Catalytic Computing Between L and P

Ian Mertz
University of Toronto
mertz@cs.toronto.edu

December 3, 2019
Oral Qualifying Examination, DCS, University of Toronto

Abstract

We survey the catalytic computing framework introduced by Buhrman et al. (STOC 2014). While the model for catalytic computing in Buhrman et al. uses a large amount of (catalytic) space, we also discuss algorithms for the Tree Evaluation Problem of Cook et al. (TOCT 2012), which operates in a model of low space. In particular, we discuss how the catalytic approach of computing large fan-in sums and products may be improved, and the possible implications of further improvements to the complexity landscape between L and TC^1.

Contents

1 Introduction: The catalytic computing model **1**

2 Computation models **1**

3 History **3**
3.1 Constant size register programs and NC^1 3
3.2 The Tree Evaluation Problem 4
3.3 Catalytic results 5

4 Catalytic sum and product lemmas **6**

5 Complexity classes between L and TC^1 **8**

6 Open problems **9**

References **10**

1 Introduction: The catalytic computing model

Consider an everyday desktop, where there are two main types of memory available for use: a huge amount of hard drive space meant for storing files, and a small memory bank in RAM, the L_1 cache, and other working space which is kept clean for use in actually running computer programs. Besides not being optimized for speed, one obvious objection to using the HDD for actually running computation is that it tends to get filled up with movies, games, and other files that we wouldn't want to write over in the course of running programs. Even so, it seems like a waste to leave such a huge memory bank untouched most of the time.

In the *catalytic computing* model, introduced by Buhrman et al. [BCK+14], we simulate this situation by considering a Turing machine with two tapes: a small (think $O(\log n)$ for this survey) work tape like the ones we usually have in space-bounded computation, and an exponentially bigger "catalytic tape" which is in an unknown inital state. The program is free to use the work tape with no restrictions, and is also free to use the catalytic tape provided that at the end of the computation the tape is returned to its original configuration. Of course since the initial configuration is unknown and possibly incompressible, it is not at all obvious that the latter tape helps us in any way.

In light of this intuition, the results of [BCK+14] can appear very surprising at first glance. They show that CL, the class of problems solved by a catalytic Turing Machine where the work tape has length $O(\log n)$ and the catalytic tape has length $n^{O(1)}$ [1] captures all the power of TC^1, the class of log-depth poly-size circuits with unbounded fan-in $\wedge$, $\vee$, and MAJ gates. Since TC^1 contains AC^1, a circuit class known to contain NL, and since $\mathsf{TC}^0 \not\subseteq \mathsf{AC}^0$ it would be reasonable to conjecture that $\mathsf{TC}^1 \not\subseteq \mathsf{AC}^1$, and thus that $\mathsf{NL} \subsetneq \mathsf{CL}$.

We survey the application of the techniques of catalytic computing to problems and classes within TC^1, in hopes of shedding light on the power of L versus CL. In Section 2 we formally introduce all computation models and complexity classes that will appear in this survey. In Section 3 we discuss some background on catalytic computing, including two lines of work instrumental in conceptualizing the model, as well as follow-up work since the initial publication of [BCK+14]. In Section 4 we state some key technical lemmas of [BCK+14] and our recent improvements to these lemmas. In Section 5 we review some important classes in between L and P, and discuss where catalytic computing techniques may play a role in resolving longstanding open relations between these classes. Finally in Section 6 we propose three concrete open problems related to these open relations and to catalytic computing in general.

2 Computation models

We first introduce branching programs, although the main focus of this survey will be on the register program model defined subsequently. Note that for this paper we only discuss *uniform* computation models, without going into too much detail about which notion of uniformity we use (see e.g. [AGM16]).

Definition 1 (Branching program)**.** Let $\mathcal{R}$ be any finite set (we usually assume $\mathcal{R} = \{0, 1\}$). A *deterministic layered branching program* for computing a function $f : \mathcal{R}^n \to O$ is a directed acyclic graph $B = (V, E)$ with labels on each node and edge, plus a value $\ell \in \mathbb{N}^+$, satisfying the following conditions:

– there exists a single source node and $|O|$ sink nodes

[1] We will focus on the case of catalytic logspace (CL), but their results hold more generally for $\mathsf{SPACE}(s(n))$ as long as $s(n) = \Omega(\log n)$

- every non-sink node is labeled with a value $j \in [\ell]$ (we assume the source node is labeled with $j = 1$) and an input variable x_i; additionaly the fan-out of each non-sink node is $|\mathcal{R}|$ and each labeled outgoing edge is labeled with a unique element of $\mathcal{R}$
- every sink is labeled with the value ℓ and a unique element of O
- every edge going from a node labeled with j goes to a node labeled with $j + 1$
- on input $\alpha \in \mathcal{R}^n$, the unique sink reached from the source on the path following the edge labeled α_i at each vertex labeled x_i is labeled with $f(\alpha)$

The *size* of B is $|V|$, the *width* of B is $\max_{j\in[\ell]} |\{v \in V | v \text{ has label } j\}|$ and the *length* of B is ℓ.

A branching program over $\mathcal{R}$ of width w and length ℓ can be simulated by a Turing Machine with space $O(\log w + \log \ell + \log |\mathcal{R}|)$ by simply keeping an address $(k, j) \in [w] \times [\ell]$ for the current node as well as the value in $\mathcal{R}$ of the label x_i for the given input. If $\mathcal{R}'$ is such that $|\mathcal{R}'| < |\mathcal{R}|$, by representing each element of $\mathcal{R}$ as $c = \log_{|\mathcal{R}'|} |\mathcal{R}|$ elements of $\mathcal{R}'$, a branching program over $\mathcal{R}$ of width w and length ℓ can also be simulated by a branching program over $\mathcal{R}'$ of width w and length $c\ell$.

Definition 2 (Register program)**.** Let $\mathcal{R}$ be any ring. A *register program* for computing a function $f : \mathcal{R}^n \to O$ is a pair $P = (\{R_j\}, \mathit{inst})$ where each R_j is a register holding a value in $\mathcal{R}$ and *inst* is a sequence of instructions of the form $R_j \leftarrow R_j + v_\ell v_r$ for $v_\ell, v_r \in \mathcal{R} \cup \{x_i\} \cup \{R_j\}$. The the *space* of P is $|\{R_i\}|$ and the *length* of P is $|\mathit{inst}|$.

We additionally say that P is *skew* if $v_\ell \in \mathcal{R} \cup \{x_i\}$ for every instruction in *inst*. It is not hard to see that for a finite $\mathcal{R}$, every register program of space s and length ℓ can be simulated by a layered branching program of width $s^{|\mathcal{R}|}$ and length $\ell + 1$.

Before moving on to catalytic computation in earnest, we introduce a third computation model, the circuit, which will mostly function as a backdrop to the result presented in Section 3, but which we will revisit in our discussion of future directions in Section 5.

Definition 3 (Circuit)**.** Let $\mathcal{R}$ be a set and for all $d \geq 1$ let $\{\circ_d\}$ be a set of functions from $\mathcal{R}^d$ to $\mathcal{R}$. A *circuit* for computing a function $f : \mathcal{R}^n \to O$ is a directed acyclic graph $C = (V, E)$ with labels on each node and edge, satisfying the following conditions:

- there exists a single source node
- every sink node is labeled with $x \in \{x_i\} \cup \mathcal{R}$
- every non-sink node with fan-in d is labeled with a function in $\{\circ_d\}$
- on input $\alpha \in \mathcal{R}^n$, the output of the source node, where the output of a node is defined in a bottom-up fashion by replacing all x_i with α_i, is $f(\alpha)$

The *size* of C is $|V|$ and the *depth* of C is the length of the longest source-sink path.

If $\mathcal{R} = \{0, 1\}$ we say C is a *Boolean circuit*, whereas if $\mathcal{R}$ is a ring we say C is an *arithmetic circuit*. For now we assume all circuits have polynomial size and all functions $\circ$ are fan-in at most 2 and define the two classes needed to begin our study of catalytic computing.

Definition 4. An NC^1 circuit is Boolean circuit C where $\circ_1 = \{\neg\}$, $\circ_2 = \{\wedge, \vee\}$, and where the depth of C is at most $O(\log n)$. For a ring $\mathcal{R}$ a $\#\mathsf{NC}^1(\mathcal{R})$ circuit is an arithmetic circuit C over $\mathcal{R}$ where $\circ_2 = \{\oplus, \otimes\}$, and where the depth of C is at most $O(\log n)$.

3 History

To review the history of catalytic computing, we first turn our attention to earlier works that inspired the model. The first is a pair of seminal results from over two decades prior: Barrington's Theorem for NC^1 [Bar89] and Ben-Or and Cleve's construction for $\#\mathsf{NC}^1$ [BoC92]. We introduce these results in full, as both are as simple as they are profound. The second is the Tree Evaluation Problem ($\mathsf{TreeEval}$), introduced by Cook et al. [CMW+12] as a candidate for separating L from P. We introduce the problem and motivate how catalytic computing arose to throw a wrench into a seemingly reasonable assumption on space-bounded complexity which was being used in the context of $\mathsf{TreeEval}$.

3.1 Constant size register programs and NC^1

Theorem 1 (Barrington's Theorem). *NC^1 is precisely the class of functions that can be computed by a poly-size layered branching program of width $O(1)$. More specifically, let C be a fan-in two Boolean circuit of depth $d \leq c \log n$. Then there exists a width-five layered branching program of length $4^d \leq n^{2c}$ computing the same function as C.*

To begin our discussion of the results of [Bar89, BoC92], let us try and understand why Barrington's Theorem [Bar89] was such a bolt from the blue in 1989. One common way of viewing circuits is as massively parallel computation, where poly(n) processors working in parallel on each layer allows us to compute tricky functions in logarithmic or even constant time. This time would appear to come at the cost of maintaining and computing a large number of bits in parallel, and furthermore it's not at all clear that taking polynomial time would allow us to save much space. Intuitively to compute the output node of a fan-in two circuit, we need to have both of its inputs in hand at some point in time, meaning at best we hold on to one, say the value of the left child, while we compute the other. Now if we want to compute the value of the right child, the same reasoning applies: at some point we need to have the values of both of *its* children, meaning we at least need to hold on to one while we compute the other. Applying this reasoning inductively, a log-depth fan-in two circuit would seem to require storing at least one bit at each level; this is a style of argument called *pebbling* [PH70, CMW+12].

By contrast, Barrington's Theorem states that a width five branching program, the equivalent of a machine with only $\log_2 5 < 3$ bits of working memory, is enough to compute an NC^1 circuit, and by extension *any* poly-size formula in polynomial time. The details of how to compute internal $\wedge$ or $\vee$ gates involve actions on a permutation group which we leave for interested readers to seek out. Instead we turn now to [BoC92] to see how a similar staggering result came out for arithmetic circuits, using the same ideas as Barrington's Theorem but in an even simpler form.

Theorem 2. *For any ring $\mathcal{R}$, $\#\mathsf{NC}^1(\mathcal{R})$ is precisely the class of functions that can be computed by a poly-size skew register program with $O(1)$ registers over $\mathcal{R}$. More specifically, let C be a fan-in two arithmetic circuit over $\mathcal{R}$ of depth $d \leq c \log n$. Then there exists a skew register program of length $4^d \leq n^{2c}$ with three registers over $\mathcal{R}$ computing the same function as C.*

Proof. We define our register program inductively on the nodes of C. For any node g in C let f_g be the function computed at g, and we will build a program P_g computing f_g; thus our final goal is to define P_o for the output gate o. More specifically, for three registers R_1, R_2, R_3, each holding a value in $\mathcal{R}$, let $P_g(R_1, R_2, R_3)$ be the program that sets

$$R_1 = \tau_1 + \tau_2 \cdot f_g$$

$$R_2 = \tau_2 \qquad R_3 = \tau_3$$

where τ_i is the initial value stored in register R_i. Thus if we can define $P_o(R_1, R_2, R_3)$, then setting $\tau_1 = \tau_3 = 0$ and $\tau_2 = 1$ it is clear that $P_o(R_1, R_2, R_3)$ computes the same function as C.

Let g be a leaf of C labeled with $x \in \{x_1 \dots x_n, \overline{x_1} \dots \overline{x_n}\} \cup \mathcal{R}$. Then the program $P_g(R_1, R_2, R_3)$ is as follows:

1: $R_1 \leftarrow R_1 + R_2 x$ $\qquad \triangleright\ R_1 = \tau_1 + \tau_2 \cdot x$

Let g be an internal node of C of height d with children g_ℓ, g_r, and inductively assume that we have programs P_{g_ℓ} and P_{g_r} of length 4^{d-1} for computing f_{g_ℓ} and f_{g_r} respectively, as well as their inverses. First consider the case when $f_g = f_{g_\ell} + f_{g_r}$. Then the program $P_g(R_1, R_2, R_3)$ is as follows:

1: $P_{g_\ell}(R_1, R_2, R_3)$ $\qquad \triangleright\ R_1 = \tau_1 + \tau_2 \cdot f_{g_\ell}$
2: $P_{g_r}(R_1, R_2, R_3)$ $\qquad \triangleright\ R_1 = \tau_1 + \tau_2 \cdot f_{g_\ell} + \tau_2 \cdot f_{g_r}$

Finally consider the case when $f_g = f_{g_\ell} \cdot f_{g_r}$. Then the program $P_g(R_1, R_2, R_3)$ is as follows:

1: $P_{g_r}^{-1}(R_1, R_3, R_2)$ $\qquad \triangleright\ R_1 = \tau_1 - \tau_3 \cdot f_{g_r}$
2: $P_{g_\ell}(R_3, R_2, R_1)$ $\qquad \triangleright\ R_3 = \tau_3 + \tau_2 \cdot f_{g_\ell}$
3: $P_{g_r}(R_1, R_3, R_2)$ $\qquad \triangleright\ R_1 = \tau_1 - \tau_3 \cdot f_{g_r} + (\tau_3 + \tau_2 \cdot f_{g_\ell}) \cdot f_{g_r}$
4: $P_{g_\ell}^{-1}(R_3, R_2, R_1)$ $\qquad \triangleright\ R_3 = \tau_3$

Since we make at most four calls to programs of length 4^{d-1}, P_g has length at most 4^d as claimed. Furthermore the only instructions are those at the leaves, which are skew instructions. □

The power of Theorem 2 is largely in how the induction is defined. At each layer we define programs which only affect one register, leaving everything else untouched. Since the programs are skew, they are also invertible by simply running the instructions in reverse and switching $+/-$ signs. To maintain these invariants, inverse programs are used to reset all registers besides the target register, but more importantly by rotating the target, source, and working registers, and by adding and subtracting values both before and after computing one of the leaves recursively (in this case P_{g_ℓ}), we can use our third register R_3 to get an $R_2 \cdot f_{g_\ell} f_{g_r}$ term while canceling out the contribution of τ_3 itself.

These results laid the foundation for catalytic computing by showing the power of inverse functions and arithmetic manipulation to create "transparent programs", register programs which only leave one register affected and which work regardless of the starting configuration of the registers. Before moving on to TreeEval we note that these results are for a very different regime than the catalytic model discussed before, namely the case where very low space of any kind is available for use. Below L this is ill-suited for the Turing Machine model, where a polynomial length register program would require $O(\log n)$ space just to keep track of a program counter, whereas for branching programs and register programs themselves this seems just as natural as the catalytic model. Moving up to L, since Barrington's Theorem and Theorem 2 work with only a constant amount of memory, it would be interesting to see what could be done with register program with $O(\log n)$ register space, which would immediately give us new catalytic techniques for L itself. We return to this idea in later sections.

3.2 The Tree Evaluation Problem

The results of [Bar89, BoC92] saw numerous applications in the decades that followed [BIS90, BBC+95, ÀJ95], but the work that led to the formulation of catalytic computing came from a much different line of work, namely lower bounds against L. The Tree Evaluation Problem, formulated by Cook et al. [CMW+12] nearly two decades after [BoC92], has been a leading candidate for separating L from P since its inception.

Definition 5 (Tree Evaluation Problem [CMW+12], $\mathsf{TreeEval}_{h,k}$)**.** The *tree evaluation problem* $\mathsf{TreeEval}_{h,k}$ is parameterized by a height h and an alphabet size k. The input is a full binary tree of height h, where every leaf is labeled with an element of $[k]$ and every internal node is labeled with a function from $[k] \times [k]$ to $[k]$. The output is the value of the root of the tree, where the tree is evaluated bottom-up in the natural way. We will often omit the subscripts and write TreeEval.

The input to $\mathsf{TreeEval}_{h,k}$ has size $(2^{h-1} - 1)k^2 \log k + 2^{h-1} \log k = O(2^h \operatorname{poly}(k))$. The problem is in P: it can be solved in polynomial time by evaluating every node, starting from the leaves, in an order that ensures a node's two children get evaluated before its parent. As with Barrington's Theorem, a pebbling lower bound shows that for a layered branching program, there is some layer which is holding at least one value in $[k]$ for each of the h levels of the tree, which would mean this level has width k^h. Formalizing the basic algorithm which computes the instance in a bottom-up manner, a careful analysis gives us a branching program of size $\Theta((k+1)^h) = \omega(2^h \operatorname{poly}(k))$.

Unlike with our intuition about the algorithms of [Bar89, BoC92], these pebbling based lower bounds actually do translate to tight $\Omega(k^h)$ lower bounds in certain restricted settings. In the *read-once* restriction the branching program only looks at each bit of the input at most once, while in the *thrifty* restriction the branching program can only read bits corresponding to the actual evaluation of the tree (so for example if the children of a node v evaluate to x and y, the branching program must not read any values of the function at v other than the value at (x, y)). The basic algorithm we introduced before fulfills both of these conditions, but either one of them is enough to guarantee a lower bound of $\Omega(k^h)$, and neither of these restrictions assume any other structure on the branching program such as being layered.

However, the strategy of proving these same lower bounds to the case of general branching programs[2] was, in the words of [Kou16], to "prove the lower bounds under essentially the assumption that the extra space does not help and then justify this assumption." This assumption, as in the case of NC^1 and $\#\mathsf{NC}^1$, is natural, persuasive, and ultimately not correct. The paradigm of catalytic computing came with [BCK+14] a few years after TreeEval, and while their results gave no explicit algorithms for TreeEval they posed an existential challenge to the natural pebbling strategy.

In recent work with James Cook [CM19], we brought this challenge more directly to TreeEval, using and improving a key lemma from [BCK+14]. While [BCK+14] simulates circuits by using the large catalytic space to evaluate and store all values at a given level simultaneously, in TreeEval all nodes have fan-in two, and so as with the proof of Theorem 2 the strategy goes node by node instead, allowing us to avoid using a large catalytic tape. More importantly, in this proof we concretely see how catalytic computing dodges both the read-once assumption (by recomputing nodes many times) and the thrifty assumption (by summing over all potential queries at each node), and while our upper bounds don't yet go beyond the basic algorithm for most values of h and k, the cost actually comes in the length of the program, while the width is far less than the pebbling lower bound predicts.

3.3 Catalytic results

To understand the follow-up work to [BCK+14], we first state their central result.

Theorem 3. $\mathsf{TC}^1 \subseteq \mathsf{CL} \subseteq \mathsf{ZPP}$

In an attempt to understand the power of catalytic space, last year Buhrman et al. [BKLS18] extended the model to non-deterministic computation, and showed a catalytic version of the

[2] It should be noted that this work was not inspired by pebbling lower bounds for restricted models. Initial work on TreeEval was specifically for the case of unrestricted branching programs; the read-once and thrifty lower bounds came much later, not earlier.

Immerman-Szelepcsényi theorem [Imm88, Sze88], namely that assuming $\mathsf{SPACE}(n) \not\subseteq \mathsf{SIZE}(2^{\epsilon n})$ for some constant $\epsilon > 0$, then $\mathsf{CNL} = \mathsf{coCNL}$. This year Gupta et al. [GJST19] extended the model to unambiguous non-deterministic computation as well, and showed that $\mathsf{CUL} = \mathsf{CNL}$ under the same assumption as $\mathsf{CNL} = \mathsf{coCNL}$. Note that [BKLS18] showed an upper bound of $\mathsf{CNL} \subseteq \mathsf{ZPP}$ as in the deterministic case, and so it remains open whether or not CL or CNL are in $\mathsf{P} \subseteq \mathsf{ZPP}$.

Another recent result is that of Potechin [Pot16], which is connected to deterministic catalytic computation for much larger space. Consider the catalytic model for branching programs, where the input layer has 2^a nodes representing the initial configuration of a binary catalytic registers, and with the output nodes being labeled with both an output of the function and the initial configuration (e.g. $2 \cdot 2^a$ nodes for a binary function). Then for the case of $a = 2^n$, [Pot16] shows that *any* function can be computed by a branching program of width $2 \cdot 2^a$ and length $2n$, a factor two off in the length from the absolute best branching program possible.

For a more in-depth survey we defer interested readers to [Kou16].

4 Catalytic sum and product lemmas

In this section we state a number of key lemmas from [CM19]. We also discuss where potential improvements to these lemmas could be found, and how they would translate to further results for $\mathsf{TreeEval}$. In particular, we look at method for computing $+$ and $\times$ catalytically.

The starting point is a lemma which appears in [BCK+14] as Lemma 4, but which is apparent from the proof of Theorem 2. Recall that τ_i is the initial value stored in register R_i. We say that P *transparently computes* $R = \tau + v$ to mean that all other registers are left unchanged, or more specifically

$$R = \tau + v$$

$$R' = \tau' \quad \forall R' \neq R$$

Lemma 4 (Basic Sum Lemma). *Let R_ℓ, R_r and R_p be distinct registers. Let P_ℓ be an invertible program which transparently computes v_ℓ into register R_ℓ and let P_r be an invertible program which transparently computes v_r into register R_r. Then there exists an invertible program P_p which transparently computes $v_\ell v_r$ into R_p, i.e.*

$$R_p = \tau_p + v_\ell \cdot v_r$$

$$R_i = \tau_i \quad \forall i \neq p$$

P_p uses only the three registers R_v, R_ℓ, R_p (not counting any space used by the programs P_ℓ and P_r) and makes two calls to P_ℓ and P_r each, plus four basic instructions of the form $R_p \leftarrow R_p \pm R_\ell R_r$.

For convenience, from now on

Our first improvement to the Basic Sum Lemma will be to parallelize it. Consider the case when we want to run many different iterations of Basic Sum Lemma, where each iteration uses different but possibly overlapping v_ℓ and v_r values. In particular we will let $\overrightarrow{R_p}$, $\overrightarrow{R_\ell}$, and $\overrightarrow{R_r}$ be distinct i, j, and k-dimensional vectors of registers, respectively. We will let P_ℓ and P_r transparently compute some values $\overrightarrow{v_\ell}$ and $\overrightarrow{v_r}$ into $\overrightarrow{R_\ell}$ and $\overrightarrow{R_r}$ respectively. For each $x \in [i]$, $R_{p,x}$ has some target value $v_{p,x}$ given by

$$v_{p,x} = \sum_{(y,z) \in S_x} v_{\ell,y}^{e_{\ell,x,y}} \cdot v_{r,z}^{e_{r,x,z}}$$

for some sets $S_x \subseteq [j] \times [k]$ and polarities $e_{\ell,x,y}, e_{r,x,z} \in \{0,1\}$ (recall that $x^1 = x$ and $x^0 = \overline{x}$). While one solution would be to run Basic Sum Lemma separately for each x and each $(y,z) \in S_x$,

resulting in $2ijk$ recursive calls to P_ℓ and P_r each, it turns out that the same two calls each as in Basic Sum Lemma suffices.

Lemma 5 (Parallel Sum Lemma)**.** *Let P_ℓ and P_r be invertible programs which transparently compute $\overrightarrow{R_\ell} = \overrightarrow{\tau_\ell} + \overrightarrow{v_\ell}$ and $\overrightarrow{R_r} = \overrightarrow{\tau_r} + \overrightarrow{v_r}$ respectively. For all $x \in [i]$ let S_x, $\{e_{\ell,x,y}\}$, $\{e_{r,x,z}\}$ be such that*

$$v_{p,x} = \sum_{(y,z)\in S_x} v_{\ell,y}^{e_{\ell,x,y}} \cdot v_{r,z}^{e_{r,x,z}}$$

Then there exists an invertible program P_p which transparently updates

$$\overrightarrow{R_p} = \overrightarrow{\tau_p} + \overrightarrow{v_p}$$

P_p uses only the $i+j+k$ registers $\overrightarrow{R_p}, \overrightarrow{R_\ell}, \overrightarrow{R_r}$ (not counting any space used by the programs P_ℓ and P_r). P_p uses two calls to P_ℓ and P_r each, plus $4ijk$ basic instructions of the form $R_{p,x} \leftarrow R_{p,x} \pm R_{\ell,y} R_{r,z}$.

Our second improvement to Basic Sum Lemma will be to work for d-ary products. A similar lemma implicitly appears in [BCK+14] as Lemma 8, but we state it with the exact parameters.

Lemma 6 (Large Product Lemma)**.** *Let $R_0, \ldots, R_d$ be distinct registers. Let $P_1 \ldots P_d$ be invertible programs where P_i transparently computes $R_i = \tau_i + v_i$. Then there exists an invertible program P which transparently updates*

$$R_0 = \tau_0 + \prod_{i\in[d]} v_i$$

P uses the $d+1$ registers $R_0, \ldots, R_d$ plus d additional registers (not counting any space used by the programs P_i) and makes d^2 calls to each P_i plus $\mathrm{poly}(d)$ basic instructions of the form $R_p \leftarrow R_p \pm \prod_{i\in[d]} R_i$.

Our third improvement to Basic Sum Lemma is to perform parallelized d-ary products, or in other words to perform Parallel Sum Lemma and Large Product Lemma simultaneously. We will let $\overrightarrow{R_p}$ and $\overrightarrow{R_c}$ be distinct i and j-dimensional vectors of registers, respectively, and we will let P_c transparently compute some values $\overrightarrow{v_c}$ into $\overrightarrow{R_c}$. For each $x \in [i]$, $R_{p,x}$ has some target value $v_{p,x}$ given by

$$v_{p,x} = \sum_{t\le k} \prod_{y\in[j]} v_{c,y}^{e_{t,x,y}}$$

for some $k \le 2^j$ and polarities $e_{t,x,y} \in \{0,1\}$. Unfortunately the proof of Large Product Lemma does not lend itself to parallelization, and so we will need an exponential number of calls to P_c. We also generalize P_c and P_p to be able to compute subsets of v_c and v_p, rather than always computing the full vectors.

Lemma 7 (Parallel Large Product Lemma)**.** *For all $T \subseteq [j]$ let $P_c(T)$ be an invertible program which transparently computes*

$$R_{c,y} = \tau_{c,y} + v_{c,y} \quad \forall y \in T$$

For all $x \in [i]$ let k, $\{e_{t,x,y}\}$ be such that

$$v_{p,x} = \sum_{t\le k} \prod_{y\in[j]} v_{c,y}^{e_{t,x,y}}$$

Then for every $S \subseteq [i]$ there exists an invertible program $P_p(S)$ which transparently computes

$$R_{p,x} = \tau_{p,x} + v_{p,x} \quad \forall b \in S$$

P_p uses only the $i+j$ registers $\overrightarrow{R_p}$ and $\overrightarrow{R_c}$ (not counting any space used by the programs $P_c(T)$) and makes one call to each $P_c(T)$, plus $2^{O(i+j)}$ basic instructions of the form $R_{p,x} = R_{p,x} + c_{t,x} \prod_{y\in[j]} R_{c,y}^{e_{t,x,y}}$ for constants $c_{t,x}$.

While the exponential loss in Parallel Large Product Lemma may seem large, for $\mathsf{TreeEval}_{h,k}$ if we encode our values from $[k]$ in $d = \log k$ binary bits, then Parallel Large Product Lemma gives us a way to compute the function at a given node using $2k^2$ recursive calls to its children, leading to a branching program of k^{2h} length and $2^{O(\log k)}$ width. As discussed before while this gives an algorithm that is ultimately polynomially less efficient than the basic algorithm, it does so only in the length, while the width corresponding to the storage space is $O(\log k)$ rather than the $\Omega(h \log k)$ pebbling lower bound. Further optimizing the encoding gives the following breakthrough algorithm.

Theorem 8 (Hybrid algorithm, [CM19])**.** *For every $\epsilon > 0$ there exists a branching program solving $\mathsf{TreeEval}_{h,k}$ with length $(\frac{k}{\epsilon h}+1)^{2h}$ and width $(\frac{k}{\epsilon h}+1)^{3\epsilon h}$. In particular for any constant $\epsilon > 0$ there is a constant $\epsilon' > 0$ such that there exists an algorithm solving $\mathsf{TreeEval}_{h,k}$ with size $k^{(1-\epsilon)h}$ for any $h \geq k^{1/2+\epsilon'}$, where $\epsilon' \to 0$ as $\epsilon \to 0$.*

5 Complexity classes between L and TC^1

Before we discuss the open problems raised by Parallel Large Product Lemma, we briefly introduce circuit classes beyond NC^1 and $\#\mathsf{NC}^1$. Recall from Section 2 our more general definition of Boolean and arithmetic circuits, where we allow functions of fan-in greater than 2. We define the following Boolean circuit classes, which all have depth $O(\log n)$ and size poly(n).

- SAC^1: $\circ_2 = \{\vee\}$, $\circ_d = \{\wedge\}$ for all $d \leq \text{poly}(n)$, sink nodes may also be labeled with literals $\overline{x_i}$
- AC^1: $\circ_1 = \{\neg\}$, $\circ_d = \{\wedge, \vee\}$ for all $d \leq \text{poly}(n)$
- $\mathsf{AC}^1[m]$: $\circ_1 = \{\neg\}$, $\circ_d = \{\wedge, \vee, \mathsf{MOD}_m\}$ for all $d \leq \text{poly}(n)$
- TC^1: $\circ_1 = \{\neg\}$, $\circ_d = \{\wedge, \vee, \mathsf{MAJ}\}$ for all $d \leq \text{poly}(n)$

We also have corresponding arithmetic classes, albeit with a few twists including two different corresponding SAC^1 classes[3]. Again we assume that all classes have depth $O(\log n)$ and size poly(n).

- $\mathsf{VP}(\mathcal{R})$: $\circ_2 = \{\otimes\}$, $\circ_d = \{\oplus\}$ for all $d \leq \text{poly}(n)$
- $\Lambda\mathsf{P}(\mathcal{R})$: $\circ_2 = \{\oplus\}$, $\circ_d = \{\otimes\}$ for all $d \leq \text{poly}(n)$
- $\#\mathsf{WSAC}^1(\mathcal{R})$: $\circ_d = \{\otimes\}$ for all $d \leq O(\log n)$, $\circ_d = \{\oplus\}$ for all $d \leq \text{poly}(n)$
- $\#\mathsf{AC}^1(\mathcal{R})$: $\circ_d = \{\oplus, \otimes\}$ for all $d \leq \text{poly}(n)$

Most of these classes are standard and follow conventions codified in [Vol99]; while many readers may be familiar with VP [Val79] we emphasize again that we are discussing *uniform* circuit classes, and in particular we consider the *Boolean Part* wherein all inputs and outputs to our arithmetic circuit models are in $\{0,1\}$ no matter what ring $\mathcal{R}$ is allowed internally. This allows us to define a hierarchy of both Boolean and arithmetic circuit classes together, which is partially displayed in Figure 1. The classes $\Lambda\mathsf{P}$ and $\#\mathsf{WSAC}^1$ were defined in [AGM16], which also contains a more comprehensive hierarchy for all classes listed.

[3] While we defined SAC^1 with $\vee$ restricted to fan-in 2 and $\wedge$ unbounded, it is known that reversing them is equivalent, as SAC^1 (like the other classes) is closed under complement [BCD$^+$89].

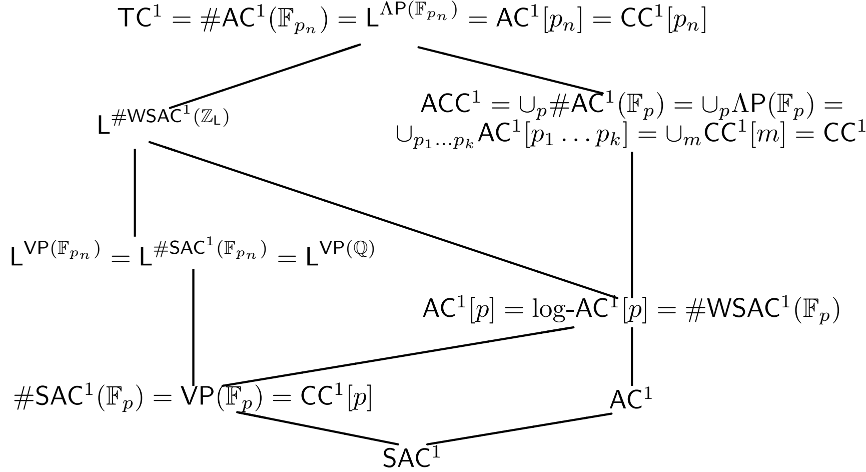

Figure 1: Circuit class relations (appears in [AGM16] as Figure C1). Not pictured: $\mathsf{NL} \subseteq \mathsf{SAC}^1$, but the $\mathsf{SAC}^1 \subseteq \mathsf{VP}(\mathbb{F}_p)$ is only known non-uniformly [GW96], while $\mathsf{NL} \subseteq \mathsf{AC}^1$ uniformly.

6 Open problems

Question 1: can Parallel Large Product Lemma be improved? The most pressing open question presented by these results is whether or not there exists a version of Parallel Large Product Lemma making fewer recursive calls, or equivalently if the construction in Large Product Lemma can be made amenable to parallelization. Should we be able to get a version of Parallel Large Product Lemma making $t(j)$ recursive calls for a function $t = o(2^j)$, it would translate to a $\mathsf{TreeEval}_{h,k}$ algorithm of size $(t(\log k))^{O(h)}$.

Cook et al. [CBM+09] offer a prize for any algorithm which, for a fixed h, proves $\mathsf{TreeEval}_{h,k} \in O(k^{h-\epsilon})$ for any constant $\epsilon > 0$. If $t(j) = \text{poly}(j)$ as in Large Product Lemma then $\mathsf{TreeEval}_{h,k}$ would have a branching program of size $(\log k)^{O(h)}$ for all h, k, which would be more than sufficient to claim the prize. Even more drastically, to prove $\mathsf{TreeEval} \in \mathsf{L}$ it would be sufficient to show it for $t(j) = O(1)$ as in Parallel Sum Lemma.

Question 2: do Large Product Lemma or Parallel Large Product Lemma give L more power to simulate circuit classes? Recall that $\mathsf{VP}(\mathcal{R})$ is the class of functions with arithmetic circuits of fan-in 2 $\otimes$ gates and unbounded fan-in $\oplus$ gates. While the simulation of TC^1 in [BCK+14] requires polynomial space to store all values at some level of the circuit simultaneously, the approach of using Parallel Sum Lemma to solve $\mathsf{TreeEval}$ allows us to compute unbounded fan-in $\oplus$ gates without ever storing all the inputs in memory, although it does require all inputs to be able to be derived from small space storage. It is not obvious at all that the techniques presented in Section 4 can be adapted to compute any of the classes in Section 5 without using the polynomial-length catalytic tape, but doing so would be a huge breakthrough in our understanding of L.

Recall also that $\#\mathsf{WSAC}^1(\mathcal{R})$ is the class of functions with arithmetic circuits of logarithmic fan-in $\otimes$ gates and unbounded fan-in $\oplus$ gates. With Parallel Large Product Lemma we can compute unbounded sums of logarithmic fan-in times operations within a polynomial number of recursive calls. If we could make any progress towards Question 1 and towards simulating VP in L, would this also make progress towards the same for $\#\mathsf{WSAC}^1$? This would be very surprising, as $\#\mathsf{WSAC}^1$ is known to contain NL uniformly.

Question 3: can circuits also apply Parallel Large Product Lemma? The Immerman-Landau Conjecture [IL95] states that the determinant of integer matrices is complete for TC^1. Mentioned in the work of [BCK+14] was that catalytic computing offers a potential counterargument to this conjecture; namely, since TC^1 can be simulated by $\mathsf{VP}(\mathbb{Q})$ circuits, being able to compute the determinant in TC^1 would imply that arbitrary polynomials of degree $n^{O(\log n)}$ computable by poly-size arithmetic circuit over $\mathbb{F}_{p_n}$ could be computed by arithmetic circuits over $\mathbb{Q}$ with degree only poly(n). While this seems highly unlikely at first glance, the first hints at a counterexample were given in [AGM16], who showed that $\#\mathsf{AC}^1(\mathbb{F}_2) = \#\mathsf{WSAC}^1(\mathbb{F}_2)$. While this proof involves more sophisticated techniques such as Toda polynomials [Tod91], probabilistic AND gates [AJMV98], and group generators, it seems worthwhile to investigate whether the relatively simple procedure given in the proof of Parallel Large Product Lemma can be utilized by circuit classes well within TC^1 in some way to give more collapses in this way.

Acknowledgements Part of this work appears in joint work with James Cook [CM19] as well as joint work with Eric Allender and Anna Gál [AGM16]. The author thanks Stephen Cook and Toniann Pitassi for many helpful discussions, as well as James Cook and Morgan Shirley for feedback on earlier drafts of this survey. The author was supported by NSERC.

References

[AGM16] Eric Allender, Anna Gál, and Ian Mertz. Dual vp classes. *computational complexity*, 25:1–43, 2016.

[ÀJ95] Carme Àlvarez and Birgit Jenner. A note on logspace optimization. *computational complexity*, 5(2):155–166, Jun 1995.

[AJMV98] 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.

[Bar89] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[BBC+95] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, 1995.

[BCD+89] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18:559–578, 1989. See Erratum in SIAM J. Comput. 18, 1283.

[BCK+14] Harry Buhrman, Richard Cleve, Michal Kouckỳ, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 857–866. ACM, 2014.

[BIS90] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within nc1. *Journal of Computer and System Sciences*, 41(3):274 – 306, 1990.

[BKLS18] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic space: Non-determinism and hierarchy. *Theory Comput. Syst.*, 62(1):116–135, 2018.

[BoC92] Michael Ben-or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, February 1992.

[CBM+09] Stephen Cook, Mark Braverman, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Branching programs: Avoiding barriers. Talk at Barriers Workshop at Princeton, August 2009. URL: https://www.cs.toronto.edu/~sacook/barriers.ps.

[CM19] James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem, 2019.

[CMW+12] Stephen Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, January 2012.

[GJST19] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Unambiguous catalytic computation. *Electronic Colloquium on Computational Complexity (ECCC)*, 26:95, 2019.

[GW96] Anna Gál and Avi Wigderson. Boolean complexity classes vs. their arithmetic analogs. *Random Struct. Algorithms*, 9(1-2):99–111, 1996.

[IL95] N. Immerman and S. Landau. The complexity of iterated multiplication. *Information and Computation*, 116:103–116, 1995.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[Kou16] Michal Koucký. Catalytic computation. *Bulletin of the EATCS*, 118, 2016.

[PH70] Michael S. Paterson and Carl E. Hewitt. Comparative schematology. In Jack B. Dennis, editor, *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127. ACM, 1970.

[Pot16] Aaron Potechin. A note on amortized branching program complexity, 2016.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[Tod91] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20:865–877, 1991.

[Val79] L.G. Valiant. Completeness classes in algebra. In *Proc. 11th ACM STOC*, pages 249–261, 1979.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag New York Inc., 1999.