

CMPT 710 - Complexity Theory: Week 8

Valentine Kabanets

November 3, 2005

1 Parallel Computation

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 **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 $\text{NC} = \cup_i \text{NC}_i$.

Thus, 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 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, 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 $F(x_1, \dots, x_n)$ 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 $1/3$ and $2/3$ of the size of the whole tree, then we are done. Otherwise, pick the subtree that is bigger than $2/3$ of the size of the original tree, 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 loses at least one leaf.) Let $\hat{F}(x_1, \dots, x_n, z)$ be the formula F with the subformula

F' replaced by a new variable z . Then

$$F(x_1, \dots, x_n) = (\hat{F}(x_1, \dots, x_n, 1) \wedge F'(x_1, \dots, x_n)) \vee (\hat{F}(x_1, \dots, x_n, 0) \wedge \neg F'(x_1, \dots, x_n)).$$

Now, we recursively re-balance the formulas $\hat{F}(x_1, \dots, x_n, 1)$ and $F'(x_1, \dots, x_n)$. 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 $O(\log n)$.

2 Examples of problems in NC_1

Boolean matrix multiplication

Given two $n \times n$ Boolean-valued matrices A, B , the goal is to compute their product $C = AB$. Note that $C[i, j] = \bigvee_{k=1}^n A[i, k] \wedge B[k, j]$. 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 $A[i, k] \wedge B[k, j]$. Thus, each entry of the matrix C can be computed in $O(\log n)$ depth.

3 Constant-depth circuits: AC^0

We also consider Boolean circuits of constant depth. If the fan-in remains at most 2, such circuits compute functions that do not depend on all of its inputs. To make things more interesting, we allow the fan-in to be unbounded. The resulting class of polysize circuits of constant depth (and unbounded fan-in) is called AC^0 .

AC^0 is a relatively weak class. For example, the Parity of n -bit strings cannot be computed in AC^0 . (However, the proof of this result is rather involved, and is one of the few successes of complexity theory in proving some kind of circuit lower bounds.) On the other hand, adding two n -bit numbers $a = a_1 \dots a_n$ and $b = b_1 \dots b_n$ can be done in AC^0 .

The idea is to compute for each bit position i , whether there is carry into that position. This computation can be done independently (in parallel) for each bit position i . It is easy to see that the carry into position i is 1 iff there exists an index $j > i$ that generated a carry (i.e., $a_j = b_j = 1$) and that carry was propagated all the way to i (i.e., for each $i < k < j$, we have $a_k = 1$ or $b_k = 1$). It is now easy to construct a constant-depth (unbounded fan-in) circuit computing the carry c_i for each position i . Then using this carry computing circuit, we can easily compute each bit in the sum $a + b$ in AC^0 .