Chapter 10

Computing With Circuits

For the next few lectures, we will deal with circuit complexity. First we will concentrate on small depth circuits. As the depth of a circuit corresponds to parallel computation time, small depth circuit complexity captures efficient parallel computation. Before we prove circuit lower bounds, which show cannot be computed by small depth circuits, we should gain some appreciation of what can be done by these circuits. So we first exhibit some computational power of these circuits. We start with one of the simplest computations: integer addition.

10.1 Binary Addition

Given two binary numbers, $a = a_{n-1} \ldots a_1 a_0$ and $b = b_n b_{n-1} \ldots b_1 b_0$, we can add the two using the *school method* – adding each column from right (the least significant bit) to left (the most significant bit), with carry bit along the way. In other words, $r = a + b$, where

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

can be accomplished by first computing $r_0 = a_0 \oplus b_0$ ($\oplus$ is exclusive or, which can be further written out in terms of $\vee$ and $\wedge$) and computing a carry bit, $c_1 = a_0 \wedge b_0$. Then, 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, for $k \geq 1$ 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 faster in parallel? Note that the computation expressed above is sequential. Before

computing r_k, one needs to compute all the previous output bits. Of course, one can always 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 this argument via CNF (or DNF) would give exponential size circuit in general.

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. This is usually called *Carry-Look-Ahead*. 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,

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

which can be rewritten as

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

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.

10.2 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, $L \in \mathrm{AC}^0$ if for every n, there exists a circuit C_n with unbounded fan-in gates AND, OR and NOT, such that

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

We showed that addition is in AC^0 (formally, we should define a language version of addition, for example, the ith bit of the sum). 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 for every n, there exists a circuit C_n with unbounded fan-in gates AND, OR and NOT, such that

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

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 (for uniform NC and AC classes) are easy to see:

$$\text{AC}^0 \subseteq \text{NC}^1 \subseteq \text{AC}^1 \subseteq \text{NC}^2 \subseteq \ldots \subseteq \text{P}$$

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 $\text{AC}^0 \neq \text{NC}^1$. 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, by divide and conquer. Later, we will show that parity cannot be done in AC^0.

10.3 Multiplication

Now that we have shown binary addition to be in AC^0, we turn our attention to binary multiplication. Again, the *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) numbers together. Thus, multiplication is nothing more than the addition of these n numbers, each of at most $2n-1$ bits, as shown in the example below.

			1	0	0	1
$\times$			1	1	0	1
			1	0	0	1
	1	0	0	1		
1	0	0	1			
1	1	1	0	1	0	1

So the question is how to add m binary numbers, each of at most m bits. To add these numbers quickly, we introduce a new trick called 3-2 trick. This trick is to take 3 numbers a, b, c, and output 2 new numbers d, e, such that the sum of the 2 new numbers is the same as the sum of the 3 input numbers: $a+b+c = d+e$. To do this, we imagine doing an addition at every bit position i separately. Given a_i, b_i and c_i, we can produce the sum as two bits, $uv \in \{00, 01, 10, 11\}$. The school method would have set the leading bit u as the carry bit and propogated onward. Instead we write down v as the ith bit of d, and u as the $i+1$th bit of e, where d and e are the two output numbers. More precisely, for $0 \leq i < n$ in parallel, we take 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 constant depth with bounded fan-in circuit (which are sometimes called NC^0).

To make use of this for the addition of m numbers, we apply divide and conquer. After $O(\log m)$ parallel rounds of the 3-2 tricks, we are left with two numbers to add, at which point we add them using the carry-look-ahead method. Thus, the parallel time (depth) to add the m m-bit numbers can be done in NC^1.

Other operations that 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. (For division, the uniformity of the circuit had been a tricky issue. For many years one can only achieve P-uniformity; but recently it has been shown that it can be done in logspace-uniform NC^1.)

10.4 Inner Product, Matrix Powers and Triangular 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^m? Using repeated squaring, it is clear we need only $O(\log m)$ levels of matrix squaring and multiplying. In particular, for a $n \times n$ matrix A with each entry an $O(n)$-bit integer, $A^{O(n)}$ can be computed in NC^2.

Our next goal is to solve linear equations. Recall that a linear system is of the form $Ax = b$ where A is a matrix of coefficients, b is a vector of constants and x is a vector of unknowns. To simplify our discussion, suppose that A is a non-singular $n \times n$ matrix.

We first consider the simplest case. Suppose A is lower triangular with unit diagonals; that is, A is of the form

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

We first write $A = I - \Delta$ where Δ is strictly lowe triangular (i.e. has zeros along the diagonal as well as above the diagonal). Note that Δ^2 has zeros in not only the upper triangular and

diagonal entries, but also the diagonal just above the main diagonal. Δ is called a *nilpotent* matrix, in that for some m, $\Delta^m = 0$. In fact $\Delta^n = 0$. By analogy of geometric sereis, we write

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

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

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

Of course, so far this argument is dubious, since it is arrived at by analogy. This argument can be made rigorous. However we will be theoretical computer scientist for once, a.k.a. mathematicians in a hurry, and justify this simply by verifying that the finite sum exhibited indeed is the inverse of $I - \Delta$, QED. Thus, if our linear system is such that A is lower triangular, (as A is non-singular, by dividing out the diagonal entries, we can assume it has unit diagonals), we can compute the 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 entire process is certainly in NC^2.

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

10.5 Determinant and Linear Equations

We show how to solve general linear equations in NC^2.

The goal is to solve a given system of linear equations: $Ax = b$, where A is a non-singular $n \times n$ matrix, b an n dimensional column vector and $x = (x_1, x_2, \ldots, x_n)$ is the vector of unknowns. Our aim is to compute the determinant of A in NC^2. Once we do that, the solution x can be found by using Cramer's rule: $x_i = \det(A_i)/\det(A)$, where A_i is the matrix obtained by replacing the ith column of A by the column vector b.

10.5.1 Trace of a matrix

Trace of an $n \times n$ matrix A, denoted $\mathrm{Tr}(A)$, is the sum of its diagonal entries. That is,

$$\mathrm{Tr}(A) = \sum_{i=0}^{n} a_{ii}$$

We next state some properties of trace.

PROPOSITION 10.1 *For two $n \times n$ matrices A and B, $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$.*

Proof.

$$\mathrm{Tr}(AB) = \sum_i \sum_k a_{ik} b_{ki} = \sum_k \sum_i a_{ik} b_{ki} = \sum_k \sum_i b_{ki} a_{ik} = \mathrm{Tr}(BA)$$

♣

The following lemma is well-known for a spectural decomposition of a matrix. More precise information of the matrix J (called Jordan normal form) is known, but we will only need this form. We will use the lemma to establish more properties of trace and eigenvalues.

LEMMA 10.2 *Let A be an $n \times n$ matrix. There exist $n \times n$ matrices T and J such that $A = T^{-1}JT$, where J is an upper triangular matrix whose n diagonal entries are the n eigenvalues of A.*

PROPOSITION 10.3 *Let A be an $n \times n$ matrix and $\lambda_1, \lambda_2, \ldots, \lambda_n$ be its eigenvalues. Then, for $k \geq 0$,*

$$\mathrm{Tr}(A^k) = \sum_{i=1}^{n} \lambda_i^k.$$

Proof. Let $A = T^{-1}JT$. Then we have

$$A^k = (T^{-1}JT)^k = T^{-1}J^kT$$

So,

$$\mathrm{Tr}(A^k) = \mathrm{Tr}(T^{-1}J^kT) = \mathrm{Tr}(T^{-1}TJ^k) = \mathrm{Tr}(J^k)$$

The second equality follows from Proposition 10.1. Note that J is an upper triangular matrix whose n diagonal elements are $\lambda_1, \lambda_2, \ldots, \lambda_n$. Hence, the diagonal elements of J^k are $\lambda_1^k, \lambda_2^k, \ldots, \lambda_n^k$. So,

$$\mathrm{Tr}(J^k) = \sum_{i=1}^{n} \lambda_i^k$$

♣

Next, we can express the determinant of a matrix as the product of its eigenvalues.

PROPOSITION 10.4 *For an $n \times n$ matrix A with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, we have*

$$\det(A) = \prod_{i=1}^{n} \lambda_i$$

Proof. Consider the decomposition $A = T^{-1}JT$. Then,

$$\det(A) = \det(T^{-1}JT) = \det(J) = \prod_{i=1}^{n} \lambda_i$$

♣

From last section, for $1 \leq k \leq n$, we know how to compute A^k in NC^2. Hence, we can also compute $\mathrm{Tr}(A^k) = \sum_{i=1}^{n} \lambda^k$. Notice that we have computed all these sums without explicitly computing $\lambda_1, \lambda_2, \ldots, \lambda_n$. Our aim is to compute $\det(A)$, the product of eigenvalues. One could explicitly compute these eigenvalues and then take their product. But, computing eigenvalues is at least as hard as computing the determinant. Instead, we will directly compute their product using the known traces $\mathrm{Tr}(A^k)$, $1 \leq k \leq n$.

Think of the eigenvalues as unknowns or variables. Then, $\mathrm{Tr}(A^k)$ and $\det(A)$ are polynomials over these variables. These polynomials are *symmetric polynomials* and we can express the polynomial $\det(A)$ in terms of the polynomials $\mathrm{Tr}(A^k)$. Towards that end, we shall diverge a little bit and study the symmetric polynomials.

10.5.2 Symmetric polynomials

We will focus on polynomials over n variables $x_1, x_2, \ldots, x_n$. A polynomial over these variables is symmetric, if renaming the variables does not change the polynomial. Formally,

Definition 10.5 (Symmetric polynomial) *A polynomial p on $x_1, x_2, \ldots, x_n$ is* symmetric *if $\forall \sigma \in S_n$ (i.e., for all possible permutations of 1 through n), $p(x_1, \ldots, x_n) = p(x_{\sigma_1}, x_{\sigma_2}, \ldots, x_{\sigma_n})$.*

Two families of symmetric polynomials are important for our purpose here: Newton's symmetric polynomials and elementary symmetric polynomials. We first give Newton's symmetric polynomials. For $0 \leq k \leq n$, the kth Newton's symmetric polynomial is given by

$$s_k = \sum_{i=1}^{n} x_i^k$$

It is clear that these polynomials are symmetric. We could define s_k for $k > n$, but we will only need $0 \leq k \leq n$. Note that $s_0 = n$.

We now give the elementary symmetric polynomials. For $1 \leq k \leq n$, the kth elementary symmetric polynomial is given by

$$p_k = \sum_{1 \leq i_1 < i_2 < \ldots < i_k \leq n} x_{i_1} x_{i_2} \ldots x_{i_k}$$

Since every choice of k variables appears as a product exactly once, these polynomials are symmetric. It is convenient to define $p_0 = 1$. Note that $p_n = \prod_i x_i$.

(Notice the correspondence between these polynomials and trace and determinant we discussed before. In particular, $\mathrm{Tr}(A^k) = \sum_{i=1}^n \lambda_i^k$ is the kth Newton's symmetric polynomial, whereas, $\det(A) = \prod_{i=1}^n \lambda_i$ is the nth elementary symmetric polynomial.)

One can express the elementary symmetric polynomials in terms of Newton's polynomials. For example,

$$p_1 = \sum_{i=1}^{n} x_i = s_1.$$

Similarly, since $s_2 = \sum_{i=1}^n x_i^2$,

$$p_2 = \sum_{1 \le i < j \le n} x_i x_j.$$

Thus,

$$s_1^2 = \left(\sum x_i\right)^2 = s_2 + 2p_2.$$

Therefore,

$$p_2 = \frac{1}{2}(s_1^2 - s_2).$$

In fact any symmetric polynomial can be expressed as a polynomial of Newton's polynomials as well as expressed as a polynomial of elementary symmetric polynomials:

THEOREM 10.6 (FUNDAMENTAL THEOREM OF SYMMETRIC POLYNOMIALS) *Every polynomial on $x_1, \ldots, x_n$ is symmetric iff it can be expressed as a polynomial of $s_1, \ldots, s_n$ (as well as $p_1, \ldots, p_n$).*

We can use this theorem to relate the elementary symmetric polynomials and Newton's polynomials. But, we need more specific formulas relating the two families of polynomials. So, next we state and prove a relationship between these two families that is useful in computing determinants in NC^2.

THEOREM 10.7 *For $1 \le k \le n$,*

$$p_k = \frac{1}{k}(p_{k-1} \cdot s_1 - p_{k-2} \cdot s_2 + \cdots \pm p_0 \cdot s_k)$$

Proof. For $k, m \ge 0$, define the polynomial:

$$f_k^m = \sum_{1 \le i_1 < \ldots < i_k \le n j \notin \{i_1, \ldots, i_k\}} x_{i_1} x_{i_2} \ldots x_{i_k} x_j^m$$

How many terms are there to sum over? Note that there are two distinct parts, the product $x_{i_1} x_{i_2} \ldots x_{i_k}$ and x_j^m. There are $\binom{n}{k} \cdot (n-k)$ terms. Clearly, the boundary cases are $f_0^m = \sum_{j=1}^n x_j^m = s_m$ and $f_k^0 = (n-k) \cdot p_k$.

We now derive a formula for $p_k \cdot s_m$, when $k \geq 1$ and $m \geq 0$:

$$\begin{aligned} p_k \cdot s_m &= \left(\sum_{1 \leq i_1 < \ldots < i_k \leq n} x_{i_1} x_{i_2} \ldots x_{i_k} \right) \cdot \left(\sum_{j=1}^{n} x_j^m \right) \\ &= \sum_{1 \leq i_1 < \ldots < i_k \leq n, j \notin \{i_1, \ldots, i_k\}} x_{i_1} x_{i_2} \ldots x_{i_k} x_j^m + \sum_{1 \leq i_1 < \ldots < i_k \leq n, j \in \{i_1, \ldots, i_k\}} x_{i_1} x_{i_2} \ldots x_{i_k} x_j^m \end{aligned}$$

The first sum is f_k^m. What about the second sum? It's a bit tricker, but a moment reflection convinces us that it is nothing but f_{k-1}^{m+1}, as each term in f_{k-1}^{m+1} occurs eactly once in the second sum. So, for $k, m \geq 1$, we have

$$p_k \cdot s_m = f_k^m + f_{k-1}^{m+1}$$

For $k = 0$, since $p_0 = 1$, $p_0 \cdot s_m = s_m = f_0^m$.

Now consider the alternating sum:

$$\begin{aligned} & p_k \cdot s_0 - p_{k-1} \cdot s_1 + p_{k-2} \cdot s_2 - \ldots + (-1)^k p_0 \cdot s_k \\ = \; & (f_k^0 + f_{k-1}^1) - (f_{k-1}^1 + f_{k-2}^2) + \ldots + (-1)^{k-1} (f_1^{k-1} + f_0^k) + (-1)^k f_0^k \end{aligned}$$

This is a telescoping sum, where all the terms cancel out, except for f_k^0. By definition, $f_k^0 = (n-k)p_k$. Hence,

$$(n-k)p_k = p_k \cdot s_0 - p_{k-1} \cdot s_1 + p_{k-2} \cdot s_2 + \cdots + (-1)^k p_0 \cdot s_k$$

Note that $s_0 = n$ and $p_0 = 1$. So,

$$(-1)^{k-1} p_1 s_{k-1} + \cdots + p_{k-2} s_2 - p_{k-1} s_1 + k p_k = (-1)^{k-1} s_k.$$

♣

10.5.3 Csanky's Algorithm for Determinant

Recall that we are interested in computing the determinant of a given non-singular $n \times n$ matrix A. Let its eigenvalues be $\lambda_1, \lambda_2, \ldots, \lambda_n$. We can compute $S_k = \mathrm{Tr}(A^k)$ in NC^2. Our goal is to compute,

$$\det(A) = P_n = \lambda_1 \lambda_2 \ldots \lambda_n$$

Think of the eigenvalues as variables or unknowns. Then, S_k and P_k can be viewed as polynomials. For each $1 \leq k \leq n$, S_k is nothing but the kth Newton's symmetric polynomial and P_k is nothing but the kth elementary symmetric polynomial. We have these equalities relating P_k and S_k. Now, the idea is to treat $\{P_k\}$ as the unknowns and solve for $\{P_k\}$ as a system of linear equations. We can write

$$(-1)^{k-1} S_{k-1} P_1 + \cdots + S_2 P_{k-2} - S_1 P_{k-1} + k P_k = (-1)^{k-1} S_k.$$

Recall that we started off with the problem of solving linear system of equations. Have we made any progress? Yes! The crucial point is that, now the system is lower triangular! To be more pedantic, let us write the above system in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -\frac{S_1}{2} & 1 & 0 & \cdots & 0 \\ \frac{S_2}{3} & -\frac{S_1}{3} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \frac{(-1)^{n-1}S_{n-1}}{n} & \frac{(-1)^{n-2}S_{n-2}}{n} & \frac{(-1)^{n-3}S_{n-3}}{n} & \cdots & 1 \end{bmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ \vdots \\ P_n \end{pmatrix} = \begin{pmatrix} S_1 \\ -\frac{S_2}{2} \\ +\frac{S_3}{3} \\ \vdots \\ \frac{(-1)^{n-1}S_n}{n} \end{pmatrix}$$

We already saw how to solve system of linear equations whose coefficient matrix is lower triangular in NC^2. Using that method, we can solve the above system of equations and compute $P_1, P_2, \ldots, P_n$. Finally, $P_n = \det(A)$ is the required answer.