The Computational Complexity Column

by

Jacobo Torán

Dept. Theoretische Informatik, Universität Ulm
Oberer Eselsberg, 89069 Ulm, Germany
`jacobo.toran@uni-ulm.de`
`http://theorie.informatik.uni-ulm.de/Personen/jt.html`

Polynomial size log depth circuits: Between NC^1 and AC^1

Meena Mahajan*

Abstract

1 Introduction

When a theoretical computer scientist asks me my area of research, I usually say complexity theory. This is often followed by the question "what kind of complexity theory" to which I inevitably reply "inside P". And usually the questioning stops there. In this brief survey, I would like to go further, and describe some of my favourite complexity classes. They all lie in the range between NC^1 and AC^1; hence this title. I cannot even begin to attempt being

*The Institute of Mathematical Sciences, CIT Campus, Chennai 600041, India.

exhaustive, and I apologize in advance to those whose favourite results I have omitted. Much of this material (and much more!) can be found in the text [42] and the surveys [1, 23].

2 Principal classes between NC^1 and AC^1

Consider (uniform) families of polynomial size log depth circuits with internal AND and OR gates, and literals / constants at the leaves. (No internal negations, without loss of generality). Restricting the gates to have constant fanin gives the complexity class NC^1; leaving it unrestricted (limited, of course, by the circuit size itself) gives AC^1.

Without loss of generality, we can assume that our circuits are layered: gates appear in layers, and wires connect adjacent layers in one direction. The maximum number of gates at any one layer of the circuit is called the width of the circuit.

Several well-known classes are sandwiched in between; let's take a look at each of these.

$$\begin{tikzcd}
 & & \mathrm{LogDCFL} \arrow[dr] & & \\
\mathrm{NC}^1 \arrow[r] & \mathrm{DLOG} \arrow[ur] \arrow[dr] & & \mathrm{LogCFL} \arrow[r] & \mathrm{AC}^1 \\
 & & \mathrm{NLOG} \arrow[ur] & &
\end{tikzcd}$$

Figure 1: The landscape between NC^1 and AC^1

2.1 NC^1

At the lower end, we begin with the class NC^1. NC^1 has many equivalent characterizations. It equals the class of languages accepted by

- alternating Turing machines in logarithmic time ALOGTIME (under appropriate uniformity conditions). ([34])

- poly size programs over finite monoids BP-M.
  A program over a monoid $M = (S, \circ)$ is a list of instructions of the form $\langle i, a, b\rangle$ where $i \in [n]$, $a, b \in S$. The instruction $\langle i, a, b\rangle$, on input $x \in \{0,1\}^n$, evaluates to a if $x_i = 1$, to b otherwise. The entire program,

on input $x \in \{0,1\}^n$, thus constructs a word $w \in S^*$. The program accepts x if w evaluates to a designated value in M.

A simple divide-and-conquer approach establishes that programs over finite monoids can be evaluated in NC^1. Barrington [4] established the converse by showing that over any non-solvable group, words can be constructed to code logical AND and negation. In particular, he used the permutation group S_5; thus programs over this group are complete for NC^1.

- bounded-width poly size branching programs BWBP.
  These are constant-width layered graphs (one for each input length) with designated start and finish vertices s, t, and edges labelled by literals or constants. An input is accepted if the corresponding graph has an st path where all edges are labelled by 1 or true literals.

  It is folklore (and easy to see) that programs over monoids can be described in this form, and vice versa.

  Such programs are also equivalent to skew circuits, where OR gates are unrestricted but AND gates can have at most one input that is not a literal or a constant.

- bounded-width poly size circuits SC^0.
  SC is the class of polynomial size poly logarithmic width circuits (width $O(\log^i n)$ for SCi). (Again, wlog the circuits can be assumed to have negations only at the leaves.) In simulating a circuit by a Turing machine, width roughly translates to space and size to time; thus SC corresponds in the uniform setting to a simultaneous time-space bound. (SC stands for Steve's Classes, named after Stephen Cook who proved the first non-trivial result about polynomial time log-squared space PLoSS, i.e. SC2, in [11]. See for instance [22]). As described above, Barrington's result places NC^1 inside skew SC^0. But even non-skew SC^0 is easily seen to be inside NC^1, since only a constant amount of memory is needed to evaluate the gates of the circuit layer by layer. Thus SC^0 equals NC^1.

- poly size formulae F, even when restricted to log-width formula LWF.
  A formula is a circuit where each gate has fanout at most 1. NC^1 circuits can be converted to formulae by duplication; the blow-up in size is still within a polynomial. Conversely, any formula can be restructured into an equivalent one with polynomial blow-up in size and logarithmic depth; a non-uniform way to do this was first described in [7, 35], while it can be done uniformly as in [28]. Further, any log depth formula can

be restructured to log width (at the expense of depth, of course), as observed in [20]; thus LWF = F = NC^1.

- predicates expressed in first-order logic, augmented with a group quantifier or a monoidal quantifier Q_G over any non-solvable group G, FO[Q_G]. [28].

2.2 AC^1

At the higher end, we have AC^1. Less is known about AC^1; it equals the class of languages accepted by

- alternating Turing machines using logarithmic space and making at most logarithmic alternations between universal and existential states on any computation path ASP,ALT(log,log) (under appropriate uniformity conditions).

- Concurrent read, concurrent write PRAMs working in logarithmic time with polynomially many processors.

Now consider the intermediate classes.

2.3 DLOG

DLOG is the class of languages accepted by deterministic logspace machines. It also equals the class of languages accepted by log width poly size circuits SC^1. DLOG equals sentences expressible in FO augmented with deterministic transitive closure FO[DTC], [18], and it follows from [32] that DLOG also equals FO + symmetric transitive closure FO[STC].

2.4 NLOG

NLOG is the class of languages accepted by nondeterministic logspace machines. The inductive counting technique of Immerman and Szelepcsényi [19, 37] shows that NLOG is closed under complementation. An equivalent formulations of NLOG is the class of languages accepted by uniform poly size skew circuits (or branching programs); see [39]. In descriptive complexity, NLOG is characterized by sentences in first-order logic with positive transitive closure FO[pos TC], see [18].

2.5 LogCFL

LogCFL, by definition, is the class of languages reducible via logspace many-one reductions to some context-free language. It follows that each such language can be accepted by a machine which has logspace to perform the reduction, and a nondeterministic finite control and a stack to then parse the CFL in polynomial time. Such machines are called AuxPDA(poly), and Sudborough showed that they accept exactly LogCFL ([36]). That is, allowing arbitrary interleaving of the two types of computation involved – (1) deterministic logspace reduction, and (2) nondeterministic PDA – is no more powerful than performing these two phases sequentially. (An aside: the polynomial time restriction is necessary, since Cook [10] showed that in unbounded time, and even in exponential time, deterministic PDA augmented with logspace worktape capture all of P.)

Using the notion of realizable pairs of surface configurations, Ruzzo showed [33] that AuxPDA(poly) can be simulated by alternating TMs using logspace and having poly-sized proof trees. What is a proof-tree? Consider the computation graph of a logspace-bounded ATM, where nodes are time-stamped configurations. (The logspace bound ensures a poly-sized graph; the time-stamping ensures that the graph is acyclic.) To prove that it accepts its input, it suffices to show a sub-graph that contains (1) the initial configuration, (2) both children of each universal node included, (3) at least one child of each existential node included, and (4) only accepting configurations as leaves. Such a sub-graph, unfolded or expanded out by duplicating nodes if necessary so that it is a tree, is what we call a proof-tree. It is easy to see that poly-sized graphs can have exponential-sized proof-trees. Ruzzo's proof shows that to describe the computations of AuxPDA(poly), poly-sized proof trees suffice. Conversely, if a logspace-bounded ATM has, for each accepted input, a proof-tree of size at most $t(n)$, then an AuxPDA can accept the same language in time $t(n)$. Thus we have a characterization of LogCFL via ATMs: LogCFL = ASP,TRSZ(log,poly). Note that the above proof-tree definition can be applied to circuits as well. Using a very nice tree-cutting argument, Venkateswaran showed [38] that a poly-sized circuit of any depth, but with a poly-size bound on its proof trees, can be flattened to log depth, at the cost of increasing the fanin of OR gates. This is the circuit class SAC^1, semi-unbounded alternating circuits. The converse simulation is direct, giving ASP,TRSZ(log,poly)=SAC^1.

An interesting offshoot of Venkateswaran's construction is that each AuxPDA(poly) can be simulated by an AuxPDA(poly) whose stack height never grows beyond $O(\log^2 n)$. (Only $O(\log n)$ pairs of surface configurations, each needing $O(\log n)$ bits, need to be stacked.)

More recently, in [27], McKenzie, Rienhardt and Vinay gave a direct proof that ASP,TRSZ(log,poly) is in LogCFL, thus eliminating the need for the elaborate construction of Sudborough.

The class of all CFLs is not closed under complementation. Nonetheless, one could expect that a logspace reduction closure captures complements as well, and indeed this is the case. Interestingly, none of the above forms directly show that LogCFL is closed under complementation. The SAC^1 formulation was used by Borodin et al [6] to apply inductive counting and thus establish this closure. This closure captures a certain symmetry between the OR and AND operators: as long as one of them has bounded arity, we are within LogCFL.

Bedard, Lemieux and McKenzie gave yet another characterization of LogCFL in [5]. Generalising the programs-over-monoids framework of Barrington, they show that LogCFL equals languages accepted by programs over groupoids. These are algebraic structures where a non-associative binary operator * on a set A is defined. Given a word $w \in A^*$, consider all possible ways of parenthesising it to apply *. These different ways yield a set of possible values S(w). Acceptance is defined in terms of S(w) containing some designated element, or equalling some designated set. By imposing syntactic conditions on programs over groupoids, NC^1, DLOG and NLOG can also be captured in this framework [5, 25].

The framework of [5] directly leads to a logical characterization as well: LogCFL is exactly those languages whose membership is expressible in first-order logic augmented by groupoidal quantifiers. A more detailed treatment of this characterization can be found in [24].

2.6 LogDCFL

LogDCFL is the class of languages reducible via logspace many-one reductions to some deterministic context-free language. As in the case of LogCFL, the two computation phases in deciding membership in a LogDCFL language can be interleaved [36]; thus LogDCFL equals DAuxPDA(poly). It is also characterized in the PRAM model: it is the restriction of AC^1 to concurrent read owner-write (CROW) PRAMs, see [15, 16]. One of the most non-trivial properties about LogDCFL is that it is contained in SC^2; this was shown by Cook in [11]. No subclass of NC containing LogDCFL is known to be inside SC, though a possibly incomparable chunk of NC consisting of randomized (bounded two-sided error) logspace is also known to be in SC [30]. Surprisingly, we do not yet know how to combine these two constructions to place randomized poly time AuxPDA inside SC.

2.7 A formal language view

For many reasons, AC^1 is not as interesting formally as the classes within it. The main reason is to do with proof-tree size: AC^1 circuits can have exponentially large proof trees. This crucially impacts arithmetic versions of these circuits; we will come to that shortly. Another is that there is no neat characterization of AC^1 via formal language classes. From the formal-language-theoretic point of view, we have the following containment diagram:

$$\begin{tikzcd}
 & & \text{DCFL} \arrow[dr] & \\
\text{Regular} \arrow[r] & \text{Det Linear CFLs: DLin} \arrow[ur] \arrow[dr] & & \text{CFLs} \\
 & & \text{Linear CFLs: Lin} \arrow[ur] &
\end{tikzcd}$$

Figure 2: Formal Language Classes

All the containments are proper, and DCFL and Lin are incomparable. Applying very weak closures to these classes – uniform FO projections – gives exactly the complexity classes of Figure 1. Notice that AC^1 is not covered here. The jump from CFLs to context-sensitive languages is too big; closure of CSLs gives all of PSPACE. We need something much smaller to capture exactly AC^1.

2.8 Completeness

Here is a partial list of problems complete for each of these classes:

LogCFL BlockChoice(Dyck-2), the hardest CFL: Given a sequence of blocks, each containing a list of strings, can we pick exactly one string from each block so that their concatenation, in that order, is in Dyck-2 (the language of balanced parentheses with two types of parentheses)?
Non-zero Tame Tensor Formula [13]: Given a tensor formula satisfying a certain "tameness" property, determine whether it is non-zero.
Semi-extended regular expression membership [31]: Given an expression r over some alphabet Σ where r is like a regular expression but is also allowed to use $\cap$, and given a string $x \in \Sigma^*$, determine whether $x \in L(r)$.

NLOG Reachability in a directed acyclic graph.
2-CNF-SAT.

Regular expression membership [21]: Given a regular expression r over some alphabet Σ and a string $x \in \Sigma^*$, determine whether $x \in L(r)$.

LogDCFL BlockChoice(Dyck-2), the hardest DCFL: let Dyck-2 be over $\{a, b, c, d\}$ with a and c opening, and matched by b and d respectively. Given a string $x_0 \in (a + c)^+$, and a sequence of blocks $B_1, \ldots, B_k$ each consisting of one string in $b(a + c)^*$ and one in $d(a + c)^*$, can we pick exactly one string from each block so that their concatenation, with x_0, is in Dyck-2?

DLOG Reachability in an undirected graph, presented by its adjacency lists. [32]
Remains hard even if the graph is a two-tree forest. [12]
Bipartiteness: given an undirected graph, determine if it is bipartite.

NC^1 Reachability in a bounded-width layered graph.
The Boolean Formula Value problem.
The word problem over the group S_5 (for that matter, over any finite non-solvable monoid).
Fixed Regular expression membership: For a fixed regular expression r over some alphabet Σ, given a string $x \in \Sigma^*$, determine whether $x \in L(r)$.

3 Lesser-known classes

By varying parameters appropriately between NC^1 and LogCFL, we get some lesser-known classes in this range:

3.1 Syntactic restrictions

- BP-width: Within polysize, constant-width BPs gives NC^1 and unbounded width BPs give NLOG. One could thus consider width $w(n)$ BPs, for w a function of n. Vinay showed [41] that for each polylog $w \in O(\log n^i)$, the corresponding class is closed under complement. But nothing much more is known. For instance, even the smallest class here, log-width BPs, lying between NC^1 and DLOG, is not known to capture any natural problem in this range.

- OR fanin: Within poly size log depth circuits with constant AND fanin, varying OR fanin from constant to polynomial takes us from NC^1 to SAC^1 (i.e. LogCFL). What about OR fanin $f(n)$ where f is, say, polylog? Again, Vinay showed closure under complement, [41]. Also, just as SAC^1 contains NLOG (at f = poly), each of these classes contains the corresponding BP-width-constrained class described above. But do they capture any natural problems?

- Circuit-width: Constraining circuit width alone to polylog gives the SC hierarchy, and very little of the NC hierarchy is known to lie within it. However, the defining property separating LogCFL from P is poly size proof trees (also referred to as poly degree). One could combine a width restriction with a degree restriction to obtain a sub-hierarchy of SC within LogCFL. Limaye et al [26] define what they call small SC denoted sSC: its ith level has poly size poly degree $O(\log^i)$ width circuits. Again, each level here contains the corresponding width-constrained BPs, though no relationship with the constrained-OR-fanin circuits is known. Though these classes are not yet known to be closed under complement, [26] shows that co-sSC^i is in sSC^{2i}. At the smallest level, sSC^0 equals SC^0, but it is not known whether sSC^1 is as powerful as SC^1.

3.2 Language/Automata-theoretic constructs

- Let us take a closer look at Figures 1, 2. NC^1 equals the closure of regular languages. Yet some non-trivial non-regular CFL families are included in it. These include parenthesis languages [8], visibly pushdown languages VPLs [3, 14], linear CFLs with an LL[1] condition [17]. (Imposing an LR[1] condition is what corresponds to determinism. Thus CFLs with an LR[1] condition equal DCFLs, linear CFLs with an LR[1] condition equal languages accepted by 1-turn DPDA, usually referred to as deterministic linear languages.)

  Let me highlight the membership in NC^1 of VPLs. Firstly, what are VPLs? These are languages accepted by visibly pushdown automata VPAs. So what are VPAs? These are PDAs with no ε moves, where the stack movement (push / no change / pop) is dictated solely by the input letter being read. They are clearly stronger than NFAs (they can accept a^nb^n: push on a, pop on b), but also weaker than PDAs (they cannot accept a^nba^n: is a a push letter or a pop letter?). In [3], it was shown that VPAs can be determinized; thus VPLs are in DCFLs. But well before this was known, these languages had been studied under the name input-driven languages. Dymond gave a nice construction [14] showing that they are in fact in NC^1. His approach is generic and works not just for VPAs but for any PDA satisfying the following:

1. no ε moves,
2. an accepting run should end with an empty stack,
3. the height of the pushdown, after processing i letters of the input, should be computable in NC^1. If there is more than one run (nondeterministic PDA), and if the height profiles across different runs are different, then the heights computed should be consistent with a single run. Furthermore, if there is an accepting run, then the heights computed should be consistent with some accepting run.

For such PDA, Dymond transforms the problem of recognition to an instance of formula value problem, and then invokes Buss's ALogTime algorithm [8] for it.

VPAs satisfy these conditions (with appropriate padding to satisfy condition (2)). But much more can be achieved via condition (3). The height profiles of all runs in a VPA are the same, and can be computed in TC0. Understanding exactly what can be placed inside NC^1 by carefully using Dymond's proof is a nice question.

An interesting proper generalization of VPAs are what Caucal introduced in [9] and calls synchronized PDA. Languages accepted by these are contained in DPDA but incomparable with DLin. Does their closure create a new class between NC^1 and LogDCFL, or does it collapse to one of these or even to DLOG?

- The fact that the logspace closure of Lin is NLOG is interesting. The machine model for Linear CFLs is PDA which, on each run, make at most 1 turn on the stack. That is, no stack symbols are pushed after the first pop move. Thus the machine model for the logspace closure of Lin is AuxPDA(poly) making 1-turn in stack movement. This suggests a fine gradation between NLOG and LogCFL parameterized by the number of turns the AuxPDA is allowed to make. A similar gradation arises between DLOG and LogDCFL by considering the deterministic counterpart.

3.3 Counting constructs

- Unambiguity: Between DLOG and NLOG lies, quite naturally, unambiguous logspace ULOG. Similarly, LogUCFL lies between LogDCFL and LogCFL. Interestingly, the correspondence between the formal language class and the complexity class is not known to hold here: the logspace closure of unambiguous CFLs viz. LogUCFL, is contained in unambiguous logspace machines UAuxPDA(poly), but the converse is not known, and similarly for unambiguous Linear CFLs. There are also close relationships in the PRAM model: while LogDCFL is characterized by log time CROW PRAMs, LogUCFL is contained in by log time CREW PRAMs, which correspond to a strong form of unambiguity in AC^1 circuits. There are several subtleties in the definition of unambiguous machines/circuits: is there at most one accepting path, or at most one path from the initial to any configuration, or at most one path between any pair of configurations? For more about these nuances, see [23, 29].

- Randomization: Between ULOG and NLOG lies one-sided-error randomized logspace RLOG: either none, or overwhelmingly many, accepting runs. Nisan showed [30] that RLOG (and even its two-sided-error version) is contained in SC^2. No class above RLOG is known to be in SC. A natural containment to expect, since LogDCFL is also in SC^2, is that randomized LogCFL, RLogCFL, is also in SC^2. So far this has not been shown to hold. But another interesting set of questions here is to do with the appropriate definition of RLogCFL itself. LogCFL has multiple characterizations, each of which can be randomized to give competing definitions for RLogCFL:

  1. a randomized logspace reduction to some CFL
  2. a randomized AuxPDA(poly) with bounded error
  3. a randomized AuxPDA(poly) with stack-height bounded by $O(\log^2 n)$ and bounded error
  4. an SAC^1 circuit with polynomially many supplementary random input bits, and bounded error

  Which of these truly reflects RLogCFL?

4 Arithmetization

In this section, I will briefly discuss arithmetizations of these classes over integers. There are two standard ways to arithmetize a circuit class: (1) assuming there are negations only at the leaves, replace AND and OR gates by $\times$ and $+$ gates respectively, or (2) count the number of proof-trees. Both give the same class of functions. Equivalent models can be appropriately arithmetized: For programs over monoids, consider an NFA corresponding to the monoid, view the instructions as projections transforming an input, and count the number of accepting paths of the NFA on the transformed version. For branching programs, count the number of *st* paths. For LogCFL, count the number of parse trees in the target CFL. For NLOG and AuxPDA(poly), count the number of accepting paths. And so on.

The arithmetization of AC^1 is not very interesting. Within log depth, a circuit can, starting with 0s and 1s, compute numbers that need exponentially many bits in their binary representation. This is because they can have exponentially large proof trees. For feasible computation, we may be justified in restricting attention to poly size arithmetic circuits that compute numbers with feasible representation. This corresponds to poly size circuits with poly degree, and over integers, is essentially the same as Valiant's class VP. It also corresponds directly to an arithmetization of one characterization of LogCFL, namely, ASP,TRSZ(log,poly).

Interestingly, all arithmetizations of LogCFL coincide: poly size poly degree arith circuits, poly size log depth $\#SAC^1$, number of accepting paths in AuxPDA(poly) machines, number of parse trees in a CFL, number of good parenthesizations of a word over a groupoid. Venkateswaran's tree-cutting construction [38] placing LogCFL in SAC^1 is not parsimonious; it does not give a one-to-one correspondence between accepting paths of the AuxPDA(poly) machine and proof trees of the SAC^1 circuit. To establish the equivalence of #AuxPDA(poly) and $\#SAC^1$, two independent and different constructions were described by [40] and [29] (see also [2]). These can also be thought of as tree-cutting, but the cuts are applied more carefully to uniquely halve the degree in a constant number of stages. Since these techniques apply to the Boolean case as well, we have three different proofs that LogCFL is in SAC^1.

Since unbounded addition and bounded multiplication are both in NC^1, it is easy to see that $\#SAC^1$ is in Boolean NC^2.

Over NLOG too, the two arithmetizations coincide: number of accepting paths in an NLOG machine, and number of proof trees in a poly size skew circuit, both give the function class #L.

For LogDCFL and DLOG, it is not clear how to define an arithmetic version. One possibility is to consider functional versions FLOG and FLogDCFL. But this is not entirely satisfactory because in this kind of framework, we expect arithmetization to yield more power. Another possibility for DLOG is to consider $\#SC^1$, since DLOG equals SC^1. But $\#SC^1$ can compute infeasible values, so this is an unreasonable class. Yet another possibility is to consider poly degree SC^1 circuits, $\#sSC^1$. But it is not even known whether sSC^1 is as powerful as SC^1, so we may be restricting ourselves too much this way. An interesting spin on $\#sSC^1$ is that it is contained in both $\#SAC^1$

and Boolean SC^2. Thus, inverting the question of "How much of NC is in SC?", it gives a piece of SC inside NC.

At NC^1, the picture is considerably murkier. Recall that Boolean NC^1 has multiple characterizations. $\#NC^1$ as log depth arithmetic circuits has been studied quite a bit since first formally defined in CMTV. There it is shown that #BWBP equals #BP-M (or #BP-NFA, as referred to in [26]), contains functional NC^1, and is contained in $\#NC^1$. But the reverse containments are still intriguingly open; we know that $\#NC^1$ can be computed by Boolean poly size circuits of bounded fanin and depth $O(\log n \log^* n)$, but the $log^* n$ factor remains. However, if the constant -1 is allowed, we get the classes $GapNC^1$ and GapBWBP, and these are known to coincide, and they are contained in FLOG.

It turns out that #LWF and #F both equal $\#NC^1$. On the other hand, #BP-VPA equals #BWBP: adding a visibly pushdown stack to an NFA not only does not increase the complexity of the language class, it does not increase the complexity of the counting function class as well. Another arithmetization of a class equivalent to NC^1 is $\#sSC^0$; this contains #BWBP and is contained in FLOG, but no relationship to $\#NC^1$ is known. These results are described in [26].

Language classes can be defined based on these arithmetizations; the predicates typically applied to NC^1 and NLOG are:

Is the # function greater than 0?	yielding NC^1, NLOG
Is the Gap function greater than 0?	yielding PNC^1, PL
Are two # (or Gap) functions equal?	yielding $C{=}NC^1$, $C{=}L$

With these predicates, the multitude of arithmetic classes around NC^1 gives rise to a host of language classes between NC^1 and DLOG. I hope that the true picture is considerably simpler.

Acknowledgements

The idea of writing this survey arose after a talk I gave at a Dagstuhl seminar on Circuits, Logic and Games in November 2006, and I thank Heribert Vollmer for the invitation to participate in the seminar. I gratefully acknowledge many helpful and insightful discussions over several occasions with Eric Allender, Nutan Limaye, Kamal Lodaya, Pierre McKenzie, Antoine Meyer, Raghavendra Rao, Jayalal Sarma, H Venkateswaran and V Vinay.

References

[1] E. Allender. Arithmetic circuits and counting complexity classes. In J. Krajicek, editor, *Complexity of Computations and Proofs*, Quaderni di Matematica Vol. 13, pages 33–72. Seconda Universita di Napoli, 2004. An earlier version appeared in the Complexity Theory Column, SIGACT News 28, 4 (Dec. 1997) pp. 2-15.

[2] E. Allender, J. Jiao, M. Mahajan, and V. Vinay. Non-commutative arithmetic circuits: depth reduction and size lower bounds. *Theoretical Computer Science*, 209:47–86, 1998.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

[4] D. Barrington. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[5] F. Bedard, F. Lemieux, and P. McKenzie. Extensions to Barrington's M-program model. *Theoretical Computer Science*, 107:31–61, 1993.

[6] A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal of Computation*, 18(3):559–578, 1989.

[7] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.

[8] S. Buss. The Boolean formula value problem is in ALOGTIME. In *STOC*, pages 123–131, 1987.

[9] D. Caucal. Synchronization of pushdown automata. In *Proc. 10th Developments in Language Theory Conference, LNCS 4036.* Springer, 2006.

[10] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of Assoc. Comput. Mach.*, 18:4–18, 1971.

[11] S. A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *STOC*, pages 338–345, 1979.

[12] S. A. Cook and P. McKenzie. Problems complete for L. *Journal of Algorithms*, 8:385–394, 1987.

[13] C. Damm, M. Holzer, and P. McKenzie. The complexity of tensor calculus. *Computational Complexity*, 11(1/2):54–89, January 2003.

[14] P. W. Dymond. Input-driven languages are in log n depth. In *Information processing letters*, pages 26, 247–250, 1988.

[15] P. W. Dymond and W. L. Ruzzo. Parallel RAMs with owned global memory and deterministic context-free language recognition. *J. ACM*, 47(1):16–45, 2000. extended abstract in ICALP 86: LNCS 226 pp. 95–104.

[16] H. Fernau, K.-J. Lange, and K. Reinhardt. Advocating ownership. In V. Chandru and V. Vinay, editors, *Proc. 16th FST&TCS, LNCS 1180*, pages 286–297. Springer, December 1996.

[17] O. Ibarra, T. Jiang, and B. Ravikumar. Some subclasses of context-free languages in NC^1. *Information Processing Letters*, 29:111–117, 1988.

[18] H. Immerman. Languages which capture complexity classes. *SIAM J. Comput.*, 4:760–778, 1987.

[19] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, Oct 1988.

[20] S. Istrail and D. Zivkovic. Bounded width polynomial size Boolean formulas compute exactly those functions in AC^0. *Information Processing Letters*, 50:211–216, 1994.

[21] T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40:25–31, 1991.

[22] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 67–161. 1990.

[23] K.-J. Lange. Complexity and structure in formal language theory. *Fundamenta Informaticae*, 25:327–352, 1996. Preliminary version in Proceedings of 8th IEEE Structure in Complexity Theory Conference, 1993, 224–238.

[24] Lautemann, McKenzie, Schwentick, and Vollmer. The descriptive complexity approach to LOGCFL. *JCSS: Journal of Computer and System Sciences*, 62, 2001.

[25] F. Lemieux. *Finite Groupoids and their Applications to Computational Complexity.* PhD thesis, McGill University, 1996.

[26] N. Limaye, M. Mahajan, and B. V. R. Rao. Arithmetizing classes around NC^1 and L. In *STACS, LNCS vol. 4393*, 2007.

[27] P. McKenzie, K. Reinhardt, and V. Vinay. Circuits and context-free languages. In *Proceedings of 5th Annual Internat. Conf. on Computing and Combinatorics (COCOON), Tokyo, Japan, LNCS 1627*, pages 194–203, 1999.

[28] D. A. Mix-Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.

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

[30] N. Nisan. $\mathrm{RL} \subseteq \mathrm{SC}$. *Computational Complexity*, 4(11):1–11, 1994.

[31] Petersen. The membership problem for regular expressions with intersection is complete in LOGCFL. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science LNCS 2285*, pages 513–522, 2002.

[32] O. Reingold. Undirected *st*-connectivity in logspace. In *Proc. 37th STOC*, pages 376–385, 2005.

[33] W. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[34] W. Ruzzo. On uniformity within NC^1. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[35] P. Spira. On time hardware complexity tradeoffs for boolean functions. In *Proceedings of 4th Hawaii International Symposium on System Sciences*, pages 525–527, 1971.

[36] I. Sudborough. On the tape complexity of deterministic context-free language. *Journal of Association of Computing Machinery*, 25(3):405–414, 1978.

[37] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[38] H. Venkateswaran. Properties that characterize LogCFL. *Journal of Computer and System Sciences*, 42:380–404, 1991.

[39] H. Venkateswaran. Circuit definitions of nondeterministic complexity classes. *SIAM J. on Computing*, 21:655–670, 1992.

[40] V. Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proceedings of 6th Structure in Complexity Theory Conference*, pages 270–284, 1991.

[41] V. Vinay. Hierarchies of circuit classes that are closed under complement. In *CCC '96: Proceedings of the 11th Annual IEEE Conference on Computational Complexity*, pages 108–117, Washington, DC, USA, 1996. IEEE Computer Society.

[42] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag New York Inc., 1999.