The Division Breakthroughs

Eric Allender [1]

1 Introduction

All of us learn to do arithmetic in grade school. The algorithms for addition and subtraction take some time to master, and the multiplication algorithm is even more complicated. Eventually students learn the division algorithm; most students find it to be complicated, time-consuming, and tedious. Is there a better way to divide?

For most practical purposes, the correct way to answer this question is to consider the time-complexity of division; what is the fastest division algorithm? That is *not* the subject of this article. I am not aware of any recent breakthrough on this question; any good textbook on design and analysis of algorithms will tell you about the current state of the art on that front.

Complexity theory gives us an equally-valid way to ask about the complexity of division:

In what complexity class does division lie?

One of the most important subclasses of P (and one of the first to be defined and studied) is the class L (deterministic logarithmic space). It is easy to see how to add and subtract in L. It is a simple exercise to show that multiplication can be computed in logspace, too. However, it had been an open question since the 1960's if logspace machines can divide.

This was fairly annoying.

Let me give an example, to illustrate how annoying this was.

We like to think of complexity classes as capturing fundamental aspects of computation. The question of whether a particular problem lies in a complexity class or not should not depend on trivial matters, such as minor issues of encoding. As long as a "reasonable" encoding is used, it should not make much difference exactly how the problem is encoded. For example, a computational problem involving numbers should have roughly the same complexity, regardless of whether the numbers are encoded in base ten, or base two, or some other reasonable notation. Unfortunately, it was not known how convert from base ten to base two in logspace, and thus one could not safely ignore such matters when discussing the class L.

Breakthrough number 1: [20] *Division is in Logspace.*

[1]Department of Computer Science, Rutgers University, `allender@cs.rutgers.edu`. This work was supported in part by National Science Foundation Grant CCR-0104823.

As a consequence, related problems (such as converting from base ten to base two) also lie in logspace.

Complexity theorists are not happy until they have pinpointed the "right" complexity class for a problem. That is, they want to find the complexity class for which a problem is complete; this corresponds to a tight lower bound on the complexity of a problem. In the case of division, defining the "right" complexity class takes a bit of explanation, as does defining the notion of "completeness". I'll provide the necessary definitions later. For now, let's state the result:

Breakthrough number 2: [26] *Division is complete for* DLOGTIME-*uniform* TC^0.

This latest breakthrough was presented at ICALP 2001 by Bill Hesse, then a student at the University of Massachusetts. He received the *best paper award* for Track A at ICALP 2001 (combined with the best student paper award). A journal publication containing this and earlier results of [10] (on which [26] builds) is available as [24].

All of these results build on the earlier work of Beame, Cook, and Hoover ([14]).

In the following sections, I will provide the necessary background about the complexity classes I'll be discussing, and then I'll present the history of these breakthroughs, and the main ideas involved. In a closing section, I'll discuss some of the applications that these advances have already found.

2 Background on Complexity Classes

In order to understand the recent progress on division, it is necessary to understand the significance of the complexity classes involved. In this article, we shall be concerned almost exclusively with subclasses of P. Figure 2 lists some of the classes that we will focus on, along with a list of some problems that are complete for each class under $\leq_m^{\mathrm{AC}^0}$ reductions. (For small complexity classes, $\leq_m^{\mathrm{AC}^0}$ is one of the most natural notions of reducibility to consider. For more background on $\leq_m^{\mathrm{AC}^0}$ you can consult an earlier edition of this column [6].)

Deterministic and nondeterministic logspace (L and NL) are probably familiar to the reader. #L is the logspace-analog of the class #P; #L is the class of functions f for which there exists a nondeterministic logspace machine M such that $f(x)$ is the number of accepting computations of M on input x. GapL is the class of all functions that can be expressed as the difference of two #L functions. Additional background about these complexity classes can be found in an earlier survey I wrote [7], and in the excellent textbook by Vollmer [41].

The remaining two complexity classes in Figure 2 are circuit complexity classes.

Complexity Class	Complete Problem
GapL	Determinant of Integer Matrices
#L	Counting paths in a DAG
Mod_pL	Determinant of Integer Matrices mod p
NL	Shortest paths, Transitive Closure
L	Graph Acyclicity, Tree Isomorphism
NC^1	Regular sets, Boolean Formula Evaluation
TC^0	

Figure 1: Some complexity classes, and some sample sets complete under AC^0 reductions.

NC^1 is the class of languages A for which there exist circuit families $\{C_n : n \in \mathbb{N}\}$ where each circuit C_n

- computes the characteristic function of A on inputs of length n,
- consists of AND and OR gates of fan-in two,
- has depth $O(\log n)$ (and consequently has size $n^{O(1)}$).

TC^0 is the class of languages A for which there exist circuit families $\{C_n : n \in \mathbb{N}\}$ where each circuit C_n

- computes the characteristic function of A on inputs of length n,
- consists of MAJORITY gates (with no bound on the fan-in),
- has depth $O(1)$
- has size $n^{O(1)}$.

It will cause no confusion to use the terms NC^1 and TC^0 also to refer to classes of *functions* computed by these classes of circuits, instead of merely focusing on *languages*. For instance, the $\leq_m^{\text{AC}^0}$ reducibility mentioned earlier comes from the class AC^0, which is defined the class of functions f for which there exist circuit families $\{C_n : n \in \mathbb{N}\}$ where each circuit C_n

- computes f on inputs of length n,
- consists of AND and OR gates (with no bound on the fan-in),
- has depth $O(1)$

- has size $n^{O(1)}$.

The circuit classes NC^1, TC^0, and AC^0 each come in different flavors corresponding to different *uniformity conditions*. As defined above, these classes are *nonuniform*. That is, there is *no restriction* on how difficult it is to compute the function $n \mapsto C_n$ (i.e., on how hard it is to *build* the circuits). In order to obtain subclasses of P, it is necessary to impose a “P-uniformity” condition. That is, the function $n \mapsto C_n$ must be computable in polynomial time. Even the P-uniformity condition does not seem to be strong enough to define subclasses of L; this leads us to consider L-uniformity. In the same way, L-uniformity is awkward when we want to consider subclasses of NC^1. We seem to have started down a slippery slope of increasingly more restrictive uniformity conditions, and it is natural to wonder if there is any uniformity condition that is particularly natural or preferable to others.

There is a consensus in the community of researchers in circuit complexity that the “right” uniformity condition is DLOGTIME-uniformity. For the rest of this paper, any reference to “uniform” circuits means “DLOGTIME-uniform” circuits, unless some other uniformity condition is explicitly mentioned. For this paper, you won’t need to be concerned with the details of this uniformity condition; for details you can consult [37, 13, 41]. (The “straightforward” notion of DLOGTIME-uniformity needs to modified a bit in order to give a satisfactory uniformity condition for NC^1 [37].) What gives rise to this consensus?

The answer to this question lies in the fact that most members of the complexity theory community are more comfortable programming than building circuits. They prefer to have a machine model that they can program in. Thus it is very desirable that uniform NC^1 correspond to logarithmic time on an alternating Turing machine [37] and uniform AC^0 correspond to logarithmic time on an alternating Turing machine making $O(1)$ alternations [15]. Similarly, uniform TC^0 corresponds to logarithmic time and $O(1)$ “alternations” on a threshold machine [34, 4].

Further support for this uniformity condition comes from a series of striking connections to finite model theory. A language is in uniform AC^0 if and only if it can be viewed as the class of finite models of a first-order formula. That is, a single formula (with existential and universal quantifiers) defines an entire language, as opposed to having a different circuit (i.e., a Boolean formula) for each input length. The reader can find out more about this connection between logic and complexity in an earlier edition of this column [27] or in the text by Immerman [28]. Lindell gives yet another characterization of uniform AC^0 [32], lending more support to this choice of uniformity condition. When we augment the existential and universal quantifiers with “majority” quantifiers (i.e., instead of asserting that a predicate holds for all or some elements, we assert that it holds for “most” domain elements), then we obtain an equivalent characterization of uniform TC^0.

For this reason, uniform AC^0 is frequently referred to as FO (for "first order"), and uniform TC^0 is frequently referred to as FOM (for "first-order with MAJORITY").

The logical framework gives rise to a natural notion of reducibility. Suppose that language A can be expressed by a first-order formula (or a FOM formula) with a new predicate symbol Q. Then we say that A is in FO $+ \, Q$ (or FOM $+ \, Q$).

There are yet more types of reducibility that we'll need. The alert reader will have noticed that Figure 2 does not list *any* complete problems for TC^0 under $\leq_m^{\mathrm{AC}^0}$ reducibility. This is because it is widely believed that no such language or function exists! On the other hand, TC^0 does have several natural problems that are complete under $\leq_T^{\mathrm{AC}^0}$ reductions, including MAJORITY, integer multiplication, and sorting. Function g is $\leq_T^{\mathrm{AC}^0}$ reducible to f if there is a uniform family of polynomial-size, $O(1)$-depth circuits of AND, OR, and NOT gates and "f gates" (i.e., gates with m input wires and r output wires, where for each m-bit input y, the r output wires take on the r-bit value $f(y)$), where the circuit family computes g.

In the same way, we can define $\leq_T^{\mathrm{TC}^0}$ reductions.

We now know that division is also complete for TC^0 under $\leq_T^{\mathrm{AC}^0}$ reductions. (It had been known for a while that multiplication reduces to division, and thus division was known to be hard for TC^0. In fact, division had been known to be in P-uniform TC^0 ever since it was observed in [35, 36] that the algorithm of [14] can be implemented in P-uniform TC^0. The breakthrough of [26] is that division is in DLOGTIME-uniform TC^0.

3 Background on Division

All of the recent work on division builds on the work of Beame, Cook, and Hoover [14]. Beame, Cook, and Hoover make use of the fact that, for small enough u, $1/(1-u) = \sum_{i=0} u^i$. Thus to divide x by y, we first let j be roughly the number of bits in y, so $2^{j-1} \leq y < 2^j$, and let $u = 1 - (y/2^j)$. Thus $x/y = x2^{-j}(\sum_{i=0} u^i)$, which can be approximated to n bits of accuracy by computing $x/y = x2^{-j}(\sum_{i=0}^{n} u^i)$. Since addition of polynomially-many numbers can be performed in uniform TC^0, this entire algorithm can be viewed as a $\leq_T^{\mathrm{TC}^0}$ reduction from division to the problem of computing the powers u^i. For our purposes, we will focus on the more general problem of ITERATED MULTIPLICATION (given n integers, each having n bits, compute their product).

That is, the argument of Beame, Cook, and Hoover shows that if ITERATED MULTIPLICATION is in FOM, so is division. Accordingly, most of the work in [14] focuses on presenting efficient circuits for ITERATED MULTIPLICATION.

The central idea of all the TC^0 algorithms for DIVISION and related problems is that of *Chinese remainder representation (CRR)*. An n-bit number is uniquely determined by its residues modulo polynomially many primes, each having $O(\log n)$ bits. (The Prime Number Theorem guarantees that there will be more than enough primes of that length.) More precisely, let $m_1, \ldots, m_k$ be a sequence of primes, each having $O(\log n)$ bits. Let $M = \prod_{i=1}^{k} m_i$. Any number $X < M$ can be represented uniquely by the sequence $(x_1, \ldots, x_k)$ with $X \equiv x_i \pmod{m_i}$ for all i. The sequence $(x_1, \ldots, x_k)$ is called the CRR$_M$ of X. If M is clear from context, we will simply call this the CRR of X.

For each number i, let C_i be the product of all the m_j's except m_i, and let h_i be the inverse of C_i modulo m_i. It is easy to verify that X is congruent modulo M to $\sum_{i=1}^{k} x_i h_i C_i$. In fact X is *equal*, as an integer, to $(\sum_{i=1}^{k} x_i h_i C_i) - rM$ for some particular number r, called the *rank* of X with respect to M (denoted rank$_M(X)$).

Here I am following the convention introduced in [10] of using capital letters (such as X, M, etc.) to refer to numbers with bit length polynomially-related to n (call these "long numbers"), and lower-case letters (such as r, x_i, etc.) to refer to numbers with bit length $O(\log n)$ (call these "short numbers"). In particular, note that r is a short number.

The algorithm of Beame, Cook, and Hoover can be summed up in the following three lines:

1. Converting from binary to CRR is in L-uniform TC^0.

2. ITERATED MULTIPLICATION is in L-uniform TC^0, if the input and output are in CRR.

3. Converting from CRR to binary is in P-uniform TC^0.

Let us consider the first two of these points.

To convert an n-bit number X from binary to CRR we merely need to find $x_{i,j} = 2^j \pmod{m_i}$ for each modulus m_i and each $j < n$ such that bit j of X is 1. Taking the sum $\sum_j x_{i,j} \bmod m_i$ gives us the value x_i in the CRR of X. Since the values $2^j \bmod m_i$ are easy to compute in L, this part of the argument is established.

Computing ITERATED MULTIPLICATION in CRR is easy, when we observe that we can add the discrete logs. More precisely, each prime modulus m_i has a generator g_i generating the (cyclic) multiplicative group of the integers mod m_i. That is, for each $x < m_i$ there is a number $\ell(x)$ such that $x \equiv g_i^{\ell(x)} \pmod{m_i}$. We are given a sequence of numbers $X_1, \ldots, X_n$ in CRR, and we want to compute $\prod_j X_j$. Thus, for each modulus i we want to compute $\prod_j X_j \equiv \prod_j x_{j,i} \pmod{m_i} \equiv \prod_j g_i^{\ell(x_{j,i})} \pmod{m_i} \equiv g_i^{\sum_j \ell(x_{j,i})} \pmod{m_i}$.

In logspace, it is easy to build a table of discrete logs for each small modulus, and hardwire this into an L-uniform TC^0 circuit. Thus we can find the discrete logs of each $x_{j,i}$ and we add them mod m_i, and then (again using our discrete log table) find the result of raising g_i to that power. This gives us one component of our answer in CRR.

The remaining part of the algorithm in Beame, Cook, and Hoover is converting from CRR to binary. As presented in [14] (see also [29]), the basic approach was to note that X is equal to $(\sum_{i=1}^{k} x_i h_i C_i) - rM$. If the binary representation of M was given, then the value $(\sum_{i=1}^{k} x_i h_i C_i)$ could be computed in binary. It was not clear how to compute the number r (the rank of X), but since r is a short number, there are not many possible values for r. Thus the circuit could try all possible values of $(\sum_{i=1}^{k} x_i h_i C_i) - rM$ and pick the right one. The bottleneck was that nobody knew how to compute the binary representation of M in logspace (although it was easy to compute this in polynomial time).

A new approach was needed.

4 Breaking the Logspace Barrier

Andrew Y. Chiu received his MS degree from the University of Wisconsin at Milwaukee in August, 1995. A mathematical prodigy, he subsequently left computer science to enter law school.

His MS thesis [19] remained unknown to most of the community for several years. No paper summarizing its contributions was presented at any of the conferences where researchers usually announce their latest theorems. No technical report was published on ECCC or on any of the other repositories for such material. We all owe a great debt to Chiu's advisor, George Davida, and to his collaborator Bruce Litow, for preparing a journal paper building on his work [20].

Chiu's MS thesis [19] shows that division and ITERATED MULTIPLICATION lie in uniform NC^1.

In this survey, I'll sketch for now only a proof that these problems lie in L-uniform TC^0. As observed in the previous section, it is sufficient to show that one can convert from CRR to binary in L-uniform TC^0.

Following the development in [24], I'll actually state and sketch a slightly stronger result. Let POW(a, i, b, p) be defined to be true if and only if $a^i \equiv b \pmod{p}$ where a, b, i and p each have $O(\log n)$ bits, and p is prime. We'll show that converting from CRR to binary is in FOM $+$ POW. It is easy to see that POW is computable in logspace, as desired.

The reader can check that the L-uniform TC^0 circuits for converting from binary to CRR and for computing ITERATED MULTIPLICATION in CRR representation can actually be implemented in $\mathrm{FOM}+\mathrm{POW}$. Thus, if we can show that converting from CRR to binary is in $\mathrm{FOM}+\mathrm{POW}$, it will follow that division lies in this class. (In fact, it is shown in [10] that division is *complete* for $\mathrm{FOM}+\mathrm{POW}$, and that it follows from a well-known number-theoretic conjecture that POW lies in FOM. Both of these latter results are superseded by [26, 24].)

In this overview, I will state and give a hint of the main lemmas. For more details, the reader can consult [24].

Lemma 4.1 *Let p be a short prime. Then the binary representation of $1/p$ can be computed to $n^{O(1)}$ bits of accuracy in* $\mathrm{FO}+\mathrm{POW}$.

Proof. The reader may easily verify that if p is an odd prime, then the the kth bit of the binary expansion of the rational number $1/p$ is the low-order bit of $2^k \bmod p$. (Alternatively, a proof is presented in [24].) ■

It is not at all obvious how to tell, given two numbers in CRR, which is larger. Logspace algorithms for this were presented in [21, 22]. Using the preceding lemma, we obtain yet another algorithm.

Lemma 4.2 *Let X and Y be numbers less than M given in CRR$_M$ form. In* $\mathrm{FOM}+\mathrm{POW}$ *we can determine whether $X < Y$.*

Proof. Clearly, $X < Y$ if and only if $X/M < Y/M$. Thus it is sufficient to show that we can compute X/M to polynomially-many bits of accuracy.

Recall that $X = (\sum_{i=1}^{k} x_i h_i C_i) - \mathrm{rank}_M(x)M$. Thus X/M is equal to

$$(\sum_{i=1}^{k} x_i h_i (1/m_i)) - \mathrm{rank}_M(x).$$

The reader can verify that the numbers x_i, $C_i \bmod m_i$, and h_i are easy to obtain in $\mathrm{FOM}+\mathrm{POW}$. By Lemma 4.1, each summand can be computed in $\mathrm{FOM}+\mathrm{POW}$ to $n^{O(1)}$ bits of accuracy. Hence we obtain polynomially-many bits of the binary representation of $(\sum_{i=1}^{k} x_i h_i (1/m_i))$, which is equal to $X/M + \mathrm{rank}_M(X)$. Since the rank is an integer, X/M is simply the fractional part of this value. ■

A crucial insight of [20] is that it is easy to change from CRR representation with one set of moduli, to CRR representation using another set of moduli. This is called *changing the CRR basis*.

Lemma 4.3 *Given X in CRR$_M$ and a short prime p, we can compute $X \bmod p$ in* $\mathrm{FOM}+\mathrm{POW}$.

Proof. Assume wlog that p does not divide M. In this case, consider the CRR base $M' = Mp$. We would like to compute X in CRR$_{M'}$, since this would give us $X \bmod p$.

Trying each of the $p = n^{O(1)}$ possible values i for $X \bmod p$, we obtain the $CRR_{M'}$ of $n^{O(1)}$ different numbers $X_0, X_1, \ldots, X_{p-1}$, one of which is X. It is easy to see that X is the only one of these numbers that is less than M. Since we can compute the $CRR_{M'}$ of M, the lemma now follows from Lemma 4.2. ■

Another important insight of [20] is that it is easy to divide by products of distinct short primes.

Lemma 4.4 *Let $b_1, \ldots, b_\ell$ be distinct short primes, B be the product of the b_i's, and let X be given in* CRR$_M$ *form. Then we can compute $\lfloor X/B \rfloor$, also in* CRR$_M$ *form, in* FOM + POW.

Proof. Assume without loss of generality that B divides M. (Otherwise, extend the basis, using Lemma 4.3.) Thus let $M = BP$ where $P = \prod_{i=1}^{k} p_i$.

In FOM + POW we can compute the following quantities:

- B in CRR$_M$ (by adding the discrete logs modulo each m_i),
- The CRR$_M$ of $S = (\sum_{i=1}^{\ell} x_i h_i (B/b_i))$, where h_i is the multiplicative inverse of $B/b_i \bmod b_i$,
- $Y = X - S$ in CRR$_M$,
- $B^{-1} \bmod P$ (i.e., the unique number $T < P$ such that $BT \equiv 1 \pmod{P}$; this can be computed in CRR$_P$ by merely inverting each nonzero component of the CRR$_M$ of B).

The important things to note are that $S \equiv X \bmod B$, and also S is only larger than B by a polynomial factor. Since Y is a multiple of B, Y/B is an integer less than P. Thus if we compute YT in CRR$_P$ we have the CRR$_P$ of the integer Y/B, and from this we can compute Y/B in CRR_M.

Therefore $\lfloor X/B \rfloor$ differs from Y/B by at most an additive term of $n^{O(1)}$. That is, we can compute a list of $n^{O(1)}$ consecutive values, one of which is equal to $\lfloor X/B \rfloor$. We can find the correct value by determining the value j such that $(YT + j)B \le X < (YT + j + 1)B$. ■

Theorem 4.5 *Let X be given in* CRR$_M$ *form. Then we can compute the binary representation of X in* FOM + POW.

Proof. As observed in [20], it is sufficient to show that we can compute the CRR$_M$ of $\lfloor X/2^k \rfloor$ for any k. This is because, to get the k-th bit of a number X that is given to us in CRR, we compute $u = \lfloor X/2^k \rfloor$ and $v = \lfloor X/2^{k+1} \rfloor$, and note that the desired bit is $u - 2v$.

First, we create numbers $A_1, \ldots, A_k$, each a product of polynomially many short odd primes that do not divide M, with each $A_i > M$. Let $P = M \prod_{i=1}^k A_i$, and compute X in CRR$_P$. By Lemma 4.4 (or directly) we can compute $(1 + A_i)/2$ in CRR$_P$. It is easy to show that $(\prod_{i=1}^k (A_i + 1)) / \prod_{i=1}^k A_i < 1 + (k/M)$.

Note that in FOM $+$ POW we can compute the CRR$_P$ representation of $Q = \lfloor X \prod_{i=1}^k ((1 + A_i)/2) / \prod_{i=1}^k A_i \rfloor$. But $X \prod_{i=1}^k ((1 + A_i)/2) / \prod_{i=1}^k A_i$ is equal to $(X/2^k)(\prod_{i=1}^k (A_i + 1)) / \prod_{i=1}^k A_i < (X/2^k)(1 + (k/M))$. We determine which of $\{Q, Q - 1\}$ is the correct answer by checking if $Q2^k > X$. ■

5 Division in Uniform TC^0

In order to present Hesse's FOM division algorithm, it is useful to define parameterized versions of the three problems we have been studying thus far. Let $b(n)$ be a function on $\mathbb{N}$. (We will need to consider only $b(n) \in \{k \log \log n,\ k \log n,\ k \log^2 n,\ n\}$ for various constants k.) Define

- DIVISION$_{b(n)}$ to be the problem of computing $\lfloor X/Y \rfloor$ where X and Y are integers with $b(n)$ bits.
- ITERATED MULTIPLICATION$_{b(n)}$ to be the problem of taking $b(n)$ numbers $X_1, \ldots, X_{b(n)}$ (each with $b(n)$ bits) as input, and computing $\prod_i X_i$.
- POW$_{b(n)}$ to be the problem of computing POW(a, i, b, p), where each of a, i, b, p has $b(n)$ bits.

Thus the preceding section showed that, for some k, FOM $+$ POW$_{k \log n}$ contains both ITERATED MULTIPLICATION$_n$ and DIVISION$_n$.

The same analysis shows that for any reasonable $b(n)$, DIVISION$_{b(n)}$ and ITERATED MULTIPLICATION$_{b(n)}$ lie inside FOM$+$POW$_{k \log b(n)}$. A direct argument shows that for small enough $b(n)$ (for instance, for $b(n) = O(\log \log n)$), POW$_{b(n)}$ is in FO. We thus have the following corollary:

Corollary 5.1 *For any constant k, uniform* TC^0 *contains* DIVISION$_{k \log^2 n}$ *and* ITERATED MULTIPLICATION$_{k \log^2 n}$.

Hesse's theorem showing that division is in uniform TC^0 thus follows from the preceding corollary and the following theorem.

Theorem 5.2 *[26] For some constant k,* POW *is in*
FOM+ ITERATED MULTIPLICATION$_{k \log n}$+ DIVISION$_{k \log^2 n}$.

Proof. We are given a, b, i and p and we want to determine if $a^i \equiv b \pmod p$. Again, we will resort to the Chinese Remainder Theorem.

In FO we can find a list of $k = o(\log n)$ primes $d_1, \ldots, d_k$, such that for all j, $d_j < 2 \log p$ and d_j does not divide $p - 1$. (In this overview, I'll ignore the details of how to find this list.) Furthermore, we can find such a list where $D = \prod_j d_j$ itself has only $O(\log n)$ bits.

Our next step is to compute, for each j, the value $a_j = a^{\lfloor (p-1)/d_j \rfloor} \bmod p$. To do this, compute $p_j = p \bmod d_j$. (Since these numbers are very small, this can be done in FO.) In FO find the multiplicative inverse of a in the integers mod p (call this a^{-1}, and (using ITERATED MULTIPLICATION$_{k \log n}$ and DIVISION$_{k \log^2 n}$) compute a^{-p_j}. One can show that there is exactly one number $x < p$ such that $x^{d_j} \equiv a^{-p_j} \pmod p$, and that furthermore, this x is the value a_j that we seek. Since d_j is quite small, once again we can find the x such that $x^{d_j} \equiv a^{-p_j} \pmod p$ using ITERATED MULTIPLICATION$_{k \log n}$ and DIVISION$_{k \log^2 n}$.

Let $s = \lfloor iD/(p-1) \rfloor$. (We can think of s as being a gross approximation to i – but one having some nice properties.) Since $s < D$, s has a representation $(s_1, \ldots, s_k)$ in CRR$_D$. Since $i, D, p - 1$ and s are all short numbers, s and the CRR$_D$ of s can be computed in FO. If we define D_j to be D/d_j, and u_j to be $s_j D_j^{-1} \bmod d_j$, then by the Chinese Remainder Theorem we have $s \equiv \sum_j u_j D_j \bmod D$.

Using ITERATED MULTIPLICATION$_{k \log n}$ and DIVISION$_{k \log^2 n}$, we can compute the value $A = \prod_{j=1}^{k} a_j^{u_j} \bmod p$. An important insight of Hesse [26] is that A in some sense is "close" to a^i. More precisely, observe that we can compute the value $u = i - \sum_{j=1}^{k} u_j \lfloor (p-1)/d_j \rfloor \bmod p-1$ in FO. Then $a^i \equiv a^u A \pmod p$. Hesse furthermore is able to show that $u < \log^2 n$. Thus (by first computing $a^{\log n}$, if need be) we can compute a^u. Since we already have A, and since $a^i = a^u A$, we have succeeded in computing a^i, as desired. ■

6 Applications

The new division algorithms have already found application in a rather diverse collection of settings. Here is a small sample.

6.1 Graph Isomorphism

Although there is a long history of research on the graph isomorphism problem (GI), there has been very little progress on the problem of proving *lower bounds* on the complexity of graph isomorphism – until recently.

In [40], Torán shows that GI is hard for NL and for Mod_pL under $\leq_m^{\mathrm{AC}^0}$ reductions. Using the L-uniform TC^0 circuits of [20] for converting from CRR to binary, Torán was able to build on those hardness results and show that GI is hard for the apparently-larger class NC^1(GapL) under logspace many-one reductions. Using the improved uniformity provided by [26, 24], it now follows that GI is hard for NC^1(GapL) under $\leq_m^{\mathrm{AC}^0}$ reductions. (In an earlier version of this paper [8], I listed this as open problem, and instead stated only a weaker result.)

6.2 Time-Space Tradeoffs

In a recent edition of the Computational Complexity Column [33], Dieter van Melkebeek provided a survey of recent progress on time-space tradeoffs. Most of the results surveyed there are for SAT and related problems on deterministic and nondeterministic machines using sublinear space. In yet another example of the observation that "upper bounds yield lower bounds", the new division algorithm of [20] enabled the authors of [11] to transfer these lower bound techniques from the domain of nondeterministic computation to the realm of unbounded-error probabilistic computation.

6.3 Eulerian Paths

Toda was one of the first authors to show that some natural problems are complete for GapL [39]. Most of the reductions presented in [39] are very restrictive, showing that problems are hard for #L or GapL under $\leq_m^{\mathrm{AC}^0}$ reductions, or even under a more restrictive notion known as projections. However, there is a glaring example where he was forced to consider a very powerful notion of reducibility.

An *Euler tour* is a closed path in a graph that traverses each edge. Toda shows in [39] that counting the number of Eulerian tours in a graph is $\leq_T^{\mathrm{AC}^0}$-reducible to the problem of computing the determinant of an integer matrix. Thus it lies in AC^0(GapL). He was unable to show that counting Euler tours is hard for GapL under $\leq_T^{\mathrm{AC}^0}$ reductions, but he could show that GapL reduces to counting Euler tours under P-uniform TC^0 reductions, via a reduction that involves division. By making use of the results of [26, 24], we now see that the P-uniformity condition can be replaced by DLOGTIME-uniformity.

6.4 Arithmetic Circuits

The complexity classes #AC^0 and GapAC^0 were introduced in [1] and have been studied in [12] and [9]. (See also the survey article on arithmetic circuits [7], and the material on arithmetic circuits in [41].) The main motivation for introducing and studying these classes comes from the fact that they give rise to several characterizations of TC^0.

However, there was a problem with these characterizations – some of them were not known to hold in the uniform setting. For instance, four different language classes arising from arithmetic AC^0 circuits were shown to coincide with TC^0 in the non-uniform and P-uniform settings, but were not known to coincide in the DLOGTIME-uniform setting. Some more of these classes were shown to coincide in [12], but there still remained a question as to whether these classes were really the same as DLOGTIME-uniform TC^0.

It is an immediate consequence of [26, 24] that all of the classes introduced in [1] coincide with TC^0 even in the uniform setting.

Another important class of arithmetic circuits arises by arithmetizing NC^1 circuits. This yields the classes #NC^1 and GapNC^1, which have received attention in [18, 7, 41]. I conjecture in [7] that the functions in #NC^1 and GapNC^1 actually *coincide* with the functions in (Boolean) NC^1. (This conjecture is based on a very efficient simulation of arithmetic circuits by Boolean circuits, first presented by Jung [30].) However, until the work of Chiu, Davida, and Litow, it was not even known whether the functions in these classes can be computed in logspace. Now, we know that they can be; that is, every function in GapNC^1 is computable in logspace.

6.5 Powering in Finite Fields

It was shown in [2] that the techniques of [24] can be used to show that powering in finite fields of polynomial size can be performed in FO. This is used in [2] to show that there is a set that is complete for NP under DLOGTIME-uniform AC$[\oplus]$ circuits that is not complete under $\leq_m^{\mathrm{AC}^0}$ (even non-uniform reductions). This improves a result that appeared in an earlier version [3], where a more powerful uniformity notion was used instead.

6.6 Sparse Complete Sets

In [16], Cai and Sivakumar showed that if there is a sparse set that is hard for P under $\leq_m^{\mathrm{AC}^0}$ reductions, then P is equal to L-uniform TC^0. In [17] they proved an analogous result, showing that if there is a sparse set that is hard for NL under

$\leq_m^{\mathrm{AC}^0}$, then NL = L-uniform TC^0. The reason for having an L-uniformity condition, instead of the more-natural DLOGTIME-uniformity condition, was because their construction required some precomputation involving finite fields; in particular it was necessary to perform powering in small finite fields. By making use of the improved powering algorithm of [2], it follows that if there is a sparse hard set for P (or for NL) under $\leq_m^{\mathrm{AC}^0}$ reductions, then P (NL, respectively) is equal to DLOGTIME-uniform TC^0.

6.7 Additional Applications

Several additional applications of the improved division algorithms are surveyed in [24] (including division of polynomials, iterated multiplication of polynomials, power series computation, and applications in proof theory). The reader is referred to [24] for details.

7 Small Space Bounds

It is observed in [24] that the division algorithm of [20] provides a new translational lemma for small space bounds.

Usually a lower bound on the complexity of the binary encoding of a set follows from a bound on the complexity of the unary encoding. (The unary encoding of $A, \mathrm{un}(A)$, is defined to be $\{0^x : x \in A\}$.) This follows from a standard *translation lemma*, such as:

Lemma 7.1 (Traditional Translation Lemma) *If* $s(\log n) = \Omega(\log \log n)$ *is fully space-constructible, then the first statement below implies the second:*

- $A \in \mathrm{dspace}(s(n))$.
- $\mathrm{un}(A) \in \mathrm{dspace}(\log n + s(\log n))$.

The converse also holds, if $s(\log n) = \Omega(\log n)$.

Note in particular that this translation lemma does not allow one to derive any lower bound on the space complexity of A, assuming only a logarithmic lower bound on the space complexity of $\mathrm{un}(A)$.

There is another reasonable way to define small space complexity classes. Define DSPACE$(s(n))$ to be the class of languages accepted by Turing machines that begin their computation with a worktape consisting of $s(n)$ cells (delimited by endmarkers), as opposed to the more common complexity classes dspace$(s(n))$

where the worktape is initially blank, and the machine must use its own computational power to make sure that it respects the space bound of $s(n)$. Viewed another way, DSPACE$(s(n))$ is simply dspace$(s(n))$ augmented by a small amount of "advice", allowing the machine to compute the space bound. (This model was defined under the name "DEMONSPACE" by Hartmanis and Ranjan [25]. See also Szepietowski's book [38] on sublogarithmic space.)

DSPACE$(s(n))$ seems at first to share many of the properties of dspace$(s(n))$. In particular, it is still relatively straightforward to show that there are natural problems, such as the set of palindromes, that are not in DSPACE$(o(\log n))$.

The efficient division algorithm of [20] provides a new translation lemma.

Lemma 7.2 New translation lemma *Let* $s(n) = \Omega(\log n)$ *be fully space-constructible. Then the following are equivalent:*

- $A \in$ dspace$(s(n))$
- un$(A) \in$ DSPACE$(\log \log n + s(\log n))$.

Corollary 7.3 *In order to show NP is not contained in* L, *it suffices to present a set* $A \in NP$ *such that* un$(A) \notin$ DSPACE$(\log \log n)$.

At first glance, this corollary may seem surprising, since there are sets in NP (such as the set of prime numbers) whose unary encoding is known *not* to be in dspace$(\log \log n)$ [23]. It might seem as if the computational power of the classes dspace$(\log \log n)$ and DSPACE$(\log \log n)$ might not be so very different. One consequence of our new insight into division is that it is now clear that the DSPACE classes can carry out simulations that seem impossible in the dspace model.

References

[1] M. Agrawal, E. Allender, and S. Datta. On TC^0, AC^0, and Arithmetic Circuits. *J. Computer and System Sciences* **60**:395–421, 2000.

[2] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi, and Steven Rudich. Reducing the Complexity of Reductions. *Computational Complexity* **10**:117-138, 2001.

[3] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi, and Steven Rudich. Reducing the Complexity of Reductions. *Proc. 29th ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 730–738.

[4] E. Allender. The Permanent Requires Large Uniform Threshold Circuits. *Chicago Journal of Theoretical Computer Science*, article 7, 1999.

[5] E. Allender. News from the Isomorphism Front. In The Computational Complexity Column, *EATCS Bulletin* **66**:73–82, October, 1998.

[6] E. Allender. Some Pointed Questions about Asymptotic Lower bounds, and News from the Isomorphism Front. In "Current Trends in Theoretical Computer Science: Entering the 21st Century," G. Păun, G. Rozenberg, and A. Salomaa, ed., World Scientific Press, 2001, pp. 25–41. The referenced material in this article originally appeared in [5].

[7] E. Allender. Making Computation Count: Arithmetic Circuits in the Nineties. In the Complexity Theory Column, edited by Lane Hemaspaandra, *SIGACT News* **28**, 4 (December, 1997) pp. 2-15.

[8] E. Allender. The Division Breakthroughs. In The Computational Complexity Column (ed. L. Fortnow), *EATCS Bulletin* **74**:61–77, June, 2001.

[9] E. Allender, Andris Ambainis, David A. Mix Barrington, Samir Datta, and Huong LêThanh. Bounded Depth Arithmetic Circuits: Counting and Closure. In *Proc. 26th International Colloquium on Automata, Languages, and Programming (ICALP)*, 1999, Lecture Notes in Computer Science 1644, pp. 149–158.

[10] E. Allender, D. Mix Barrington, and W. Hesse. Uniform Circuits for Division: Consequences and Problems. In *Proc. 16th Annual IEEE Conference on Computational Complexity*, 2001, pp. 150–159.

[11] E. Allender, Michal Koucky, Detlef Ronneburger, Sambuddha Roy, and V. Vinay. Time-Space Tradeoffs in the Counting Hierarchy. To appear in Theory of Computing Systems. An earlier version appeared in *Proc. 16th Annual IEEE Conference on Computational Complexity*, 2001, pp. 295–302.

[12] A. Ambainis, D. Mix Barrington, H. LêThanh. On Counting AC^0 Circuits with Negative Constants. In *Proc. 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 1998, Lecture Notes in Computer Science 1450, pp. 409–417.

[13] D. A. M. Barrington, N. Immerman, and H. Straubing. On Uniformity within NC^1. *J. Computer and System Sciences*, **41**:274–306, 1990.

[14] P. Beame, S. Cook and J. Hoover. Log Depth Circuits for Division and Related Problems. *SIAM J. Comput.*, **15**:994–1003, 1986.

[15] S. Buss, The Boolean Formula Value problem is in ALOGTIME. In *Proc. 19th ACM Symposium on Theory of Computing (STOC)*, 1987, pp. 123–131.

[16] J.-Y. Cai and D. Sivakumar. Resolution of Hartmanis' Conjecture for NL-Hard Sparse Sets. *Theoretical Computer Science*, **240**:257–269, 2000.

[17] J.-Y. Cai and D. Sivakumar. Sparse Hard Sets for P: Resolution of a Conjecture of Hartmanis. *J. Computer and System Sciences*, **58**:280–296, 1999.

[18] H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic NC^1 computation. *J. Computer and System Sciences*, **57**:200–212, 1998.

[19] A. Chiu. Complexity of Parallel Arithmetic Using the Chinese Remainder Representation. Master's thesis, U. Wisconsin-Milwaukee, 1995. G. Davida, supervisor.

[20] A. Chiu, G. Davida, and B. Litow. Division in Logspace-Uniform NC^1. *RAIRO Theoretical Informatics and Applications* **35**:259–276, 2001.

[21] G. I. Davida and B. Litow. Fast Parallel Arithmetic via Modular Representation. *SIAM J. Comput.*, **20**:756–765, 1991.

[22] Paul F. Dietz, Ioan I. Macarie, and Joel I. Seiferas. Bits and Relative Order from Residues, Space Efficiently. *Information Processing Letters*, **50**:123–127, 1994.

[23] J. Hartmanis and L. Berman. On Tape Bounds for Single Letter Alphabet Language Processing. *Theoretical Computer Science* **3**:213–224, 1976.

[24] W. Hesse, E. Allender, and D. Mix Barrington. Uniform Constant-Depth Threshold Circuits for Division and Iterated Multiplication. *J. Computer and System Sciences* **65**:695–716, 2002.

[25] J. Hartmanis and D. Ranjan. Space Bounded Computations: Review and New Speculation. In *Proc. 14th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 1989, Lecture Notes in Computer Science 379, pp. 49–66.

[26] W. Hesse. Division is in Uniform TC^0. In *Proc. 28th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2001, Lecture Notes in Computer Science 2076, pp. 104–114.

[27] N. Immerman. Progress in Descriptive Complexity. The Computational Complexity Column, *EATCS Bulletin* **67**:72–80, February, 1999.

[28] N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.

[29] Neil Immerman and Susan Landau. The Complexity of Iterated Multiplication. *Information and Computation* **116**:103–116, 1995.

[30] H. Jung. Depth Efficient Transformations of Arithmetic into Boolean Circuits. In *Proc. 5th Fundamentals of Computation Theory (FCT)*, 1985, Lecture Notes in Computer Science 199, pp. 167–173.

[31] I. Macarie. Space-Efficient Deterministic Simulation of Probabilistic Automata. *SIAM J. Comp.* **27**:448-465, 1998.

[32] Steven Lindell. A Purely Logical Characterization of Circuit Uniformity. In *Proc. 7th Annual IEEE Structure in Complexity Theory Conference*, 1992, pp. 185–192.

[33] D. van Melkebeek. Time-Space Lower Bounds for Satisfiability. This volume. An earlier version appeared in The Computational Complexity Column, *EATCS Bulletin* **73**:57-77, February, 2001.

[34] I. Parberry and G. Schnitger. Parallel Computation with Threshold Functions. *J. Computer and System Sciences*, **36**:278–302, 1988.

[35] J. Reif, On Threshold Circuits and Polynomial Computation. In *Proc. Annual 2nd IEEE Structure in Complexity Theory Conference*, 1987, pp. 118–123.

[36] J. Reif and S. Tate. On Threshold Circuits and Polynomial Computation. *SIAM J. Comput.*, **21**:896–908, 1992.

[37] W. L. Ruzzo. On Uniform Circuit Complexity. *J. Computer and System Sciences*, **21**:365–383, 1981.

[38] A. Szepietowski. *Turing Machines with Sublogarithmic Space*. Lecture Notes in Computer Science 843, Springer-Verlag, 1994.

[39] S. Toda. Counting Problems Computationally Equivalent to Computing the Determinant. Technical Report CSIM 91-07, Department of Computer Science, University of Electro-Communications, Tokyo, Japan, 1991.

[40] J. Torán. On the Hardness of Graph Isomorphism. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, 2000, pp. 180–186.

[41] H. Vollmer. *Introduction to Circuit Complexity*. Springer-Verlag, 1999.