Computing with a full memory: Catalytic space

Harry Buhrman*[1,2], Richard Cleve†[3], Michal Koucký‡[4],
Bruno Loff[1], and Florian Speelman§[1]

[1]CWI, Amsterdam
[2]University of Amsterdam
[3]University of Waterloo
[4]Charles University, Prague

Abstract

We define the notion of a *catalytic-space* computation. This is a computation that has a small amount of clean space available and is equipped with additional auxiliary space, with the caveat that the additional space is initially in an arbitrary, possibly incompressible, state and must be returned to this state when the computation is finished. We show that the extra space can be used in a nontrivial way, to compute uniform TC^1-circuits with just a logarithmic amount of clean space. The extra space thus works analogously to a catalyst in a chemical reaction. TC^1-circuits can compute for example the determinant of a matrix, which is not known to be computable in logspace.

In order to obtain our results we study an algebraic model of computation, a variant of straight-line programs. We employ register machines with input registers $x_1, \ldots, x_n$ and work registers $r_1, \ldots, r_m$. The instructions available are of the form $r_i \leftarrow r_i \pm u \times v$, with u, v registers (distinct from r_i) or constants. We wish to compute a function $f(x_1, \ldots, x_n)$ through a sequence of such instructions. The working registers have some arbitrary initial value $r_i = \tau_i$, and they may be altered throughout the computation, but by the end all registers must be returned to their initial value τ_i, except for, say, r_1 which must hold $\tau_1 + f(x_1, \ldots, x_n)$. We show that all of Valiant's class VP, and more, can be computed in this model. This significantly extends the framework and techniques of Ben-Or and Cleve [BC92].

Upper bounding the power of catalytic computation we show that catalytic logspace is contained in ZPP. We further construct an oracle world where catalytic logpace is equal to PSPACE, and show that under the exponential time hypothesis (ETH), SAT can not be computed in catalytic sub-linear space.

*Supported in part by EU project SIQS.
†Supported in part by Canada's NSERC, CIFAR, and the U.S. ARO.
‡Supported in part by (FP7/2007-2013)/ERC Consolidator grant LBCAD no. 616787.
§Supported by the NWO DIAMANT project and EU project SIQS.

1 Introduction

Imagine the following scenario. You want to perform a computation that requires more memory than you currently have available on your computer. One way of dealing with this problem is by installing a new hard drive. As it turns out you have a hard drive but it is full with data, pictures, movies, files, *etc.* You don't need to access that data at the moment but you also don't want to erase it. Can you use the hard drive for your computation, possibly altering its contents temporarily, guaranteeing that when the computation is completed, the hard drive is back in its original state with all the data intact? One natural approach is to compress the data on the hard disk as much as possible, use the freed-up space for your computation and finally uncompress the data, restoring it to its original setting. But suppose that the data is not compressible. In other words, your scheme has to always work no matter the contents of the hard drive. Can you still make good use of this additional space?

In order to study this question we define the following model of computation, which we call *catalytic space*[1]. We equip the standard Turing machine model — which has input, output, and work tapes — with an additional auxiliary tape. We assume that the Turing machine halts on every input and call it *catalytic* if at the end of every computation, the auxiliary tape is unaltered for *every* possible initial setting of its content. As usual in space-bounded computation we limit the amount of work space by a function $s(n)$, usually logarithmic or polynomial. We define the class $\mathsf{CSPACE}(s(n))$ to be the class of sets that are computed by catalytic Turing machines whose work-tape is bounded by $s(n)$ tape cells, and whose auxiliary space is bounded by $2^{s(n)}$ cells.

Intuition tells us that the auxiliary tape is not very useful since its contents must be present in some way at every step of the computation and if these contents are incompressible, effectively no extra space is available. Surprisingly it appears that that $\mathsf{CSPACE}(\log n)$ is more powerful than ordinary logspace ($\mathsf{DSPACE}(\log n)$ or L), for we show that $\mathsf{TC}^1 \subseteq \mathsf{CSPACE}(\log n)$. Note that TC^1 contains NL and even $\#\mathsf{L}$ and other classes that are conjectured to be different from L. We remark that, although the catalytic requirement of the auxiliary space suggest the computation is reversible, it is not sufficient to have just reversibility, since reversible computation schemes [Ben73, LMT97, BTV01] usually require the initial configuration of all the space cells to be set to some fixed initial value, for example all blanks. However, a stronger version of reversibility, that we call *transparent computation*, suffices. Our reversibility framework is related to the work of Ben-Or and Cleve [BC92] but goes beyond it. We show that the techniques of Ben-Or and Cleve stop at the class of problems that are reducible to iterated matrix product (GapL), whereas our model is able to compute TC^1.

We don't know what the exact power of catalytic logspace is, but show that it is contained in ZPP. It could be possible that every problem in P is computable in $\mathsf{CSPACE}(\log n)$. This would be remarkable. It could be of practical interest in situations when additional clean space is not available, for example when the main memory of a computer is filled with data of an ongoing background computation which may be temporarily stopped, but requires the memory to be unaltered when it continues. On the other hand, $\mathsf{CSPACE}(\log n)$ might be a proper subset of P. There remains the possibility that $\mathsf{CSPACE}(\log n) = \mathsf{L}$. If this is the case then our result implies $\mathsf{L} = \mathsf{NL}$, and the intuition that an additional full memory is useless could lead to an approach for proving this collapse. Lastly, we present an oracle relative to which $\mathsf{CSPACE}(\log n) = \mathsf{PSPACE}$, showing the potential, at least in a relativized world, of the auxiliary tape. We also show that under the exponential-time hypothesis [IP99], $\mathsf{SAT} \notin \mathsf{CSPACE}(o(n))$.

[1] Catalysis refers to the situation where the rate of a chemical reaction is increased by participation of a substance which is not consumed and is available unaltered after the reaction has taken place.

2 Preliminaries

To put our results into a proper context we need to review several problems and related complexity classes.

L, NL, LOGCFL. By L we denote the class of problems solvable in logspace, by NL the class of problems solvable *non-deterministically* in logspace, and by LOGCFL the class of problems that are logspace many-one reducible to context-free languages. Another equivalent characterization of LOGCFL is as the class of languages accepted by non-deterministic logspace-bounded auxiliary push-down automata (AuxPDAs) running in polynomial time [Sud78].

NC^i, SAC^i, AC^i, TC^i. These are classes of boolean functions computed by polynomial-size circuits of depth $(\log n)^i$. The different classes differ by the set of gates that are allowed in the circuit. NC^i-circuits consist of input gates, constant (0/1) gates, binary (fan-in-2) AND and OR gates, and unary NOT gates. SAC^i-circuits additionally allow for the OR gates to have arbitrary fan-in. AC^i-circuits allow for both AND and OR gates to have arbitrary fan-in. TC^i-circuits are additionally allowed to have MAJ gates of arbitrary fan-in (a MAJ gate decides whether most of its input bits are 1).

GapL, #LOGCFL. We also consider counting classes: GapL is the class of functions obtained by counting the difference between the number of accepting and rejecting paths of a non-deterministic logspace machine; $\#\mathsf{LOGCFL}$ is the class of functions that count the number of accepting paths of AuxPDAs running in logarithmic space and polynomial time.

$\mathsf{VP}(R)$, $\mathsf{SkewVP}(R)$. Finally, we will also work with *algebraic* circuits that operate over some ring R. When R is the ring of integers $\mathbb{Z}$, these are also called *arithmetic* circuits. Valiant's class $\mathsf{VP}(R)$ [Val79] is the class of (families of) multivariate polynomials over R, computed by algebraic circuits using addition and multiplication gates over R, that have size and degree $n^{O(1)}$ (where n is the number of variables). $\mathsf{SkewVP}(R)$ is the class of multivariate polynomials which can be computed by $\mathsf{VP}(R)$-circuits, with the further restriction that each multiplication gate is binary and such that one of its inputs is either a constant or an input variable.

$\#\mathsf{NC}^i(R)$, $\#\mathsf{SAC}^i(R)$, $\#\mathsf{AC}^i(R)$. These are classes of families of multivariate polynomials over R that are computed by polynomial-size algebraic circuits of depth $(\log n)^i$. Again, these classes differ only by the set of gates that are allowed. $\#\mathsf{NC}^i(R)$-circuits consist of input gates, constant gates (one such gate for each element in R), and binary addition and multiplication gates. $\#\mathsf{SAC}^i(R)$-circuits further allow for the addition gates to have arbitrary fan-in. $\#\mathsf{AC}^i(R)$-circuits can have both addition and multiplication gates of arbitrary fan-in.

Beside circuit families over a ring R that is the same for all input lengths we also consider circuit families where the circuit for inputs of length n computes over a ring R_n, e.g., $\#\mathsf{NC}^1(M_{n^2\times n^2}(\mathbb{Z}))$ consists of families of multivariate polynomials over the ring of integer matrices, where the size of the matrices is n^2 for n being the number of matrix variables.

$\mathsf{DET}_{n,R}$, $\mathsf{IMM}_{n,m,R}$. By $\mathsf{DET}_{n,R}$ we denote the problem of computing a determinant of an $n \times n$ matrix over a ring R. By $\mathsf{IMM}_{n,m,R}$ we denote the problem of computing the product of n matrices, each over the ring R of dimension $m \times m$. We can omit the subscripts when the ring or dimensions are understood from the context. Typically we may think of R being the ring of integers, and $m = n$.

Relationship among these concepts. We now present known relationships among these classes; see Figure 1 for an overview.[2] It is standard knowledge that $\mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{SAC}^1 \subseteq \mathsf{AC}^1 \subseteq$

[2]Below and in Figure 1, inclusion is not meant in a set-theoretic sense, and should be interpreted with the usual caveats that apply to complexity classes; for instance, $\mathsf{NL} \subseteq \mathsf{GapL}$ in the sense that the characteristic function of any NL decision problem is in GapL; or, to give another example, $\#\mathsf{AC}^0(\mathbb{Z}_p) \subseteq \mathsf{TC}^0$ in the sense that any polynomial in $\#\mathsf{AC}^0(\mathbb{Z}_p)$ can be computed in TC^0 using the canonical encoding of $\mathbb{Z}_p$ (see Section 4.1) But describing this with full

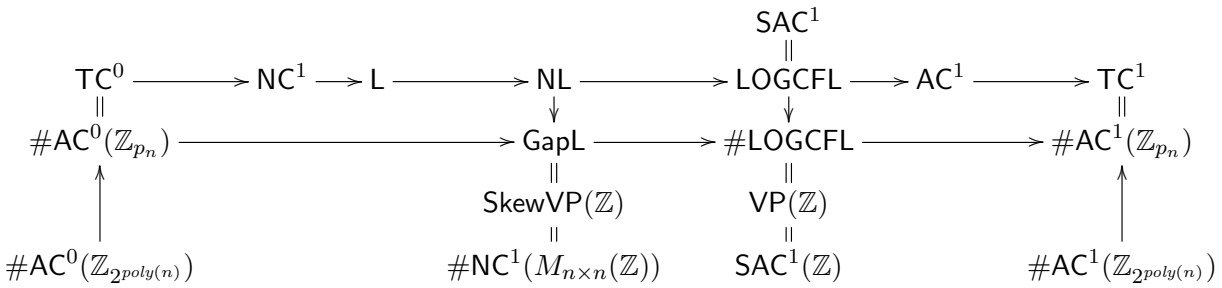

Figure 1: Inclusion diagram for all the classes.

TC^1, but none of these inclusions is known to be proper. TC^0 is known to contain problems such as computing the sum and the product of n-many n-bit integers, computing the division of two n-bit integers, *etc* [BCH86, RT92, HAM02]. It is also (not-as-well) known that $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{SAC}^1 = \mathsf{LOGCFL}$ [Ven91].

The complexity of computing the determinant characterizes GapL. More precisely, f is in GapL if and only if it is logspace many-one reducible to $\mathsf{DET}_{n,\mathbb{Z}}$ [Tod91, Dam91, Vin91, Val92]. Cook and others [Coo85, AO94] have shown that the class of problems logspace many-one reducible to DET is the same as the class of problems logspace many-one reducible to IMM.[3] Taken over the integers, $\mathsf{SkewVP}(\mathbb{Z})$ equals GapL [Tod92], and also $\#\mathsf{NC}^1(M_{n^{O(1)}\times n^{O(1)}}(\mathbb{Z}))$, the class of log-depth fan-in-2 circuits over integer matrices.[4]

$\#\mathsf{SAC}^1(R)$ is actually a characterization of $\mathsf{VP}(R)$ [VSBR83], a deep result of depth-reduction for algebraic circuits. Taken over the integers $\mathsf{VP}(\mathbb{Z})$ equals $\#\mathsf{LOGCFL}$ [Vin91].

The question posed by Valiant [Val79] about the relationship between the determinant and $\mathsf{VP}(\mathbb{Z})$, namely, whether evaluating a $\mathsf{VP}(\mathbb{Z})$ circuit reduces to evaluating the determinant of a matrix that is at most polynomially larger in size (or, equivalently, whether $\mathsf{SkewVP}(\mathbb{Z}) = \mathsf{VP}(\mathbb{Z})$), is no different to the question about the relationship between GapL and $\#\mathsf{LOGCFL}$.

[AAD00, BFS92, RT92] establish a relationship between the classes TC^i and $\#\mathsf{AC}^i(R)$ over various integral rings and finite fields. For instance, it is shown in [RT92] that $\mathsf{TC}^i \subseteq \#\mathsf{AC}^i(\mathbb{Z}_{p(n)})$, where $p(n)$ is any prime number larger than the maximum fan-in of the TC^i circuit to be simulated (for inputs of length n), and that, conversely, $\#\mathsf{AC}^i(\mathbb{Z}_{f(n)}) \subseteq \mathsf{TC}^i$ holds for any function $f(n) = O(2^{poly(n)})$.

A remark on TC^1 versus GapL. Immerman and Landau [IL95] conjecture that computing determinant over the integers is hard for TC^1. However, there is evidence suggesting that this is not the case. Namely, it is known that TC^1 circuits can evaluate $\#\mathsf{AC}^1$ circuits over $\mathbb{Z}_m$, the ring of integers mod m, for exponentially large m.[5] If the Immerman-Landau conjecture were true then $\#\mathsf{SAC}^1$ circuits over the integers — which compute polynomials of degree polynomial in the number of inputs — could simulate TC^1, and hence $\#\mathsf{AC}^1$. But the latter can have super-polynomial

precision would give a cluttered, poorer exposition.

[3] A function f is logspace many-one reducible to the determinant if there is a function g computable in logspace such that $f(x)$ (viewed as a number written in binary) is equal to the determinant of matrix $g(x)$.

[4] This follows from [BC92, Cle89], see Theorem 2 below.

[5] This is because TC^0 circuits can evaluate an iterated sum and iterated product of integers, as well as compute the remainder mod m. TC^1 circuits cannot evaluate $\#\mathsf{AC}^1$ circuits over unbounded integers since $\#\mathsf{AC}^1$ circuits represent polynomials of degree up to $n^{O(\log n)}$, and hence the encoding of their output may require super-polynomially many bits.

degree! This conclusion can not be ruled out entirely, because while polynomials of $n^{O(1)}$ degree over integer variables can not simulate polynomials of larger degree over integer variables, they could still conceivably simulate polynomials of $n^{\log n}$ degree over integers modulo 2^n ($\mathbb{Z}_{2^n}$). But this does seem unlikely.

3 Transparent computation

The model for transparent computation is a variant of straight-line programs. The computational device is a *register machine* equipped with read-write working registers $\vec{r} = r_1, r_2, \ldots, r_m$ and read-only input registers $\vec{x} = x_1, \ldots, x_n$. Each register x_i or r_i holds a value from some designated ring R. The standard set of instructions — called *standard basis* — consists of *instructions* of the form $r_i \leftarrow r_i \pm u \times v$, where u and v are either elements of R ("constants"), or registers different from r_i, and the $+, -$ and $\times$ are the operations of R. These instructions are said to be *reversible*, and for an instruction I, its *inverse* I^{-1} is I with the $+$ or $-$ interchanged.[6] Moreover when at least one of the u and v is an input register or constant we call the instruction *skew*, and the *skew basis* is the standard basis restricted to skew instructions

A *program* for this register machine is a sequence of reversible instructions, and we also call these programs *reversible*. Thus for a reversible program $P = I_1, I_2, \ldots, I_\ell$ we let the *inverse program* P^{-1} be $I_\ell^{-1}, I_{\ell-1}^{-1}, \ldots, I_1^{-1}$. It is easy to verify that P, P^{-1} computes the identity.

We say that a program P *uses register* r if one of its instructions involves this register, e.g., $r_1 \leftarrow r_1 + r_4 \cdot r_7$ uses registers r_1, r_4 and r_7.

We say that $f(\vec{x})$ can be *computed transparently* into a register r_i if there is a reversible program P that when executed on registers $r_1, r_2, \ldots, r_m$ with initial values $\tau_1, \tau_2, \ldots, \tau_m$ ends with value $\tau_i + f(\vec{x})$ in register r_i; the other registers may contain arbitrary values at the end of the computation. (We will always use τ_i to denote the initial value held in register r_i before executing a program.) Clearly, if we have a program that transparently computes f into a register r we can modify it by relabeling registers to compute f transparently into a different register. We may also want P to transparently compute a vector of functions $(f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x}))$ into registers $r_{i_1}, r_{i_2}, \ldots, r_{i_k}$, meaning that the execution of P ends with the value $\tau_{i_j} + f_j(\vec{x})$ in each register r_{i_j}.

Transparent computation is a very special type of reversible computation as it has the additional property that the computation is meaningful regardless of the initial setting of the working registers.[7] Hence the choice of name: the computation is "transparent," in the sense that it somehow *sees through* the contents of the working registers. This property is *not* universally shared by reversible models of computation. Our model is a variant of the model considered by Coppersmith and Grossman [CG75], and by Ben-Or and Cleve [BC92].

[6] Generally speaking, the reversibility property would hold for any instruction of the form $r_i \leftarrow \sigma_{\vec{x},\vec{r}_{\neq i}}(r_i)$, where $\sigma_{\vec{x},\vec{r}_{\neq i}}$ is a permutation of R which may arbitrarily depend on the input registers and on the work registers other than r_i. Also, in principle, different registers could work over different domains. In this paper we do not make use of these possibilities, but they may appear in future work.

[7] Furthermore, and quite remarkably, the following can be shown: let $R(t)$ be the contents of the working registers after executing t instructions of some transparent program, and let $X = X_1, \ldots, X_n$ be the input; then for any t, $I(R(t) : X) = I(R(0) : X)$, where I denotes the common information, either in the Shannon or Kolmogorov sense (input and registers must be suitably specified, respectively as a distribution or as a binary string, in order to fit in either framework; details are left to the reader).

In particular, if the initial contents of the registers are independent of the input ($I(R(0) : X) = 0$), then at any point in the computation, the register machine knows nothing about the input, other than whichever specific register X_i it might be accessing directly (as when executing the instruction $r \leftarrow r + X_i$, for instance).

It should be noted, however, that if one is to look at two distinct time-steps t_1 and t_2, some information about X could be derived, i.e., it could hold that $I(R(t_1), R(t_2) : X) > I(R(0) : X)$.

Definition 1. $\mathsf{TP}(R, s, m)$ *is the class of functions transparently computed by reversible programs over the standard basis over ring* R*, having at most* s *instructions and using at most* m *registers.* $\mathsf{TP}(R)$ *is the class of (families of) functions in* $\mathsf{TP}(R, poly, poly)$*.* $\mathsf{SkewTP}(R, s, m)$ *and* $\mathsf{SkewTP}(R)$ *are analogously defined for the skew basis.*

Coppersmith and Grossman [CG75] have shown that the class $\mathsf{TP}(\mathbb{Z}_2, 2^{O(n)}, O(1))$ contains all boolean functions (cf. [Cle89]). The reason why we are interested in transparent computation is because it allows us to restore the work registers to their initial values. For suppose that we have a reversible program P that transparently computes $f(\vec{x})$ into register r_1, while freely modifying the contents of other registers. Then we can take the program P': $r \leftarrow r - r_1, P, r \leftarrow r + r_1, P^{-1}$, where r is a register not used by P. While this new program still transparently computes $f(\vec{x})$ into r, all of the remaining registers are returned to their initial value. We then say that P' *cleanly* (as well as transparently) *computes* $f(\vec{x})$ into register r.

Uniformity. Our class $\mathsf{TP}(R)$ is a non-uniform class. Naturally, we may consider also its uniform variant. All our results in which we simulate circuits by transparent programs essentially preserve the uniformity, so a uniform family of circuits is simulated by a uniform family of transparent programs. There is only a slight loss in our Powering Lemma where we hardwire binomial coefficients into the transparent program. Since the necessary binomial coefficients can easily be computed in logarithmic space the resulting transparent program is still at least logspace uniform if the circuit family is. This also affects all our results that use the Powering Lemma, including our main result on simulation of TC^1. A possible way to avoid this loss in uniformity is to construct very uniform transparent programs that would compute the binomial coefficients.

3.1 Previous results on this model

It is a natural question to ask: what functions can be transparently computed by small programs over the standard basis, or over other bases? We do not have a precise answer to this question but we will be able to show that all functions in the circuit class TC^1 can be computed transparently by polynomial size programs over the standard basis. This greatly extends the result of Ben-Or and Cleve [BC92] who in essence show that any function in NC^1 can be computed transparently by a polynomial size program using three registers. Cleve in his thesis [Cle89] shows a result slightly stronger than [BC92], namely that iterated matrix product can be computed transparently by polynomial size programs over the standard basis. Indeed, an inspection of the proof, together with the technique of Ben-Or and Cleve, shows that the iterated matrix product can be computed transparently by polynomial size programs over the skew basis. In particular, iterated matrix product of n matrices can be represented by a formula over $R^{m\times m}$ of depth $\log n$. Using the same techniques, we can prove a tight characterization of $\mathsf{SkewTP}(R)$.

Theorem 2. *Let* $f(x_1, \ldots, x_n)$ *be a polynomial over a ring* R*.*

(a) If f *can be represented as an entry of a* d*-depth formula over the ring* $M_{m\times m}(R)$*, where each entry in each matrix input to this formula is either an element of* R*, or* $\pm x_i$ *for some* i*, and* $m = poly(n)$*, then* f *is in* $\mathsf{SkewTP}(R, O(m^3 4^d), O(m^2))$*.*

(b) If f *is in* $\mathsf{SkewTP}(R, s, m)$*, then* f *can be represented as an entry in the product of* s*-many* $(m+1)\times(m+1)$ *such matrices.*

Proof. The first part is a restatement of Theorem 3.3.1 of [Cle89]. For the given parameters, it follows that $f \in \mathsf{SkewTP}(R, O(m^3 4^d), O(m^2))$. The only minor difference is that Theorem 3.3.1

of [Cle89] uses standard basis instructions and not our skew basis. However, the inspection of the proof together with the technique of Ben-Or and Cleve [BC92] shows that the theorem is true also for the skew basis.

Now suppose that $f \in \mathsf{SkewTP}(R, S, m)$. Consider the $(m+1)$-dimensional vector $R_0 = (0, \ldots, 0, 1)$, where the first m entries represent the values of registers $r_1, \ldots, r_n$ used by the program, and the last entry represents a constant one. The skew instruction $r_i \leftarrow r_i \pm r_j \cdot v$, where v is either an element of R or a variable x_i, can be represented by the $(m+1) \times (m+1)$ matrix having 1 on the diagonal, $\pm v$ in the (j, i) position, and 0 elsewhere. The instruction $r_i \leftarrow r_i + v$ can be represented by an identity matrix with the entry $(m+1, i)$ set to v. These matrices will act on the vector R_0 in the same way as their corresponding instructions. If the program transparently computes f into r_1 then the $(1, m+1)$ entry of the product of the matrices corresponding to the program gives f. For each instantiation of $\vec{x}$, this product can be computed by a balanced ($O(\log n)$-depth) tree of product gates over the ring $M_{m \times m}(R)$. ■ From the GapL-completeness of IMM over $\mathbb{Z}$, we get:

Corollary 3. $\mathsf{SkewTP}(\mathbb{Z}) = \mathsf{GapL} = \mathsf{SkewVP}(\mathbb{Z}) = \#\mathsf{NC}^1(M_{n^{O(1)} \times n^{O(1)}}(\mathbb{Z}))$.

3.2 Getting more

The previous characterization tells us that, to go beyond GapL, we can not restrict ourselves to skew instructions. We will now show how to use reversible programs to transparently compute $\#\mathsf{SAC}^1(R)$. We must then be able to transparently compute binary product and unbounded sum.

Lemma 4 (Binary product)**.** *Let r_0, r_1, r_2, r_3, r_4 be registers over some ring R. There are reversible programs I_1, I_2, I_3 over the standard basis using registers over R such that for any reversible program P that does not use r_0, r_3 and r_4 and that transparently computes $r_1 \leftarrow \tau_1 + f_1(\vec{x})$ and $r_2 \leftarrow \tau_2 + f_2(\vec{x})$, the program I_1, P, I_2, P^{-1}, I_3 computes $r_0 \leftarrow \tau_0 + f_1(\vec{x}) \times f_2(\vec{x})$. The total length of I_1, I_2, I_3 is eight instructions.*

Proof. The following program computes the required product. The right-hand side indicates the result of applying the instructions on the left-hand side.

1. $r_0 \leftarrow r_0 + r_1 r_2 + r_1 r_4 + r_3 r_2$ // $r_0 = \tau_0 + \tau_1\tau_2 + \tau_1\tau_4 + \tau_3\tau_2$
2. P // $r_i = \tau_i + f_i(\vec{x})$, for $i = 1, 2$
3. $r_3 \leftarrow r_3 + r_1$ // $r_3 = \tau_3 + \tau_1 + f_1(\vec{x})$
   $r_4 \leftarrow r_4 + r_2$ // $r_4 = \tau_4 + \tau_2 + f_2(\vec{x})$
   $r_0 \leftarrow r_0 + r_1 r_2$ // $r_0 = \tau_0 + f_1(\vec{x})f_2(\vec{x}) + \tau_1(\tau_4 + \tau_2 + f_2(\vec{x})) + (\tau_3 + \tau_1 + f_1(\vec{x}))\tau_2$
4. P^{-1} // $r_i = \tau_i$, for $i = 1, 2$
5. $r_0 \leftarrow r_0 - r_1 r_4 - r_3 r_2$ // $r_0 = \tau_0 + f_1(\vec{x})f_2(\vec{x})$

The first statement, which can be implemented using three standard basis instructions, forms I_1; the statements from line 3 form I_2; and the two instructions corresponding to line 5 form I_3. ■

Lemma 5 (Unbounded sum)**.** *Let $r_0, r_1, r_2, \ldots, r_k$ be registers over some ring R. There are reversible programs I_1 and I_2 over the standard basis using registers over R such that for any reversible program P that does not use r_0 and that for each $i = 1, \ldots, k$ transparently computes $r_i \leftarrow \tau_i + f_i(\vec{x})$, the program I_1, P, I_2 computes $r_0 \leftarrow \tau_0 + \sum_{i=1}^{k} f_i(\vec{x})$. The total length of I_1, I_2 is $2k$.*

Proof. The following program computes the sum.

1. For each $i = 1, \ldots, k$ do $r_0 \leftarrow r_0 - r_i$.
2. P
3. For each $i = 1, \ldots, k$ do $r_0 \leftarrow r_0 + r_i$.

The first statement which corresponds to k standard basis instructions forms I_1, and the k instructions from line 3 form I_2. ■

Corollary 6. *If R is a ring and f is computed by a depth-d arithmetic circuit with w wires and s gates for binary product and unbounded fan-in addition, then*

$$f \in \mathsf{TP}(R, O(dw2^{d+1}), O(s)) .$$

Proof. Let C be the depth-d circuit for f of given properties. Let us assume that C is layered, that is, each gate at level ℓ takes as its inputs gates at level $\ell - 1$. For every gate g_i of C we will have an auxiliary register r_i into which we will transparently compute the value of g_i. We will compute the values of gates inductively level by level.

If g_i is an input gate then it corresponds either to a constant $c \in R$ or to an input variable x_j. In the former case the instruction $r_i \leftarrow r_i + c$ transparently computes the value of g_i, and in the latter case $r_i \leftarrow r_i + x_j$ does the job. A concatenation of such instructions in arbitrary order for all the input gates gives a program that simultaneously and transparently computes the values of input gates into their associated registers.

Assume that we already have a program $P_{\ell-1}$ that simultaneously and transparently computes the values of gates at the level $\ell - 1$ into appropriate registers. If g_i is a gate at level ℓ then it is either the sum of the values of gates at the level $\ell - 1$ or their binary product. By the Unbounded Sum Lemma or by the Binary Product Lemma, there are programs I_1^i, I_2^i, I_3^i such that $I_1^i, P_{\ell-1}, I_2^i, P_{\ell-1}^{-1}, I_3^i$ transparently computes g_i into r_i. (We can and will assume that I_1^i, I_2^i, I_3^i use different auxiliary registers for different i.) If $g_{i_1}, g_{i_2}, \ldots, g_{i_k}$ are the gates at level ℓ then

$$P_\ell = I_1^{i_1}, \ldots, I_1^{i_k}, P_{\ell-1}, I_2^{i_1}, \ldots, I_2^{i_k}, P_{\ell-1}^{-1}, I_3^{i_1}, \ldots, I_3^{i_k}$$

computes simultaneously and transparently the values of the gates at level ℓ into appropriate registers. In this way we obtain a program P_d for transparently computing the value of C.

If the size of the program P_ℓ is S_ℓ then $S_\ell \leq 2S_{\ell-1} + 4w_\ell$, where w_ℓ is the number of wires leading into the gates at the level ℓ. The number of input gates can be bounded by w, so $S_1 \leq w$. Thus $S_\ell \leq 6w2^{\ell-2} \leq w2^{\ell+1}$. Each gate uses at most three registers, and hence our final program will use $O(s)$ registers. This is under the assumption that C is layered. Any circuit can be transformed into a layered one while increasing its number of wires by a factor of at most d. ■

We thus get a potentially larger class of functions than that of Ben-Or and Cleve:

Corollary 7. *For any ring R, $\#\mathsf{SAC}^1(R) \subseteq \mathsf{TP}(R)$. In particular,*

$$\#\mathsf{LOGCFL} = \#\mathsf{SAC}^1(\mathbb{Z}) = \mathsf{VP}(\mathbb{Z}) \subseteq \mathsf{TP}(\mathbb{Z}).$$

3.3 Getting TC^1

To go even further and obtain TC^1 we will need the ability to compute the n-th power of a gate. We will show how to do this over commutative rings, but we do not know how to proceed in the non-commutative case.

The following lemma gives a small-length program for computing the iterated product of registers.

Lemma 8 (Iterated product)**.** *There is a program P with $2k+1$ instructions from the standard basis over R that transparently computes, for every $i \leq k$,*

$$r_i \leftarrow \tau_i + m_1 \times \ldots \times m_i,$$

where $m_1, \ldots, m_k$ are either input registers, work registers (different from the r_i), or constants.

Proof. The following program computes the product.

1. For $i = k, \ldots, 2$ do $r_i \leftarrow r_i - r_{i-1} \times m_i$.
2. $r_1 \leftarrow r_1 + m_1$
3. For $i = 2, \ldots, k$ do $r_i \leftarrow r_i + r_{i-1} \times m_i$. ■

Notice that this lemma is different from the binary product or unbounded sum lemmas, in that we do not prove how to inductively compute the iterated product of the outputs of some given program. In fact, we currently do not know how to prove this.

To compute the n-th power over commutative rings, we will need the following variant of the usual binomial expansion.

Lemma 9. *For any elements a, x of a commutative ring, and any integer $k \geq 1$, the following holds:*

$$(a+x)^k = x^k + \sum_{i=1}^{k} (-1)^{i-1} \binom{k}{i} a^i (a+x)^{k-i}$$

Proof. Let us consider the binomial expansion of $(a + x - a)^k$.

$$\begin{aligned} x^k = (a+x-a)^k &= \sum_{i=0}^{k} \binom{k}{i} (-a)^i (a+x)^{k-i} \\ &= (a+x)^k + \sum_{i=1}^{k} (-1)^i \binom{k}{i} a^i (a+x)^{k-i} \end{aligned}$$

Now the lemma immediately follows. ■

Lemma 10 (Powering)**.** *Let k be a positive integer. Let r_0 and r be registers over some* commutative *ring R. There are programs I_1, I_2 and I_3 over the standard basis registers over R such that for any program P that does not use any registers used by I_1, I_2, I_3 other than r and that transparently computes*

$$r \leftarrow \tau + f(\vec{x}),$$

the program I_1, P, I_2, P^{-1}, I_3 computes

$$r_0 \leftarrow \tau_0 + [f(\vec{x})]^k.$$

The total length of I_1, I_2, I_3 is $O(k)$, and $O(k)$ registers are used.

Proof. Assume we have auxiliary registers $r_1, r_2, \ldots, r_k$. Then for the constants $c_i = (-1)^{i-1}\binom{k}{i}$, $i = 1, \ldots, k$, the following program computes the power of $f(\vec{x})$.

1. For $i = 1, \ldots, k$ do $r_0 \leftarrow r_0 + c_i \cdot r_i \cdot r^{k-i}$.
2. P
3. For $i = 1, \ldots, k$ do $r_i \leftarrow r_i + r^i$.
   $r_0 \leftarrow r_0 + r^k$.
4. P^{-1}
5. For $i = 1, \ldots, k$ do $r_0 \leftarrow r_0 - c_i \cdot r_i \cdot r^{k-i}$.

This can be seen as before by carefully tracking the contents of the registers, and eventually by applying Lemma 9. By the Iterated Product Lemma the first line can be implemented using a program over the standard basis of size $O(k)$. This will be I_1. Similarly, line 3 and line 5 can each be implemented by a similar-size program I_2 and I_3, respectively. This would give programs of size $O(k)$ using $O(k)$ registers. ■

Lemma 11 (Exact value)**.** *Let p be a prime, R be the field $\mathbb{Z}_p$, and $s \in R$. Let $r_0, r_1, r_2, \ldots, r_k$ be registers over R. There are programs I_1, I_2 and I_3 over the standard basis using registers over R such that for any program P that does not use r_0 and that transparently computes for each $i = 1, \ldots, k$*

$$r_i \leftarrow \tau_i + f_i(\vec{x}),$$

the program I_1, P, I_2, P^{-1}, I_3 computes

$$r_0 \leftarrow \tau_0 + [\![\sum_{i=1}^{k} f_i(\vec{x}) \neq s]\!],$$

where $[\![\sum_{i=1}^{k} f_i(\vec{x}) \neq s]\!]$ equals 1 if $\sum_{i=1}^{k} f_i(\vec{x}) \neq s$ and equals 0 otherwise. The total length of I_1, I_2, I_3 is $O(p+k)$, and $O(p)$ registers are used.

Proof. By the Unbounded Sum Lemma we have programs I_1' and I_2' such that for any program P that simultaneously and transparently computes $r_i \leftarrow r_i + f_i(\vec{x})$, the program $P' = I_1', P, I_2'$ transparently computes $\sum_{i=1}^{k} f_i(\vec{x}) - s$ into an auxiliary register r. The total length of I_1', I_2' is $2k + 1$. Notice, $\sum_{i=1}^{k} f_i(\vec{x}) - s$ is non-zero iff $\sum_{i=1}^{k} f_i(\vec{x}) \neq s$. Since R is a field of size p, by Fermat's little theorem, $(\sum_{i=1}^{k} f_i(\vec{x}) - s)^{p-1}$ is one iff $\sum_{i=1}^{k} f_i(\vec{x}) - s$ is non-zero. Hence, by the Powering Lemma, we have programs I_1'', I_2'', I_3'' such that $I_1'', P', I_2'', P'^{-1}, I_3''$ transparently computes $(\sum_{i=1}^{k} f_i(\vec{x}) - s)^{p-1}$, i.e., $[\![\sum_{i=1}^{k} f_i(\vec{x}) \neq s]\!]$. Setting $I_1 = I_1'', I_1'$, setting $I_2 = I_2', I_2'', (I_2')^{-1}$ and setting $I_3 = (I_1')^{-1}, I_3''$ gives the required programs. Their total length is $2(2k+1) + O(p)$. ■

Corollary 12. *Let a function f be computed by a depth-d boolean circuit consisting of at most s MAJ-gates, each of fan-in at most k. Let $p > k$ be a prime. Then $f \in \mathsf{TP}(\mathbb{Z}_p, O(dpks4^d), O(dksp))$.*

Proof. First, notice that MAJ gates can be simulated using the Exact Value gates. Indeed, let $b_1, b_2, \ldots, b_k$ be bits where k is even. Then

$$[\![\sum_{j=1}^{k/2} [\![\sum_{i=1}^{k} b_i \neq j]\!] \neq k/2]\!]$$

if and only if

$$\sum_{i=1}^{k} b_i > k/2.$$

Similarly for odd k. Hence, the depth-d circuit C for f consisting of MAJ gates has an equivalent depth-$2d$ circuit C' consisting of the Exact Value gates. The number of gates in C' is at most $O(ks)$. Making C' layered may increase the number of gates by a factor of $2d$. Using the same technique as in the proof of Corollary 6 we can transparently simulate the computation of C' by a reversible program. Each gate of C' will require additional computation of size $O(k+p)$, and uses $O(p)$ registers. Since, there are $O(dks)$ gates this will contribute by $O(dks(k+p))$ instructions using $O(dksp)$ registers. However, as we proceed layer by layer in constructing the program for C', the number of instructions gets multiplied by a factor of at most 2^{2d} as the instructions for each gate get copied twice at each sub-sequent layer. Hence, in total we obtain a program of length $O(2^{2d}(dk^2s + dkps)) = O(4^d dkps)$. ■

Allender and Koucký [AK10] show that for any $\epsilon > 0$, one can simulate MAJ-gate of fan-in n by a uniform constant depth circuit of polynomial size consisting of MAJ-gates of fan-in at most n^ϵ. Hence, in the previous lemma we could use polynomially smaller primes for the cost of increasing the size of the resulting program by a polynomial factor. We can state our main technical result.

Theorem 13. *For any sequence of primes $(p_n)_{n\in\mathbb{N}}$ of size polynomial in n, $\mathsf{TC}^1 \subseteq \mathsf{TP}(\mathbb{Z}_{p_n})$.*

Note, we can find polynomially large primes in logspace so if f is computable by a logspace uniform family of TC^1 circuits then f is transparently computable by a logspace uniform family of polynomial size transparent programs.

Because of the relationship between TC^1 and $\#\mathsf{AC}^1$ the previous theorem allows us to simulate the computation of $\#\mathsf{AC}^1$ circuits over $\mathbb{Z}_m$, the ring of integers modulo m, where m can be exponentially large. Because the degree of the polynomials computed by $\#\mathsf{AC}^1(\mathbb{Z}_m)$ circuits can be as high as $n^{\log n}$, this seems to give a significant improvement over GapL and $\#\mathsf{LOGCFL}$.

4 Catalytic computation

A *catalytic Turing machine* is a Turing machine equipped with a read-only input tape, a work tape[8], and an extra tape — the *auxiliary tape*. For every possible initial setting of the auxiliary tape, at the end of the computation the catalytic Turing machine must have returned the tape to its initial contents.

We say a language L is decided by a catalytic Turing machine M if for any string $\mathbf{x}$, and for any string $\mathbf{a}$ representing the initial contents of the auxiliary tape, $M(\mathbf{x}, \mathbf{a})$ halts with contents of the auxiliary tape being exactly $\mathbf{a}$ and $M(\mathbf{x}, \mathbf{a})$ accepts if and only if $\mathbf{x} \in L$.

Definition 14. *Let $S, S_a : \mathbb{N} \to \mathbb{N}$. We define the class $\mathsf{CSPACE}(S(n), S_a(n))$ to be the set of all languages that can be decided by a catalytic machine using $O(S(n))$ space of the work tape and $O(S_a(n))$ auxiliary space of the auxiliary tape, for an input of length n.*

As a notational shorthand we define $\mathsf{CSPACE}(S(n)) = \mathsf{CSPACE}(S(n), 2^{O(S(n))})$ as the set of languages that can be decided by a catalytic machine with a work tape of size $S(n)$. We take the auxiliary space exponential in $S(n)$, the largest amount of auxiliary space which can be addressed when using the machine's work tape.

[8]For simplicity, the Turing machine's alphabet is assumed to be $\{0, 1\}$, but the model naturally extends to larger alphabets.

We will pay the most attention to the setting where the machine has work tape of logarithmic size, which we call catalytic logspace or $\mathsf{CSPACE}(\log n)$.

4.1 Simulation of transparent computation by catalytic computation

Our goal is to present now several surprising containments in the catalytic logspace. To achieve that, we will show how to simulate transparent programs in catalytic logspace, how to extract the value of a function from the transparent computation, and how to deal with uniformity issues.

Let us first observe that, in the same way in which one can compose logspace reductions, we can compose constantly many reductions running in catalytic logspace into a single reduction that will also run in catalytic logspace. In this case the total work space will be roughly the sum of the work space used by each of the reductions, but the same auxiliary space can be reused by each of the reductions, since it is returned to its original content after each use. We will heavily use such compositions in this section.

Before proceeding further let us specify what we mean by a uniform sequence of rings.[9] We say that a map $h : R \to \{0,1\}^*$ is a *compact encoding* of the ring R if h is a bijection between R and the lexicographically first $|R|$ strings of length $\ell = \lceil \log_2 |R| \rceil$.[10] We say that a family of rings $(R_n)_{n=1}^{\infty}$ is *logspace uniform*, if there are logspace-bounded Turing machines M, M_+, M_c, M_s and a family $(h_n)_{n=1}^{\infty}$ of compact encodings of $(R_n)_{n=1}^{\infty}$, such that (1) on input $(1^n, h_n(u) \circ h_n(v))$, M outputs $h_n(u \circ v)$, where $u, v \in R$ and $\circ \in \{+, -, \times\}$; (2) with $(1^n, h_n(v))$ written on a read-only tape and $h_n(u)$ written on a read-write tape, M_+ transforms $h_n(u)$ *in-place* into $h_n(u+v)$ for any $u, v \in R_n$ (possibly using $O(\log n)$ of extra space); (3) on input 1^n, M_c outputs $h_n(-1), h_n(0), h_n(1)$ and M_s outputs $|R_n|$.

Examples of logspace uniform families are $(\mathbb{Z}_2)_{n=1}^{\infty}$ and $(\mathbb{Z}_{2^n})_{n=1}^{\infty}$. More generally, if a sequence of numbers $m_1, m_2, \ldots$ is itself logspace uniform in the usual sense then $(\mathbb{Z}_{m_n})_{n=1}^{\infty}$ is logspace uniform. (This follows since addition, multiplication and taking remainder are all computable in logspace, and adding and subtracting two integers can be done *in-place*.) In the case of $\mathbb{Z}_m$, we will make use of the *canonical* compact encoding mapping $n \in \mathbb{Z}_m$ to the n-th $\lceil \log m \rceil$-bit string in the lexicographical order. In this case, the encoding of the binomial coefficients $\binom{n}{k}$ can be computed in $O(\log m)$ space, which will be important for the TC^1 simulation in Section 3.3.

The following is our key simulation lemma.

Lemma 15 (Catalytic simulation)**.** *For any logspace uniform family of rings $(R_n)_n$, there is a logspace catalytic machine M that on input (P, x) outputs $f(x)$, where P is a transparent program using registers $r_1, r_2, \ldots, r_m$ over $R_{|x|}$ that transparently computes $f(x)$ into r_1. Furthermore, M uses $(m \cdot \lceil \log_2 |R_{|x|}| \rceil)^2$ bits of auxiliary space, and logarithmic (in terms of length of P and x) amount of work-space.*

Proof. The machine M will compute $f(x)$ by simulating P in the auxiliary space. Let $n = |x|$. To simulate registers $r_1, \ldots, r_m$ of P the machine will view its auxiliary space as consisting of blocks each having $b = \lceil \log_2 |R_n| \rceil$ bits. Each of the blocks may be used as a register.

Consider first the case when $|R_n|$ is a power of two. Then the first m blocks of the auxiliary space can be used to represent the values of registers $r_1, \ldots, r_m$. As the sequence of rings is uniform, in

[9] The well-endowed rings defined by Borodin, Cook and Pippenger [BCP83] are similar, but have different requirements.

[10] The encoding is called compact because in some cases using the lexicographically first $|R|$ strings forces the encoding to be unnatural. This happens in the case of prime fields $\mathbb{F}_{p^n}$ for $p > 2$ and $n > 1$, where the most natural encoding would be n blocks of $\lceil \log_2 p \rceil$ bits, each holding a $\mathbb{Z}_p$ coefficient; but such a natural encoding does not map into the lexicographically first strings of $n\lceil \log_2 p \rceil$ bits, so it is not a *compact* encoding! We will need the encoding to be compact in order to simulate register machines using a full memory.

logspace we can simulate any instruction in the standard basis. Hence, in logspace we can simulate P. To compute the value $f(x)$, we can design a reduction that first outputs the content of r_1, that is the initial content τ_1 of the first block of the auxiliary space, then simulates P and again outputs the content of r_1, this time holding the value $\tau_1 + f(x)$, and finally runs P^{-1} which restores the original content of the auxiliary space. Clearly, this is a reduction running in catalytic logspace. By composing this reduction with one which subtracts the two output values obtained by the previous reduction, we get a program computing $f(x)$.

When $|R_n|$ is not a power of two, we will proceed similarly but we have to represent registers differently. We split our auxiliary space into m groups of mb blocks (each block having b bits as before). Two possibilities may happen: either there is a group in which none of the blocks represents a value from R_n, or each group has a block that represents a value from R_n.

In the first case, if b bits do not represent a value from R_n, then — because our encoding of R_n is compact — they have their first bit set to one. Thus in this case there is a group of mb blocks where the first bit of each block is set to one. These mb bits can be used to simulate m registers of P. We will first erase them, then simulate P, output the content of the first register, which holds $f(x)$, and in the end reset the mb bits back to one.

In the second case, we will use the first block representing a value from R_n in the i-th group to represent the register r_i. Since during the simulation of P, register r_i always contains a value from R_n, it is uniquely determined during the whole computation and we can locate it in logspace. Using the same strategy as in the case of R_n having size of power of two we can compute $f(x)$ while restoring the auxiliary space to its original contents. ■

We remark that we could save on the auxiliary space, and instead of using $(m \cdot \lceil \log_2 |R_{|x|}| \rceil)^2$ bits of auxiliary space, we could use only $O(m \cdot \lceil \log_2 |R_{|x|}| \rceil)$ bits if we were to use some stronger compression of the high order bits in the case when there are insufficiently many blocks representing values from R_n.

It is clear that if a sequence of programs $(P_n)_n$ is logspace constructible — where the programs are over some logspace constructible sequence of rings and P_n transparently computes f_n into a register r_1 — then we can compute the function family $(f_n)_n$ in catalytic logspace.

Corollary 16. *Let $(P_n)_n$ be a logspace uniform sequence of programs over some logspace constructible sequence of rings. Let P_n transparently compute f_n into a register r_1. Then the function family $(f_n)_n$ is in catalytic logspace.*

We remark that our constructions of transparent programs in Section 3 are all logspace-uniform. Thus, from the results in Section 3 we conclude, quite surprisingly, that a computer which has plenty of occupied memory is (to the extent we believe that $\mathsf{TC}^1 \not\subseteq \mathsf{L}$) more powerful than one that does not.

Theorem 17. $\mathsf{TC}^1 \subseteq \mathsf{CSPACE}(\log n)$, *for logspace uniform TC^1.*

The Ben-Or & Cleve construction of Theorem 2(a) is also uniform. From this (using Chinese remaindering computable in logspace) we obtain a result incomparable to the above:

Theorem 18. *Iterated matrix product of n matrices over $\mathbb{Z}$, each of dimension $m(n) \times m(n)$, can be computed in logspace with $O(m(n)^2 \cdot \log n)$ bits of auxiliary space. In particular, the iterated matrix product of n matrices over $\mathbb{Z}$, each of dimension $2^{\sqrt{\log n}} \times 2^{\sqrt{\log n}}$, can be computed in logspace with sub-polynomial ($2^{O(\sqrt{\log n})}$) auxiliary space.*

Thus even if the auxiliary space is of less than polynomial size, in catalytic logspace we can still compute functions that are not known to be in the ordinary logspace.

4.2 Upper bounds

Let $\mathsf{ZTIME}(T(n))$ be the set of languages decidable by a zero-error probabilistic Turing machine that runs in expected time $O(T(n))$ for any input of length n.

Theorem 19. $\mathsf{CSPACE}(S(n)) \subseteq \mathsf{ZTIME}(2^{O(S(n))})$.

Proof. Consider an input x of length n, and let $s = O(S(n))$ be the available space on the work tape and s_a be the size of the auxiliary tape of the machine M. Since the total space available to the catalytic machine equals $s + s_a$, it has at most $O(2^{s+s_a})$ possible configurations. We take s_a to be at most $2^{O(s)}$.

When running M with input x and auxiliary start a, the machine can visit any configuration only once, since otherwise it would never halt. Similarly, a catalytic Turing machine can also not have any configuration in common between a computation starting with a or one with $a' \neq a$, for a certain input x; from that point on they would run the same computation, so the restored auxiliary part at halting would be incorrect for at least one of them.

Because of this uniqueness property, we can bound the expected runtime of a catalytic computation by simple counting. Note that the total number of different configurations that a Turing machine of memory $s + s_a$ can have is bounded by $O(2^{s_a+s+\log s_a+\log s})$, where we need the logarithmic terms to account for the location of the tape heads. Let $\textsc{time}\, M(x, a)$ denote the computation time of M on input x with the auxiliary tape initialized to a. Then it holds that

$$\sum_{a=0}^{2^{s_a}-1} \textsc{time}\, M(x, a) \leq O(2^{s_a+s+\log s_a+\log s}) .$$

Dividing by 2^{s_a} gives

$$\mathop{\mathbb{E}}_{a \in_R \{0,1\}^{s_a}} [\textsc{time}\, M(x, a)] \leq 2^{O(s)},$$

where we use that $\log s_a = O(s)$. Now the inclusion in $\mathsf{ZTIME}(2^{O(S(n))})$ directly follows: a simulating zero-error probabilistic machine can just run the same computation as M, randomly generating bits of a as needed, and halt in expected time $2^{O(s)}$. ■

In particular, for catalytic logspace, $\mathsf{CSPACE}(\log n) \subseteq \mathsf{ZPP}$.

A natural question to ask is: can a catalytic machine directly simulate deterministic Turing machines that use strictly more space, by having a translation for every instruction? From the previous theorem it follows that the answer is no. (Lack of this type of simulation of course does not rule out the possibility that the catalytic machine could decide languages that need more space, it only hints that such a construction can not use another Turing machine as a black box.)

Corollary 20. *No step-by-step simulation of deterministic space $\omega(S(n))$ is possible in catalytic space $S(n)$.*

Proof. There is some computation M on space $\omega(S(n))$ that uses time $t = 2^{\omega(S(n))}$ for all inputs of length n. Let x be an input of length n. Suppose that M has a step-by-step catalytic simulation M', which runs in space $s = S(n)$ with auxiliary space s_a.

By the definition of a step-by-step simulation, we have that

$$\forall a \in \{0,1\}^{s_a}\ \textsc{time}\, M'(x, a) \geq \textsc{time}\, M(x) \geq 2^{\omega(s)}.$$

From the proof of Theorem 19 we know that on expectation over a, M' must have $\textsc{time}\, M'(x, a) \leq O(2^s)$, a contradiction. ■

Corollary 21.
If $\mathsf{ZPP} = \mathsf{L}$ *then* $\mathsf{CSPACE}(S(n)) = \mathsf{DSPACE}(S(n))$.

Proof. The Corollary follows from Theorem 19. Using padding, we have that $\mathsf{ZPP} = \mathsf{L}$ implies $\mathsf{ZTIME}(2^{S(n)}) \subseteq \mathsf{DSPACE}(S(n))$, giving $\mathsf{CSPACE}(S(n)) \subseteq \mathsf{ZTIME}(2^{S(n)}) \subseteq \mathsf{DSPACE}(S(n))$. ■

Corollary 22. *The exponential-time hypothesis [IP99] implies that* $\mathsf{SAT} \notin \mathsf{CSPACE}(o(n))$.

Proof. The exponential-time hypothesis says that $\mathsf{SAT} \notin \mathsf{BPTIME}(2^{o(n)})$. From this it directly follows that $\mathsf{SAT} \notin \mathsf{ZTIME}(2^{o(n)})$ and by Theorem 19 this implies $\mathsf{SAT} \notin \mathsf{CSPACE}(o(n))$. ■

4.3 Oracle results for catalytic computation

We can show an oracle relative to which $\mathsf{CSPACE}(\log n) = \mathsf{PSPACE}$.

Theorem 23. *There exists an oracle* A *such that*

$$\mathsf{DSPACE}^A(2^{\Omega(S(n))}) = \mathsf{CSPACE}^A(S(n))$$

The intuition behind the proof is as follows. Any auxiliary string is either compressible, in which case we can replace it by a compressed version and use the now-available free space, or hard to compress, in which case we can make some non-trivial use of it — in this case as a 'password' for the oracle that can not be found by a small-space computation.

Some care has to be taken when interpreting oracle results for space-bounded computation. For example, there are oracles relative to which classic results like Savitch's theorem and the Immerman-Szelepcsènyi theorem do not hold.

Proof. Kolmogorov complexity will give us the notion of compressibility:

Definition 24. *Fix some choice* U *for a universal Turing machine, and let* x, y *be two binary strings. The Kolmogorov complexity of a* x *relative to* y, *denoted* $C(x|y)$ *is the size of the smallest program* p *for machine* U *that outputs* x *on input* y *(i.e.,* $U(p, y) = x$*). The Kolmogorov complexity of* x, *denoted* $C(x)$, *is* $C(x|\varepsilon)$.

Fact 25 (Chain Rule [ZL70])**.** $C(x, y) \geq C(x) + C(y|x) - 4 \log C(x, y) - O(1)$.

We will construct an oracle A such that, relative to this oracle, a catalytic computation with work-tape space $s = S(n)$ can simulate a deterministic computation that uses space $2^{s/16}$. As a minor technical restriction, consider $S(n)$ such that $2^{S(n)/8} = \omega(n)$, i.e., $S(n)$ is at least $c \log(n)$ for $c > 8$.

Let a be a bit-string of length 2^s, the arbitrary initial contents of the auxiliary tape.

The oracle A will be given by four distinct *parts*, which we first describe informally. The first part checks if the (relative) Kolmogorov complexity of a given string is low. The second and third part can be respectively used to compress or decompress a given string. The fourth part, for which the definition is slightly more involved, gives access to a complete set for the large space computation when given a string with high complexity.

$$\begin{aligned}
A_1 &= \left\{ \langle 1, s, a, a' \rangle \middle|\, |a| = 2^{s/8} \text{ and } C(a|a') < \frac{3}{4}s \right\} \\
A_2 &= \left\{ \langle 2, a, a', i, b \rangle \middle| b \text{ is the } i\text{-th bit of the smallest } p \text{ such that } U(p, a') = a \right\} \\
A_3 &= \left\{ \langle 3, a, p, i, b \rangle \middle| b \text{ is the } i\text{-th bit of } U(p, a) \right\} \\
A' &= A_1 \cup A_2 \cup A_3
\end{aligned}$$

Now let $K^O_{f(n)}$ be a complete language for space $f(n)$ relative to oracle O. We define A_4 in stages, where the complete set is given relative to only the previous stages.

$$A_4^{(n)} = \left\{ \langle 4, a, x \rangle \middle| |x| = n \text{ and } C(a) \geq 2^{S(n)/8} \text{ and } x \in K^{A' \cup A_4^{<n}}_{2^{S(n)/16}} \right\}$$
$$A_4 = \bigcup_n A_4^{(n)}$$

Here $A_4^{<n} = \bigcup_{i=1}^{n-1} A_4^{(i)}$. Now the oracle A is the union of these parts, $A = A' \cup A_4$.

Let us give an algorithm to decide any given language $L \in \mathsf{DSPACE}^A(2^{S(n)/16})$. We divide the first $2^{s/4}$ bits of a into $2^{s/8}$ parts each of size $2^{s/8}$ and name the parts $a_1, \ldots, a_{2^{s/8}}$. Let $a_{<i}$ be the concatenation of a_1 up to a_{i-1}.

Starting with $i = 1$, ask part 1 of the oracle if $C(a_i | a_{<i}) < \frac{3}{4}s$. If that is not the case, increment i and repeat. If that *is* the case, then use the second part of the oracle to find the compressed version of a_i (given $a_{<i}$). Then store the compressed string version in our ordinary memory of size s, and erase the a_i part in the auxiliary tape. This frees up $2^{s/8}$ bits of memory, which we can use to decide if $x \in L$. When we are done with that, we can use the third part of the oracle to decompress a_i back into the auxiliary tape.

If none of the a_i for $i \in \{1, \ldots, 2^{s/8}\}$ are compressible given $a_{<i}$, we can show a lower bound for the Kolmogorov complexity of a using the chain rule:

$$\begin{aligned} C(a_1, a_2, \ldots, a_{2^{s/8}}) &\geq \sum_{i=1}^{2^{s/8}} (C(a_i | a_{<i}) - 4 \log C(a) - O(1)) \\ &\geq 2^{s/8} \left(\frac{3}{4}s - \frac{4}{8}s - O(1) \right) \\ &\geq 2^{s/8} \end{aligned}$$

(for s sufficiently large). Now we can use a as a high complexity 'password' for the fourth part of the oracle.

No machine in space $o(2^{s/8})$ can make a query of complexity as large as a. To see this, consider the configuration of the machine (including the input tape) before it starts writing the first character of any query q to the oracle tape. This configuration can be stored using $O(2^{S(n)/16}) + n = o(2^{S(n)/8})$ bits, but it contains all the information needed to produce q — a contradiction if q has Kolmogorov complexity at least $2^{s/8}$.

This implies that machines with space $2^{S(n)/16}$, on an input of length n, cannot distinguish $A' \cup A_4^{<n}$ from A, because they cannot query any string in $A_4^{(i)}$, for $i \geq n$. For any n it then holds that $K^{A' \cup A_4^{<n}}_{2^{S(n)/16}} = K^A_{2^{S(n)/16}}$, for the accessible strings of length n, and hence, having access to the string a and the oracle A, our catalytic machine can decide $K^A_{2^{S(n)/16}}$ (and therefore whether $x \in L$) by using the part 4 of the oracle. ■

Theorem 26. *There is an oracle B such that $\mathsf{NL}^B \not\subseteq \mathsf{CSPACE}^B(\log n)$.*

Proof. A Baker-Gill-Solovay [BGS75] construction works: from the proof of Theorem 19 we know that a Turing machine M deciding a language in $\mathsf{CSPACE}(\log n)$ has to run in average polynomial time, averaged over all possible auxiliary starting contents a. Therefore for any input x there is always an a for which M makes only polynomially many queries, and we apply the construction for that starting state — we diagonalize against the machine M at a string in the oracle that is

not queried by $M(x, a)$. Because the outcome of the catalytic computation should be correct for all possible starting values, the existence of a value a such that the machine fails implies that the machine does not correctly decide the language. ■

Acknowledgments

The third author acknowledges stimulating discussions with Steve Cook on L versus P which motivated this research. He also thanks Pierre McKenzie for helpful conversations.

References

[AAD00] M. Agrawal, E. Allender, and S. Datta. On TC0, AC0, and arithmetic circuits. *Journal of Computer and System Sciences*, 60(2):395–421, 2000.

[AK10] E. Allender and M. Koucký. Amplifying lower bounds by means of self-reducibility. *Journal of the ACM*, 57(3), 2010.

[AO94] E. Allender and M. Ogihara. Relationships among PL, #L, and the determinant. In *Proceedings of the Ninth Annual Structure in Complexity Theory Conference*, pages 267–278, 1994.

[BC92] M. Ben-Or and R. Cleve. Computing algebraic formulas using a constant number of registers. *SIAM Journal on Computing*, 21(1):54–58, 1992.

[BCH86] P. Beame, S. Cook, and H. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15(4):994–1003, 1986.

[BCP83] A. Borodin, S. Cook, and N. Pippenger. Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Information and Control*, 58(1–3):113–136, 1983.

[Ben73] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 1973.

[BFS92] J. Boyar, G. Frandsen, and C. Sturtivant. An arithmetic model of computation equivalent to threshold circuits. *Theoretical Computer Science*, 93(2):303–319, 1992.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the $\mathcal{P} =? \mathcal{NP}$ question. *SIAM Journal on Computing*, 4(4):431–442, 1975.

[BTV01] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. In *Proceedings of the 28th ICALP*, 2001.

[CG75] D. Coppersmith and E. Grossman. Generators for certain alternating groups with applications to cryptography. *SIAM Journal on Applied Mathematics*, 29(4):624–627, 1975.

[Cle89] R. Cleve. *Methodologies for Designing Block Ciphers and Cryptographic Protocols*. PhD thesis, University of Toronto, 1989.

[Coo85] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[Dam91] C. Damm. DET=$L^{(\#L)}$. Technical Report Informatik-Preprint 8, Fachbereich Informatik der Humboldt–Universität zu Berlin, 1991.

[HAM02] W. Hesse, E. Allender, and D. A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002.

[IL95] N. Immerman and S. Landau. The complexity of iterated multiplication. *Information and Computation*, 116(1):103–116, 1995.

[IP99] R. Impagliazzo and R. Paturi. The complexity of k-sat. In *Proceedings of the 14th CCC*, pages 237–240, 1999.

[LMT97] K. J. Lange, P. McKenzie, and A. Tapp. Reversible space equals deterministic space. In *Proceedings of the 12th CCC*, 1997.

[RT92] J. Reif and S. Tate. On threshold circuits and polynomial computation. *SIAM Journal on Computing*, 21(5):896–908, 1992.

[Sud78] I. H. Sudborough. On the tape complexity of deterministic context-free languages. *Journal of the ACM*, 25(3):405–414, July 1978.

[Tod91] S. Toda. Counting problems computationally equivalent to computing the determinant. *Technical Report CSIM*, 91-07, 1991.

[Tod92] S. Toda. Classes of arithmetic circuits capturing the complexity of computing the determinant. *IEICE Transactions on Information and Systems*, E75-D:116–124, 1992.

[Val79] L. G. Valiant. Completeness classes in algebra. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, STOC '79, pages 249–261, New York, NY, USA, 1979. ACM.

[Val92] L. G. Valiant. Why is Boolean complexity theory difficult? In *Poceedings of the London Mathematical Society symposium on Boolean function complexity*, pages 84–94. Cambridge University Press, 1992.

[Ven91] H. Venkateswaran. Properties that characterize LOGCFL. *Journal of Computer and System Sciences*, 43(2):380–404, 1991.

[Vin91] V. Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proceedings of the Sixth Annual Structure in Complexity Theory Conference*, pages 270–284, 1991.

[VSBR83] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal on Computing*, 12(4):641–644, 1983.

[ZL70] A. K. Zvonkin and L. A. Levin. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematics Surveys*, 256:83–124, 1970.