1 Parallel computation

1.1 NC

Imagine a Boolean formula on \( n \) variables. Suppose that we apply the appropriate electric currents to the inputs. How long will take for these currents to “propagate” through the formula, yielding the value of the formula on the given inputs? A moment’s thought suggests that this time is proportionate to the depth of the formula. Thus, the smaller the depth, the faster we can compute the formula value on any given input.

The considerations above show the importance of the following complexity classes (actively studied by Nick Pippenger, and named in his honor \( \text{NC} \), for “Nick’s Class”, by Steve Cook):

\[
\text{NC}^i = \{ L \mid L \text{ is decided by a family of polysize circuits of depth } O(\log^i n) \}
\]

The class \( \text{NC} = \bigcup_i \text{NC}^i \).

Thus, \( \text{NC}^1 \) is the class of languages decided by polysize circuits of logdepth. In general, almost all Boolean functions need circuits of linear depth. So, those Boolean functions that can be computed by shallow circuits are the functions computable in parallel, as the depth of a circuit corresponds to the parallel time.

Some comments on the definition of NC. For \( \text{NC}^1 \), it does not matter whether we consider polysize circuits of logdepth or formulas of logdepth! This is because any polysize circuit of depth \( O(\log n) \) can be easily converted into a formula of the same depth \( O(\log n) \), by “unwinding” the underlying graph into a tree (i.e., each gate gives rise to as many copies of itself as there are paths from that gate to the output gate of the circuit). (Check this!)

Also, more interestingly, the class of languages computable by logdepth formulas is the same as that computable by polysize formulas (without any depth restrictions)! The reason is that any given polysize formula can be “re-balanced” to become of logdepth. The details follow.

Let \( F(x_1, \ldots, x_n) \) be a formula of size \( s \), where \( s \in \text{poly}(n) \). Then it is possible to show that \( F \) will contain a subformula \( F' \) of size \( t \), where \( s/3 \leq t \leq 2s/3 \). (Think of a tree with
two subtrees: left and right. If either left or right subtree is of size between \( s/3 \) and \( 2s/3 \), then we are done. Otherwise, pick the subtree that is bigger than \( 2s/3 \), and continue with that subtree. Sooner or later, we will come across a subtree whose size is in the required range, since after each step our subtree looses at least one leaf.) Let \( \tilde{F}(x_1, \ldots, x_n, z) \) be the formula \( F \) with the subformula \( F' \) replaced by a new variable \( z \). Then

\[
F(x_1, \ldots, x_n) = (\tilde{F}(x_1, \ldots, x_n, 1) \land F'(x_1, \ldots, x_n)) \lor (\tilde{F}(x_1, \ldots, x_n, 0) \land \neg F'(x_1, \ldots, x_n)).
\]

Now, we recursively re-balance the formulas \( \tilde{F}(x_1, \ldots, x_n, 1) \) and \( F'(x_1, \ldots, x_n) \). Then we plug the resulting balanced formulas into the right-hand side of the expression for \( F \) given above.

Each recursive call adds at most 3 to the depth of the formula. On the other hand, since after each recursive call the size of the formula gets shrunk by a factor \( 2/3 \), there can be at most \( \log_{3/2} |F| \) nested recursive calls (i.e., the depth of the recursion is at most \( O(\log n) \)). Thus, in total, the depth of the formula obtained at the end of this recursive re-balancing will be \( O(\log n) \).

Thus, the above arguments show the following:

**Theorem 1.** The class NC\(^1\) is exactly the class of Boolean function families computable by polysize formulas.

No superlinear lower bounds are known for any explicit (i.e., not random) boolean function, even if we restrict circuits to be of \( O(\log n) \) depth. The following open question was posed by Leslie Valiant in the 80s, and is still open!

**Valiant’s Challenge:** For some explicit boolean function \( f \) (say, in \( \text{NP} \)), prove that \( f \) cannot be computed by any circuit of size \( O(n) \) and simultaneously depth \( O(\log n) \).

### 1.2 Boolean matrix multiplication in NC\(^1\)

Given two \( n \times n \) Boolean-valued matrices \( A, B \), the goal is to compute their product \( C = AB \). Note that \( C[i, j] = \lor_{k=1}^n A[i, k] \land B[k, j] \). For each triple \( i, k, j \), we can compute the AND of \( A[i, k] \) and \( B[k, j] \) in depth 1. Then, for each pair \( i, j \), we can construct a binary tree of depth \( \log n \) that computes the OR of the \( n \) terms \( A[i, k] \land B[k, j] \). Thus, each entry of the matrix \( C \) can be computed in \( O(\log n) \) depth.

### 1.3 Adding \( n \) \( n \)-bit numbers in NC\(^1\): “3 for 2” Trick

Given \( n \) numbers \( A_1, \ldots, A_n \), where each \( A_i \) is an \( n \)-bit non-negative integer, we can add them up by a \( O(\log n) \)-depth NC\(^1\) circuit as follows. Partition the numbers into \( n/3 \) groups of 3 numbers each. To each triple of these numbers, say, \( A, B, C \), apply a constant-depth circuit to be described later that produces a pair of new numbers, say, \( X, Y \), so that

\[
A + B + C = X + Y.
\]
After this stage, we get \((2/3)n\) new numbers that need to be added up, in order to compute the sum of the original \(n\) numbers. (The bit-sizes of these new numbers are just slightly bigger, by at most 1 bit). Then we recurse. After \(O(\log n)\) iterations, we end up with just 2 numbers that need to be added up, which we do by an obvious circuit of depth \(O(\log n)\). As overall we have \(O(\log n)\) recursive stages, with each of them (except the very last addition of two numbers) computable by circuits of constant depth, we get the overall circuit depth \(O(\log n)\), as required.

Finally, we describe the reduction of 3-number addition to 2-number addition. Say we are given \(A = A_{n-1} \ldots A_0\), \(B = B_{n-1} \ldots B_0\), and \(C = C_{n-1} \ldots C_0\) (in binary). Define \(X = X_{n-1} \ldots X_0\) so that, for each \(0 \leq i \leq n - 1\),

\[X_i = A_i \oplus B_i \oplus C_i\]

(the parity of the \(i\)th bits in \(A\), \(B\), and \(C\)). Define \(Y_n \ldots Y_1 Y_0\) so that \(Y_0 = 0\) and, for all \(1 \leq i \leq n\),

\[Y_i = 1 \text{ iff } A_{i-1} + B_{i-1} + C_{i-1} > 1\]

(the integer sum of the \(i\) bits, viewed as one-bit integers, is bigger than 1, i.e., we have a carry into position \(i\) from the sum of the \(i\)th bits of \(A, B, C\)). We claim that \(A + B + C = X + Y\).

(Exercise!) Finally, as each bit of \(X\) and each bit of \(Y\) is a function of some 3 bits in \(A, B, C\), we can construct a constant-depth circuit for computing the strings \(X\) and \(Y\) from the strings \(A, B, C\), as promised.

2 \(TC\)

Here we allow unbounded fanin gates, that is, each gate may have an arbitrary number of inputs coming in (as opposed to the case of \(NC\) above where gates had fanin at most 2). As the gates, we allow MAJORITY functions (on any number of inputs). Recall that \(\text{MAJORITY}_n(x_1, \ldots, x_n) = 1 \text{ iff } \sum_{i=1}^{n} x_i \geq n/2\). We also allow constants 0, 1, as well as NOT gates.

Note that

\[\text{AND}(x_1, x_2) = \text{MAJORITY}_3(x_1, x_2, 0),\]

and

\[\text{OR}(x_1, x_2) = \text{MAJORITY}_2(x_1, x_1).\]

Thus MAJORITY gates can easily simulate any AND and OR gates.

We define the class \(TC^i\) to be the class of polyzise circuits of depth \(\log^i n\), using MAJORITY gates (as well as constant gates, and NOT gates). The interesting case is \(TC^0\): the class of functions computable by constant-depth such circuits (of polynomial size).

Recall that \(\text{PARITY}_n(x_1, \ldots, x_n) = \oplus_{i=1}^{n} x_i\) is the parity of the given \(n\) bits.

Theorem 2. \(\text{PARITY}_n \in \text{TC}^0\).
Proof. Define \( Thr_{n,k}(x_1, \ldots, x_n) = 1 \) iff \( \sum_{i=1}^{n} x_i \geq k \) to be a threshold gate. (Note that \( \text{Majority}_n = Thr_{n,n/2} \) is a special case.) First, it is easy to see (by using extra variables set to constants, as in our example above for simulating AND-gates with MAJORITY gates) that any \( Thr_{n,k} \) can be implemented using a single Majority gate. (Exercise.)

Next, using two threshold gates, we can define for every integer \( 0 \leq k \leq n \), a \( \text{TC}^0 \) circuit that outputs 1 iff the input \( x \) has exactly \( k \) ones in it. Our circuit is

\[
\text{Exact}_{n,k} = Thr_{n,k} \land \neg Thr_{n,k+1}.
\]

Finally, the parity of \( x_1 \ldots x_n \) is 1 iff \( \sum_{i=1}^{n} = a \), for some odd integer \( 1 \leq a \leq n \). Thus, we can compute the parity of \( x_1 \ldots x_n \) as

\[
\lor_{a=2k+1,0 \leq k < n/2} \text{Exact}_{a,n}.
\]

Since AND and OR gates can be simulated threshold gates (as observed above), we conclude that the described circuit is indeed a desired \( \text{TC}^0 \) circuit for Parity.

\[\square\]

3 AC

Here we allow \textit{unbounded fanin} gates, that is, each gate (AND, OR) may have an arbitrary number of inputs coming in (as opposed to the case of \( \text{NC} \) above where gates had fanin at most 2).

We define the class \( \text{AC}^i \) to be the class of languages decided by a family of polysize circuits of depth \( \log^i n \) (where AND and OR gates have unbounded fanin).

The following inclusion is easy: \( \text{NC}^i \subseteq \text{AC}^i \subseteq \text{NC}^{i+1} \), for any \( i \geq 0 \).

3.1 Addition in \( \text{AC}^0 \)

An example of problem in \( \text{AC}^0 \): ADDITION (given \( n \)-bit numbers \( a, b \), compute their sum \( a + b \)). Let \( a = a_{n-1}a_{n-2} \ldots a_1a_0 \) and \( b = b_{n-1}b_{n-2} \ldots b_1b_0 \) be the binary representations of \( a \) and \( b \), respectively. Let \( c = c_n \ldots c_0 \) be the binary representation of their sum. For each bit \( c_i \), we have \( c_i = a_i + b_i + \text{carry}_i \), where the addition is modulo 2, and \( \text{carry}_i \) is 1 if there is a carry into position \( i \), and 0 otherwise. We can define \( c_i \) by the following formula:

\[
c_i \equiv \exists 0 \leq j < i (a_j \& b_j) \forall i > k > j (a_j \lor b_j).
\]

The meaning of the formula is: to have a carry into position \( i \), there must exist a position \( j < i \) where the carry got started (i.e., \( a_j = b_j = 1 \)) and in all positions \( k \) between \( i \) and \( j \) the carry got propagated (i.e., each \( k \) is such that at least one of \( a_k \) or \( b_k \) is 1).

Finally, we can covert the above formula into an \( \text{AC}^0 \) circuit by simulating \( \exists \) with unbounded fanin OR-gates, and \( \forall \) with unbounded fanin AND-gates.
3.2 FO and AC$^0$

More generally, there is a one-to-one correspondence between (uniform) AC$^0$ and First-Order (FO) formulas that also have $+$ and $\ast$ operations. Here an FO formula has a predicate symbol $R(i)$ which encodes an input binary string $X$: the $i$th bit of $X$ is 1 iff $R(i)$ is true. The FO formula has variables $x, y, z, \ldots$ which range over possible positions in the input string $X$. So, if $X$ is of length $n$, then the variables $x, y, z, \ldots$ range over log $n$-bit numbers. Finally, the FO formula may have $\exists$ and $\forall$ quantifiers over the variables, and may use $+$ and $\ast$ on the variables.

Given such a FO formula, we can talk about strings accepted by the formula. A given $n$-bit string $X$ is encoded by a predicate $R(i) = X_i$, for $1 \leq i \leq n$. If the formula with this instantiation of the predicate $R$ is true, then we say that the string $X$ is accepted by (or, more formally, is a model of) the formula. Thus, a given FO formula can be thought of as defining a language (the set of all strings accepted by the formula). It turns out that the class of such languages is exactly the class uniform AC$^0$; here a language $L$ is in uniform AC$^0$ if there is a very efficient algorithm for constructing a circuit on $n$ inputs. One actually needs the notion of DLOGTIME-uniformity, which roughly means that given $n$ and gate indices $i$ and $j$ (all in binary), the algorithm runs in linear time and outputs the labels of the gates (AND/OR/NOT) as well as decides if the gates $i$ and $j$ are connected to each other in the circuit for $L$ restricted to $n$-bit inputs.

3.3 Lower bounds

One of the few successes in complexity theory is that the function PARITY (given a binary string of length $n$, compute the sum of its bits modulo 2) is not in AC$^0$.

**Theorem 3.** $\text{PARITY}_n \not\in \text{AC}^0$.

In fact, to compute PARITY of $n$-bit strings using depth $d$, one needs $\Theta(2^{n/(d-1)})$ size circuits (which is tight). We will sketch the proof next time. As an exercise, you are invited to show this bound for the case of depth $d = 2$. 

5