Electronic Colloquium on Computational Complexity, Report No. 65 (2000)

Uniform Circuits for Division: Consequences and Problems

Eric Allender[*] David A. Mix Barrington[†]

September 7, 2000

Abstract

The essential idea in the fast parallel computation of division and related problems is that of Chinese remainder representation (CRR) – storing a number in the form of its residues modulo many small primes. Integer division provides one of the few natural examples of problems for which all currently-known constructions of efficient circuits rely on some sort of extra information or *non-uniformity*; the major stumbling block has seemed to be the difficulty of converting from CRR to binary. We give new bounds on the nonuniformity required for division; it is necessary and sufficient to be able to compute discrete logarithms modulo an $O(\log n)$ bit number. In particular, we show that the necessary uniformity predicates lie in a class that (provably) does not contain L.

The fact that CRR operations can be carried out in log space has interesting implications for small space classes. We define two versions of $s(n)$ space for $s(n) = o(\log n)$: dspace($s(n)$) as the traditional version where the worktape begins blank, and DSPACE($s(n)$) where the space bound is established by endmarkers before the computation starts. We present a new translational lemma, and derive as a consequence that (for example), if one can improve the result of [14] that $\{0^n : n \text{ is prime}\} \notin \text{dspace}(\log\log n)$ to show that $\{0^n : n \text{ is prime}\} \notin \text{DSPACE}(\log\log n)$, it would follow that L $\neq$ NP.

1 Introduction

The exact complexity of division, powering, and iterated multiplication of integers has been a major open problem since Beame, Cook, and Hoover [7]

[*]Rutgers University, `allender@cs.rutgers.edu`

[†]U. of Massachusetts, `barring@cs.umass.edu`

ISSN 1433-8092

showed these problems to be in P-uniform TC^0 in 1986[1]. (TC^0 is the set of problems solvable by threshold circuits of constant depth and polynomial size, "P-uniform" means that these circuits can be constructed by a poly-time Turing machine.) In a recent breakthrough, Chiu, Davida and Litow [9] showed these problems to be in L-uniform TC^0, where the circuits can be constructed in log space. They thus also solved an even older open problem by showing these problems to be solvable in log space itself.

Here we examine two implications of this new result and some of the new problems that it suggests. First, what are the prospects for improving the result to place these problems in the most natural version of "uniform TC^0", that of log-time uniform circuits or problems definable by first-order formulas with MAJORITY quantifiers? We show that that in Immerman's descriptive complexity setting [18], the new construction leads to first-order formulas with MAJORITY quantifiers and only a single extra numerical predicate, for calculating powers modulo a number of $O(\log n)$ bits. (Thus the threshold circuits are uniform if only the values of this predicate are provided.) But, we show, this new predicate lies in a class that *provably* does not contain L, strongly suggesting that the full power of L-uniform TC^0 is not needed to solve these problems.

We then consider the implications of the new log space algorithms for the study of small-space complexity classes. Most prior work on Turing machines with $O(\log\log n)$ space, for example, has assumed that the work tape starts out blank, with no marker to indicate the end of the available space. We call this class dspace$(\log\log n)$, in contrast to the class DSPACE$(\log\log n)$ where this initial marker is given.

Lower bound results for the DSPACE classes exist – for example the set of palindromes is not in DSPACE$(o(\log n))$.

The space-efficient CRR algorithms allow us to prove more efficient translational arguments, showing that the *unary* languages in DSPACE$(\log\log n)$ are simply the unary encodings of the languages in log space. This highlights the difference between dspace and DSPACE classes. For example, a classic result of Hartmanis and Berman [14] says that the set of unary strings of prime length is not in $\mathit{dspace}(o(\log n))$. The new translational lemma shows that proving an analogous result for DSPACE$(\log\log n)$ would separate the classes L and NP.

In Section 2 we review the history and context of these numeric problems.

[1][7] claimed only P-uniform NC^1, but it was observed later in [22] that their algorithm is implementable in TC^0.

In Section 3 we outline the new proof of Chiu, Davida, and Litow [9] that the necessary CRR operations for division can be carried out in log space, and we show that this construction is uniform, given a predicate for discrete logs of small numbers. In Section 4 we explain the possible consequences of this fact for improving the uniformity of division circuits. Translational lemmas for small space-bounded classes are presented in Section 5.

2 Circuits for Division: An Overview

We are concerned with the complexity of three basic problems in integer arithmetic (with input and output in binary representation):

- Division: Given a number X of n bits and a number Y of at most n bits, find $\lfloor X/Y \rfloor$,
- Powering: Given a number X of n bits and a number k of $O(\log n)$ bits, find X^k, and
- Iterated Multiplication: Given n numbers $X_1, \ldots, X_n$, each of at most n bits, find the product $X_1 X_2 \ldots X_n$.

Beame, Cook, and Hoover [7] showed that each of these problems can be solved by a family of *threshold circuits* of constant depth and polynomial size. A threshold circuit is made up of (unweighted) *threshold gates*, which add up their Boolean inputs and output a Boolean that is true iff the sum exceeds a certain threshold, a parameter of the gate. As always in circuit complexity, a key property of the circuit family is its degree of *uniformity.* The circuits of [7] are P-uniform, in that they can be constructed in time polynomial in n. Thus they showed that these problems are in the class *P-uniform* TC^0.

Recently Chiu, Davida, and Litow [9] have dramatically improved this result by constructing circuit families for these problems that are constructible in *log space* (hence putting the problems in *L-uniform* TC^0). A logspace machine can simulate a constant-depth threshold circuit if it can construct it, so this result also solves the longstanding open problem of putting these problems into the class L, deterministic log space.

Here we present the argument of [9], paying close attention to the use of non-uniformity in the circuits. To do this we use the formalism of *descriptive complexity* developed by Immerman [5, 4, 18], where constant-depth circuit families are presented in the form of *first-order formulas.* For example, the complexity class FO (or log-time uniform AC^0 [5]) consists of those languages that can be described by first-order formulas where the variables range over the positions in the input string, there are atomic formulas for equality, order, addition, and multiplication of these input positions (as well as the contents of the input at each position), and first-order quantifiers $\exists$ and $\forall$ range over the input positions. The larger class FOM ("first-order with MAJORITY") is the same except that MAJORITY *quantifiers* may be used in the formulas — this class is equal in power to log-time uniform families of threshold circuits of constant depth and polynomial size. In FOM one can multiply two n-bit numbers, add together n n-bit numbers, and of course carry out all operations in FO.

Allowing the family of threshold circuits to become less uniform corresponds in the descriptive complexity model to adding new *numerical predicates* to the first-order-with-majority formula. A numerical predicate is one that depends only on the numbers of the input positions that form its arguments, rather than on the contents of the input. For example, with the numerical predicate $T(i, j)$, meaning "the i-th Turing machine halts on input j", one can easily construct a first-order formula representing a nonrecursive language. This language would be recognized by a family of constant-depth, polynomial-size circuits, but these circuits themselves would be a nonrecursive set. As we restrict ourselves to additional predicates that are more computable, the corresponding circuit families become more uniform. For example, a language is in P-uniform TC^0 iff it can be described by a first-order-with-majority formula where all the numerical predicates are P-computable, and analogously for L-uniform TC^0. In fact, there is a *single* numeric predicate NUM_P such that P-uniform TC^0 is equal to first-order-with-majority augmented with NUM_P, which we may denote as FOM+NUM_P. For more details, see [1, Theorem 5.1]. A similar construction yields a predicate NUM_L, such that logspace-uniform TC^0 is equal to FOM+NUM_L.

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 of $O(\log n)$ bits. In many cases we can determine each of these residues in parallel, allowing a great savings in depth. Of course, our problems specify that their input and output must be in ordinary binary notation, so we are faced with the problem of converting to and from CRR.

In particular, consider the expressive power of the *power predicate* "$a \equiv b^i \pmod{m}$" where a, b, i and m each have $O(\log n)$ bits. Since the multiplicative group of a prime number is cyclic, we can identify a generator of this group (the least g such that $g^{m-1} \equiv 1 \pmod{m}$ and no smaller power of g is 1) and compute *discrete logarithms* modulo m for each number, if m is prime. More precisely, there is a first-order formula $\mathrm{GEN}(g, m)$ that has POW as a predefined predicate, that is true if and only if g is the least generator of the multiplicative group mod m. Thus finding a generator can be accomplished in the complexity class FO+POW. We will also call this a FO *reduction* to POW from the problem of finding generators. (Note that FO reductions are equivalent to uniform AC^0-Turing reductions, as considered in [28], where oracle gates are used, instead of predefined predicate symbols. In particular, note that our use of the term "FO reduction" in this paper is more general than in [18], where this term is used to denote AC^0 many-one reducibility.) Similarly, it is easy to see that computing discrete logs mod m can be performed in $\mathrm{FO} + \mathrm{POW}$.

Finally, note that *if the input and output are in CRR*, the iterated multiplication problem simply reduces to the iterated addition problem (by adding the discrete logs), showing that iterated multiplication (in CRR) is in FOM once the power predicate is present (i.e., it is in the class FOM+POW).

This construction was used in [7], but additional work is required in order to compute these functions in binary, instead of CRR. In order to convert into and out of CRR, Beame, Cook and Hoover needed an additional predicate: the binary representation of the product of the first n^3 primes. While the power predicate is easily seen to be computable in logspace, this prime-product predicate was not known to be so computable. The central contribution of [9] is to develop better methods for working with CRR, so that the prime-product predicate is no longer needed. We will show below that their construction is entirely in $\mathrm{FOM} + \mathrm{POW}$. Among other things this shows that the power predicate, the essential ingredient in converting a binary number *into* CRR, is powerful enough (along with FOM operations) to get a number *out of* CRR into binary. In Section 4 we will discuss the exact complexity of the power predicate and the prospects for placing DIVISION in FOM itself.

3 The Proof of Chiu, Davida, and Litow

We will refer to numbers with polynomially many bits as *long numbers*, and denote them by capital letters. Numbers of $O(\log n)$ bits will be called *short*, and denoted by small letters. We are given two long numbers X and Y and asked to find $Z = \lfloor X/Y \rfloor$. Note that it suffices to find a small number of candidates for Z, as in FOM we can compute ZY for any candidate and then verify that $X - ZY$ is non-negative and less than Y.

To fix notation, we now recapitulate the development of CRR. If we are given a sequence of distinct primes[2] $m_1, \ldots, m_k$, each a short number, let M be their product. Any number $X < M$ can be represented uniquely as $(x_1, \ldots, x_k)$ with $X \equiv x_i \pmod{m_i}$ for all i. For each number i, let C_i be the product of all the m_j's except m_i, let h_i be the inverse of C_i modulo m_i, and let D_i be $h_i C_i$. It is easy to verify that X is congruent modulo M to $\sum_{i=1}^{k} x_i D_i$. In fact X is *equal*, as an integer, to $(\sum_{i=1}^{k} x_i D_i) - rM$ for some particular number r, called the *rank* of X with respect to M. Note that r is a short number. It is roughly equal to the sum of the k rational numbers $x_i D_i / M$ or $x_i h_i / m_i$, each of which is between 0 and m_i.

Lemma 3.1 *If $X, m_1, \ldots, m_k$ are each given in binary and $X < M$, we can compute $(x_1, \ldots, x_k)$ (the CRR_M form of X) in $\mathrm{FOM} + \mathrm{POW}$.*

Proof. For each modulus m_i and each $j < n$ we must calculate $2^j \pmod{m_i}$ (given by the power predicate), add the results (using iterated addition in FOM), and take the result modulo m_i (in FO). ∎

Lemma 3.2 *[11, 20] The rank of X with respect to M is computable in $\mathrm{FOM} + \mathrm{POW}$.*

Proof. (This result was first shown by Davida and Litow [11], but here we sketch an easier argument by Macarie [20].) Note that by Lemma 3.1 we can assume that X is given in CRR_M form. If we approximate each of the numbers $x_i D_i / M$ to $O(\log n)$ bits of accuracy and then add the approximations (using iterated addition, in FOM), we get the right answer *unless* r is very close to an integer. If it is, we need to know whether $\sum_{i=i}^{k} x_i D_i$ is just above or just below a multiple of M. But as shown in [20],

[2]Much of the development of CRR can be carried out equally well if the moduli are *powers* of distinct primes. In our argument the principal CRR modulus M will always be a product of distinct primes, though other CRR moduli may not be.

if this sum is close to a multiple of M we can replace this problem with an equivalent one, of whether another sum is just above or just below a multiple of M/m_k. This problem is then either solved by approximation, or reduced to an equivalent problem for a multiple of $M/m_{k-1}m_k$, and so on until the problem is solved at some level or *all* the primes are eliminated. The various levels of the recursion can be carried out simultaneously in parallel, since they do not depend on the answers at previous levels. Also note that the power predicate is used in this construction, both to convert X to CRR_M and to calculate the numbers D_i. ■

Lemma 3.3 *If p is a short prime power, then the iterated product problem modulo p is in* $\mathrm{FOM} + \mathrm{POW}$*.*

Proof. We have seen that if p is *prime*, we find a generator, take discrete logs of the factors, add them together, and raise the generator to the result. But in fact this strategy can be extended to work for all prime powers. If $p = q^e$ with q a prime, in FO we can find the prime factorization of p and hence compute q and e. The multiplicative group of the ring of integers modulo q^e is well-known to be cyclic *unless* $q = 2$ (and thus we compute iterated product as in the prime case). But if $q = 2$ this multiplicative group has two generators, one of which has order 2, so one can essentially take discrete logs here as well. ■

Lemma 3.4 *If X is given in* CRR_M*, and p is a short prime power, then we can compute X modulo p in* $\mathrm{FOM} + \mathrm{POW}$*.*

Proof. We have a representation of the integer X as $(\sum_{i=1}^{k} x_i D_i) - rM$. We can calculate each of the terms $x_i D_i$ modulo p, and M modulo p, using iterated product modulo p. Since we can also calculate the rank in $\mathrm{FOM} + \mathrm{POW}$, we are done. ■

Remark: Although it will not be needed later, we note that the previous two lemmas hold where p is *any* short number, not merely a prime power. If we know the residue of a number modulo each of the prime-power factors of p, we can compute the residue modulo p by simply guessing it (in FO) and verifying that it has the correct residue for each factor.

Proposition 3.5 *Let $b_1, \ldots, b_\ell$ be short powers of distinct 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* $\mathrm{FOM} + \mathrm{POW}$*.*

Proof. For each b_i, we compute X modulo b_i and thus get X into CRR_B form. (Recall that this is possible even though the b_i may be short prime powers rather than short primes.) There is a unique number E, equal to the residue of X modulo B, that is less than B and has this CRR_B form. Using the algorithm of Lemma 3.4, we can now put E into CRR_M form, by finding its residue modulo each of the numbers m_i. Then we can calculate $X - E$, a multiple of B, in CRR_M form.

It is easy to calculate B in CRR_M form. Let N be the product of those prime factors $n_1, n_2, \ldots, n_{l'}$ of M that do not divide B. Let $(y_1, y_2, \ldots, y_{l'})$ be the CRR_N form of B. Given B in CRR_M form it is easy in FOM to compute the lists $(n_1, n_2, \ldots, n_{l'})$ and $(y_1, y_2, \ldots, y_{l'})$, since sorting is in FOM. B^{-1} modulo N is easy to compute in CRR_N, by computing y_i^{-1} mod n_i. The number we want, $\lfloor X/B \rfloor$, is exactly B^{-1} times $X - E$. We can compute this number in CRR_N form, getting the right answer because $\lfloor X/B \rfloor < N$ (as $X/B < M/B \leq N$). To get the remaining components of the CRR_M form of $\lfloor X/B \rfloor$, we need only apply Lemma 3.4 for each prime factor of M that divides B. ∎

Proposition 3.6 *If Y is any long number given in CRR_M, we can find (in FOM+POW) a number D (also in CRR_M) such that (a) D is the product of short powers of distinct primes and (b) the rational number Y/D is between $1/2$ and 1.*

Proof. Clearly such a D exists with $Y \leq D \leq 2Y$, because given any sequence of short odd primes we can multiply them together one at a time until the product exceeds Y, remove the last prime, and then multiply by 2 until the result exceeds Y. To do this effectively, however, we need to be able to take two numbers in CRR_M and determine which is bigger, which can be done in FOM + POW by Lemma 3.7 and its corollary. ∎

Lemma 3.7 *Let $A < M$ be a number given in CRR_M form. In FOM + POW, we can compute whether $A < M/2$.*

Proof. Write A as $(\sum_{i=1}^{k} a_i D_i) - rM$ where r is the rank of A with respect to M. So $2A$ is $(\sum_{i=1}^{k} (2a_i) D_i) - 2rM$. Define c_i for each i so that $c_i \equiv 2a_i \pmod{m_i}$. The vector $(c_1, \ldots, c_k)$ is the CRR_M form of a unique number $C < M$, which is either $2A$ (if $A < M/2$) or $2A - M$ (otherwise). We know that each c_i is either $2a_i$ or $2a_i - m_i$, so define tM to be the sum of $m_i D_i$ for all those i where we must subtract off m_i (note that t is a short number and, by Lemma 3.4, t can be computed in FOM + POW by computing it modulo a short power of 2). Thus $2A$ is equal to $(\sum_{i=1}^{k} c_i D_i) - 2rM + tM$. But we can calculate the rank s of C, so that C is $(\sum_{i=1}^{k} c_i D_i) - sM$. So now $2A = C$ iff $2r - t = s$, an easily testable condition. ■

Corollary 3.8 *[11, 12] Let X and Y be numbers less than M given in CRR$_M$ form. In* FOM + POW *we can determine if $X < Y$.*

Proof. First compare X and Y with $M/2$. If one is less than $M/2$ and one is greater, then it is easy to determine if $X < Y$. In the remaining case where either both or neither are less than $M/2$, it is easy to see that $X < Y$ if and only if $Y - X$ is less than $M/2$. ■

Theorem 3.9 *[9]* Division, *with input and output in binary, is in* FOM+ POW.

Proof. Given X and Y, choose D as above and let u be the rational number $1 - Y/D$, so that $u < 1/2$ and D/Y is $1/(1-u) = \sum_{j=0}^{\infty} u^j$. Our desired number $\lfloor X/Y \rfloor$ is the floor of $(X/D)(D/Y)$ or X/D times $\sum_{j=0}^{\infty} u^j$.

We will find numbers N and A such that N is easy to compute, A is a product of distinct short odd primes (not including those dividing D), and the rational number N/A is within $\epsilon < 2^{-2n}$ of $1/(1-u)$. Then because DA is a product of short odd prime powers, we can compute the floor of XN/DA exactly. This number differs from the floor of X/Y by at most $\epsilon X/D$, which is less than 1, so we have our two candidates for the floor of X/Y.

To do this we create numbers $A_1, \ldots, A_{2n}$, each a product of polynomially many short primes and each at least $2n$ bits long. Our number A will be the product of all the A_i's. For each i we pick a number t_i such that t_i/A_i is within 2^{-2n} of u. This is done by taking t_i to be the floor of $(D-Y)A_i/D$, in CRR$_M$, which we can compute because D is a product of short prime powers.

Then for each number $j < 2n$ we can calculate the product for all $i < j$ of t_i/A_i, with the numerator expressed in CRR$_M$, and note that this rational number is within 2^{-2n} of u^j. Multiplying this by A_i/A_i for all $j \leq i \leq 2n$, we get a product N_j/A which is within 2^{-2n} of u^j, with N_j in CRR$_M$. Summing the N_j for all $j < 2n$ gives a number N in CRR$_M$ such that N/A is very close to $1/(1-u)$, as desired.

Of course this gives us the desired quotient only in CRR$_M$, not in binary. But it is easy, given division of arbitrary n-bit integers with result in CRR, to convert a number from CRR to binary. To get the k-th bit of a number Z that is given to us in CRR, we compute $u = \lfloor Z/2^k \rfloor$ and $v = \lfloor Z/2^{k+1} \rfloor$, and note that the desired bit is $u - 2v$. We get this bit as a CRR number, but it is easy to recognize the CRR forms of the numbers 0 and 1. (By the same token, it is now possible in FOM+POW to convert numbers from any base to another, by first converting to CRR.) ■

4 Consequences

What does this new algorithm finally tell us about the complexity of DIVISION? In one sense the circuit complexity of DIVISION has been well-understood since [7]; DIVISION can be computed by threshold circuits of constant depth and polynomial size, and since MAJORITY is reducible to DIVISION, we cannot hope to put DIVISION into a smaller circuit class.

The remaining question, of course, is *how uniform* the threshold circuits can be made to be. The main result of [9] is that the P-uniform circuits of [7] can be made L-uniform, with the important consequence that DIVISION is in L itself. Our analysis of their algorithm tells us something more, that DIVISION is in the class we have called FOM+POW. We think that a closer analysis of this class will tell us something about the prospects for placing DIVISION within FOM itself, and open up some other interesting complexity questions as well.

Here is a list of problems that have been known since [7] to be in P-uniform TC^0 but are not known to be in FOM:

- DIVISION
- POW
- ITERATED MULTIPLICATION
- POWERING
- CONVERTING CRR TO BINARY[3]
- CONVERTING BINARY TO CRR
- DIVISIBILITY (i.e., given X and Y, does X divide Y?)

[3]To be completely formal, the statement of this problem should include a specification of the moduli used in CRR. For the purposes of this paper any reasonable definition is sufficient, and hence we leave this unspecified.

All of these problems are in FOM + POW. For some of these problems, this is optimal, as the following observations show.

Proposition 4.1 Division *is complete for* FOM + POW *under* FO *reductions.*

Proof. Note first that Beame, Cook, and Hoover presented a FO reduction from Powering to Division in [7]. Thus, to solve the POW predicate $a \equiv b^i \pmod{m}$ we can follow the reduction of [7] to use Division to compute b^i, and then divide b^i by m and compare the answer to a. Similarly, it is well-known that Majority is FO reducible to Division; since Majority is reducible to multiplication [8] and Powering is reducible to to Division [7], it suffices to reduce multiplication to Powering. But this is easy, since $XY = [(X+Y)^2 - X^2 - Y^2]/2$. ■

Although the other problems are not known to be hard under FO reductions, some of them are complete under FOM reductions.

Proposition 4.2 POW*,* Iterated Multiplication*, and* Powering *are complete for* FOM + POW *under* FOM *reductions.*

Proof. For POW this is a trivial observation. Since all of these problems are in FOM + POW, they are all FO-reducible to Division. An argument in [7] reducing Division to Powering is easily seen to provide a FOM reduction. And, of course, Powering is a special case of Iterated Multiplication. ■

It is interesting to note that many number theorists conjecture that 2 is a generator of the multiplicative group mod p for a constant fraction of all m-bit primes [16, 17]. If this conjecture is true, then most of the argument of Section 3 can be carried out using only primes of this sort. (There are some minor technical modifications that need to be made to the argument in Section 3 in order to completely do away with the use of prime powers and the use of primes for which 2 is not a generator. We suppress those details here.) Thus, if the conjecture is true, it follows that Converting Binary to CRR is also complete for FOM + POW, since testing if 2 is a generator can be checked in FO if one is able to convert 2^i to CRR, and in the same way discrete logs can be found modulo such primes p. Note also that Converting Binary to CRR is FO-reducible to Divisibility, since determining if X is equivalent to a mod p is equivalent to checking if p divides $X - a$. Thus we conjecture that both of these problems are complete for FOM + POW under FOM-reducibility.

There is some irony in the fact that we do not know how to show that CONVERTING CRR TO BINARY is hard for FOM + POW under FOM reductions, although in some sense it was precisely this problem that was the source of the P-uniformity in the algorithm of [7].

The class FOM+POW clearly lies somewhere between FOM itself and L-uniform TC^0. Is the full power of L-uniform TC^0 needed to perform integer division? We cannot answer this question definitively, since for all we know FOM and L could be identical, but we can give significant evidence that it is *not*. To do this we will need to take a closer look at the complexity of the power predicate.

A recent paper of Barrington, Kadau, Lange, and McKenzie [6] looked at groups presented as multiplication tables and the complexity of various problems including that of computing powers. These results apply directly to the group of integers modulo m (where m is polynomial in n) because the product operation of this group is FO computable. They showed that powering in such a group, and thus our power predicate, is in a new complexity class they called FOLL.

In [6] the class FOLL is defined to be those languages defined by first-order formulas with a quantifier block iterated $O(\log \log n)$ times, or equivalently languages recognized by uniform circuit families (of AND and OR gates) of depth $O(\log \log n)$, polynomial size, and unbounded fan-in. FOLL clearly contains FO and is contained in uniform AC^1, and both containments are proper, but little else is known about FOLL. For example, is it contained in L, NL, or SAC^1?

We do at least know, by well-known lower bounds on circuit size and depth (e.g.,[25]), that the parity language is *not* in FOLL. Since FOLL is closed under FO reductions, it follows that *no language* in FOLL can be complete under such reductions for any class including parity, in particular for L, NC^1, or FOM.

The power predicate is shown to be in FOLL in [6], as a special case of computing powers in any group given by its multiplication table. The key step in this computation is to note that a^{jk}, for example, is *FO* computable from the complete table of j-th and k-th powers, since $a^{jk} = b$ iff $\exists c : (a^j = c) \wedge (c^k = b)$. Thus each round of FO computation *squares* the highest power computed, and (with some other clauses in the definition) after $O(\log \log n)$ rounds all powers polynomial in n can be computed.

It is also useful to consider a graph-theoretical approach to these problems. Define REACH($\log n$) to be the problem, given a directed graph G with n vertices, and given two vertices s and t, of determining if there is a path from s to t with length at most $\log n$. Similarly, define $\mathrm{REACH}_1(\log n)$ to be the same problem restricted to graphs with outdegree 1. The problems $\mathrm{REACH}(\log n)$ and $\mathrm{REACH}_1(\log n)$ may be viewed as "scaled-down" versions of the standard complete problems for NL and L, respectively. The relevance of these problems is illustrated by the following inclusions:

$$\mathrm{FO} + \mathrm{POW} \subseteq \mathrm{FO} + \mathrm{REACH}_1(\log n) \subseteq \mathrm{FO} + \mathrm{REACH}(\log n) \subseteq \mathrm{FOLL}$$

The first of these inclusions can be seen by considering a directed graph where the nodes are the elements of the multiplicative group mod m, we fix an element a, and every node g has an edge to the node ga and the node g^2. By the familiar repeated squaring algorithm, every power a^i of a is at the end of a path of length $O(\log i)$ from the identity node, where each choice of squaring or multiplying by a is given by one of the bits of i. To calculate a^i, it suffices to *follow* a particular path of length $O(\log n)$ in a graph. It is easy to modify this construction and build a graph of outdegree one (by noticing that the graph is leveled, and deleting one edge leaving each node). The second inclusion is trivial, and the third inclusion follows from the familiar Savitch algorithm.

We think it is reasonably likely that DIVISION is in FOM, the fully uniform version of TC^0. We have seen that this happens if and only if $\mathrm{POW} \in \mathrm{FOM}$. In particular, the full power of L-uniformity (i.e., the numeric predicate NUM_L) does not seem to be needed, since POW and $\mathrm{REACH}(\log n)$ lie in FOLL, and in particular $\mathrm{REACH}_1(\log)$ is an "exponentially-easier" special case of the standard complete problem for L.

Is DIVISION in uniform NC^1? Here we refer to the most robust definition of "uniform NC^1", that of Ruzzo [23], in terms of the "extended connection language" of a circuit. According to this notion of uniformity, uniform $\mathrm{NC}^1 = \mathrm{ATIME}(\log n)$. Again, DIVISION lies in uniform NC^1 if and only if $\mathrm{POW} \in \mathrm{ATIME}(\log n)$.

One attack on this problem would be to try to show $\mathrm{REACH}_1(\log n) \in \mathrm{ATIME}(\log n)$. A positive answer to this question would resolve *two* major open questions about uniform NC^1: it would include DIVISION and we could redefine it in terms of the simpler "direct connection language" being in DLOGTIME rather than the extended connection language. (That is, the two most natural definitions for uniform NC^1 proposed in [23] would coincide.)

It is conceivable that $\mathrm{REACH}_1(\log n)$ is in FO. This hypothesis implies that L can be simulated by unbounded fan-in circuits of depth $O(\log n/(\log\log n))$, rather than the depth $O(\log n)$ given by Savitch's theorem. Similarly, showing that $\text{REACH}_1(\log n) \in \text{FOM}$ would yield threshold circuits of depth $O(\log n/(\log\log n))$ for every problem in L. Note that NC^1 does have unbounded fan-in circuits of depth $O(\log n/(\log\log n))$ [8].

5 Small space-bounded complexity classes

For many people working in computational complexity theory, space-bounded computation only "begins" with logarithmic space. To be sure, there is a large literature with dealing with space bounds between $\log\log n$ and $\log n$. (For example, see [19] for a perspective on the sequence of difficult papers leading up to a separation of the bounded-alternation hierarchy for sublogarithmic-space-bounded machines.) Nonetheless, this work relies on the automata-theoretic limitations of the small-space-bounded machine. For instance, if $s(n) = o(\log n)$ is a fully-space-constructible function, then there is a constant k such that, for infinitely many n, $s(n) < k$. This provides easy proofs of lower bounds for the space complexity of many languages, such as the proof in [14] that the set $\{0^n : n \text{ is prime}\}$ cannot be accepted in space $o(\log n)$.

However, it is still an open question whether the set of (binary encodings of) primes can be accepted in space $o(\log n)$. How can this be? Surely the binary encoding of a set cannot be easier than the unary encoding of the same set!

Let us see why this is still an open question. 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, using a standard *translation lemma*, such as:

Lemma 5.1 (Traditional Translation Lemma) *If $s(\log n) = \Omega(\log\log n)$ is fully space-constructible, then the first statement below implies the second:*

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

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 $\text{un}(A)$. As an example to see that this is unavoidable, consider the regular set $A = 10^*$. Arguing as in [14] it is easy to see that $\mathrm{un}(A) = \{0^{2^k} : k \in \mathbb{N}\}$ is not in $\mathrm{dspace}(o(\log n))$. (Every infinite unary language in $\mathrm{dspace}(o(\log n))$ has an infinite regular subset; $\mathrm{un}(A)$ does not.)

There is another reasonable way to define space complexity classes. Let $\mathrm{DSPACE}(s(n))$ 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 $\mathrm{dspace}(s(n))$ 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, $\mathrm{DSPACE}(s(n))$ is simply $\mathrm{dspace}(s(n))$ augmented by a small amount of "advice", allowing the machine to compute the space bound. (This model was defined under the name "DEMON-SPACE" by Hartmanis and Ranjan [15]. See also Szepietowski's book [27] on sublogarithmic space.)

$\mathrm{DSPACE}(s(n))$ seems at first glance to share many of the properties of $\mathrm{dspace}(s(n))$. In particular, it is still relatively straightforward to show that there are natural problems, such as the set of palindromes, that are not in $\mathrm{DSPACE}(o(\log n))$. (This follows from a simple crossing-sequence and Kolmogorov-complexity argument [15].)

The main contribution of this section is an easy argument, showing that the efficient division algorithm of [9] provides a new translation lemma.

Lemma 5.2 New translation lemma *Let $s(n) = \Omega(\log n)$ be fully space-constructible. Then the following are equivalent:*

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

Proof. For the forward direction, it is sufficient to present a small-space algorithm for $\mathrm{un}(A)$.

Note that $\log\log n$ space can hold the binary representation of a short prime p. Thus on input 0^n, a $\mathrm{DSPACE}(\log\log n)$ machine can compute the pieces of the Chinese Remainder Representation of n.

Thus, by [9], in space $\log(|n|) = \log\log n$ we can compute the bits of the binary representation of n. Thus, on input 0^n a Turing machine can simulate a $s(|n|)$-space-bounded computation (of a machine having input n) in space $s(\log n)$.

For the converse, given a Turing machine accepting $\mathrm{un}(A)$ in space $\log\log(x) + s(\log x)$ on input 0^x, we want to use $\log(|x|) + s(|x|) = O(s(|x|))$ space to determine if $x \in A$. We provide merely a sketch here.

The most naïve approach to carry out this simulation will not work, since we do not have enough space to record the location of the input head in a simulated computation on 0^x, and thus we cannot perform a step-by-step simulation. However, we do have enough space to carry out a simulation until either

(a) the input head returns to an endmarker without repeating a worktape configuration, or

(b) some worktape configuration is repeated.

In case (a), a step-by-step simulation is sufficient. In case (b), we can determine the period of the loop, and (doing some simple arithmetic) we can determine the state the machine will be in when it encounters the other end marker.

Thus in either case, the simulation can proceed. ■

Corollary 5.3 *Let $\mathcal{C}$ be any complexity class. In order to show $\mathcal{C}$ is not contained in* L, *it suffices to present a set $A \in \mathcal{C}$ such that* $\mathrm{un}(A) \notin \mathrm{DSPACE}(\log\log n)$.

We remark that the argument above can easily be adapted to show that the unary languages in $\mathrm{NSPACE}(\log\log n + \log(s(n)))$ are exactly the unary encodings of languages in $\mathrm{NSPACE}(s(n))$. It should be remarked that a different translational method was presented by Szepietowski [26] for relating the L = NL question to the $\mathrm{dspace}(\log\log n) = \mathrm{nspace}(\log\log n)$ question. However, as we have seen, there is no direct analog to Corollary 5.3 for the dspace or nspace classes.

In fact, it is not very difficult to show that there are unary languages in P (and even in $\mathrm{dspace}((\log\log n)^2)$) that are not in $\mathrm{DSPACE}(\log\log n)$. A straightforward delayed diagonalization (as in [14]) can be used to construct such a set $A \subseteq 0^*$. Note that this does not prove P $\neq$ L, since $\mathrm{un}(A)$ (a very sparse set) *is* in $\mathrm{DSPACE}(\log\log n)$. Stating this another way, the unary set $A \in \mathrm{dspace}((\log\log n)^2)$ is equal to $\mathrm{un}(B)$ for some $B \in \mathrm{dspace}((\log n)^2)$, where B is not known to be in P.

Observe that all unary languages in $\mathrm{NSPACE}(\log\log n)$ are in FO. This follows since if B is a unary language in $\mathrm{NSPACE}(\log\log n)$, then $B = \mathrm{un}(A)$ for some $A \in$ NL. Thus, by [21] (see also [13]), $A \in \mathrm{RUD} = \bigcup_k \Sigma_k\mathrm{TIME}(n)$. It was observed in [2] that $B = \mathrm{un}(A) \in$ FO if and only if $A \in$ RUD.

In some ways, DSPACE($\log\log n$) is a more natural class than dspace($\log\log n$), in the sense that this class is related to a natural class of branching programs, whereas no similar characterization is known for dspace($\log\log n$). The following definitions make this precise.

For this extended abstract, we assume the reader is familiar with basic definitions regarding branching programs. A branching program is *leveled* if the vertex set can be partitioned into *columns*, where all edges from vertices column i go to vertices in column $i+1$. We need *not* assume that all vertices in a given column query the same input location. We assume that vertices are labeled by a pair (c, j) where c is the number of the column, and j is the index of the node within column c. The *width* of a branching program is the maximum number of vertices in any column. In this paper, we consider only deterministic branching programs though parallel results on NSPACE classes and nondeterministic branching programs (or "contact schemes") can be obtained by the same techniques.

Theorem 5.4 *A is accepted by log-time-uniform branching programs of polynomial size and width $O(\log^{O(1)} n)$ if and only if A is FO-reducible to a language accepted by an oblivious* DSPACE($\log\log n$) *machine.*

Proof. First, consider a language accepted by an oblivious machine M with a worktape of size $O(\log\log n)$. By definition of "oblivious", the input location scanned by M at time t can be computed in FO. Thus it is an easy matter to construct a branching program with a node for each worktape configuration on each level, with edges simulating M's transition function. The resulting branching program will be FO-uniform, and this can be transformed into an equivalent log-time uniform branching program by standard techniques.

Conversely, let A be accepted by a log-time uniform leveled branching program of width $\log^{O(1)} n$. It is easy to show that there is a *FO* reduction that, given an input string x, produces a sequence of the form

$$\#\#f_1\#f_2\#\ldots\#f_t\#\#$$

where t is the number of columns, and each f_i is a function $f_i : \{1, \ldots w\} \to \{1, \ldots w\}$, where $w = \log^{O(1)} n$ is the width of the branching program, with the property that $f_i(j) = j'$ iff the branching program, when in vertex j in column i, moves to vertex j' in column $i+1$ when querying the specified bit of x.

Note that an input x is accepted by M if and only if $f_t(f_{t-1}(\ldots(f_1(1))\ldots))$ is an accepting state of M. We encode each function f in the sequence as a list

$$(1, f(1))(2, f(2))\ldots(w, f(w)).$$

Note that there is an oblivious machine with space bound $O(\log\log n)$ that takes such a sequence of functions as input and computes the composition. ■

Essentially equivalent observations appear elsewhere. For instance, it is shown in [10] that leveled branching programs of width $O(2^{s(n)})$ correspond to non-uniform finite automata with space bound $s(n)$.

We do not know if the restriction to oblivious machines is necessary. If the behavior of a machine's input head is allowed to depend on the input contents, then the machine potentially has access to the $\log n$ bits of memory contained in the input head position. This might allow an otherwise space-bounded machine to solve L-complete problems. For example, the "non-uniform automata" of [3] are oblivious, correspond to constant-width poly-size branching programs and have the power of NC^1. But as shown by Barrington and Immerman (reported in [10]), if the obliviousness restriction is removed, the same machines have the power of general poly-size branching programs or L. But these machines make important use of non-uniformity, in the form of a read-only "program tape". It is not clear whether a DSPACE($\log\log n$) machine, for example, would be able to exploit the input position in the same way.

It is easy to see that unary languages (and even languages in 0^*1^*) in DSPACE($\log\log n$) are accepted by machines whose input heads sweep back and forth across their input. Thus one can also express questions about DSPACE($\log\log n$) in terms of the "sweeping automata" of [24]. For instance, if one can show that the set $\{0^n1^m : n \text{ is prime}\}$ is not accepted by a sweeping automaton with $\log^{O(1)} n$ states, then $\mathrm{L} \neq \mathrm{NP}$.

6 Acknowledgments

Eric Allender was supported by NSF grant CCR-9734918. Much of this work was carried out during the March 2000 McGill Invitational Workshop on Complexity Theory – the authors thank the organizer Denis Thérien and all the other participants. We also thank Dieter van Melkebeek, Samir Datta, Michal Koucky, Rüdiger Reischuk, and Sambuddha Roy for helpful discussions.

Additional work on this project was carried out during the Park City Mathematics Institute's summer program in July and August 2000, supported by the Clay Mathematics Institute. The authors thank PCMI, CMI, Alexis Maciel, and the students in the PCMI undergraduate where this material was presented.

References

[1] E. Allender. P-uniform circuit complexity. *J. ACM* **36**:912–928, 1989.

[2] E. Allender and V. Gore. On strong separations from AC^0. In *Advances in Computational Complexity Theory*, Jin-Yi Cai, ed., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 13, AMS Press, 1993, pp. 21–37.

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

[4] D. A. M. Barrington and N. Immerman. Time, hardware, and uniformity. In *Complexity Theory Retrospective II*, L. A. Hemaspaandra and A. L. Selman, eds., Springer-Verlag, 1997, pp. 1–22.

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

[6] D. A. M. Barrington, P. Kadau, K.-J. Lange, and P. McKenzie. On the complexity of some problems on groups given as multiplication tables. *Proc. 15th IEEE Conference on Computational Complexity*, 2000, pp. 62–69.

[7] P. Beame, S. Cook and J. Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, **15**:994–1003, 1986.

[8] A. K. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM J. Comput.*, **13**:423–439, 1984.

[9] A. Chiu, G. Davida, and B. Litow. NC^1 Division. Preliminary version. Available at `http://www.cs.jcu.au/~bruce/papers/crr00.ps.gz`.

[10] C. Damm and M. Holzer. Inductive Counting for Width-Restricted Branching Programs. *Information and Computation* **130**:91–99, 1996.

[11] G. I. Davida and B. Litow. Fast parallel arithmetic via modular representation. *SIAM J. Comput.*, **20**:756–765, 1991.

[12] 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.

[13] L. Fortnow Time-space tradeoffs for satisfiability. *Journal of Computer and System Sciences* **60**:336–353, 2000.

[14] J. Hartmanis and L. Berman. On tape bounds for single letter alphabet language processing. *Theoretical Computer Science* **3**:213–224, 1976.

[15] J. Hartmanis and D. Ranjan. Space bounded computations: Review and new speculation. In *MFCS '89: Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 379, Springer-Verlag, 1989, pp. 49–66.

[16] D. R. Heath-Brown. Artin's conjecture for primitive roots. *Quart. J. Math. Oxford (2)* **37**:27–38, 1986.

[17] C. Hooley. *Applications of Sieve Methods to the Theory of Numbers.* Cambridge Tracts in Mathematics No. 70, 1970.

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

[19] M. Liśkiewicz and R. Reischuk. Computing with sublogarithmic space. In *Complexity Theory Retrospective II*, L. A. Hemaspaandra and A. L. Selman, eds., Springer-Verlag, 1997, pp. 197–224.

[20] I. Macarie. Space-efficient deterministic simulation of probabilistic automata. *SIAM J. Comp.* **27**:448-465, 1998.

[21] V.A. Nepomnjaščiĭ. Rudimentary predicates and Turing calculations. *Soviet Math. Dokl.* **11**:1462–1465, 1970.

[22] J. Reif and S. Tate. On threshold circuits and polynomial computation. *SIAM J. Comput.*, **21**:896–908, 1992.

[23] W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 21:365–383, 1981.

[24] M. Sipser. Lower bounds on the size of sweeping automata. *Journal of Computer and System Sciences* **21**:195–202, 1980.

[25] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. *Proc. 19th ACM Symposium on Theory of Computing (STOC)*, 1987, pp. 77–82.

[26] A. Szepietowski. If deterministic and nondeterministic space complexities are equal for $\log\log n$, then they are also equal for $\log n$. *Theoretical Computer Science*, **74**:115–119, 1990.

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

[28] C. B. Wilson. Decomposing NC and AC. *SIAM J. Comput.* **19**:384–396, 1990.