Using the Chinese Remainder Theorem, we can compute perm B by computing perm B mod p_i, $1 \leq i \leq k$. For each i, $1 \leq i \leq k$, we replace all -2 entries in B by $p_i - 2$; modulo p_i, they are the same. We then compute the permanent of this matrix and reduce modulo p_i to get perm B mod p_i. The advantage of this is that we have now reduced the problem to that of computing the permanents of matrices with small nonnegative entries only.

All that remains is to show how to reduce the computation of the permanent of a matrix over $\{0, 1, 2, p-2\}$ to the problem of computing the permanent of a matrix over $\{0, 1\}$.

Recall the equivalence between the permanent of a matrix and the cycle covers of a directed graph. We must reduce the problem of computing the number of cycle covers of a weighted directed graph with positive integral weights to the problem of computing the number of cycle covers of an unweighted directed graph. This is accomplished by replacing every weighted edge with a subgraph consisting of several new vertices and edges.

The following figure shows this construction for an edge of weight 3.

The above process is repeated for each edge in G. The resulting graph G' is unweighted. Each cycle cover in G involving edges (u_i, v_i) with weights m_i, $1 \leq i \leq n$, is simulated by $m_1 m_2 \cdots m_n$ cycle covers in G', each of weight 1, thus the permanents are the same. Also, G' can be constructed in polynomial time.

This completes the proof that the problem of counting the number of perfect matchings in a bipartite graph (equivalently, counting the number of cycle covers in a directed graph) is $\#P$-complete.

Lecture 28 Parallel Algorithms and *NC*

Parallel computing is a popular current research topic. The successful design of parallel algorithms requires identifying sources of data independence in a problem that allow it to be decomposed into independent subproblems, which can then be solved in parallel. This process often involves looking deeply into the mathematical structure of the problem.

Aside from specific architectures such as the hypercube, there are many different general models of parallel computation in use. Among the most popular are:

- *Parallel Random Access Machines (PRAMs).* A PRAM consists of a set of processors that have access to a common shared memory. Each processor may have registers and local memory of its own. We charge one time unit for a memory access (which many consider an unreasonable assumption). PRAMs can be exclusive or concurrent read and exclusive or concurrent write, giving four versions, denoted CRCW, CREW, ERCW, EREW. An EREW PRAM does not allow processors to read and write simultaneously to the same memory location, and requires the programmer to insure that this does not happen. A CRCW PRAM does allow this, and resolves conflicts arbitrarily.

- *Vector machines.* This model can be *SIMD (Single Instruction Multiple Data)* or *MIMD (Multiple Instruction Multiple Data).* The processors are arranged in an array and all execute synchronously. The SIMD

machines all execute the same instruction, but execute it on different data. Processors communicate by message passing.

- *Boolean and arithmetic circuits.* These are essentially dags with input nodes, output nodes, and basic bit operations or arithmetic operations associated with internal nodes. This model is quite common, especially in the theory of *NC*. The size of the circuit (number of nodes) corresponds roughly to the number of processors in a PRAM, and the depth of the circuit (length of the longest path from an input to an output) corresponds to time. Since each circuit has only a fixed number of input nodes, there must be a different circuit for each input length.

Many object to these models on the grounds that they do not adequately capture the "communication bottleneck", since communication complexity is not usually counted. These arguments do have merit, and one should not immediately take a parallel complexity bound obtained in one of these models as an accurate indication of the performance one would expect of a parallel implementation under current technology. However, independent of whether or not the complexity bounds are realistic, the important matter is to identify the fundamental sources of independence in a computational problem that allow efficient parallelization. These are mathematical properties that transcend technology; they will be there to exploit in any parallel machine or machine model now or in the future.

28.1 The Class *NC*

The complexity class *NC* plays the same role in parallel computation that P plays in sequential computation. A problem is considered to be "efficiently parallelizable" (at least in theory) if it can be shown to be in *NC*. The name *NC* stands for *Nick's Class*, after Nick Pippenger, who invented it.

Like P, the definition of *NC* is quite robust in the sense that it is impervious to minor perturbations of the machine model. It is the class of problems that can be solved on a PRAM in $(\log n)^{O(1)}$ or polylogarithmic time using $n^{O(1)}$ or polynomially many processors. It can also be defined as the class of problems accepted by a *uniform* family of Boolean circuits, one for each input length, of polylogarithmic depth and polynomial size. The uniformity condition says essentially that the n^{th} circuit in this family is easily constructed, and is a technical condition that allows circuits and PRAMs to simulate each other efficiently. See the survey paper [23] for details.

The question $NC \stackrel{?}{=} P$ is analogous to the $P \stackrel{?}{=} NP$ question. There is an *NC* reducibility relation and a notion of P-completeness with respect to that reducibility relation. There is a set of problems known to be P-complete, among them the circuit value problem [67] and max flow [42]. The classes P and *NC* are equal if any of these problems turn out to be in *NC*.

28.2 Parallel Matrix Multiplication

To illustrate, we give a simple parallel algorithm to compute the product of two $n \times n$ matrices in time $1 + \log n$ with n^3 processors. We use the arithmetic circuit model.

Let A and B be two $n \times n$ matrices. We assume that the entries A_{ij} of A and B_{ij} of B are available at the n^2 input nodes of the circuit. Recall that

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \,. \tag{36}$$

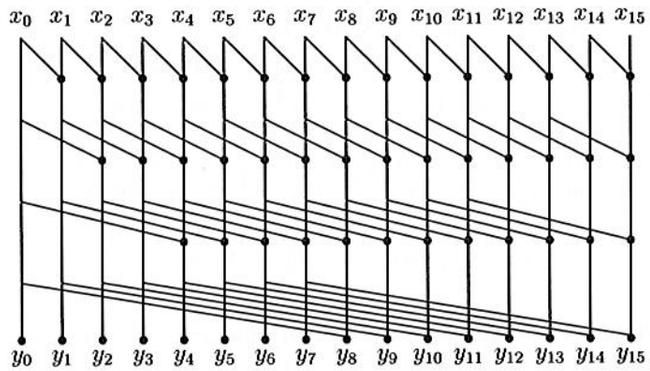
In parallel, compute the n^3 products $A_{ik}B_{kj}$ for each triple i, j, k. This can be done in one step, since we have n^3 processors. Then allocate n processors to each pair i, j and compute the sums (36) from the data computed in the first step. This sum can be obtained in $\log n$ time in parallel by placing each of the n summands at the leaves of a complete binary tree, and summing adjacent pairs. This requires $\log n$ stages, since at each stage the number of data items is halved. The value at the root of the binary tree is the sum of the elements at the leaves.

28.3 Parallel Prefix

This circuit is a very useful subroutine in many parallel algorithms. Suppose we have n elements $x_0, x_1, \ldots, x_{n-1}$ and a binary operation $\cdot$ that is associative but not necessarily commutative. We wish to compute the *prefix products* y_i, $0 \le i \le n-1$, where

$$y_i = x_0 \cdot x_1 \cdot x_2 \cdots x_i \,.$$

Consider the following circuit with n input gates and n output gates. The i^{th} input gate receives x_i and the i^{th} output gate gives y_i. In the first step, every processor i passes its data to processor $i+1$, and the two data items are multiplied. In the next stage, data is passed from each i to $i+2$; in the next stage, from i to $i+4$; and so on for $\log n$ stages. The following illustration gives the circuit for $n = 16$.



This construction works even if n is not a power of 2. See [68] for an alternative construction.

This parallel algorithm has a particularly nice implementation on a hypercube. We can embed the circuit of 2^n processors on a hypercube of dimension n in such a way that all message routing can be done with no collisions and no message travels more than a distance of 2 on the cube.

This embedding will be defined in terms of the *Gray representation* of the numbers in the set $\{0, 1, 2, \ldots, 2^n - 1\}$, as opposed to the usual binary representation. Both representations pair elements of this set with the n-bit binary strings in a one-to-one fashion. In the natural order

$$0 < 1 < 2 < \cdots < 2^n - 1 \,,$$

the corresponding sequence of strings in the binary representation is obtained by starting from $0 \ldots 0$ and successively adding 1 in binary. For example, for $n = 4$ we get the sequence

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, ..., 1111 .

In the Gray representation, the sequence is

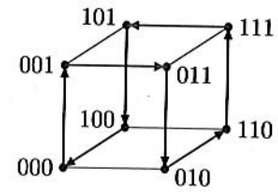
0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, ..., 1000 .

Each element is obtained from the last by flipping one bit. If we graph the sequence of bits that are flipped, the picture looks similar to an English ruler with demarcations for inches, half inches, quarter inches, and so forth.

Now consider the unit cube in n-dimensional Euclidean space. Its vertices are points with Euclidean coordinates $(a_0, \ldots, a_{n-1})$ where each $a_i \in \{0, 1\}$. We map the processor that is i^{th} from the left in the parallel prefix circuit to the point of the cube whose Euclidean coordinates give i in the Gray representation. For $n = 3$, the Gray ordering is

000, 001, 011, 010, 110, 111, 101, 100

and this corresponds to the following Hamiltonian circuit in the cube:



It is easy to convert back and forth between the binary and Gray representations. Let b_i and g_i denote the binary and Gray representations of i respectively. Then the j^{th} bit of g_i (counting from the left and starting at 0) is the exclusive-or of the j^{th} and $j - 1^{\text{st}}$ bits of b_i, and the j^{th} bit of b_i is obtained from g_i by taking the exclusive-or of the j^{th} bit of g_i and all bits to its left. Converting b_i to g_i takes time $O(1)$ with n processors and converting g_i to b_i takes time $O(\log n)$ with n processors using parallel prefix.

In the next lecture we will see how to characterize these operations algebraically. This will give a convenient means for proving properties of binary and Gray representations and of routing on the hypercube. We will then use these tools to analyze our hypercube implementation of parallel prefix.

Lecture 30 Integer Arithmetic in NC

30.1 Integer Addition

Addition of two n-bit binary numbers can be performed in $\log n$ depth with n processors. We will use parallel prefix to calculate the carry string. Once the carry is computed, the sum is easily computed in constant time with n processors by taking the exclusive-or of the two summands and the carry string.

The carry string is defined as follows:

- The lowest order carry bit is always 0.
- If the i^{th} bits of the two summands (counting from the right) are both 0, then the $i+1^{\text{st}}$ bit of the carry will be 0, irrespective of the i^{th} bit of the carry.
- If the i^{th} bits of the two summands are both 1, then the $i+1^{\text{st}}$ bit of the carry will be 1, irrespective of the i^{th} bit of the carry.
- If the i^{th} bits of the two summands are 0 and 1, then the $i+1^{\text{st}}$ bit of the carry will be the same as the i^{th} bit of the carry. In this case we say that the carry is *propagated* from i to $i+1$.

To compute the carry using parallel prefix, we will use a three element algebra $\{0,1,p\}$ with associative binary operation $\cdot$ defined below. Intuitively, the

element 0 means, "carry 0", the element 1 means "carry 1", and the element p means, "propagate the carry from the previous bit position".

The binary operation $\cdot$ is defined by the following table:

$\cdot$	0	1	p
0	0	0	0
1	1	1	1
p	0	1	p

In other words, for any $x \in \{0,1,p\}$,

$$\begin{aligned} 0 \cdot x &= 0 \\ 1 \cdot x &= 1 \\ p \cdot x &= x\,. \end{aligned}$$

Note that $\cdot$ is associative but not commutative: $0 \cdot 1 = 0$ but $1 \cdot 0 = 1$.

Let u be a string over $\{0,1,p\}$ with a 0 in position 0, a 0 in position $i+1$ if the i^{th} bits of the two summands are both 0, a 1 in position $i+1$ if the i^{th} bits of the two summands are both 1, and a p in position $i+1$ if one of the i^{th} bits of the two summands is 0 and the other is 1. The string u can be computed in constant time from a and b with n processors. The carry string is obtained by computing the suffix products of u.

Example 30.1 Let $a = 100101011101011$ and $b = 110101001010001$. The string u over $\{0,1,p\}$ for these two numbers, the carry string obtained as the suffixes of u, and the binary sum are as illustrated.

$$\begin{aligned} u &= 1p01010p1ppp0p10 \\ carry &= 1001010110000110 \\ a &= 100101011101011 \\ b &= 110101001010001 \\ \hline sum &= 1011010100111100 \end{aligned}$$

□

30.2 Integer Multiplication

Consider a multiplication problem involving two n-bit binary numbers. The grade school algorithm for multiplication gives n partial sums, which then can

be added to get the product. For example,

$$\begin{array}{r} 101101 \\ \times\ 101011 \\ \hline 101101 \\ 101101 \\ 000000 \\ 101101 \\ 000000 \\ +\ 101101 \\ \hline 11110001111 \end{array} \qquad (37)$$

This can be done in time $O((\log n)^2)$ with $O(n^2)$ processors in a straightforward way. First compute all the bits of the partial sums, then add the partial sums in pairs in a tree-like fashion. It takes constant time to compute the partial sums with $O(n^2)$ processors, and $O((\log n)^2)$ to do the additions.

By being slightly more clever, we can reduce the time to $O(\log n)$ by reducing the problem of adding three n-bit binary numbers to adding two $n+1$-bit binary numbers. Look at the partial sums obtained by adding each 3-bit column individually:

$$\begin{array}{r} 101100111 \\ 101011100 \\ +\ 101111101 \\ \hline 10 \\ 01 \\ 11 \\ 10 \\ 10 \\ 10 \\ 11 \\ 00 \\ +\ 11 \\ \hline \end{array}$$

Rearranging, we get

$$\begin{array}{r} 10 \\ 01 \\ 11 \\ 10 \\ 10 \\ 10 \\ 11 \\ 00 \\ +\ 11 \\ \hline \end{array} \longrightarrow \begin{array}{r} 101111101 \\ +\ 101000110 \\ \hline \end{array}$$

Thus, in constant time we have reduced the problem of adding three binary numbers to adding two binary numbers. To apply this to the multiplication problem (37), we partition the partial sums into sets of three and perform this step in parallel for all the sets. This reduces the problem of adding n numbers to the problem of adding $\frac{2n}{3}$ numbers. We repeat this step until we have only two numbers, then we just add them using the $O(\log n)$ time addition algorithm described above. After the first stage, we have $\frac{2}{3}n$ numbers; after the second stage, $(\frac{2}{3})^2 n$, and so on. The number of numbers decreases geometrically, thus there are only $O(\log n)$ stages. Each stage takes $O(1)$ time and $O(n^2)$ processors.

30.3 Integer Division

We wish to do integer division with remainder in *NC*. That is, given binary numbers s and t, compute the unique quotient q and remainder r such that $s = qt + r$ and $0 \le r < t$.

Our algorithm is based on *Newton's method*, a useful technique for approximating roots of differentiable functions. Newton's method works as follows. Starting from an initial guess x_0, compute a sequence of approximations

$$x_{i+1} \;=\; x_i - \frac{f(x_i)}{f'(x_i)}\,, \qquad (38)$$

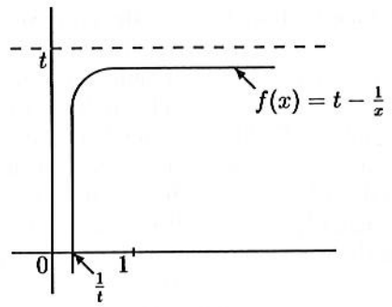
where $f' = df/dx$. For real-valued functions of a real variable, this is equivalent to finding the line tangent to the curve $y = f(x)$ at x_i and taking x_{i+1} to be the point where that line intersects the x axis.

In general, Newton's method is not guaranteed to converge to a root. However, if the function is well-behaved and the initial guess x_0 is close enough to a root, then the method converges very quickly: the number of bits of accuracy roughly doubles with each iteration. In this application, although we are using an approximation technique, we will be using only exact binary arithmetic (no floating point), and will obtain an exact solution.

We first show how to approximate the reciprocal $\frac{1}{t}$ of a given number t in binary. We will do this by approximating the root of the function

$$f(x) \;=\; t - \frac{1}{x}$$

using Newton's method.



In this case, $f'(x) = x^{-2}$, and (38) becomes

$$x_{i+1} = 2x_i - tx_i^2 .$$

We take as our first approximation x_0 the unique fractional power of 2 in the interval $(\frac{1}{2t}, \frac{1}{t}]$. This can be found in $O(\log n)$ time by finding the unique power of 2 in the interval $[t, 2t)$ and taking its reciprocal by reversing the order of the binary digits and placing a binary point after the first 0. We then iterate Newton's method to get the sequence of approximations $x_0, x_1, x_2, \ldots$. These approximations blast in toward $\frac{1}{t}$ quickly: we start with an error of at most $\frac{1}{2t}$, and at each step we roughly square the error, thus doubling the number of bits of accuracy. This is called *quadratic convergence.*

Lemma 30.2 *The sequence $x_0, x_1, \ldots$ obtained from Newton's method is non-decreasing and converges quadratically to $\frac{1}{t}$.*

Proof. By definition,

$$\frac{1}{2t} < x_0 \leq \frac{1}{t} ,$$

or in other words,

$$0 \leq 1 - tx_0 < \frac{1}{2} .$$

For $i \geq 0$,

$$\begin{aligned} 1 - tx_{i+1} &= 1 - t(2x_i - tx_i^2) \\ &= (1 - tx_i)^2 . \end{aligned}$$

It follows by induction that

$$\begin{aligned} 1 - tx_i &= (1 - tx_0)^{2^i} \\ &< 2^{-2^i} , \end{aligned}$$

thus

$$\frac{1}{t} - x_i < \frac{1}{2^{2^i} t} .$$

From these facts we can conclude that

$$\frac{1}{2t} < x_0 \leq x_1 \leq x_2 \leq \cdots \leq \frac{1}{t} .$$

□

After $k = \lceil \log\log \frac{s}{t} \rceil$ iterations we have

$$1 - tx_k < \frac{t}{s} .$$

From this and the fact that $x_k \leq \frac{1}{t}$ we have that

$$0 \leq \frac{s}{t} - sx_k < 1 .$$

Therefore the desired integer part of $\frac{s}{t}$ is either $\lfloor sx_k \rfloor$ or $\lceil sx_k \rceil$, and the remainder can be found by subtracting.

Each Newton iteration took $O(\log n)$ time (we did not do enough iterations to let the numbers get too big) and we needed $\log\log \frac{s}{t} = O(\log n)$ iterations.

For some interesting ramifications of the division problem, including an $O(\log n)$-depth circuit for integer division under a slightly weaker uniformity condition, see [9].