Adaptive logspace
and depth-bounded
reducibilities

José Luis Balcázar

Report LSI–91–14
Adaptive logspace and depth-bounded reducibilities

José L. Balcázar

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya (ed. FIB)
08028 Barcelona, SPAIN

Abstract: We discuss a number of results regarding an important subject: the study of the computational power of depth-bounded reducibilities, their use to classify the complexity of computational problems, and their characterizations in terms of other computational models. In particular, problems arising in the design of concurrent systems are studied, and two kinds of logarithmic space reductions are defined. The first one is nonadaptive and equivalent in many respects to the oracle set model. The second one provides a notion of adaptive logspace reducibility which turns out to characterize precisely depth-bounded reductions. The closures of NP under these reducibilities are also treated.

This is a conference delivered at Structure in Complexity Theory 6th Annual Conference, Chicago 1991, and appears in the proceedings in the present form.

Resumen: Discutimos una serie de resultados relacionados con un tema importante: el estudio del poder de cálculo de las reducibilidades acotadas en profundidad, su uso para clasificar la complejidad de problemas computacionales, y sus caracterizaciones mediante otros modelos de cálculo. En particular, estudiamos problemas sugeridos por el diseño de sistemas concurrentes y definimos dos tipos de reducibilidad en espacio logarítmico. El primer tipo es inadaptivo y equivalente en muchos sentidos al modelo de conjunto oráculo. El segundo proporciona una noción de reducibilidad adaptativa en espacio logarítmico que caracteiza exactamente las reducibilidades de profundidad acotada. También tratamos los cierres de NP bajo estas reducibilidades.

Adaptive logspace and depth-bounded reducibilities

José L. Balcázar
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya (ed. FIB)
08028 Barcelona, SPAIN

Abstract

We discuss a number of results regarding an important subject: the study of the computational power of depth-bounded reducibilities, their use to classify the complexity of computational problems, and their characterizations in terms of other computational models. In particular, problems arising in the design of concurrent systems are studied, and two kinds of logarithmic space reductions are defined. The first one is nonadaptive and equivalent in many respects to the oracle set model. The second one provides a notion of adaptive logspace reducibility which turns out to characterize precisely depth-bounded reductions. The closures of NP under these reducibilities are also treated.

1 Introduction

This text presents some results obtained at the author's department, together with the environment, developed by many researchers, in which these results play a role. These results are presented here following a common thread underlying them: the study of the computational power of depth-bounded reducibilities and their use to classify the complexity of computational problems.

Depth-bounded circuits form an interesting model of computation: they provide a theoretically sound model (among many others) for the concept of "problem solvable by a feasibly parallel computer with an extremely efficient use of time". The definition, proposed in [30], is notably robust, admitting a large number of very different characterizations by other parallel (or even sequential, as we shall see) computation models; moreover, by using this computational model a large number of interesting, naturally motivated problems become precisely classified in terms of their sequential or parallel complexity.

In order to compare the complexity of computational problems, various notions of reducibility have been successfully used in many settings. Depth-bounded circuits provide such a notion of reducibility [35]. We will use depth-bounded reducibilities to characterize the precise complexity of several problems related to the design of concurrent systems. These results were announced in [3], [6], and [9].

The peculiar properties of depth-bounded reducibilities, as compared to other reducibilities based on Turing machines, were proved in [35]. In particular, a major difference arising among various computational models when used for computing reductions is the degree of adaptiveness of the reducibilities. We know of no formal definition of the concept of adaptiveness, but the intuition is the following: in a reduction that calls repeatedly a problem as subroutine, adaptiveness is the ability of sending as parameters of the calls data that depend inherently of the answers obtained from previous calls. On the contrary, in a nonadaptive model, the parameters for all the calls are predictable in advance without actually making any of these calls. We will use sometimes adaptiveness as a relative concept, always in an intuitive fashion.

Circuit-based reducibilities turn out to be much
more adaptive than space-bounded machines, and this fact allowed Wilson [35] to prove the corresponding reducibilities different. Several modifications of the model allow for finer relationships which point out the precise differences in computational power: in [36] the Turing machine model is equipped with a capability of stacking half-constructed queries, thereby enlarging its power by the addition of a certain degree of adaptiveness; while in [2] width and size restrictions on circuits are considered, in order to reduce their adaptiveness.

We propose here a different way of increasing the power of logarithmic space reductions by endowing them with a controlled amount of adaptiveness. To accomplish this, we work for a large part of this paper with functions instead of sets: we use machines with output and we reduce functional problems to functional problems. The oracle access mechanisms for functional oracles can be thought of in two natural but very different ways. We study both here. The first one is nonadaptive and equivalent in many respects to the oracle set model. The second one provides a notion of adaptive logspace reducibility which turns out to characterize precisely depth-bounded reductions. Using this notion and some more technical advances, the work reviewed here solved open problems from [2], [34], and [37]. The results we will describe regarding this subject of research appear in [4] and [15]; a few of them are announced here for the first time, and will be published in appropriately authored papers in the near future.

Moreover, the adaptive logarithmic space reducibility provides a sequential model of parallel computation, in the sense that we prove that it offers a measurable resource essentially equivalent to parallel time: the number of queries to a trivial oracle (namely the identity function). We obtain several results along the lines of the Similarity and Duality principles of Hong [21], from which we argue that the resource “number of queries” indeed allows one to see our model of adaptive logspace machines, in a sense, as a model of parallel computation.

Finally let us point out that although many of our results are stated in terms of function classes, most of them also hold for their respective language classes, as revealed by simple inspections of the proofs given in the references.

2 Computational models

Sets will always be understood here as 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; this condition is implicitly given if the function belongs to a circuit class, and we explicitly assume it for all our functions here. Although for particular functions this makes a difference, all the classes we work with are closed under appropriate paddings allowing us to accept this assumption without any loss of generality. The identity function is denoted \(id\).

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 [7] for definitions and basic facts about these classes (but do not forget about the unboundedness the output tape assumed here).

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 unbounded or explicitly bounded 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.

2.1 Depth-bounded reducibilities

These reducibilities are based on functions computed by boolean circuits, possibly including oracle gates. Depth-bounded boolean circuits provide definitions of complexity classes included in P, corresponding to very efficient parallel computation on a feasible amount of hardware: the NC and AC hierarchies. Basic facts about these classes can be found in [8].

Circuits with bounded fan-in are finite directed acyclic graphs with nodes or gates up to indegree 2. Nodes of indegree zero are the input nodes \( z_1, z_2, \ldots, z_n \) or nodes labelled 0 or 1; nodes with indegree one are labelled \( \neg \), and nodes with indegree two are labelled \( \land \) or \( \lor \). Some of the nodes are specified as output nodes. In circuits with unbounded fan-in there is no restriction on the indegree of the \( \lor \) and \( \land \) nodes. Then these nodes are also called existential and universal, respectively. To compute a function, circuit families \( \{ C_n \} = (C_1, C_2, \ldots) \) are required, where each circuit in the family computes the function for a fixed input length.

The NC and AC hierarchies contain all those functions that are computable by bounded fan-in, respectively unbounded fan-in uniform circuits of polynomial size and polylogarithmic depth. 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 \( O(\log^k n) \). Regarding the uniformity condition, we will assume all our circuits to be logspace uniform; logtime uniformity ([13], [27]) could be used as well.

It is known that other parallel computation models such as Parallel RAMs are equivalent to unbounded fan-in circuits [33]; bounds on the number of processors correspond to bounds on the size of the circuit, and bounds on the PRAM time correspond to bounds on the depth of the circuit. Thus \( AC^k \) is formed by all the problems that can be solved in a PRAM within \( O(\log^k n) \) time for constant \( k \) and using polynomially many processors; in particular, \( AC^0 \) is constant time on a PRAM and \( AC^1 \) is logarithmic time. \( AC^0 \) contains some problems with long histories, for instance the addition of two integer numbers [16], [27].

We consider also circuits consisting of threshold gates: these evaluate to one when the number of inputs with value one reaches a certain threshold, fixed for that gate. The class \( TC^k \) contains the problems solvable by uniform circuits of threshold gates with depth bounded by \( O(\log^k n) \) [28]. In particular, \( TC^0 \) contains very natural and interesting problems such as the multiplication of two integer numbers [16], [27]. Since threshold gates can simulate AND and OR gates by using as threshold the number of inputs and 1 respectively, we have that \( AC^k \subseteq TC^k \). In [1] and [19] it is proved that for \( k = 0 \) this inclusion is strict, since the majority problem \( \text{MAJ} \), which consists of deciding whether the input contains at least as many ones as zeros,

\[
\text{MAJ} = \{ w \in \{0, 1\}^* \mid |w_1| \geq |w_0| \}
\]

cannot be solved by a constant depth polynomial size unbounded fan-in circuit having only AND and OR gates. The proof does not require any uniformity condition on the \( AC^0 \) circuits.

In order to use circuits to compute reducibilities we employ oracle gates, which compute the value \( f(z) \) of a functional oracle \( f : \{0, 1\}^l \rightarrow \{0, 1\}^m \) for an input \( z \) (see [17]). In unbounded fan-in circuits, oracle nodes have depth 1. In the case of bounded fan-in 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. Thus, by definition, in NC circuits an oracle gate with \( l \) inputs contributes \( \log l \) to the depth of the circuit.

Let \( F \) be a function class. Then \( NC^k(F) \) and \( AC^k(F) \) denote the class of functions computed by logspace uniform NC, respectively AC circuits
of depth $O(\log^k n)$ which contain functional oracle nodes for a function $f \in \mathcal{F}$.

Certain nonadaptive versions of relativizations of both $\text{NC}^1$ and $\text{AC}^0$, denoted here by $\text{NC}^1_\mathcal{F}$ and $\text{AC}^0_\mathcal{F}$, are also of interest. They are defined as the class of functions computed by (uniform) $\text{NC}^1$ or, respectively, $\text{AC}^0$ circuits that have at most one oracle node on each path from an input node to an output node.

Our descriptions of circuits are natural extensions of well-accepted proposals [17], [32]. Each gate $i$ of a circuit is described by a tuple $(t, t, p_1, p_2, \ldots, p_m)$ specifying the name $i$ of the gate, the type $t$, and the name $p_j$ of the $j$th input to gate $i$. The description of the circuit consists of a sequence of descriptions of all its gates. This is valid for both $\text{NC}$ and $\text{AC}$ circuits, and also for circuits with oracle gates; in this case those $p_j$ which specify an input coming from an oracle gate should also specify from which output of the oracle gate.

Our convention regarding oracle gates includes a as particular case circuits with oracle sets (see e.g. [35], [37]). Conversely, a 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 that uses as oracle a set formed by those pairs $a \leq b$ for which the $i$th bit of $g(x)$ equals $b$ ([17], [37]); this construction does not increase the depth nor raises problems with uniformity conditions.

2.2 Nonadaptive logspace reducibility

Given a class of functions $\mathcal{F}$, we consider here the class of functions reducible to this class in logarithmic space. More precisely, we define the class $\text{FL}(\mathcal{F})$ as the class of functions computable within logarithmic work space, using an oracle function from $\mathcal{F}$; we assume here that the same tape is used both 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 time bound is forced by the logarithmic space bound of the working tapes. We will refer to this kind of reducibility as functional logspace Turing reducibility. This model bears close resemblance to the oracle set model, and therefore we will use the same notation for both, the difference lying only in whether the class of oracles is a class of functions or a class of sets.

Note that even if this model is restricted to read the oracle answer only once, no computational power is lost: the machine 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 that this model is nonadaptive: the answers obtained from previous queries cannot be relevant for the construction of forthcoming ones. 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 [25].

Inherently nonadaptive reducibilities are obtained in an obvious manner if the model used to define them is allowed only one query to be relevant for each result. For instance, the reducibility denoted by $\text{FL}_1(\cdot)$ is given by restricting the number of queries to 1 only. This is similar to the "metric" reducibility defined for polynomial time in [24], since $f \in \text{FL}_1(g)$ if and only if there exist $h_1, h_2 \in \text{FL}$ such that $f(x) = h_1(x, g(h_2(x)))$ for all $x$.

This will be weaker in principle than $\text{FL}(\cdot)$; thus, as a second possibility, define the class $\text{FL}_0(\mathcal{F})$ as follows: the oracle is called only once, with a sequence consisting of a polynomial number $p(n)$ of arguments $x_1, \ldots, x_{p(n)}$; as answer, a sequence consisting of the values of the function on each of the arguments is obtained. This corresponds to the view of truth-table reducibility between languages as one round of queries made in parallel to an oracle set. The already discussed reducibilities $\text{AC}^0_1(\cdot)$ and $\text{NC}^1_1(\cdot)$ are also clearly nonadaptive.
Now the following result proves the nonadaptiveness of FL(·). More details about the proof can be found in [5].

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

(i) $f \in \text{FL}(\mathcal{F})$;

(ii) $f \in \text{FL}_{\|}(\mathcal{F})$;

(iii) $f \in \text{FL}_1(\text{AC}^0_i(\mathcal{F}))$

i.e., $f(x) = h_1(x, g(h_2(x)))$

for some $h_1, h_2 \in \text{FL}$, and $g \in \text{AC}^0_i(\mathcal{F})$;

(iv) $f \in \text{FL}_1(\text{NC}^1_i(\mathcal{F}))$;

(v) $f \in \text{FL}(\text{FL}(\mathcal{F}))$.

**Proof (sketch).** For (i) implies (ii), a machine prepares as oracle query a list of all queries that could be necessary, by cycling through all query-inducing configurations and writing out the induced query. Then the complete simulation can proceed without any more queries.

For (ii) implies (iii), the circuit is used simply to replicate the computation of the oracle function. Then by decomposing the computation of a FL$_1(\cdot)$ machine in three phases, “computation before the oracle query”, “oracle query”, and “computation after the oracle query”, we obtain an appropriate decomposition $f(x) = h_1(x, g(h_2(x)))$.

It is immediate that (iii) implies (iv). To prove (iv) implies (v) it suffices to show that every function $h \in \text{NC}^1_i(\mathcal{F})$ is also in FL(\mathcal{F}). A depth-first evaluation of the circuit (even with one layer of oracle nodes) can be performed in logspace.

The proof of (v) implies (i) is similar to the proof of the transitivity of logspace reductions. It consists of a (quasi) step-by-step simulation. Queries are not really asked: a configuration that produces the current query is saved instead. When a symbol from a query answer is needed, the machine for the oracle function is started and simulated until that symbol is produced. In turn, when this machine for the oracle function requests an input symbol, it is obtained by simulating the base machine from the saved query-inducing configuration on.

It follows immediately that a function class is closed under log space functional Turing reducibility if and only if it is closed under both logspace many-one reducibility and AC$^0_i(\cdot)$ reducibility. It is interesting to remark that the closure under AC$^0_i(\cdot)$ reducibility employed in statement (iii) can be weakened at the price of a quite technical discussion. Here it is equivalent to use the closure under marked replication instead of the closure under AC$^0_i(\cdot)$. In later results it will be pointed out that for many of them it suffices to assume closure under marked replication, marked concatenation, and join: for all $f_1, f_2 \in \mathcal{F}$, the functions

$$f_{\text{rep}}(w_1 \ldots w_m \mathcal{S}) = f(w_1) \ldots f(w_m) \mathcal{S}$$

$$f_{\text{conc}}(f_1, f_2)(w_1 \& w_2 \mathcal{S}) = f_1(w_1) \& f_2(w_2) \mathcal{S}$$

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

must belong to $\mathcal{F}$.

As a consequence of proposition 1 we immediately obtain:

**Corollary 2.** The following function classes are closed under FL(·): FL, NC$^i_j$ for $i \geq 2$, AC$^j_i$ for $j \geq 1$, and TC$^i_j$ for $j \geq 1$.

The closure of the class NL under this functional reducibility FL(·) yields a very interesting class of functions, the class FL(NL), which will reappear 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 the same output on all accepting computations; see also [22], 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 FL(NL) computations. This class admits the following characterization [4], [5]:

**Proposition 3.** FNL = FL(FNL) = FL$\|$(NL) = AC$^0_i$(NL) = AC$^0$(NL) = NC$^1_i$(NL) = NC$^1$(NL).
Under the name NL*, and defined as NC^1(NL), this function class was suggested already in [17] as the function class corresponding to the language class NL.

2.3 Adaptive logspace reducibility

We introduce now an adaptive model for logarithmic space reducibility. It is natural to suspect that the source of nonadaptiveness in the reducibility discussed in the previous section is the logarithmic space bottleneck separating each two successive oracle queries. An intuitive way to obtain an adaptive model by modifying this one is to let the machine read the last answer while preparing the next query. Thus the computational model is a logarithmic space machine fitted with two oracle tapes: on one of them the argument to the functional oracle is written, and the answer of the oracle is provided on the other.

Observe that in this model the logarithmic space bound no longer enforces a time bound, nor a bound on the length of the oracle queries. Without an explicit bound on the size of the oracle tapes and without a bound on the number of queries or on the running time of the machine, already with the identity function as functional oracle any recursive enumerable function could be computed by this model, simply because the oracle query and answer tapes then function as an additional unlimited storage. Therefore, for most applications we will bound explicitly the length of the oracle queries by a polynomial on the length of the input. All oracle functions we will use provide polynomially long oracle answers.

There are two interesting ways of characterizing the class FPSPACE with this model:

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

(i) The class of functions computable by the logspace adaptive model with the identity function as an oracle with a polynomial bound on the length of the queries.

(ii) The class of functions computable by the logspace adaptive model with the identity function as an oracle with a polynomial bound on the number of queries.

**Proof.** The equivalence of FPSPACE with (i) is immediate: polynomial space clearly suffices to simulate the computation of the logspace machine with an identity function oracle, and vice versa such a machine can devote each computation phase between queries to a simulation of one step of the polynomial space machine.

The equivalence 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 FPSPACE 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 one: add to the left one successor, copy the configuration, and add to the right the other successor. 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).

For the converse, we make use of vector machines [31]. A vector machine can simulate in polylogarithmic time the computation of a standard logarithmic space machine (the proof can be found in [31] and in theorem 1.5 of [8]). For a machine with output, the boolean transitive closure used in the proof has to be replaced by a more general
transitive closure concept (page 220 in [31], section 28 of [21]). Considering the last oracle answer as input and the next oracle query as output, the computation between two queries of a logarithmic space adaptive machine can be thus simulated by a vector machine in time polylogarithmic on the maximal length $O(2^{p(n)})$ of a query, and therefore in time polynomial in the length $n$ of the input.

Repeating this simulation for the polynomially many queries we still have a polynomial time vector machine, which can be simulated in FPSPACE by the results of [31] (see also theorem 1.6 of [8]).

Another idea to prove the last inclusion, suggested by B. Jenner (personal communication), is to consider the computation of the adaptive logspace machine as the composition of polynomially many logarithmic space reducibilities; a FPSPACE machine can keep track of one configuration for each of them, simulating as much as it can and recomputing intermediate output bits whenever required. This idea generalizes one of the parts of proposition 1.

Consider now adaptive logarithmic space machines as acceptors for languages. In a similar manner as part (i) of proposition 4, we can impose now a $O(\log^k n)$ bound on the length of the queries; when it is combined with a polynomial bound on the number of queries, the classes obtained turn out to be exactly the family $\text{SC}^k$ of languages deterministically computable with $O(\log^k n)$ space and polynomial time. The proof is quite straightforward after part (i) of the previous proposition.

As we will see in later sections, the classes NC, which are in a sense duals of SC, can be obtained by switching the bound on the number of queries and the bound of the length of the oracle tapes. Thus, variants of one and the same model allow us to characterize both the class NC and the dual class SC.

From now on we assume a polynomial bound on the length of the queries. Let us present the logarithmic space reducibility for which we use this adaptive model. In order to distinguish this reducibility from the nonadaptive one presented before, we introduce a different notation for it. Let $FL_{\text{G}}[\mathcal{F}]$ be the class of functions that 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 $\mathcal{F}$ a number of times bounded by $g(n)$ for a function $g \in \mathcal{G}$. We will be mainly interested in $\mathcal{F}$ being one of the classes FL, FNL, NC$^k$, AC$^k$ for $k \geq 0$, and the class $\{id\}$ containing just the identity function $id$; for $\mathcal{G}$ we will consider the class of polylogarithmic (i.e., $O(\log^k n)$) functions and the polynomial functions. E.g., for polynomially many queries we have (see [4], [5]):

**Proposition 5.** If $\mathcal{F}$ is closed under (nonadaptive) FL(·) then $FL_{\text{poly}}[\mathcal{F}] = FP(\mathcal{F})$.

In fact, it holds that $FL_{\text{poly}}[FL(\mathcal{F})] = FP(\mathcal{F})$, and for the particular case of the identity function as oracle one can prove $FL_{\text{poly}}[\{id\}] = FL_{\text{poly}}[FL] = FL_{\text{poly}}[FNL] = FL_{\text{poly}}[FP] = FP$.

We should point out here that a model bearing some similarities to this adaptive logspace machine, the Logspace Transform Machine (LSTM), was used by Hong [20], [21] as a tool to prove simulations between several computational models. The reader who knows of this work can see that it holds that $FL_{\text{o}}[\{id\}] = (r, poly)$-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 above), both of which are lacking in the LSTM.

In fact, proposition 4 above can be interpreted as a phenomenon of “duality” in the terminology of Hong: he shows that for many computation models, appropriately defined parallel time and space are duals of each other, while appropriately defined sequential time is dual of itself. As will be manifest later on, queries are in a sense a measure of parallel time for logarithmic space adaptive ma-
chines, and thus proposition 4 amounts to proving once more this duality for our model.

3 Problems from concurrency

One reason to develop concurrent programs stems from the fact that important advantages can be gained from the use of massive parallelism. However, the current experience seems to indicate that the design of correct concurrent programs is even more difficult than in the sequential case. In the development of sequential software, the use of formal methods sometimes provides considerable help. The possibility of formalizing both specifications and implementations in the same formal language yields the potential of automated analysis, allowing for early checking of correctness and provably correct prototypes. However, it has been experienced that for concurrent systems, even for very very small ones, verification using formal systems may give rise to formidable computational problems.

We use depth-bounded reducibilities to characterize precisely the parallel complexity of three problems regarding formal models for concurrent computation: decision of firing sequences for Petri nets, trace equivalence for partially commutative monoids, and strong bisimilarity in finite transition systems. The first two turn out to be equivalent to the Majority problem, and therefore are in $\text{TC}^0$; thus they can be efficiently parallelized, allowing logarithmic time Parallel RAM algorithms, but do not admit constant time Parallel RAM algorithms with polynomially many processors. Strong bisimilarity in finite labelled transition systems, however, turns out to be $\text{P}$-complete.

3.1 Firing sequences in Petri nets

The Petri net model was one of the first models introduced to describe concurrent processes with distributed control [29]. A Petri net is a bipartite graph consisting of places $P$ and transitions $T$. Places can hold tokens. During the evolution of the net, the number of tokens in each place models some local aspect of the system. A transition can be fired if the places immediately preceding it in the graph hold a pre-specified number of tokens, and then those places lose those tokens and the places immediately following the transition gain a pre-specified number of tokens. Transitions model the events of the net; thus every sequential behavior is represented by a word $w \in T^*$. Each such word is a firing sequence of the net.

The following result from [6] classifies precisely the complexity of the decision problem for firing sequences in terms of the problem $\text{MAJ}$.

Theorem 6. Given a Petri net, the problem of deciding whether a word of $T^*$ is a firing sequence is $\text{AC}^0$-reducible to $\text{MAJ}$. Moreover, there is a Petri net such that $\text{MAJ}$ is $\text{AC}^0$-reducible to its firing sequence decision problem.

Thus this problem belongs to $\text{TC}^0 \subseteq \text{NC}^1$, but is not in $\text{AC}^0$. A proof based on expressibility of first order logics with majority quantifiers is given in [6], and explicit efficient parallel algorithms are given in [3].

3.2 Partially commutative monoids

In a monoprocessor environment, a concurrent system is fully described by the set of all sequential evolutions, which can be represented by (sets of) finite words of events called traces. It may be the case that the execution of event $x$ modifies the environment of event $y$; then these events are in conflict, and their order of execution is relevant. On the other hand, two events are independent if their order of execution is not relevant. This is formalized by means of a commutation relation in the alphabet of events. Thus two words are equivalent if one can be obtained from the other by repeatedly interchanging adjacent commuting events.

Thus a concurrent alphabet is a finite alphabet denoting the set of events, endowed with an irreflexive, symmetric binary relation called the commutation relation, extended in the natural way to a congruence over the free monoid. The quotient is called a partially commutative monoid [26], and its elements are called traces. A basic decision
problem here is: given a concurrent alphabet and two words, are they equivalent in this congruence? That is, do they model the same concurrent behavior?

A characterization of trace equivalence found in [18] can be used to obtain a fast sequential algorithm. It is based on the projection functions over subsets of the alphabet. Regarding parallel algorithms, the following result from [6] exactly classifies this problem:

**Theorem 7.** Given a concurrent alphabet, the problem of deciding whether two words are equivalent is $AC^0$-reducible to MAJ. Moreover, there is a concurrent alphabet such that MAJ is $AC^0$-reducible to its trace equivalence problem.

Therefore the trace equivalence problem belongs also to TC$^0$. Again the fact that MAJ is not in AC$^0$ implies the same lower bound for this problem: it cannot be solved in constant time by a PRAM with a polynomial number of processors. A logic-based proof is given in [6], while [3] actually presents an efficient parallel algorithm.

### 3.3 Bisimulation

Several fundamental and widely studied problems from the area of concurrent systems consist of deciding some form of equivalence of finite state systems. A capability that some people would expect from software tools for aiding the design of concurrent systems is to be able to solve such problems. Indeed, efficient sequential algorithms to solve such problems exist [23]. The crucial concept is bisimulation (defined below). A natural question to ask is: do the automatic bisimulation techniques admit fast parallel algorithms?

Unfortunately we provide strong evidence that the answer is negative. More precisely, deciding bisimulation in finite transition systems is P-complete.

A **finite labelled transition system** consists of a finite set of states and a finite alphabet of actions, related by a set of transitions similar to those of finite automata: a transition connects two states under the label of an action.

Given two states $p$ and $q$, the idea of having the same behavior is formalized by the notion of strong bisimulation: two states $p$ and $q$ are strongly bisimilar if and only if:

(i) whenever $p \xrightarrow{e} p'$ for any $p'$, then for some $q'$, $q \xrightarrow{e} q'$ and $p'$ and $q'$ are also bisimilar;

(ii) whenever $q \xrightarrow{e} q'$ for any $q'$, then for some $p'$, $p \xrightarrow{e} p'$ and $p'$ and $q'$ are also bisimilar; and

(iii) the relation is the coarsest possible fulfilling (i) and (ii).

Other relations can be defined similarly using "invisible actions". The decisional problems for these other notions are equivalent to the decision of strong bisimulations.

Then we have the following P-completeness result:

**Theorem 8.** The following problem is P-complete: given a finite transition system with two selected states $p^*$ and $q^*$, decide whether $p^*$ and $q^*$ are strongly bisimilar.

To prove that it is in P, it suffices to consider a construction in which, starting from a single equivalence class, the classes are partitioned when witnesses are found that two states cannot be bisimilar. More efficient algorithms are given in [23]. The completeness result follows from an NC reduction from the Monotone Alternating Circuit Value Problem. The reduction is given in [3] and [9]; it consists of looking at each gate as a state of a finite labelled transition system over a single letter and adding a number of gadgets and auxiliary states ensuring that gates at the same level yield bisimilar states if and only if they evaluate both to true or both to false.

Other related results can be obtained by the same proof idea, proving the P-completeness of the decision problems for the properties named Observation Equivalence and Observation Congruence.
4 Some relationships

Several characterisations 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, and 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 ([20], [21]).

Here we will show that also in our model the adaptive queries to functional oracles can be understood as dividing the computation process into phases. Thus, 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 in this way new characterizations of the AC and NC classes. More precisely, we explain first how the depth of unbounded fan-in circuits corresponds to the number of functional queries. A characterization in terms of bounded fan-in circuits also exists, and will be presented in a later subsection.

4.1 Logarithmic space versus AC

We present next a characterisation in terms of unbounded fan-in circuits. Many more details of the proof can be found in [4], [5].

Theorem 9. Let \( F \) be a function class. Then for all \( k \geq 0 \)

\[
\text{AC}^k(\text{FL}(F)) = \text{FL}_{\log^k}[\text{FL}(F)]
\]

Proof (sketch). For the inclusion from left to right, let the function \( h \) be computed by the logspace uniform family \( \{\alpha_n\} \) of \( \text{AC}^k \) circuits with oracle nodes for a function \( f \in \text{FL}(F) \). We will show how \( \alpha_n \) can be evaluated in exactly as many phases as its depth: in phase \( i \) all the gates of depth at most \( i \) will be already evaluated.

For this, the functional oracle \( g \) is chosen such that it reproduces the complete description of the circuit \( \alpha_n \), supplemented by the values of those oracle gates which are directly evaluable with the information obtained so far:

Input: \((\alpha_n)\mathcal{F}(o_1, o_2, \ldots, o_m)\mathcal{F}(x_1, x_2, \ldots, x_m)\),

where \((\alpha_n)\mathcal{F}\) is a description of a circuit \( \alpha_n \) with oracle nodes in \( f \), where all of the gates up to a certain depth are evaluated, i.e., have their value attached, \((o_1, o_2, \ldots, o_m)\) is a list of oracle gates of the circuit, and \(x_1, x_2, \ldots, x_m \in \{0, 1\}^* \) is a list of inputs to the oracle gates in the list, \( z_i \) being the input for gate \( o_i \);

Output: \((\alpha_n)\mathcal{F}(y_1, y_2, \ldots, y_m)\),

where \( y_j = f(x_j) \) for all \( 1 \leq j \leq m \).

The base machine keeps reading out the description provided by the oracle, which has \( \alpha_n \) evaluated up to depth \( i - 1 \) plus the output of all oracle gates of depth \( i \).

It produces a new, updated description by appending to the oracle gates their values and evaluating all “normal” gates of depth \( i \). It also creates a list of all those oracle gates whose inputs are now known, as well as a list of their inputs. This constitutes the next query. Logarithmic space suffices to perform these computations. After repeating this process \( c \cdot \log^k n \) times, for some constant \( c \), all gates are evaluated, and the value \( h(z) \) can be produced on the output tape.

For the inclusion from right to left, the computation of a logspace Turing transducer \( M \) with \( c \cdot \log^k n \) adaptive queries to a functional oracle \( g \in \text{FL}(F) \) can be decomposed in \( c \cdot \log^k n \) iterations of a function \( h \in \text{FL}(g) \) which, given the input \( z \), a configuration of the machine in the oracle answer state, the output up to that configuration, and the answer, outputs again the same four data for the next oracle answer configuration. Then we can construct a trivially structured \( \text{AC}^k \).
circuit consisting of a line of \( c \cdot \log^k n \) functional oracle gates for the function \( h \). The output of \( M \) is extracted from the output of the last oracle answer.

Theorem 9 implies that for any function class \( \mathcal{F} \) closed under \( \text{FL}(\cdot) \) reducibility, \( AC^k(\mathcal{F}) = \text{FL}_{\log^k}[\mathcal{F}] \). Using corollary 2 and proposition 3, we get:

**Corollary 10.** For all \( k \geq 0 \),

(i) \( AC^k(\text{FL}) = \text{FL}_{\log^k}[\text{FL}] = \text{FL}_{\log^k}[\{\text{id}\}] \);

(ii) \( AC^k(\text{FNL}) = \text{FL}_{\log^k}[\text{FNL}] \);

(iii) \( AC^k(\text{NC}^i) = \text{FL}_{\log^k}[\text{NC}^i] \) for \( i \geq 2 \);

(iv) \( AC^k(\text{AC}^j) = \text{FL}_{\log^k}[\text{AC}^j] \) for \( j \geq 1 \).

(v) \( AC^k(\text{TC}^j) = \text{FL}_{\log^k}[\text{TC}^j] \) for \( l \geq 1 \).

The second equality in (i) can be proved by arguing that the oracle gates of a \( AC^k(\text{FL}) \) circuit can be evaluated as well by the logspace base machine.

A stronger version of Theorem 9 can be obtained as follows: if \( \mathcal{F} \) is a function class that contains \( \text{FL} \) and is closed under join, marked replication, and marked concatenation, then

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

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.

### 4.2 Logarithmic space versus NC

To complete the view, we show now that we can also characterize parallel classes defined by bounded fan-in circuits. Again the proof is only sketched; it can be found in [4], [5].

**Theorem 11.** For any \( k \geq 0 \),

(i) \( \text{NC}^{k+1}(\text{FL}) \subseteq \text{FL}_{\log^k}[\text{FL}] \);

(ii) \( \text{NC}^{k+1}(\text{FNL}) \subseteq \text{FL}_{\log^k}[\text{FNL}] \).

**Proof (sketch).** We will sketch the proof of (i). The proof for (ii) is similar.

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 more or less logarithmic depth, taking into account the contribution of the oracle nodes to the total depth. The evaluation principle is however similar to that of Theorem 9, although we have to evaluate more than one level of gates in each phase.

Define the weight of a gate 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 gate \( i \) relative to a list of gates \( l = (i_0, i_1, \ldots, i_n) \) as the maximal sum of the weights of all gates on a path from a node of \( l \) to \( i \) that does not pass through any other gate of \( l \), counting the weight of \( i \) but not that of the gate of \( l \). Note that any path from \( l \) to \( i \) can be recorded with as many bits as the weighted depth of \( i \) relative to \( l \) (see also [11]). The log space machine \( T \) will query a functional oracle \( \log^k n \) times, so that after phase \( j \) all gates with weighted depth up to \( j \cdot c \cdot \log n \) relative to the input gates have been evaluated, for the appropriate constant \( c \).

The functional oracle \( h \) is defined as follows:

**Input:** A description of the partially evaluated circuit \( \alpha_n \) with some gates marked; these marked gates build a subcircuit (with oracle nodes) of depth at most \( c \cdot \log n \);

**Output:** Another description of \( \alpha_n \) where all gates \( i \) that were marked in the input description have now their correct value attached.

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

**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 \( \alpha_n \) 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;
Evaluation: a query to the oracle \( h \) yields another description of \( \alpha_n \), where all the marks set during the marking process are replaced by the appropriate values of the gates.

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 \( f(x) \).

It can be seen that 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 [17]).

To show that the marking process can be performed within logarithmic space, we use an argument that is implicit in the proof of NC\(^1\)(FL) = FL in [17] (or [11]). 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 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 \( i \) not yet evaluated 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 \( i \).  

From Theorems 9 and 11 we get several interesting corollaries. The first one is the affirmative answer [4] to questions left open by Wilson [37]: the reducibilities \( \text{AC}^k \) and \( \text{NC}^{k+1} \) coincide on the oracle classes \( L \) and \( NL \).

**Corollary 12.** For all \( k \geq 0 \),

\[
\begin{align*}
(\text{i}) \quad & \text{AC}^k(L) = \text{AC}^k(FL) = \text{NC}^{k+1}(FL) = \\
& \text{NC}^{k+1}(L) = \text{FL}_{\log^k} [FL] = \text{FL}_{\log^k} \{id\}; \\
(\text{ii}) \quad & \text{AC}^k(NL) = \text{AC}^k(FNL) = \\
& \text{NC}^{k+1}(NL) = \text{NC}^{k+1}(FNL) = \\
& \text{FL}_{\log^k} [FNL].
\end{align*}
\]

Note that for \( k = 0 \) we have: \( \text{AC}^0(FL) = \text{NC}^1(FL) = \bigcup_c \text{FL}_c[FL] = \text{FL} \) and \( \text{AC}^0(FNL) = \text{NC}^1(FNL) = NL^* = \bigcup_c \text{FL}_c[FNL] = \text{FNL} \).

Another interesting consequence is that the \( \text{FL}_{\log^k} \{\cdot\} \) operator is able to provide another analogy between the NC and the AC hierarchies, since both are built up in exactly the same manner using this operator, and we obtain one or the other just depending on the starting class.

**Corollary 13.** For all \( k \geq 0 \),

\[
\text{NC}^{k+2} = \text{FL}_{\log^k} \{\text{NC}^2\} = \text{FL}_{\log^k} \{\text{NC}^j\}
\]

with \( j \geq 2 \) and \( i + j = k + 2 \), and

\[
\text{AC}^{k+1} = \text{FL}_{\log^k} \{\text{AC}^1\} = \text{FL}_{\log^k} \{\text{AC}^j\}
\]

with \( j \geq 1 \) and \( i + j = k + 1 \). Thus in particular, \( \text{AC}^{k+1} = \text{FL}_{\log^k} \{\text{AC}^2\} \) for all \( k \geq 1 \), \( \text{NC}^{k+1} = \text{FL}_{\log^k} \{\text{NC}^2\} \) for all \( k \geq 2 \), and \( \text{NC} = \text{AC} = \bigcup_k \text{FL}_{\log^k} \{\{id\}\} \).

Observe that although all the classes of the NC hierarchy from \( \text{NC}^3 \) on, as well as the classes \( \text{NL}^* = \text{NC}^1(\text{NL}) \) and \( \text{NC}^2(\text{NL}) \), have a characterization in our model, the class \( \text{NC}^2 \) does not.

## 5 The closures of NP

The reducibilities we are studying here provide also a way of characterizing several interesting complexity classes above the class \( \text{NP} \). These classes lie between the classes \( \Theta_2 \) and \( \Delta_2 \) of the polynomial time hierarchy, and have been defined by other concepts, namely natural extensions of the boolean hierarchy [14], [34] and “binary search” processes operating over NP oracles [2].

The main results of this section are presented in detail in [15].

Let us give one among many definitions of the classes \( \text{NP}(k) \) of the boolean hierarchy over \( \text{NP} \). According to this definition, \( A \in \text{NP}(k) \) if and only if there is in \( \text{NP} \) a set \( B \) of pairs \( (z, i) \) for \( 1 \leq i \leq k \) such that the following holds: first, \( B \) has a single “step down”, that is, \( (z, i) \in B \) implies \( (z, i−1) \in B \) for all \( i > 1 \); second, \( z \)'s membership in \( A \) is given by the parity of the place where the step down appears, that is, \( z \in A \) if and only if the unique \( i \) with \( (z, i) \in B \) and \( (z, i + 1) \notin \).
\( B \) is odd. It is known that for polynomial time machines, \( k \) nonadaptive queries to an oracle in NP give the same power as one query to an oracle in NP(\( k \)) [10].

This relationship holds even if \( k \) is allowed to depend on the length of the input, provided that it is polynomial time computable. Additionally, a characterization in terms of adaptive queries exists: if a function \( r \) grows at most logarithmically, then \( r(n) \) adaptive queries to NP are as powerful as \( 2^r(n) - 1 \) nonadaptive queries, and thus as powerful as a class NP(\( 2^r(n) + O(1) \)) [10], [14], [34]. Thus a polynomially growing extension of the boolean hierarchy characterizes the class \( \Theta_2 \). A similar characterization exists for \( \Delta_2 \) [34].

In [2] it is observed that proofs involving such sets presenting exactly one step down can be frequently casted as a form of binary search over a segment of the oracle. They take advantage of this intuition to define width-bounded and size-bounded reducibilities, so that for appropriate classes \( C \) they are able to show that \( C(O(b(n))) \) is the closure of the class \( C \) under reducibilities computed by circuits of size \( \log b(n) + O(1) \). In particular, \( \Theta_2 \) is again characterized as the closure of NP under reducibilities computed by circuits of size \( O(\log n) \); thus a characterization of \( \Theta_2 \) by means of circuits is obtained. It should be pointed out that in order to define width-bounded and size-bounded reducibilities, wires coming from the inputs are not charged. See [2] for a motivation of this decision.

It is natural to ask whether other characterizations can be obtained, and principally what classes are obtained from depth-bounded reducibilities to NP sets. It is to be observed also that the results of [2] and [34] are proved only for bounds obeying certain limitations; in particular, \( \text{NP}(b) \) is characterized only for \( b \) polynomially bounded above, or in [34] for exactly exponential.

The map has been thoroughly completed in [15], where a census-counting technique is applied to characterize the reduction classes of NP under depth-bounded reductions, as well as to generalize results from [2] and [34] to arbitrary (meaningful) bounding functions. More precisely, they prove the following:

**Theorem 14.** The following classes coincide:

(i) the class of sets reducible to NP by polynomial time machines making \( O(\log^k n) \) adaptive queries;

(ii) the class of sets reducible to NP by AC\(^{k-1} \) reducibilities;

(iii) the class of sets reducible to NP by reducibilities computed by circuits of size \( O(\log^k n) \);

(iv) NP(\( 2^{O(\log^k n)} \)).

Notice also that the class of sets reducible to NP by reducibilities computed by circuits of polynomial size is \( \Delta_2 \). Similar results can be proved using the same techniques for other bounding functions. As a particular consequence, for example, let us point out that with an appropriate uniformity notion, AC\(^O \)(NP) again characterizes \( \Theta_2 \). This fits very well the characterization of AC\(^O \) as the first-order definable structures given in [27], since \( \Theta_2 \) is precisely the set of structures definable by the closure of second order existential logic (i.e. NP) by first-order constructions [14]. Characterizations of NC(NP) and relativized separations of the NC and AC hierarchies over NP have also been obtained (Castro and Seara, personal communication; see also the next section).

An additional set of characterizations in terms of adaptive log space reducibility can be obtained from theorem 9, applying the fact that AC\(^k \)(FL(NP)) = AC\(^k \)(NP):

**Corollary 15.** The sets whose characteristic functions belong to FL\(_{\log^k} \)(FL(NP)) are precisely those in the class AC\(^k \)(NP).

In fact, the argument can be also presented the other way around, proving a theorem similar to theorem 14 in terms of FL\(_{\log^k} \)[FL(NP)] and then using this last corollary (Jenner, personal communication).
6 A generalization

The careful reader of section 4 has probably noticed that theorems 9 and 11 are of very different level of generality: while the first just assumes some reasonably weak closure properties of a class in order to state the equivalence of two kinds of reducibilities to it, the second is only stated for the classes FL and FNL. The reason is that a bounded fan-in circuit with oracle gates presents some difficulties when attempting to decompose it into subcircuits of logarithmic depth. These difficulties arise from the fact that oracle gates form an undecomposable circuit of nontrivial depth, and it cannot be cut into pieces in an homogeneous, logarithmic space uniform manner.

In this section we state a yet unpublished generalization of theorem 11, giving a sufficient condition for the equivalence of $AC^k$ and $NC^{k+1}$ reducibilities to a class $C$. The author has been informed of it by B. Jenner (personal communication); the proof idea is essentially different from the one used in section 4, and has been suggested and pursued by Peter Rossmannith.

This idea consists of replacing each oracle gate with $m$ inputs in any NC circuit by a subcircuit consisting of the oracle gate followed by a chain of $\log m$ trivial identity gates. For a circuit transformed in this manner, it is no longer necessary to charge depth more than 1 for each gate, including oracle gates, and thus the problem that some gates contribute different depths to the total depth of the circuit no longer arises. There is still a uniformity problem, but now it can be solved by a leveling technique such as that of [12] as used in [37]. The following then holds:

Theorem 16. If $C$ is closed under $NC^1(-)$ reducibility, then $NC^{k+1}(C) = AC^k(C)$.

Of course this encompasses theorem 11. Moreover, it allows us also to obtain additional characterizations of the closures of NP discussed in the last section, since it can be applied to the class $NC^1(NP)$. A detailed proof and a number of applications will appear in the near future in an appropriately authored paper.

Observe however that, as pointed out in [15], it is not clear at all yet that $NC^1(NP) = AC^0(NP)$ (recall from the previous section that $AC^0(NP) = \Theta_2$). Of course an appropriate uniformity notion should be used here [27]. Proving the equality $NC^1(NP) = AC^0(NP)$ would be extremely interesting, since in principle the left hand side has a higher adaptiveness capability that the right hand side since it can include more than a constant number of oracle gates. The fact that in this case the oracle gates should have sublinearly many inputs still gives some room for discussion. On the contrary, disproving the equality $NC^1(NP) = AC^0(NP)$ (at least relativized) would be very interesting as well, since in this case we would obtain that when the unrelativized chain $AC^0 \subseteq NC^1 \subseteq L$ is relativized to NP, the first and third classes would coincide, being equal to $\Theta_2$, but the middle one would not.

Acknowledgements Building on work of other researchers, the results reported here were obtained at the author’s department by C. Álvares, J. Balcázar, J. Castro, J. Gabarró, B. Jenner (visiting from Hamburg), M. Sántha (visiting from Paris), and C. Seara. I am grateful to them for allowing me to steal plenty of complete paragraphs from their writing without a single complaint. Additionally, useful discussions with and repeated technical (and human) help from E. Allender, R. Book, B. Kitaig, P. Rossmannith, L. Rusu, J. Torán, C. Wilson, and others is gratefully acknowledged. I am indebted also to C. Álvares, J. Castro, J. Díaz, J. Gabarró, F. Green, B. Jenner, and C. Seara for proofreading under exceedingly pressing deadlines.

References


