP)$ denote the class of languages accepted by polynomial time machines that have logarithmically often query access to an NP oracle. This class, which is now widely called Θ_2 after the work of [21], has a variety of characterizations, among others by logspace Turing reductions: $\Theta_2 = L(NP) = L_{\log}(NP)$

[21]. Due to this result, clearly, similar characterizations of $P_{\log^k}(NP)$ for $k \geq 2$ by logspace Turing reductions do not seem possible, unless these classes fall down to Θ_2 .

Using the new notion of adaptive logspace Turing reducibility, we can generalize the result of Wagner as follows.

Theorem 13. For all $k \geq 0$,

$$P_{\log^{k+1}}(NP) = L_{\log^{k+1}}[FL_1(NP)] = L_{\log^k}[FL_{\log}(NP)] = L_{\log^k}[FL(NP)].$$

Here, $L_{\log^k}[\cdot]$ refers to the adaptive logspace Turing reducibility computed by logspace acceptors rather than by logspace transducers with oracle. This is essential for the proof.

Proof. We will sketch a proof of the inclusions $P_{\log^{k+1}}(NP) \subseteq L_{\log^{k+1}}[FL_1(NP)]$, $L_{\log^{k+1}}[FL_1(NP)] \subseteq L_{\log^k}[FL_{\log}(NP)]$ and $L_{\log^k}[FL(NP)] \subseteq P_{\log^{k+1}}(NP)$. The remaining inclusion is trivial.

For the first inclusion, we compute longer and longer prefixes of the $\log^{k+1} n$ binary answers of a $P_{\log^{k+1}}(NP)$ machine M for a given input of length n with the help of a suitable $FL_1(NP)$ oracle. Since prefixes of the answer string cannot be kept on the work tape of the logspace machine, they are copied directly from the answer to the query tape. With the complete answer string, one more query is sufficient to see whether M accepts the input.

For the second inclusion, consider a $L_{\log^{k+1}}FL_1(NP)$ machine M with input of length n . Again we successively compute the $\log^{k+1} n$ binary answers of M , this time in blocks of length $\log n$ by a suitable oracle function from $FL_{\log}(NP)$. By keeping all the blocks of answers already computed, $\log^k n$ many transfers of the oracle answer to the new oracle query asking for the next answer block are needed to obtain the full 0-1 answer string of M on the oracle answer tape. Then, acceptance or rejection can be computed with one further transfer and query. Note that the oracle function in $FL_{\log}(NP)$ needed here is bounded in length by $\log^{k+1} n$.

For the third inclusion, we apply the census techniques developed in [12] and [16] and used in [21] and [8] to show $L(NP) \subseteq L_{\log}(NP)$. Let M be a $L_{\log^k}[FL(NP)]$ machine with oracle f . Let f be computed by M_f . We will refer to the census of a query q of M as the number of queries answered positively by M_f on input q . Such a census is polynomial in length of q and can thus be represented by $\log n$ bits. Given the census of q an NP machine can completely simulate M_f and thus compute any bit of f (see [21]). It is not difficult to see that this extends to the computation of M as well, given each census for a sequence of subsequent queries of M . Thus, for a given input of length n a $P_{\log^{k+1}}(NP)$ machine first computes the census of all the $\log^k n$ many queries with the help of a suitable NP oracle constructing each census by a prefix technique. Then one further query is sufficient to compute acceptance or rejection of the input. ■

By applying Theorem 7, we get the following corollary, which has been shown in [10] directly without referring to adaptive logspace reducibility.

Corollary 14. [10] For all $k \geq 0$,

$$P_{\log^{k+1}}(NP) = AC^k(NP).$$

Proof. Theorem 13 and Theorem 7 together yield $P_{\log^{k+1}}(NP) = AC^k(FL(NP))$. $AC^k(FL(NP)) = AC^k(NP)$ is obtained as follows. Construct the transition matrix of a machine M_f computing the oracle function f in $FL(NP)$ with the help of one layer of NP oracle nodes. Then use one further layer of NP oracle nodes to compute each bit of f in parallel. ■

The proof technique of Theorem 13 actually also yields a characterization of $P_{\log^k}(NP)$ in terms of polylog space-bounded classes and therefore by width-bounded circuits. It holds for all $k \geq 0$,

$$P_{\log^k}(NP) \subseteq SC^k(NP).$$

It is unclear whether the reverse inclusion can be obtained, since a SC^k machine can query polynomially often. Equality holds when the number of queries of the SC^k machine is restricted to $\log^k n$.

6. Discussion

Some of the results presented in the preceding sections can be strengthened somewhat by using more technical conditions on the classes.

Say that a function class $\mathcal{F}$ is closed under *marked replication* if, for all $f \in \mathcal{F}$, the function $f_{\text{repl}} : \{0, 1\}^* \$ \rightarrow \{0, 1\}^* \$$ with

$$f_{\text{repl}}(w_1 \$ w_2 \$ \dots \$ w_m \$) := f(w_1) \$ f(w_2) \$ \dots \$ f(w_m) \$$$

is contained in $\mathcal{F}$. Similarly, closure under *marked concatenation* for a function class $\mathcal{F}$ holds if, for all $f_1, f_2 \in \mathcal{F}$, the function $\text{conc}_{f_1, f_2} : (\{0, 1\}^* \$)^2 \rightarrow (\{0, 1\}^* \$)^2$ with

$$\text{conc}_{f_1, f_2}(w_1 \$ w_2 \$) := f_1(w_1) \$ f_2(w_2) \$$$

is contained in $\mathcal{F}$. Let the *join* of two functions $f_1, f_2 : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be the function $\text{join}_{f_1, f_2} : \{0, 1\}^+ \rightarrow \{0, 1\}^+$ with

$$\text{join}_{f_1, f_2}(w) := \begin{cases} f_1(w'), & \text{if } w = 1w', \\ f_2(w') & \text{if } w = 0w'. \end{cases}$$

A function class $\mathcal{F}$ is closed under *join*, if for all $f_1, f_2 \in \mathcal{F}$ the function $\text{join}_{f_1, f_2} : \{0, 1\}^+ \rightarrow \{0, 1\}^+$ is contained in $\mathcal{F}$.

Then it can be seen that closure under marked replication can be substituted for $\text{AC}_1^0(\cdot)$ in Proposition 1.

Also, using these definitions, a slightly stronger version of Theorem 7 can be obtained as follows: Let $\mathcal{F}$ be a function class that contains FL and is closed under join, and under marked replication and marked concatenation. Then it holds:

$$\text{AC}^k(\mathcal{F}) = \text{FL}_{\log^k}[\mathcal{F}] \quad \text{for all } k \geq 0.$$

Clearly, any “reasonable” functional complexity class is closed under marked replication, marked concatenation, and join, and will, if it has some “minimal” computational power, include all functions computable with logarithmic space. However, we feel that the theorems as presented in the body of the paper cover all the interesting cases and have more natural hypothesis.

Acknowledgment

The authors are grateful to Prof. L. Ruzzo for his immediate and detailed explanation of certain closure results for NC^1 reducibility, to Chris Wilson for helpful comments on an earlier version of this paper and stimulating discussions, and to Bernd Kirsig, who pointed out to us a flaw in a previous version of this paper. The second author is indebted to Ron Book, for discussions in which he contributed insights on the relationship between queries and phases. All three authors are very grateful to the Deutsche Forschungsgemeinschaft, who supported the visit of the third author to Barcelona.

References

- [1] C. Álvarez, B. Jenner; A very hard logspace counting class; Proc. of the 5th Structure in Complexity Theory Conference (1990), pp. 154–168. (to appear in Theoretical Computer Science)
- [2] J.L. Balcázar, J. Díaz, J. Gabarró; Structural Complexity I; Springer-Verlag, Berlin, 1988.
- [3] J.L. Balcázar, J. Díaz, J. Gabarró; Structural Complexity II; Springer-Verlag, Berlin, 1990.
- [4] D.A. Mix Barrington, N. Immerman, H. Straubing; On uniformity within NC^1 ; J. of Computer and System Sciences 41,3 (1990), pp. 274–306.
- [5] A. Borodin; On relating time and space to size and depth; SIAM Journal of Computing 6,4 (1977), pp. 733–744.
- [6] A. Borodin, S.A. Cook, P. Dymond, W.L. Ruzzo, M. Tompa; Two applications of complementation via inductive counting; SIAM Journal of Computing 18,3 (1989), pp. 559–578.

- [7] S. Buss; The Boolean formula value problem is in ALOGTIME; Proc. of the 19th Annual ACM Symposium on Theory of Computing (1987), pp. 123-131.
- [8] S.R. Buss, L. Hay; On truth-table reducibility to SAT and the difference hierarchy over NP; Proc. of the 3rd Structure in Complexity Theory Conference (1988); pp. 224-233.
- [9] J. Castro, P. Rossmanith, C. Seara; in preparation.
- [10] J. Castro, C. Seara; Characterizations of some complexity classes between Θ_2^P and Δ_2^P . Report LSI-90-27, Univ. Politècnica de Catalunya.
- [11] S.A. Cook; A taxonomy of problems with fast parallel algorithms; Information and Control 64 (1985), pp. 2-22.
- [12] L.A. Hemachandra; The strong exponential hierarchy collapses; Proc. of the 19th Annual ACM Symposium on Theory of Computing (1987), pp. 110-122.
- [13] J.-W. Hong; On similarity and duality of computation (I); Information and Control 62 (1984), pp. 109-128.
- [14] J.-W. Hong; Computation: Computability, Similarity and Duality; Pitman, London, 1986.
- [15] B. Jenner; B. Kirsig; Alternierung und Logarithmischer Platz; Dissertation (in German), Universität Hamburg, 1989.
- [16] J. Kadin; $P^{NP}[\log n]$ and sparse Turing-complete sets for NP; Proc. of the 2nd Structure in Complexity Theory Conference (1987); pp. 33-40.
- [17] M.W. Krentel; The complexity of optimization problems; Proc. of the 18th Annual ACM Symposium on Theory of Computing (1986), pp. 69-76.
- [18] R. Ladner, N. Lynch; Relativization of questions about log space computability; Math. Systems Theory 10 (1976), pp. 19-32.
- [19] R. Ladner, N. Lynch, A. Selman; Comparison of polynomial-time reducibilities; Theoretical Computer Science 1 (1975), pp. 103-123.
- [20] W. Ruzzo; On uniform circuit complexity; J. of Computer and System Sciences 22 (1981), pp. 365-383.
- [21] K. Wagner; Bounded query classes, SIAM J. on Computing 19,5 (1990), pp. 833-846.
- [22] C.B. Wilson; Relativized NC; Math. Systems Theory 20 (1987), pp. 13-29.
- [23] C.B. Wilson; Decomposing NC and AC; SIAM Journal of Computing 19,2 (1990), pp. 384-396. (preliminary version in 4th Structure in Complexity Theory Conference, 1989)