CS277: Concrete Complexity **Fall 1996**

Lecture 5-6: September 10, 12, 1996

Lecturer: Uri Zwick *Scribe: Chris Umans*

5.1 The NC^k Complexity Classes

Definition 5.1 *A family of circuits is LOGSPACE-uniform if the description of the n^{th} circuit can be generated by a Turing Machine in $O(\log n)$ space.*

Definition 5.2 *The complexity class NC^k is the class of languages that can be recognized using LOGSPACE-uniform families of finite fan-in circuits with depth $O((\log n)^k)$ and size $n^{O(1)}$.*

Note that for NC^1, the $O(\log n)$ depth requirement automatically imposes the size requirement, but this is not true for $k > 1$.

Definition 5.3 *The complexity class NC (Nick's Class) = $\bigcup_{k\geq 1} NC^k$.*

In parallel computations, NC is the class of languages recognized in polylogarithmic time using a polynomial number of processors. To get an idea of where NC stands in the complexity hierarchy, note that:

$$\mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{NC}^2,$$

where L denotes LOGSPACE, and NL denotes non-deterministic LOGSPACE.

To prove that $\mathrm{NC}^1 \subseteq \mathrm{L}$, we would use Lemma 5.4, for which we sketch a proof below. (See exercise 4 for more on this topic).

Lemma 5.4 *A circuit of depth D can be simulated in $O(D)$ space.*

Proof: We can assume that the circuit is a formula. We then simulate the formula one node at a time in DFS order. At any given time, we only need to remember at most D bits to describe our current position in the tree, and possibly at most D temporary results. ∎

Addition and multiplication are in NC^1; the question of whether division is in NC^1 is an open problem. (We shall see *P-uniform* circuits for division with $O(\log n)$ depth, but it is not known if LOGSPACE-uniform division circuits exist).

5.2 Division in Depth $O(\log n)$

Recall that we saw a construction for performing division with $O(m(n))$ size and $O((\log n)^2)$ depth, where $m(n)$ is the circuit size required to multiply 2 n-bit integers. We would like to reduce the depth to $O(\log n)$. We can do this by reducing division to "powering" and powering to "product", but first we must formalize the notion of NC^1-reducibility.

Definition 5.5 *A problem f is NC^1-reducible to a problem g if there is a LOGSPACE-uniform sequence of circuits of depth $O(\log n)$ for f that use g-gates, where the depth of a g-gate with m inputs is considered to be* $\log m$.

Theorem 5.6 *The following problems are NC^1-equivalent:*

1. *DIVISION: given two n-bit integers, x and y, compute $\lfloor \frac{x}{y} \rfloor$.*
2. *POWERING: given the n-bit integer x, compute $1, x, x^2, x^3, \cdots, x^{n-1}$.*

Proof: (DIVISION $\to$ POWERING) Let x be an n-bit number, and assume x is normalized (i.e. $\frac{1}{2} \leq x < 1$). As in previous lectures, we need only compute the inverse to do division, and it is easy to see that if we let $y = 1 - x$, then:

$$\frac{1}{x} = \frac{1}{1-y}.$$

The power series expansion of this function gives us:

$$\frac{1}{1-y} = 1 + y + y^2 + y^3 + \cdots + y^{n-1} + \cdots,$$

and this series converges, since $y \leq \frac{1}{2}$.

Thus by taking the first $O(n)$ terms we can compute $\frac{1}{1-y} = \frac{1}{x}$ to the desired accuracy. Of course the first $O(n)$ terms are precisely the outputs of POWERING, and so DIVISION is NC^1-reducible to POWERING.

(POWERING $\to$ DIVISION) Let x be a n-bit integer, and define X as follows:

$$X = \underbrace{\boxed{\qquad 1 \qquad}}_{2n^2 \text{ bits}} \underbrace{\boxed{\qquad x \qquad}}_{2n^2 \text{ bits}} \underbrace{\boxed{\qquad x^2 \qquad}}_{2n^2 \text{ bits}} \cdots \underbrace{\boxed{\qquad x^{n-1} \qquad}}_{2n^2 \text{ bits}}$$

The value of X is thus:

$$X = \sum_{i=0}^{n-1} 2^{2n^2(n-i)} x^i = 2^{2n^3} \sum_{i=0}^{n-1} \left(x \cdot 2^{-2n^2} \right)^i.$$

The rightmost summation is simply a geometric series that converges to

$$\frac{1}{1 - x \cdot 2^{-2n^2}}$$

as n goes to infinity. After n terms, however, we can take the floor to get X:

$$X = 2^{2n^3} \sum_{i=0}^{n-1} \left(x \cdot 2^{-2n^2} \right)^i = \left\lfloor \frac{2^{2n^3}}{1 - x \cdot 2^{-2n^2}} \right\rfloor = \left\lfloor \frac{2^{2n^3+2n^2}}{2^{2n^2} - x} \right\rfloor .$$

So given x, we can perform one (large) division and get all of the bits needed to output the result of POWERING(x). ■

Observation 5.7 *POWERING is NC^1-reducible to PRODUCT, the problem of computing*

$$\prod_{i=0}^{n-1} x_i ,$$

where each x_i is a n-bit integer. The reduction is trivial.

Now we show how to compute products in $O(\log n)$ depth, which, by the reductions above, will allow us to do division in $O(\log n)$ depth, our original goal.

This method is due to Beame, Cook, and Hoover. Their approach is to take logs, which transforms the product problem into a summation problem, and then take anti-logs, where all of the log calculations take place in small finite fields. A sketch of the procedure follows:

1. Pick k small primes $p_1, p_2, \cdots, p_k$ such that $\prod_i p_i > 2^{n^2}$.
2. Compute all $x_{ij} = x_i \bmod p_j$.
3. Compute all $Y_j = \prod_{i=1}^{n} x_{ij} \pmod{p_j}$.
4. From $Y_1, Y_2, \cdots, Y_k$, reconstruct $Y = \prod_{i=1}^{n} x_i$.

For the first step, it is enough to take the first n^2 primes, so we choose $k = n^2$. Note that $p_k \approx k^2 \ln k$ (Prime Number Theorem), so each prime number has $O(\log n)$ bits.

For step two, we need to reduce x modulo a small prime p. There are two approaches:

1. For all $0 \le i \le n-1$, prepare the value $2^i \bmod p$, and add up the relevant values for each bit that is 1 in x, or

2. prepare an approximation q of $\frac{1}{p}$. Note that $x \bmod p = x - \lfloor \frac{x}{p} \rfloor p$, and $\lfloor \frac{x}{p} \rfloor \approx \lfloor xq \rfloor$. If we are careful in choosing the accuracy of our approximation of q, then $x - \lfloor xq \rfloor p$ will be exactly $x \bmod p$; otherwise, we may have to add or subtract p for the correct result.

For the third step, we need to compute a product modulo a small prime p. Since p is prime, $\mathcal{Z}_p$ is a cyclic group. This means that there exists an element $g \in \mathcal{Z}_p$ (called a generator), such that all of the elements $1 = g^{p-1}, g, g^2, g^3, \cdots, g^{p-2}$ are distinct, and in fact constitute exactly $\mathcal{Z}_p \setminus \{0\}$. For every $x \in \mathcal{Z}_p \setminus \{0\}$ we let $\mathrm{ind}_g(x) = i$ such that $g^i = x$. This is the index function for groups, but also gives us a discrete logarithm function (base g) in the field $\mathcal{Z}_p$. To calculate the product of $x = g^i$ and $y = g^j$, we need only compute $x \cdot y = g^{i+j}$. Note that if some x_{ij} is zero, it has no log, but then the product is zero, so we can just test for this special case with a few extra gates.

Figure 5.1 illustrates the portion of the circuit that executes steps 1-3.

The final step is to compute Y from the Y_j's. To do this, we choose U_i such that

$$U_i \equiv \begin{cases} 1 & (\mathrm{mod}\ p_i) \\ 0 & (\mathrm{mod}\ p_j, j \neq i) \end{cases}$$

By the Chinese Remainder Theorem, such a U_i exists, and we can precompute it. Finally,

$$Y = \sum_{j=1}^{k} U_j Y_j \quad (\mathrm{mod}\ \prod_j p_j)$$

This completes the Beame, Cook, and Hoover method for computing products in $O(\log n)$ depth circuits. While multiplication and summation are LOGSPACE-uniform, these product circuits are not, because we don't know if we can construct the U_i's LOGSPACE-uniformly.

5.3 Linear Lower Bounds

The highest circuit size lower bound for any function in NP is just linear!

Theorem 5.8 (Trivial Lower Bound) *If $f(x_1, x_2, \cdots, x_n)$ depends on all of its n variables, then $C_\Omega(f) \geq n - 1$, for every $\Omega \subseteq B_2$.*

Definition 5.9 (Threshold Function)

$$T_k^n(x_1, x_2, \cdots, x_n) = \begin{cases} 1 & \textit{iff } \sum_{i=1}^n x_i \geq k \\ 0 & \textit{otherwise} \end{cases}$$

Note that $T_1^n \equiv$ OR, for which the trivial lower bound is tight.

Theorem 5.10 $C_{B_2}(T_2^n) \geq 2n-1$.

Proof: The proof is by induction on n. For $n=2$, $T_2^2 = \wedge$ so $C(T_2^2)=1$. Consider a minimal size circuit for T_2^n. Look at the first gate A in a topological ordering of the circuit's DAG. Its inputs must be variables; call them x_i and x_j. Depending on the 4 possible combinations of values for x_i and x_j, one of three effective circuits is realized:

$$
\begin{aligned}
T_2^n(x_1, x_2, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_{j-1}, 0, x_{j+1}, \cdots, x_n) &= T_2^{n-2} \\
T_2^n(x_1, x_2, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_{j-1}, 1, x_{j+1}, \cdots, x_n) &= T_1^{n-1} \\
T_2^n(x_1, x_2, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_{j-1}, 0, x_{j+1}, \cdots, x_n) &= T_1^{n-1} \\
T_2^n(x_1, x_2, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_{j-1}, 1, x_{j+1}, \cdots, x_n) &= T_0^{n-2}.
\end{aligned}
$$

One of x_i and x_j must feed another gate B; otherwise these three possibilities would be selected from the output of just one gate, which can have only two values (0 and 1). WLOG, assume x_i feeds B, giving the configuration shown in Figure 5.2. Setting x_i to zero gives us a T_2^{n-1} circuit from which we can eliminate 2 gates, since the output from gate A is either a constant, x_j or $\overline{x_j}$, and gate B also depends on a single input. Thus

$$C_{B_2}(T_2^n) \geq 2 + C_{B_2}(T_2^{n-1}) \geq 2 + 2(n-1) - 1 = 2n-1$$

by the inductive assumption. ■

This general method of showing lower bounds is called "gate elimination."

Theorem 5.11 *Let* $XOR_n(x_1, x_2, \cdots, x_n) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$. *Then* $C_{U_2}(XOR_n) = 3(n-1)$.

Proof: The proof is by induction on n. First, $C_{U_2}(\mathrm{XOR}_2) = 3$. If there were just two gates, then they must be in the configuration shown in Figure 5.3a, with the input to the second gate must be a constant c. For U_2, a constant input blocks at least one value on the other wire, so two gates cannot compute the parity function. Figure 5.3b shows the three gate construction for XOR_2, which can be easily extended, giving us $C_{U_2}(\mathrm{XOR}_n) \leq 3(n-1)$.

As with the threshold function, we look at the first gate A in a topological ordering, and say it is fed by x_i and x_j. If x_i does not feed another gate B, then some value of x_j blocks it, which is impossible for the parity function. Further, gate B must feed a third gate C since it cannot be the output gate. Assigning to x_i the value that blocks gate B, we have gate B outputting a constant, gate C outputting either a constant or a unary function of its second variable, and gate A also outputting a constant or a unary function of x_j. We can eliminate all three gates to obtain XOR_{n-1}, possibly having to invert the final output depending on the constant assigned to x_i. Thus

$$C_{U_2}(\mathrm{XOR}_n) \geq 3 + C_{U_2}(\mathrm{XOR}_{n-1}) \geq 3 + 3(n-2) = 3(n-1)$$

by the inductive assumption.

■

5.4 Exercises

Ex. 2 How many wires are required for producing all prefix sums in constant depth? Gates with unbounded fan-in are allowed, but we redefine size to equal the number of wires. A depth 1 $O(n^2)$ construction is simple; see how much of an improvement can be made.

Ex. 3 Use Newton's method to construct a circuit of size $O(m(n))$ for computing the n most significant bits of $\sqrt{x}$ where x is an n-bit number. (Assume x is normalized – the most significant bit is 1). The value $m(n)$ is the size required to multiply two n-bit integers.

Ex. 4 Show that $L \subseteq \mathrm{NL} \subseteq \mathrm{NC}^2$. Hint: begin by showing that matrix multiplication is in NC^1. A LOGSPACE Turing Machine has only a polynomial number of states. Think of a graph with a node for each state, and an edge for each transition. We know in which state the TM begins; we only need to know where it ends. We can simulate one transition by one matrix multiplication...

Solution to Ex. 1 Recall that the goal in exercise 1 was to construct a $n \to 1$ selector using $2n + o(n)$ gates. We first note that it is easy to compute all of the monomials of $x_1, x_2, \cdots, x_n$ using 2^{n+1} gates. (A monomial of $x_1, x_2, \cdots, x_n$ is just $x_1^{\alpha_1} \wedge x_2^{\alpha_2} \wedge \cdots \wedge x_n^{\alpha_n}$ where $\alpha_i \in \{0, 1\}$).

If we were given all of the monomials for free, then we could compute the output x_a of the selector using $2n$ gates by computing the following:

$$x_a = \bigvee_i x_i \wedge (a_0^{\alpha_0^i} \wedge a_1^{\alpha_1^i} \wedge \cdots \wedge a_n^{\alpha_n^i}),$$

where $\alpha_0^i \alpha_1^i \cdots \alpha_n^i$ is the binary representation of i.

We construct the circuit as shown in Figure 5.5. The top level contains $\sqrt{n}$ separate $\sqrt{n} \to 1$ selectors, all of which take $a_0, a_1, \cdots, a_{k/2-1}$ as inputs on the left, where $k = \lceil \log n \rceil$. One of these circuits will output the correct x_a based on the lower order half of the selection index a. Since *all* of the selectors at this level have the same input from the left, we can afford to expend $O(\sqrt{n})$ gates computing all of the monomials of $a_0, a_1, \cdots, a_{k/2-1}$. We then use $2\sqrt{n} \cdot \sqrt{n} = 2n$ gates to actually compute the output for all of these selectors. The final $\sqrt{n} \to 1$ selector selects the "correct" output from among the $\sqrt{n}$ outputs it receives from the previous level, using the higher order half of a. This requires $O(\sqrt{n})$ gates, even if we use the "old" construction that requires $3n + o(n)$ gates for a $n \rightarrow 1$ selector. The total number of gates used is then $2n + O(\sqrt{n})$.

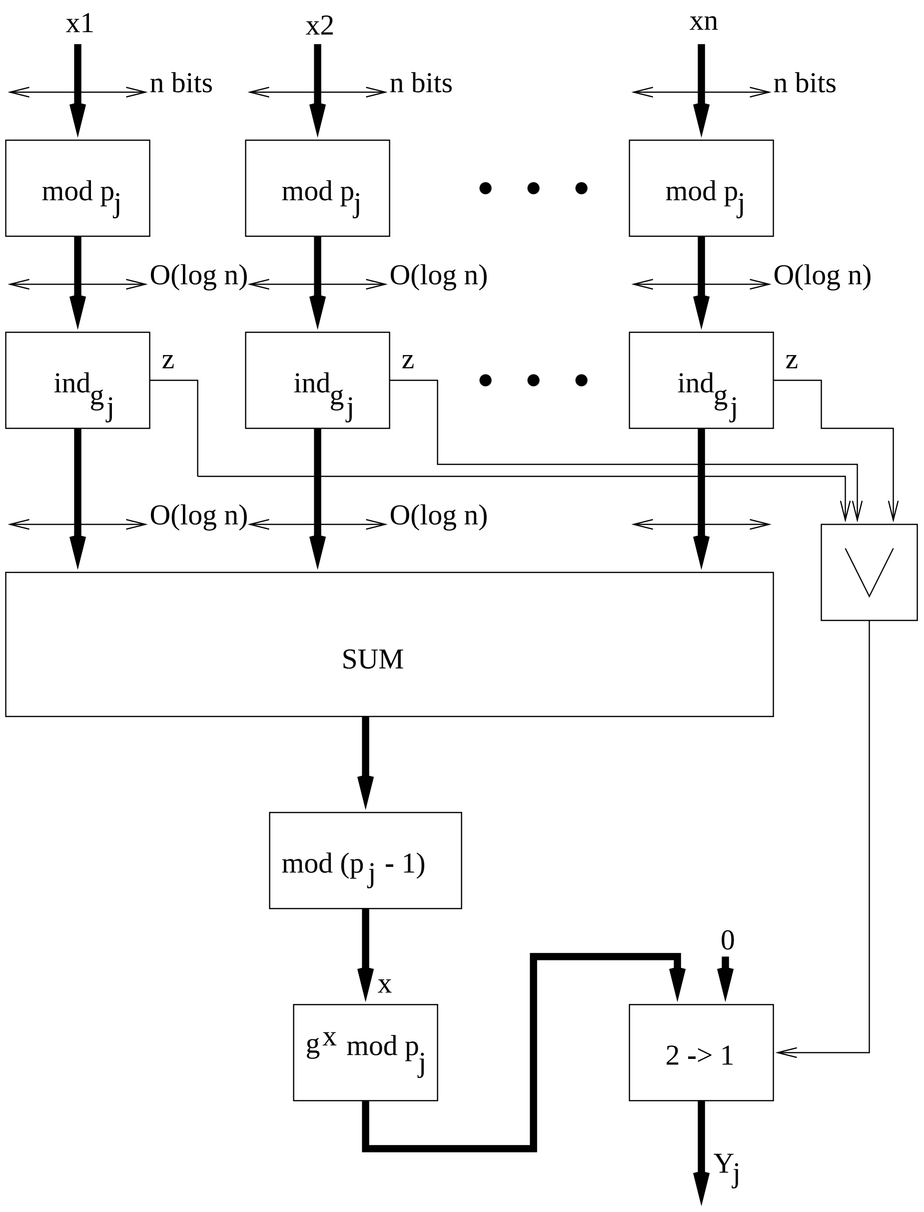

Figure 5.1: The j^{th} part of the product circuit. The z lines in the second-layer modules are 1 iff the input to that module is zero.

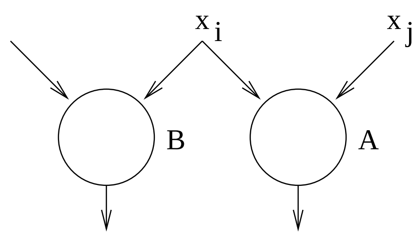

Figure 5.2: The two gates eliminated in the induction step of the proof of Theorem 5.10.

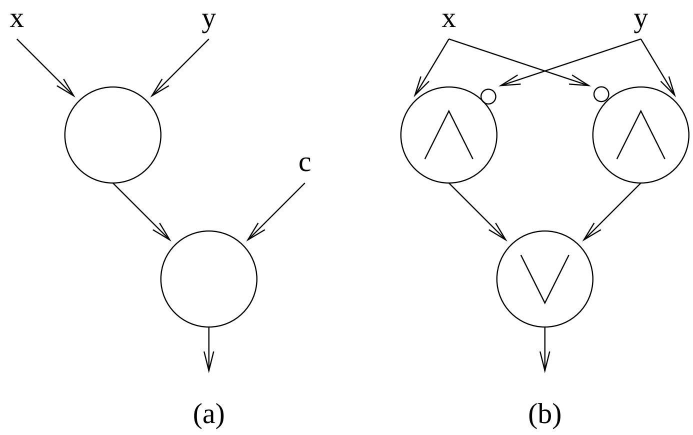

Figure 5.3: Two circuits illustrating that $C_{U_2}(\mathrm{XOR}_2) = 3$.

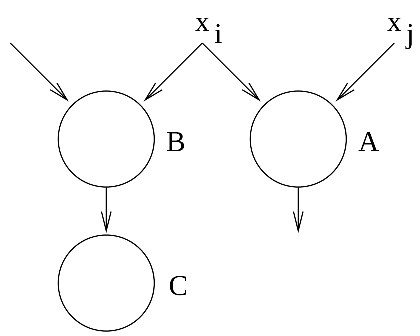

Figure 5.4: The three gates eliminated in the induction step of the proof of Theorem 5.11.

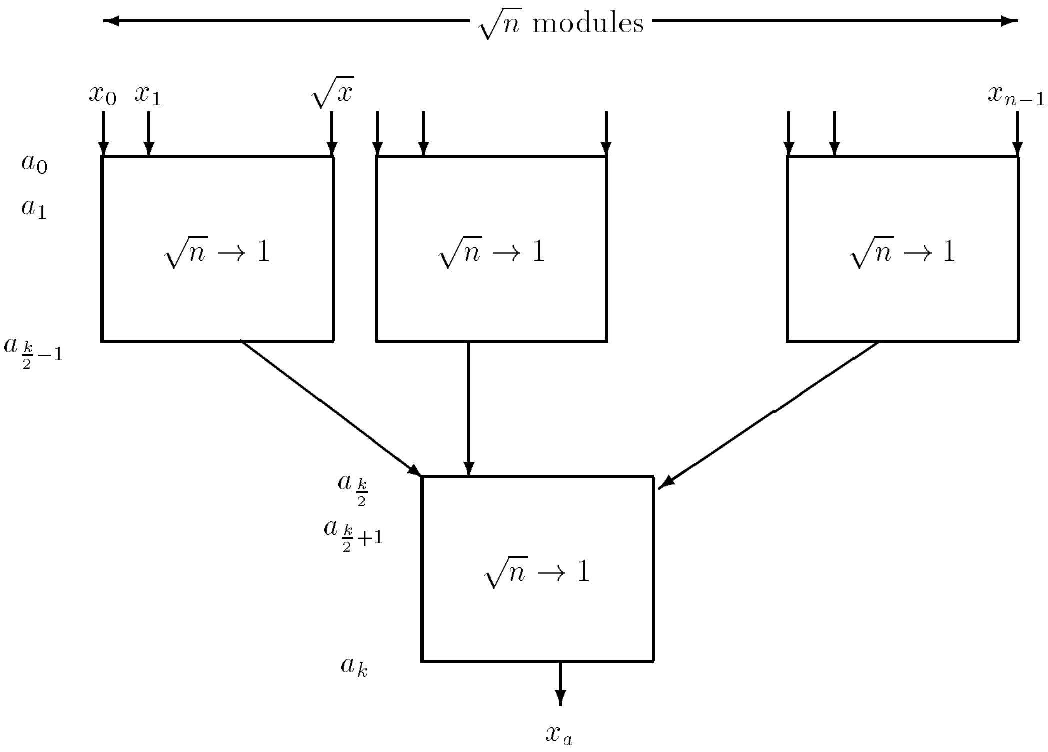

Figure 5.5: The construction of a $n \to 1$ selector circuit.