The Computational Complexity Column

by

V. Arvind

Institute of Mathematical Sciences, CIT Campus, Taramani
Chennai 600113, India
arvind@imsc.res.in
http://www.imsc.res.in/~arvind

Catalytic computation is a fascinating notion of space-restricted computation introduced and studied by Buhrman et al in their STOC 2015 article. It is motivated by reversible models of computation and has intriguing connections to classical complexity measures. In this article, Michal Koucky explains the underlying ideas and results in that work and new directions for research.

Catalytic computation

Michal Koucký*
Computer Science Institute
Charles University, Prague
koucky@iuuk.mff.cuni.cz

Abstract

Catalytic computation was defined by Buhrman et al. (STOC, 2015). It addresses the question whether memory, that already stores some unknown data that shall be preserved for later use, can be meaningfully used for computation. Buhrman et al. provide an intriguing answer to this question by giving examples where the occupied memory can be used to perform computation. In this expository article we survey what is known about this problem and how it relates to other problems.

1 Introduction

In various sciences it is customary to study complex systems in isolation to make the study tractable. This also happens in theoretical computer science where we often look at a single Turing machine solving a certain problem while ignoring the rest of the universe. For example theorems like the Space Hierarchy Theorem describe computation that happens in isolation from the rest of the world. However, in typical real world scenario computation happens in the context of some outside environment. For example, when focusing on space (memory) used by the computation we can come across the following typical situations:

1. A process (program) runs on a computer that is equipped with a hard disk containing data unrelated to the computation.

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement n. 616787. Supported in part by the Center of Excellence CE-ITI under the grant P202/12/G061 of GA ČR.

2. A process runs on a computer and simultaneously there are other running processes on the same computer occupying parts of the main memory.

3. A process invokes some internal procedure (function), the internal state of the process is stored on the stack before the invocation and upon finishing the procedure the state is restored from the stack. During the computation the procedure is using some working memory while not touching the content of the stack.

In all these scenarios the process or procedure that is running has some available working memory that it can use as it wishes but in addition to that there is a huge amount of memory that is accessible in principle but currently it stores data which have to be retained during the computation of the process or procedure. A natural question that comes up is whether the process or procedure can make some meaningful use of the extra memory under the condition that upon finishing it will be restored to its initial content. To rephrase the question:

> Can we compute more or more efficiently if we allow access to the extra memory?

Naturally, granting access to the extra memory would bring in a host of issues regarding privacy, security and reliability. Some of these can be easily dealt with using encryption. However, these issues are not the issues we are trying to address here. Our question is whether in a friendly environment such as in the third scenario one can make use of the extra memory. This could allow for more space efficient computation.

It is natural to conjecture that the extra space cannot be used meaningfully. The reason is that we do not have control over the initial content of the extra memory and we cannot just simply erase it. For example, the content of the extra memory can be incompressible (either in non-technical sense or in the sense of Kolmogorov complexity). Then we have to essentially keep the content of the extra memory in there assuming our own working memory is substantially smaller. Otherwise we could lose some information and we would not be able to restore the initial content of the extra memory when finishing.

On the other hand, if the content of the extra memory were indeed incompressible one could try to use it for derandomization using the *hardness versus randomness* paradigm [**?**, **?**]. And, if it were compressible we could compress it and use the extra space to perform our computation. So the usefulness of the extra memory is not clear cut.

Hence, we are interested in the question whether there are problems that can be solved using the extra memory regardless of its initial content but that cannot be solved using the same resources but without the extra memory.

The origins of this question can be traced back to the program of Steve Cook on separating L (problems solvable in logarithmic space) from P (problems solvable in polynomial time) [**?**]. In a project of Steve Cook and Yuval Filmus [**?**], they propose to prove lower bounds on the size of branching programs for the Tree Evaluation Problem in two steps: first prove the lower bounds under essentially the assumption that the extra space does not help and then justify this assumption. This assumption can be phrased in terms of catalytic branching programs that we will see in Section 9. The setting of the parameters for the assumption of Cook and Filmus is quite specific though so it is not clear how much the study of our general question sheds light on their problem. However, a prototypical example for our third scenario is recursive Savitch's algorithm for solving the Graph Reachability Problem, and hence understanding the general question provides insights about the relationship between L and NL (problems solvable non-deterministically in logarithmic space).

In the rest of the article we will survey what we know about the usefulness of the extra memory and we will show perhaps surprisingly that the extra memory can be used meaningfully despite the fact that we do not have control over its initial content and we have to restore it by the end of the computation. We will formalize this question in the next section. We will call the computation with the extra tape a *catalytic* computation as the extra memory serves as a form of catalyst to carry out the computation.

2 Catalytic computation

It is fairly easy to capture the scenarios described in the introduction in terms of the usual computational models. We can take either a Turing machine or a random access machine (RAM) and equip it with an extra tapes or extra region of memory that is initialized to an unknown content, it can be modified during the computation but it must be restored to its initial content by the end of computation. For clarity, we will think of Turing machines but any other computational model could easily be extended to obtain similar results. We will use the definitions of Buhrman et al. [**?**].

A *catalytic Turing machine* is a Turing machine equipped with a read-only input tape, a read-write work tape and an extra read-write 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 content. We will often refer to the auxiliary tape as the *catalytic* tape.

Definition 1. *Let $s, w : \mathbb{N} \to \mathbb{N}$ be non-decreasing functions. We say that a language L is decided by a catalytic Turing machine M in space $s(n)$ and using catalytic space $w(n)$ if on every input x of length n and arbitrary string a of length $w(n)$ written on the auxiliary tape the machine halts with a on its auxiliary tape, during its computation M uses (accesses) at most $s(n)$ tape cells on its work tape and $w(n)$ cells initially containing a on its auxiliary tape, and M correctly outputs whether $x \in L$.*

We define $\mathsf{CSPACE}(s(n), w(n))$ to be the set of all languages that can be decided by a catalytic machine in space $s(n)$ using catalytic space $w(n)$. As a notational shorthand let $\mathsf{CSPACE}(s(n)) = \mathsf{CSPACE}(s(n), 2^{O(s(n))})$

In our treatment we will assume that Turing machines work with the binary alphabet $\{0, 1\}$ but all the results can be easily extended to other alphabets.

It is natural to consider only functions $w(n)$ where $w(n) \in 2^{O(s(n))}$. The reason is that otherwise the position of the head on the auxiliary tape encodes potentially more information than the content of the work space. Indeed, if we had multiple auxiliary tapes and infinite $w(n)$ we could simulate arbitrary space computation just using the positions of the heads on the auxiliary tapes [**?**]. Such a machine would be essentially equivalent to counter machines. This justifies our definition of $\mathsf{CSPACE}(s(n))$.

We also allow only one work tape and one auxiliary tape. Since we are concerned mainly about space this is without the loss of generality as we can simulate multiple tapes on a single tape whenever $s(n) \in \Omega(\log w(n))$.

The most important class for us is the catalytic log-space, the class $\mathsf{CL} = \mathsf{CSPACE}(\log n)$. It corresponds to polynomial size catalytic tape and logarithmic work space. This seems to be the maximal reasonable auxiliary tape and the minimal reasonable work space. It is natural to compare this class to the usual deterministic logarithmic space L and non-deterministic logarithmic space NL. As we will exhibit later that both classes are contained in CL. Indeed, Buhrman et al. [**?**] show a (to the best of our knowledge) stronger statement that $\mathsf{TC}^1 \subseteq \mathsf{CL}$:

Theorem 2. *Languages that are recognized by log-space uniform families of polynomial-size Boolean circuits of logarithmic depth consisting of arbitrary fan-in* MAJ*-gates and* NOT*-gates (i.e. log-space uniform TC^1 circuits) are in CL.*

We will survey classes between L and TC^1 in Section 4. We remark that we do not know whether log-space uniform $\mathsf{TC}^1 = \mathsf{L}$ as we do not know of

any separation of L from P or even NP. So to the best of our knowledge it could be that $\mathsf{L} = \mathsf{TC}^1 = \mathsf{CL}$. That would imply a remarkable sequence of collapses of complexity classes as we will see later. However, to appreciate the power of catalytic space and CL in particular we will recall what is known about the important Graph Reachability Problem (STCONN).

3 Graph Reachability Problem

The Graph Reachability Problem is the following algorithmic problem:

Input: Graph G and two of its vertices s and t.
Output: Decide whether there is a path from s to t in G.

We denote the corresponding language $\mathsf{STCONN} = \{(G, s, t)$; there is a path from s to t in $G\}$. This is a well studied problem from the computational complexity perspective but also from practical stand point. STCONN is the standard complete problem for non-deterministic log-space, NL. Its undirected version $\mathsf{USTCONN}$, where the graphs are undirected, is known to be decidable in log-space, L. This is a celebrated result of Reingold [**?**]. Previous to this result it was known that $\mathsf{USTCONN}$ is in randomized log-space (RL) [**?**], and there is a sequence of results getting ever so closer to logarithmic space [**?**, **?**, **?**, **?**]. There are many other restrictions of Graph Reachability such as reachability on planar graphs or graphs with other special properties that people study [**?**].

For general STCONN the best space upper bound is provided by Savitch's Theorem which puts NL into $\mathsf{DSPACE}(\log^2 n)$, the class of problems solvable deterministically using $O(\log^2 n)$ space. No randomized log-space algorithm for general STCONN is known. Savitch's algorithm is very space efficient but its running time is superpolynomial, namely $n^{\Theta(\log n)}$.

When we focus on deterministic algorithms running in polynomial time, the landscape looks markedly different. Solving reachability in linear time can be accomplished by algorithms using either breadth-first search or depth-first search. However, all these algorithms typically require space at least linear in the number of vertices of the graph. The most space efficient algorithm running in polynomial time is the algorithm of Barnes et al. [**?**] that uses the unlikely space $n/2^{\Theta(\sqrt{\log n})}$. No better polynomial time algorithm is known, and the space used by this algorithm matches a lower bound on space for solving STCONN on a restricted model of computation so called *Node Naming Jumping Automata on Graphs* (NNJAG's) [**?**, **?**]. NNJAG's are a model specifically proposed for the study of STCONN and most of the known sublinear space algorithms for STCONN can be implemented on it.

Hence, any polynomial time algorithm using space less than $n/2^{\omega(\sqrt{\log n})}$ is likely to require fundamentally new ideas. It is a major challenge to design a polynomial time algorithm for STCONN working in space $O(n^\epsilon)$ for some $\epsilon < 1$.

So our currently best algorithm for STCONN runs in space $\Theta(\log^2 n)$ and all known polynomial time algorithms for this problem run in space almost linear. Compare this to the result of Buhrman et al. [?]:

Theorem 3. STCONN *can be solved on catalytic Turing machines in space* $O(\log n)$ *with catalytic space* $O(n^2 \log n)$ *and time* $O(n^9)$.

The time bound $O(n^9)$ is a crude estimate for a naive implementation of the algorithm of Buhrman et al. on catalytic Turing machines. On catalytic RAM it would achieve substantially better running time using the same space bounds. In other words, if we are allowed to use someone else's occupied memory of size $O(n^2 \log n)$, we can solve STCONN in polynomial time and logarithmic work space. In terms of work space this is exponentially better than any known polynomial time algorithm for STCONN. To us, this clearly justifies the study of the model, and conceivably there could be even practical applications of this paradigm. Even in the unlikely case of $\mathsf{CL} = \mathsf{L}$, catalytic space could provide nontrivial advantage in terms of algorithm design or the actual running time.

4 Complexity classes and problems around L

This section serves as a brief overview of the landscape surrounding L. We assume that the reader is familiar with basic concepts such as Turing machines. (More background information can be found in standard textbooks, e.g. [?, ?].) There is a surprising number of complexity classes people study that are close to L in computational power. Most of these classes come quite naturally either as classes that capture the computational complexity of some well-known problems or they correspond to some natural restriction of a more general computational device.

Problems: STCONN, USTCONN, DET, IMM. We have already seen the problems STCONN and USTCONN in Section 3. Another problem relevant to our study is the problem of computing a *determinant*. By $\mathsf{DET}_{n,R}$ we denote the problem of computing a determinant of an $n \times n$ matrix over a ring R. A closely related problem is the *Iterated Matrix Multiplication* $\mathsf{IMM}_{n,m,R}$ which is the problem of computing the product of n matrices, each over the ring R of dimension $m \times m$. Typically we may think of R being the ring of integers,

and $m = n$. We will omit the subscripts when the ring or dimensions are understood from the context. It is well-known that by results of Cook [**?**, **?**], the class of problems log-space many-one reducible to DET is the same as the class of problems log-space reducible to IMM. (A function f is *log-space many-one reducible to* the determinant if there is a function g computable in log-space such that $f(x)$ (viewed as a number written in binary) is equal to the determinant of matrix $g(x)$.)

Classes: L, NL, LOGCFL. Beside the computational classes L (problems solvable *deterministically* in logarithmic space), and NL (problems solvable *non-deterministically* in logarithmic space) we will also refer to the class LOGCFL which contains both L and NL. LOGCFL is the class of languages accepted by non-deterministic Turing machines running in polynomial-time, working in space $O(\log n)$ and using in addition to their work space an unlimited push-down stack, so called *AuxPDA*'s [**?**]. Equivalently, LOGCFL is the class of problems that are log-space many-one reducible to context-free languages.

Counting classes: #L, #LOGCFL, GapL. Instead of considering whether a non-deterministic Turing machine accepts its input on some non-deterministically chosen computational path or rejects on all of them we can count the number of accepting paths of the machine on the given input. This gives a function that maps inputs to integers. That is a more general concept than just acceptance by a non-deterministic machine which corresponds to a function mapping inputs to $\{0, 1\}$. The *counting* class #L is the class of functions obtained by counting the number of accepting paths of a non-deterministic log-space machine, and #LOGCFL is the class of functions that count the number of accepting paths of an AuxPDA running in logarithmic space and polynomial time. The complexity of computing the determinant is closely related to #L. In particular, f is log-space many-one reducible to determinant if and only if it is the difference of two functions in #L [**?**, **?**, **?**, **?**]. The class of such functions is usually denoted by GapL. DET and IMM are both in GapL.

Circuits. The above classes are defined in terms of Turing machines. We also consider functions defined in terms of *circuits*. A circuit is a computational device that consists of *gates* interconnected by *wires*. Wires carry values (typically 0 or 1) from one gate to another gate, and each gate takes its incoming values, computes a designated function (such as AND or OR) on them, and send the resulting value along all its outgoing wires. The *fan-in* of

a gate is the number of its incoming wires. If a gate has fan-in two we say it is *binary*. We say that it has *unbounded fan-in* when we do not place any restriction on its fan-in. Gates of fan-in zero are the *input gates*, each such a gate is associated with one input bit (e.g. the 17-th input bit), and when the circuit is provided with an input, the gate sends along its outgoing wires the value of the associated input bit. The *output of the circuit* is the output value of designated gates. One can represent a circuit by a directed graph where nodes represent gates and directed edges represent wires. We will consider only circuits whose graphs contain no directed cycle, i.e., they are directed acyclic graphs (DAG's). The output value of such circuits is well defined. A circuit is a Boolean circuit if it computes with the Boolean values $\{0, 1\}$.

Uniformity. For each input length n we typically have a different circuit C_n taking the appropriate number of input bits. To represent a function which takes inputs of arbitrary length, one considers families of circuits $\{C_n\}_{n\geq 1}$, where C_n computes the function on inputs of length n. If we do not put any restrictions on the circuit family $\{C_n\}_{n\geq 1}$ we can compute *any* function, even *uncomputable* one such as the Halting Problem. To restrict ourselves to computable functions we will look on circuit families $\{C_n\}_{n\geq 1}$ for which there is an algorithm that on input 1^n outputs a description of the circuit C_n. Such a family is called *uniform*. It will be *log-space uniform* if the algorithm uses work space $O(\log n)$ to compute the description of C_n. In this article when we say uniform we will mean log-space uniform unless specified otherwise. We will restrict ourselves essentially only to log-space uniform circuit families.

Size and depth. An important parameter of a circuit is its *size* which is the number of its gates. Since the graph of the circuit is acyclic each gate computes its value ones it receives its input values, and multiple gates can compute their value at the same time. Hence circuits are a model of *parallel computation*. The time to finish the evaluation of a circuit is given by the *depth* of the circuit which is the length of the longest path in the corresponding graph.

For a circuit family $\{C_n\}_{n\geq 1}$, let $s(n)$ be the size of C_n and $d(n)$ be its depth. If $s(n)$, as a function of n, is bounded by a polynomial in n then we say that the circuit family has *polynomial size*; it is *exponential size* if $s(n)$ is bounded by an exponential in n. If $d(n)$ is bounded by $O(\log n)$ we say that the family has *logarithmic depth*, if $d(n)$ is bounded by a constant, that is $d(n) \in O(1)$, then we say that the family has *constant depth*.

Any Boolean function $f : \{0, 1\}^* \to \{0, 1\}$ can be computed by family of circuits of size $O(2^n/n)$ consisting of binary AND and OR gates and unary

NOT gates (see e.g. [**?**]). For most functions this is actually the optimal circuit size as can be verified by a simple counting argument. Functions from P are computable by circuit families of polynomial size consisting of binary AND, OR and unary NOT gates. This is actually a precise characterization as a function is computable in polynomial time on a Turing machine if and only if the function is computable by a log-space uniform circuit family of polynomial size. We will look on circuit families that compute functions with complexity close to log-space.

Circuit classes: $\mathsf{TC}^0, \mathsf{NC}^1, \mathsf{SAC}^1, \mathsf{AC}^1, \mathsf{TC}^1$**.** TC^0 is the class of functions computable by families of Boolean circuits of polynomial size and constant depth that consist of MAJ gates, that is gates that output the majority value of their input bits, and unary NOT gates. NC^1 is the class of functions computable by families of Boolean circuits of polynomial size and logarithmic depth that consist of binary AND and OR gates and unary NOT gates. Equivalently, it is the class of functions computable by polynomial size Boolean *formulas* over AND, OR and NOT. SAC^1 is the class of functions computable by families of Boolean circuits of polynomial size and logarithmic depth that consist of binary AND gates, unbounded fan-in OR gates and unary NOT gates. AC^1 is the class of functions computable by families of Boolean circuits of polynomial size and logarithmic depth that consist of unbounded fan-in AND and OR gates and unary NOT gates. Finally, TC^1 is the class of functions computable by families of Boolean circuits of polynomial size and logarithmic depth that consist of MAJ gates and unary NOT gates. It is standard knowledge that $\mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{SAC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{TC}^1$, 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 n-bit integers, computing the division of two n-bit integers, etc. [**?**, **?**, **?**]. The class NL is contained in SAC^1 which is equal to $\#\mathsf{LOGCFL}$ [**?**].

Arithmetic circuits: $\mathsf{VP}, \#\mathsf{SAC}^1, \#\mathsf{AC}^1$**.** Beside circuits that operate over the Boolean domain we consider also *algebraic* circuits that operate over some ring R. When R is the ring of integers $\mathbb{Z}$, these are also called *arithmetic* circuits. The most relevant class for us is Valiant's class $\mathsf{VP}(R)$ [**?**], which is the class of functions computed by polynomial size algebraic circuits using $+$ and $\times$ gates over R, where the circuit corresponds to (represents) a multivariate polynomial of polynomial degree. An alternative characterization of $\mathsf{VP}(R)$ is as the class $\#\mathsf{SAC}^1(R)$ of functions computed by algebraic circuits of polynomial size and logarithmic depth that use binary multiplication and addition with arbitrary fan-in [**?**]. The class $\#\mathsf{AC}^1$ contains functions

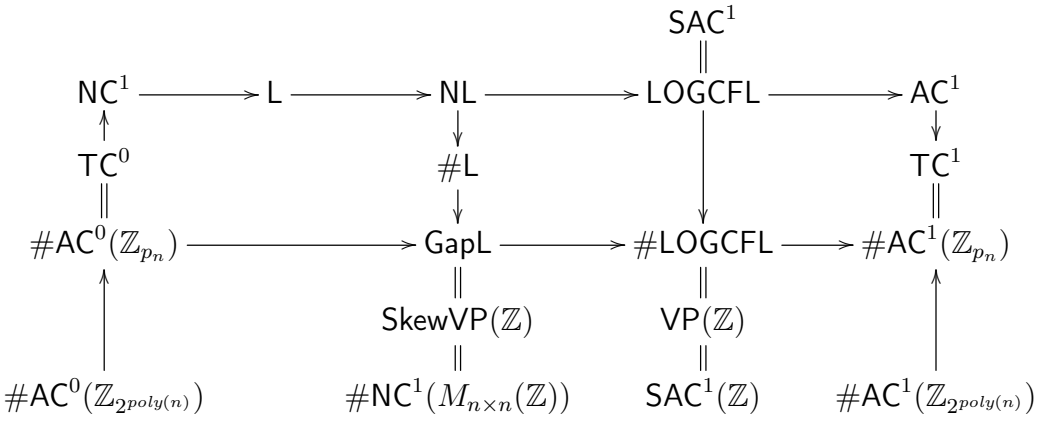

Figure 1: Inclusion diagram for various classes. All these classes fall in CL.

computed by arithmetic circuits of polynomial size and logarithmic depth where both addition and multiplication have arbitrary fan-in.

Taken over the integers, VP($\mathbb{Z}$) exactly equals #LOGCFL. *Skew* circuits are algebraic circuits where each multiplication gate is binary and restricted so that one of its inputs is either a constant or an input variable. Skew circuits over integers having polynomial size and degree compute exactly GapL [**?**], i.e., functions reducible to determinant. Hence, the question on the relationship between GapL and #LOGCFL is exactly the question posed by Valiant [**?**] about the relationship between the determinant and VP($\mathbb{Z}$), namely, whether evaluating a VP($\mathbb{Z}$) circuit reduces to evaluating the determinant of a matrix that is at most polynomially larger in size. (Valiant shows that a matrix of size $n^{\log n}$ is enough, and that this can be reduced to polynomial size in the case of skew circuits.)

Immerman and Landau [**?**] conjecture that computing determinant over the integers is hard for TC^1. 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. 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 representation of their output may require super-polynomially many bits. If 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 $\#\mathsf{AC}^1$. The latter can have super-polynomial degree which seems to go against the conjecture. The conjecture 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 $\mathbb{Z}_{2^n}$. Allender, Gál and Mertz [?] give examples of such type of phenomena. [?, ?, ?] establish relationship between TC^0 and $\#\mathsf{AC}^0$ over various integral rings and finite fields. This relationship can be translated into a similar type of relationship between TC^1 and $\#\mathsf{AC}^1$ as depicted in Figure 1.

5 Reversible computation

Our requirement on catalytic computation is to restore the initial content of its auxiliary tape by the end of a computation no matter what is its initial content. This suggest that the problem of using the catalytic tape is related to *reversible computation* as we are required to return to the same configuration of the auxiliary tape. Indeed, catalytic computation is related to reversible computation but both concepts are somewhat different.

The goal of a reversible computation is to perform each step of the computation *reversibly*. This translates into the requirement that for each configuration of the computer there is at most one other configuration which will lead to the former one in one step of the computation. In other words, the graph of all possible computational configurations of the computer consists of lines and cycles. (In *irreversible* computation multiple different configurations can lead in one step to the same configuration for example by setting to 0 a tape cell that in one of the configurations contains symbol 0 and in the other one symbol 1. Hence, the graph of configurations of an irreversible computation is formed by disjoint trees (forest).)

The interest in reversible computation is motivated by minimizing energy needed to carry out a computation. By laws of thermodynamics, irreversible steps of a computation dissipate heat [?]. Reversible computation could in principle be carried out without expending any energy. In 60's this lead to a question what functions can be computed reversibly. Bennett [?] provided an answer to this question by showing that any irreversible computation can be simulated reversibly. His technique is based on recording the history of all moves on an extra history tape. This solution requires substantial amount of extra space that has to be initially empty and that will eventually be restored to its empty state. In [?], Bennett designed a better simulation that requires only polynomial amount of extra space for recording only milestones in the history.

Lange, McKenzie and Tapp [?] came up with a different technique that is based on traversing the tree of the computer configurations in an Eulerian

fashion. This technique in principle does not need any extra space. However, when run in reverse it might not be able to recognize the correct initial configuration as there might be many initial configurations that lead to the same final configuration (they will be a part of the same tree of configurations). Hence, when the computation is run in reverse it will cycle through all these initial configurations. Thus, one either needs some easy to recognize initial configuration or extra space to keep track of the length of the computation. This extra space is proportional to the work space. The drawback of this technique is that the simulation might require exponential time. Buhrman, Tromp and Vitányi [**?**], and independently Williams [**?**] combined the two techniques to get various trade-offs between the running time and space of the simulation.

One can also build reversible Boolean circuits using various types of reversible gates such as Toffoli gates [**?**]. However, they also require extra space which essentially stores the history of the computation and must be initialized to a particular configuration. Hence, all of the above techniques require extra space that has to be initially set to particular values, e.g., blanks. This makes them unsuitable for catalytic computation as we cannot impose restrictions on the initial content of the catalytic memory. Catalytic computation is more relaxed about the reversibility, though. It does not require every step of the computation to be reversible but only that we can restore the content of the tape. For example provided that the initial content of the auxiliary tape is sufficiently compressible it is legal to compress it, perform there some irreversible computation, and then decompress the tape again. So the requirements on catalytic computation are different than on reversible computation.

There is one more technique that computes in reversible fashion, and that we did not discuss yet. Motivated by cryptographic applications, Coppersmith and Grossman [**?**] studied which permutations can be computed reversibly on a very simple model of computation. They showed that all permutations can be computed in this model where odd permutations need a single extra bit of storage used in a catalytic fashion. This aspect was noted by Bennett in his paper [**?**]. Ben-Or and Cleve [**?**] extended the computational model further to get a remarkable result that any arithmetic formula over a ring (finite or infinite) can be evaluated using only three working registers to hold values from the ring. This result was a generalization of the famous Barrington's theorem [**?**] which established that all Boolean formulas can be evaluated on *width-5 (permutation) branching programs*. The model and techniques allow one to overlay space for one computation over the space of another computation. These techniques are the basis for the proof of Theorem 2 that $\mathsf{TC}^1 \subseteq \mathsf{CL}$. We will describe the model and techniques in the next section.

6 Transparent computation

In this section we present the model of transparent computation of Buhrman et al. [?] which is a form of reversible computation. It generalizes the model of Coppersmith and Grossman [?] and Ben-Or and Cleve [?].

The model of transparent computation is a *non-uniform* model. The computational device for transparent computation is a *register* machine equipped with read-write *working registers* $r_1, r_2, \ldots, r_m$ and read-only *input registers* $x_1, \ldots, x_n$. Each register holds a value from some designated ring R. The input to the device is given in the registers $x_1, \ldots, x_n$ so, inputs of different lengths require machines with different numbers of input registers and possibly also of working registers. Each operation (*instruction*) of the machine is of the form $r_i \leftarrow r_i + f(x_1, \ldots, x_n, r_1, \ldots, r_m)$ or $r_i \leftarrow r_i - f(x_1, \ldots, x_n, r_1, \ldots, r_m)$, where the function f gives a value from R and the $+$ and $-$ operations are over the ring R. One may allow different rings for different input sizes (and registers).

Coppersmith and Grossman consider the case when $R = \mathbb{F}[2]$ and when the instructions can use arbitrary functions f. Ben-Or and Cleve consider an arbitrary ring R but allow only instructions of the form $r_i \leftarrow r_i \pm v$ and $r_i \leftarrow r_i \pm r_j * v$, where r_i and r_j are different working registers and v is either an element of R (*constant*) or one of the input registers $x_1, \ldots, x_n$. (The $*$ denotes multiplication over R.) We will call such instructions *skew bases*. Buhrman et al. [?] use arbitrary rings R and instructions of the form $r_i \leftarrow r_i \pm v$ and $r_i \leftarrow r_i \pm u * v$, where both u and v can be either elements from the ring R, input registers, or working registers different from r_i. We will call such instructions *standard bases*.

A *program* for the register machine is a sequence of operations. We call P a *transparent* program. We say that $f(x_1, x_2, \ldots, x_n)$ can be *computed transparently* into a register r_i if there is a transparent program P that when executed on registers $r_1, r_2, \ldots, r_m$ with arbitrary initial values $\tau_1, \tau_2, \ldots, \tau_m$ and the input given in registers $x_1, \ldots, x_n$, ends with value $\tau_i + f(x_1, x_2, \ldots, x_n)$ in register r_i; the other registers may contain any values at the end of the computation. However, if the other registers do not change their value we say the program is *clean*. Clearly, we are interested in clean programs, and as we will see in a moment, any program can be made clean.

Notice, $r_i \leftarrow r_i + f(x_1, \ldots, x_n, r_1, \ldots, r_m)$ is an inverse operation to $r_i \leftarrow r_i - f(x_1, \ldots, x_n, r_1, \ldots, r_m)$ provided that f does not depend on the value of r_i. This is in particular true for the standard and skew bases. Thus for a transparent program $P = a_1, a_2, \ldots, a_\ell$ we let the *reverse program* P^{-1} be $a_\ell^{-1}, a_{\ell-1}^{-1}, \ldots, a_1^{-1}$ where a_i^{-1} is the same instruction as a_i but the $+$ and $-$ are interchanged. It is easy to verify by induction on the length of P that

P, P^{-1} computes identity. Hence, all transparent programs are *reversible*.

Clearly, if we have a program that transparently computes f into a register r_i we can modify it by relabeling registers to compute f transparently into a different register. To make a program that computes $r_i \leftarrow r_i + f(\vec{x})$ in a clean fashion, we use an extra working register r'_i. Let P' be a transparent program for $r'_i \leftarrow r'_i + f(\vec{x})$. Consider the following program:

1. $r_i \leftarrow r_i - r'_i$.
2. P'
3. $r_i \leftarrow r_i + r'_i$.
4. P'^{-1}

One can easily verify that the only effect of this program is adding $f(\vec{x})$ to r_i.

Beside computing a single function in a transparent fashion one can simultaneously compute several functions $f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x})$ into registers $r_{i_1}, r_{i_2}, \ldots, r_{i_k}$ so that the execution of P ends with the value $\tau_{i_j} + f_j(\vec{x})$ in each register r_{i_j}. (In case of a clean program the values of the remaining registers should remain the same, and any program can be made clean using an additional working register.)

Observe that a transparent program P over the standard bases computes a polynomial over R in the input variables $x_1, \ldots, x_n$. The degree of this polynomial is at most exponential in the length of P, and it is at most polynomial in the length of P when P is over the skew bases. Ben-Or and Cleve [**?**] prove the following:

Theorem 4 (Ben-Or and Cleve)**.** *Let $f : R^n \to R$ be a function over some ring R.*

1. *If f can be computed by an arithmetic formula of depth d over R consisting of $+, -, *$ with variables $x_1, \ldots, x_n$ then f can be computed transparently by a program of length 4^d over skew bases using at most three working registers.*
2. *If f is computed transparently by a program of length ℓ using skew instructions and m working registers then f is computable by an arithmetic formula over R of depth $O(\log \ell \cdot \log m)$.*

The first part is the important part as it allows to evaluate an arbitrary arithmetic formula using only three registers. It is well known that any arithmetic formula can be *balanced*, i.e., transformed so that its depth becomes logarithmic in its size. This was proven originally for commutative

rings by Brent et al. [?, ?] but their argument is known to hold also for non-commutative rings. This implies that any arithmetic formula can be computed transparently by a program using three working registers whose size is polynomial in the size of the formula. This holds for *any* ring. As noted by Cleve [?] it even holds over the ring of $n \times n$ matrices over R.

That means that the Iterative Matrix Multiplication $\mathsf{IMM}_{n,n,R}$ can be transparently computed using three registers holding values from $R^{n \times n}$. Alternatively, if we view $\mathsf{IMM}_{n,n,R}$ as a function from R^{n^3} into R^{n^2} then there is a polynomial size transparent program computing $\mathsf{IMM}_{n,n,R}$ using $3n^2$ working registers with instructions over R [?]. In the next section we will describe how to simulate transparent programs on a catalytic machine. This simulation directly allows one to conclude $\#\mathsf{L} \subseteq \mathsf{CL}$ since $\mathsf{IMM}_{n,n,\mathbb{Z}}$ is hard for $\#\mathsf{L}$.

To illustrate the technique used for transparent computation we give the proof of Ben-Or and Cleve's theorem.

Proof. First, we prove Part 1. We will prove by induction on the depth d of the formula computing f a slightly stronger statement that there is a clean transparent program computing $r_i \leftarrow r_i + u * f(x_1, \ldots, x_n)$ and $r_i \leftarrow r_i - u * f(x_1, \ldots, x_n)$, where $u = 1$ or u is a working register different from r_i.

If $d = 0$ then $f(x_1, \ldots, x_n) = v$, where v is either an input variable x_i or a constant from the ring R. The required program is a single instruction $r_i \leftarrow r_i \pm u * v$, where we put $+$ or $-$ depending on whether we want to add $u * f$ to r_i or subtract it.

So assume that the claim is true for functions computed by formulas of depth less than d, where $d \geq 1$. There must be some functions $g(x_1, \ldots, x_n)$ and $h(x_1, \ldots, x_n)$ computed by formulas of depth less than d so that $f = g \diamond h$, where $\diamond \in \{+, -, *\}$. Computing $r_i \leftarrow r_i + u * f(x_1, \ldots, x_n)$ can be decomposed into two parts:

1. $r_i \leftarrow r_i + u * g(x_1, \ldots, x_n)$
2. $r_i \leftarrow r_i + u * h(x_1, \ldots, x_n)$

By the induction hypothesis we have a clean program P_g implementing the first part, and a program P_h implementing the second part, both of length at most 4^{d-1}. The concatenation of the two programs P_g and P_h yields the required program of length at most $2 \cdot 4^{d-1} \leq 4^d$. Subtraction $r_i \leftarrow r_i - u * f(x_1, \ldots, x_n)$ is done similarly.

The only remaining case is the case of $f = g * h$, and this is the place where magic happens. Let r_k be a working register different from r_i and u. Consider the program:

1. $r_k \leftarrow r_k + u * g(x_1, \ldots, x_n)$
2. $r_i \leftarrow r_i + r_k * h(x_1, \ldots, x_n)$
3. $r_k \leftarrow r_k - u * g(x_1, \ldots, x_n)$
4. $r_i \leftarrow r_i - r_k * h(x_1, \ldots, x_n)$

By induction hypothesis we have a *clean* transparent program of size at most 4^{d-1} for each of the parts. A careful inspection of the code should convince the reader that the four parts together form a clean program for $r_i \leftarrow r_i + u * f(\vec{x})$. The size of the program is at most 4^d. Subtracting $u * f$ can be done similarly, one just switches $+$ and $-$ on the second and last line. This proves the first part.

To prove the second part, observe that we can think of a transparent program as acting on a vector of registers $\tilde{r} = (r_1, r_2, \ldots, r_m, 1)$. Each instruction $r_i \leftarrow r_i + r_j * v$ corresponds to a $(m+1) \times (m+1)$ matrix obtained from the identity matrix by replacing the $((m+2-j), i)$-entry by v, and each instruction $r_i \leftarrow r_i + v$ is obtained by replacing the $(1, i)$-entry by v. Multiplying $\tilde{r}$ by the sequence of matrices corresponding to individual instructions of the transparent program gives a vector with the resulting register values.

If the program computes $r_1 \leftarrow r_1 + f(x_1, \ldots, x_n)$, then $(1,1)$-entry of the product of the matrices is the value of $f(x_1, \ldots, x_n)$. Since each entry of the product of two $(m+1) \times (m+1)$ matrices can be computed by an arithmetic formula of depth $O(\log m)$, each entry of the product of ℓ matrices can be computed by an arithmetic formula of depth $O(\log \ell \cdot \log m)$ by forming a log-depth tree of matrix products. This proves the claim. □

Theorem 4 allows one to transparently compute any function from GapL. This requires only three matrix registers and skew instructions over matrices. To go to the (possibly) higher class TC^1 Buhrman et al. [?] use the full standard bases and polynomially many registers. The extra instruction $r_i \leftarrow r_i + r_j * r_k$ allows one to efficiently compute iterated product of registers, polynomially large powers of a register, and equality test.

The main theorem of Buhrman et al. [?] regarding transparent computation is the following.

Theorem 5 (Buhrman, Cleve, Koucký, Loff, Speelman)**.** *For any sequence of primes $(p_n)_{n \in \mathbb{N}}$ of size polynomial in n, functions from TC^1 can be computed transparently using polynomially many working registers over $\mathbb{F}[p_n]$ by programs of polynomial length with instructions from the standard bases.*

This claim holds uniformly as well as non-uniformly, so if a function f is computed by log-space uniform TC^1 circuits then it is computed by log-space uniform transparent programs, that is in log-space on input 1^n one can compute a prime p_n and the description of the transparent program computing f on inputs of size n. Here each input bit is represented by one register containing either 0 or 1, and the output of the function is also either 0 or 1. Other representations are also possible.

In the next section we will show how to simulate transparent computation by catalytic machines.

7 Catalytic simulation of transparent programs

Buhrman et al. [**?**] show how to simulate transparent programs on catalytic machines. The main idea is to use the catalytic tape to simulate registers of the machine. This is fairly straightforward for rings of size 2^k, where k is some integer. Each register can be represented by a block of k bits on the catalytic tape and the work tape can be used to manipulate these registers.

Imagine that we have a function $f : R^n \to R$ computed transparently by a program P into the register r_1, i.e., $r_1 \leftarrow r_1 + f(x_1, \ldots, x_n)$. We let the catalytic machine simulate instructions of P one by one to obtain $r_1 + f(x_1, \ldots, x_n)$ on the catalytic tape. The question is how do we recover the value of $f(x_1, \ldots, x_n)$ at this point? Consider the case when R is small enough so that we can fit a value from R into the work memory of the catalytic machine.

Then to compute $f(x_1, \ldots, x_n)$ we first store the initial value of r_1 on the work tape, execute P, extract $f(x_1, \ldots, x_n)$ from the current and initial value of r_1 by subtracting them, and we recover the initial content of the catalytic memory by reversing P, i.e., running P^{-1}.

If R is large so we cannot fit the whole value of r_1 onto the work tape then we can recover $f(x_1, \ldots, x_n)$ bit by bit by repeatedly computing $r_1 \leftarrow r_1 + f(x_1, \ldots, x_n)$ and extracting a different bit during each iteration. This works well when the operations on R are simple enough so that we have enough work space to add, subtract and multiply its elements. For example, for the case of $R = \mathbb{Z}_{2^n}$, arithmetic over $\mathbb{Z}_{2^n}$ can be done in logarithmic space despite the fact that each value occupies n-bits. We have seen that $\mathsf{IMM}_{n,n,\mathbb{Z}_{2^n}}$ can be computed transparently so we can compute $\mathsf{IMM}_{n,n,\mathbb{Z}_{2^n}}$ catalytically using catalytic space $3n^3$ and logarithmic work space.

Since $\mathsf{IMM}_{n,n,\mathbb{Z}_{2^n}}$ is complete for $\#\mathsf{L}$ and catalytic space is closed under log-space reductions (even catalytic log-space reductions) one obtains that $\#\mathsf{L} \subseteq \mathsf{CL}$ and this also implies the correctness of Theorem 3.

Similarly, one can simulate transparent programs for functions in TC^1. The only difficulty is that those programs need rings of prime size. In such a situation the initial content of the catalytic tape might represent values outside of the ring R and one cannot directly compute with them. This issue can be overcome using compression as was done by Buhrman et al. to establish Theorem 2. Currently we do not know of other methods how to catalytically compute some interesting functions. One possible direction for further algorithms that we will describe in Section 11 is to use non-deterministic catalytic computation.

8 Limits on the power of catalytic space

We have seen that CL has uprising computation power. Is there any limit to that power? Naturally, $\mathsf{CL} \subseteq \mathsf{PSPACE}$ as we can trivially simulate catalytic tape by an ordinary work tape. Buhrman et al. [**?**] provide a more interesting answer: $\mathsf{CL} \subseteq \mathsf{ZPP}$. The class ZPP stands for problems solvable by zero-error randomized algorithms running in *expected* polynomial time, i.e., algorithms that on each input run in polynomial time in expectation over their random choices and whenever they stop, they provide a correct answer.

The key observation of Buhrman et al. is that a log-space catalytic computation must finish in polynomial time on *average* over the initial content of the catalytic tape. Indeed, if there are W ways how to initialize the catalytic tape then there are at most $W \cdot \mathrm{poly}(n)$ possible configurations of the whole machine on a given input. On two different initial contents of the catalytic tape the machine cannot visit exactly the same configuration as the computation would be the same from then on so it would fail to restore the catalytic tape in one of the cases. So on average, a computation can visit at most $W \cdot \mathrm{poly}(n)/W = \mathrm{poly}(n)$ configurations so, it is polynomial time on average.

To simulate CL computation probabilistically we simulate the catalytic tape on a work tape, we randomly choose its initial content and run the simulation. If the simulation finishes in $O(\mathrm{poly}(n))$ steps we use its output as it must be correct. If the simulation runs for too long, we restart it with a new random initial content of the catalytic tape.

Theorem 6 (Buhrman, Cleve, Koucký, Loff, Speelman)**.** $\mathsf{CL} \subseteq \mathsf{ZPP}$ *and more generally,* $\mathsf{CSPACE}(s(n)) \subseteq \mathsf{ZPTIME}(2^{O(s(n))})$.

An immediate consequence is that under the Exponential-Time Hypothesis [**?**] $\mathsf{SAT} \notin \mathsf{CL}$ and so $\mathsf{NP} \not\subseteq \mathsf{CL}$. It is widely believed that $\mathsf{ZPP} = \mathsf{P}$ so under standard derandomization assumptions $\mathsf{CL} \subseteq \mathsf{P}$. However, it is still possible that $\mathsf{CL} = \mathsf{PSPACE}$. Indeed, relative to an oracle this is true.

Theorem 7 (Buhrman, Cleve, Koucký, Loff, Speelman)**.** *There exists an oracle A such that* $\mathsf{CL}^A = \mathsf{PSPACE}^A$.

It is an interesting question whether one could derandomize the probabilistic simulation of CL computation. This could in principle be easier than derandomizing the whole ZPP.

9 Non-uniform catalytic computation

The catalytic computational model we have seen so far is uniform, i.e, there is a single algorithm that works for all input lengths. It is natural to consider also the *non-uniform* variant where the algorithm might be completely different for each input length. There are two standard ways how to facilitate non-uniformity: either via so called *advice* function or via some inherently non-uniform model of computation such as Boolean circuits or branching programs.

Advice function $a : \mathbb{N} \to \{0,1\}^*$ augments the usual uniform algorithm so that the algorithm on an input x of length n also gets for free the advice string $a(n)$ [**?**]. The advice might help the algorithm to decide about the input x. The length of $a(n)$ controls the *amount of non-uniformity* the algorithm receives. For example $\mathsf{L}/poly$ is the class of problems solvable in log-space with advice function of length polynomial in n, $\mathsf{L}/O(1)$ is the class of problems solvable in log-space with advice of constant length.

We can equip a catalytic machine with an advice to get classes such as $\mathsf{CL}/poly$ and $\mathsf{CL}/O(1)$. (We assume that there is a single advice for all possible initial setting of the catalytic space, and the machine has to restore the tape only with appropriate advice. This deviates from the original definition of Karp and Lipton [**?**] which would require the machine to restore the catalytic space on any advice.)

The other possibility to define a non-uniform model for space bounded computation is via branching programs. A branching program for inputs of length n is a directed acyclic graph, where each node is labeled by one of the input variables $x_1, \ldots, x_n$ except two designated nodes ACCEPT and REJECT. Each node labeled by a variable x_i has two outgoing edges, one labeled by 0, the other by 1. The computation of the branching program on an input x starts in a designated initial node INI and follows a path consistent with the input, i.e., in a node labeled by x_i we follow the edge labeled by the actual value of the i-th input bit. Once we reach either ACCEPT or REJECT the computation ends and the final node represents the output. Families of branching programs of polynomial size are known to compute functions from $\mathsf{L}/poly$.

The model that corresponds to catalytic computation are the catalytic branching programs. In the context of proving lower bounds they were originally studied by Cook and Filmus [?], and later they were investigated by Girard, Koucký and McKenzie [?]. A catalytic branching program has W initial nodes $\mathrm{INI}_1, \ldots, \mathrm{INI}_W$ and $2W$ final nodes $\mathrm{ACCEPT}_1, \ldots, \mathrm{ACCEPT}_W$ and $\mathrm{REJECT}_1, \ldots, \mathrm{REJECT}_W$. When the computation starts in INI_i it must finish in either ACCEPT_i or REJECT_i. (The W initial nodes correspond to W possibilities for initial setting of the catalytic space.) Hence, starting from INI_i the catalytic branching program computes some function f_i, and overall it computes some W-tuple of functions $(f_1, \ldots, f_W)$.

The basic question is what is the smallest size of a catalytic branching program for a given W-tuple of functions. A trivial construction of a catalytic branching program for $(f_1, \ldots, f_W)$ puts together W branching programs, each computing one of the functions. The size of such a branching program is the sum of the sizes of the W programs. Is there a more efficient way to construct catalytic branching programs?

It is tempting to conjecture that the trivial construction is the best possible. This is known for some functions, for example for a W-tuple of random functions or for functions computed by read-once Boolean formulas [?]. However, in general this is not the case as demonstrated in [?].

Theorem 8 (Girard, Koucký, and McKenzie)**.** *For any n there are functions $f_1, \ldots, f_W : \{0,1\}^n \to \{0,1\}$, $W = 2^{n/2}$, such that the minimal branching program for each f_i has size $\Omega(2^n/n)$ but the size of a catalytic branching program for $(f_1, \ldots, f_W)$ is $O(2^n/n)$.*

Thus, the trivial construction can be far from optimal. Currently, we do not know of any single *complex* function where the trivial construction of catalytic branching programs for its W-tuple would be optimal. [?] conjecture that a random function should be such an example but the counting argument which works for a W-tuple of *independent* random functions does not work for a single random function.

Girard, Koucký and McKenzie establish a correspondence between catalytic branching programs and catalytic computation. A description of a catalytic branching program of size S for W-tuple $(f, \ldots, f)$, i.e., f iterated W times, can be given as an advice to a catalytic machine working in space $\log S/W$ with catalytic space $\log W$ to compute f, and vice versa. If f can be computed by a catalytic machine in space s with catalytic space w, then 2^w-tuple of f can be computed by a catalytic branching program of size $2^w \cdot 2^s$. This works also when the catalytic machine is getting some non-uniform advice.

Using this correspondence, Girard, Koucký and McKenzie argue that if $\mathsf{NL} \not\subseteq \mathsf{L}/poly$, i.e., when non-deterministic log-space is not in non-uniform log-space, then for STCONN there are catalytic branching programs computing 2^{3n^3}-tuple of STCONN more efficiently than the trivial construction. This builds on the result of Buhrman et al. [**?**]. A similar claim holds also for complete functions in LOGCFL under this or the weaker assumption $\mathsf{LOGCFL} \not\subseteq \mathsf{L}/poly$.

We do not know of any example of a single function, where we could obtain savings over the trivial construction unconditionally. Possible candidates are symmetric functions. We have nontrivial lower bounds for them [**?**], and also they can be computed by *permutation* branching programs. That could be useful in a construction of a nontrivial catalytic branching program.

The question on the size of catalytic branching programs is related to *direct sum* type of questions for space. Consider two functions $f : \{0,1\}^n \to \{0,1\}^n$ and $g : \{0,1\}^n \to \{0,1\}$. What is the space needed to compute their composition $g(f(\cdot))$? Is it the sum of the space needed to compute each of them separately? It is easy to see that the space can be less if there is an efficient catalytic program for f. These questions are also related to the question on the depth of formulas computing composition of functions [**?**, **?**].

10 Catalytic space hierarchy

One of the first complexity questions about catalytic space one might ask is whether the catalytic space obeys some form of space hierarchy, i.e., providing the machine with more space allows one to compute more problems. It is natural to expect that such a space hierarchy should exist. However, proving it is a different matter. The model imposes *semantic* condition on the behavior of the machine, and we do not now how to enumerate correctly behaving catalytic machines. That means that diagonalization, the usual tool for proving hierarchy theorems, is not directly applicable to our model. This is similar to the situation with other semantic classes such as bounded-error probabilistic computation (BPP, etc.). However, one can apply general techniques that were developed for proving hierarchy theorems for semantically defined classes in the non-uniform setting. Using the technique of Kinne and van Melkebeek, and van Melkebeek and Pervyshev [**?**, **?**, **?**], Buhrman et al. [**?**] conclude the following.

Theorem 9. *For any integer $a \geq 1$ and real $k > 0$ there exists $k' > k$ such that*

1. $\mathsf{CSPACE}(\omega(\log n))/1 \not\subseteq \mathsf{CSPACE}(\log n)/a = \mathsf{CL}/a$,

2. $\mathsf{CSPACE}(n^{k'})/1 \not\subseteq \mathsf{CSPACE}(n^k)/a$.

Similar claim holds also for the non-deterministic catalytic space. Since

$$\mathsf{CL} \subseteq \mathsf{PSPACE} \subsetneq \mathsf{DSPACE}(n^{\omega(1)}) \subseteq \mathsf{CSPACE}(n^{\omega(1)})$$

uniformly one can conclude a much weaker statement:

$$\mathsf{CL} \subsetneq \mathsf{CSPACE}(n^{\omega(1)}).$$

Separating CL even from $\bigcup_{k>0} \mathsf{CSPACE}(n^k)$ might be difficult as we know of an oracle A where $\mathsf{CL}^A = \mathsf{PSPACE}^A$.

These statements work for classes where the catalytic space is exponential in the work space. One might ask whether $\mathsf{CSPACE}(s(n), o(w(n))) \subsetneq \mathsf{CSPACE}(s(n), w(n))$? Currently we do not know whether this is true even non-uniformly. The Iterated Matrix Multiplication of $\sqrt{w(n)} \times \sqrt{w(n)}$ matrices over $\mathbb{Z}$ is a candidate problem that is known to be in $\mathsf{CSPACE}(\log n, w(n) \cdot \log n)$ [**?**] but not known to be in $\mathsf{CSPACE}(\log n, o(w(n)))$.

11 Non-deterministic catalytic computation

Non-deterministic computation is a useful paradigm for understanding and classifying some algorithmic problems. In the context of catalytic computation non-determinism could provide an avenue for designing algorithms for problems not known to be in CL. Motivated by this, Buhrman et al. [**?**] define non-deterministic catalytic computation. There are different ways how to define non-determinism for catalytic machines, Buhrman et al. chose the following requirements:

a) **Catalicity.** For each initial setting of the catalytic tape and any choice of non-deterministic bits the machine halts and restores its catalytic tape to its initial setting.

b) **Consistency.** If a machine non-deterministically accepts an input x for some initial setting of the catalytic tape then it non-deterministically accepts x on every possible initial setting of the catalytic tape.

These requirements seem most natural as they preserve the spirit of the catalytic model. Additionally, they also allow composition of non-deterministic computation as is done for example for computing the union of two languages. We will denote by CNL the class of languages accepted non-deterministically by a catalytic machine using polynomial catalytic tape and logarithmic work tape.

For the classical computation Savitch [?] established a relationship between determinism and non-determinism: $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s^2(n))$. We do not know of similar relationship for catalytic computation. Savitch's proof goes by arguing about reachability in the graph of configurations of the machine. There seem to be various obstacles to establishing some variant of Savitch's Theorem for CNL. On a particular initial setting of the catalytic tape, the graph of reachable configurations can be exponentially large. Even if it were polynomial, it is not clear how to deterministically cycle through all the configurations that are reachable from the initial configuration. These issues seem to break Savitch's technique. Interestingly though, CNL is still in ZPP.

Theorem 10 (Buhrman, Koucký, Loff, Speelman)**.** $\mathsf{CNL} \subseteq \mathsf{ZPP}$ *and more generally,* $\mathsf{CNSPACE}(s(n)) \subseteq \mathsf{ZPTIME}(2^{O(s(n))})$.

The argument is similar to the one for deterministic CL as the average number of reachable configurations is still polynomial.

Buhrman et al. [?] provide also a variant of the Immerman-Szelepcsényi Theorem [?, ?] which shows that non-deterministic log-space is closed under the complement (i.e., complements of languages from NL are also in NL). The proof of Buhrman et al. requires use of pseudo-random generators [?] so the theorem is known to hold only under certain derandomization assumption.

Theorem 11 (Buhrman, Koucký, Loff, Speelman)**.** *If there exists* $\epsilon > 0$ *and* $L \in DSPACE(n)$ *which cannot be computed by Boolean circuits of size* $2^{\epsilon n}$ *then* $\mathsf{CNL} = \mathsf{coCNL}$.

In non-uniform setting the conclusion would hold without any assumption. The proof uses the inductive counting technique of Immerman and Szelepcsényi. To overcome the problem with exponentially many reachable configurations Buhrman et al. use the pseudo-random generator. For the actual inductive counting they do not enumerate over all possible configurations but only the reachable ones and they use finger-printing technique to distinguish them.

It was observed recently together with Tewari [?] that a similar technique should also establish an equivalent of the Reinhardt-Allender Theorem [?, ?] that $\mathsf{NL}/poly \subseteq \mathsf{UL}/poly$. UL is the class of languages accepted by a non-deterministic Turing machine running in log-space that has at most one non-deterministic accepting computation on every input. UL is the space analog of UP with the complete problem UNIQUE-SAT [?, ?].

It would be interesting to see some problems outside of TC^1 to be put in CNL. Languages in NC^2 would be natural candidates. (NC^2 is defined

similarly to NC^1 but one allows depth of $O(\log^2 n)$. It is well known that $\mathsf{TC}^1 \subseteq \mathsf{NC}^2$.)

12 Conclusions

We have seen that the catalytic space provides unexpected power to computation. There are many questions remaining to be answered. We summarize here some of the major ones.

1. Are there problems beyond TC^1 that are computable in catalytic log-space?

2. What are other techniques for using the catalytic space beyond simulating transparent computation? What is the relationship between transparent computation and catalytic computation?

3. Is catalytic log-space contained in P? Is it in NC^2?

4. Is there uniform hierarchy of catalytic space? Is there hierarchy with respect to the amount of catalytic space?

5. What is the relationship between deterministic and non-deterministic catalytic computation?

6. What can one say about *randomized* catalytic computation?

7. Is there some meaningful relaxation of catalytic computation? For example, one could allow the machine with low probability to destroy the content of the catalytic tape.

Nontrivial answers to some of these questions would provide us with more insight into the role of space in computation.

References