CS 810: Introduction to Complexity Theory 3/4/2003

Lecture 13: Circuit Complexity

Instructor: Jin-Yi Cai Scribe: David Koop, Martin Hock

For the next few lectures, we will deal with circuit complexity. We will concentrate on small depth circuits. These capture parallel computation. Our main goal will be proving circuit lower bounds. These lower bounds show what cannot be computed by small depth circuits. To gain appreciation for these lower bound results, it is essential to first learn about what can be done by these circuits. In next two lectures, we will exhibit the computational power of these circuits. We start with one of the simplest computations: integer addition.

1 Binary Addition

Given two binary numbers, $a = a_1 a_2 \dots a_{n-1} a_n$ and $b = b_1 b_2 \dots b_{n-1} b_n$, we can add the two using the *elementary school method* – adding each column and carrying to the next. In other words, $r = a + b$,

$$\begin{array}{cccccc} & a_n & a_{n-1} & \dots & a_1 & a_0 \\ + & b_n & b_{n-1} & \dots & b_1 & b_0 \\ \hline r_{n+1} & r_n & r_{n-1} & \dots & r_1 & r_0 \end{array}$$

can be accomplished by first computing $r_0 = a_0 \oplus b_0$ ($\oplus$ is exclusive or) and computing a carry bit, $c_1 = a_0 \wedge b_0$. Now, we can compute $r_1 = a_1 \oplus b_1 \oplus c_1$ and $c_2 = (c_1 \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1)$, and in general we have

$$\begin{aligned} r_k &= a_k \oplus b_k \oplus c_k \\ c_k &= (c_{k-1} \wedge (a_k \vee b_k)) \vee (a_k \wedge b_k) \end{aligned}$$

Certainly, the above operation can be done in polynomial time. The main question is, can we do it in parallel faster? The computation expressed above is sequential. Before computing r_k, one needs to compute all the previous output bits. One can express the bits r_k in CNF (or DNF) form. Such a circuit will have depth just 2 and hence, could add fast in parallel. But the size would be exponential.

It turns out that there is indeed a way to do this in fewer levels (i.e. low depth or fast parallel time) and polynomial size. In computing $r_k = a_k \oplus b_k \oplus c_k$, the main problem is computing c_k and what we would like is to be able to compute the carry bit c_k in one step for any k. Notice that for c_1 to be 1, we must have $a_0 = b_0 = 1$. In general, for c_k to be 1, a carry bit has to be generated at some position $i < k$, and should be propagated all the way through to position k. To be precise,

$$c_k = \exists i (0 \le i < k) \left[a_i \wedge b_i \bigwedge \forall j (i < j < k)(a_j \vee b_j) \right]$$

which can be rewritten as

$$c_k = \bigvee_{0 \le i < k} \left[a_i \wedge b_i \wedge \bigwedge_{i < j < k} (a_j \vee b_j) \right]$$

Notice that this computation can be done in constant depth since nothing in it depends on previous results. To do the above computation in constant depth, of course, we would need gates with arbitrary fan-in.

2 NC and AC Classes

We next define the classes AC^i and NC^i. First, we define AC^0.

Definition. AC^0 is the set of languages that have constant depth, polynomial size circuits (unbounded fan-in). Formally, $L \in \mathsf{AC}^0$ if for every n, there exists a circuit C_n such that

1. $\forall x \in \{0,1\}^n, C_n(x) = L(x)$
2. $|C_n| < n^{O(1)}$
3. $\text{depth}(C_n) = O(1)$

We showed that addition is in AC^0 (formally, we should define a language version of addition). We say that a language is in *uniform* AC^0 if the *circuit* C_n is computable in log-space. Meaning, there is a Turing machine, which given 1^n as input, outputs the circuit C_n and the machine runs in space $O(\log n)$. Next, we generalize the notion to define AC^i.

Definition. AC^i is the set of languages L such that $\forall n \exists C_n \forall x \in \{0,1\}^n$

1. $C_n(x) = L(x)$
2. $|C_n| < n^{O(1)}$
3. $\text{depth}(C_n) = O[(\log n)^i]$

AC^i allows circuits to use gates with unbounded fan-in. The class NC^i is similar to AC^i, except that only bounded fan-in gates are allowed by NC^i. More precisely, we require that each boolean $\vee$ and $\wedge$ is binary. The following containments are easy to see:

$$\mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \cdots \subseteq \mathsf{P}$$

As with other hierarchies in complexity theory, we know very little about whether this hierarchy is strict or if it collapses. One thing we know is that $\mathsf{AC}^0 \neq \mathsf{NC}^1$. This is because, one can do parity in NC^1, but not in AC^0. It is easy to see that we do parity in NC^1. Later, we will show that parity cannot be done in AC^0.

3 Multiplication

Now that we have shown binary addition to be in AC^0, we turn our attention to binary multiplication. Again, the *elementary school method* involves multiplying one of the numbers by each of the digits of the other number and adding all the results to get the final result. But since we are doing binary multiplication, each of these digits is either a 1 or a 0. Thus, we either write the number (shifted appropriately) or not, and add all of these (at most $n+1$) numbers together. Thus, multiplication is nothing more than the addition of these $n+1$ numbers as shown in the example below.

$$\begin{array}{cccccccc}
 & & & 1 & 0 & 0 & 1 \\
\times & & & 1 & 1 & 0 & 1 \\
\hline
 & & & 1 & 0 & 0 & 1 \\
 & 1 & 0 & 0 & 1 & & \\
1 & 0 & 0 & 1 & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}$$

So the question that we now wish to answer is how fast we can add k k-bit numbers together. To add these numbers quickly, we introduce a new trick called 3-2 trick. This trick is to take three numbers that you wish to add together, and output two new numbers that add to exactly the same result. In other words, to add $a+b+c$, find two new numbers d and e such that $a+b+c=d+e$. To do this, we simply add (in parallel) the three bits a_i, b_i, and c_i, and set $d_i = a_i \oplus b_i \oplus c_i$ and $e_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$. Now we have converted the addition of three numbers to the addition of two numbers in NC^0.

To make use of this (remember we're trying to add k k-bit numbers to finish multiplication), we keep applying the trick until we only have two numbers left to add at which point we add them using the method to add described above. Thus, the time to add the k k-bit numbers will take $O(\log k)$ parallel applications of the reduction. Thus, the addition of k numbers and therefore integer multiplication as well can be done in NC^1.

Other operations that also are also in NC^1 are integer division and inner product. Using the Chinese Remainder Theorem, it can be shown that division is in NC^1 and is uniform as well.

4 Matrices and Linear Systems

We now turn our attention to matrix computations and solving linear systems of the form $Ax = b$. First, consider multiplying two $n \times n$ matrices A and B. Each entry of the output matrix is an inner product of a row of A and a column of B. All these can be carried out in parallel. Each inner product involves n multiplications and then summing up the products. The multiplications can be carried out in parallel in NC^1. Summing up the n products can be done in a "binary tree fashion" with $\log n$ levels. Thus matrix multiplication can be done in NC^1. How about computing A^k? Using the method of repeated squaring, it is clear we need only $\log k$ levels of matrix squaring. In particular, for a $n \times n$ matrix A, A^n can be computed in NC^2.

Our next goal is to solve linear equations. Recall that a linear system is a bunch of equations we wish to simultaneously solve, and we can represent this as $Ax = b$ where A is the matrix of coefficients, b is the vector of constants, and x is the vector of unknowns. To simplify our computation, suppose that A is a non-singular $n \times n$ matrix.
We first consider the simplest case. Suppose A is upper triangular; that is, A is of the form

$$A = \begin{pmatrix} 1 & x_{1,2} & x_{1,3} & \dots & x_{1,n} \\ 0 & 1 & x_{2,3} & \dots & x_{2,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & x_{n-1,n} \\ 0 & \dots & \dots & 0 & 1 \end{pmatrix}$$

We first write $A = I - \Delta$ where Δ is *strictly upper triangular* (has zeros along the diagonal as well as below the diagonal). Then, Δ^2 is a slightly "smaller" matrix in that the diagonal above the main diagonal is all zeros as well. Raising Δ to successive powers leads to smaller and smaller matrices until we have $\Delta^n = \mathbf{0}$, the zero matrix. Recall that the sum of an infinite geometric series is $\frac{1}{1-r}$ where r is the multiplicative ratio. Similarly, it is tempting to say that, inverse of $I - \Delta$ is given

by,

$$\frac{1}{I-\Delta} = \sum_{i=0}^{\infty} \Delta^i$$

As $\Delta^n = 0$, only the first $n-1$ terms of the above series are non-zero. So,

$$\frac{1}{I-\Delta} = I + \Delta + \Delta^2 + \ldots \Delta^{n-1}$$

Of course, the argument we presented is dubious! But, notice that, the right hand side of the above equation is, indeed, the inverse of $(I-\Delta)$. To verify, multiply the right hand side by $(I-\Delta)$,

$$(I + \Delta + \Delta^2 + \ldots \Delta^{n-1}) \times (I-\Delta) = I - \Delta^n = I$$

(the second equality is because, $\Delta^n = \mathbf{0}$). Thus, if our linear system is such that A is upper triangular, we can find inverse of A by evaluating the above finite series. This involves only matrix multiplication and addition. Finally, the solution is given by $x = A^{-1}b$. This is entire process is certainly in NC^2 (computing Δ^n is the hardest part).

This process works great for upper triangular matrices, but we want to solve *any* linear system. We can do that using Gaussian elimination, but this process is certainly not parallel as each step depends on what was done in the previous step. Our aim to somehow reduce the problem of inverting general matrices to the problem inverting upper triangular matrices. We will do that in next lecture.