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 nth 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

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

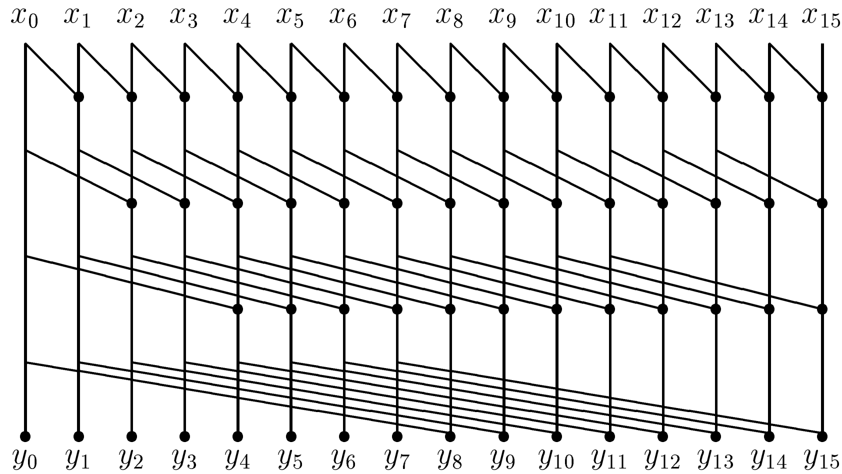
In parallel, compute the n^3 products $A_{ik}B_{kj}$ for each triple i, j, k. This can be done in one step, since we have n^3 processors. Then allocate n processors to each pair i, j and compute the sums (36) from the data computed in the first step. This sum can be obtained in $\log n$ time in parallel 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, $0 \leq i \leq n-1$, where

$$y_i \quad = \quad x_0 \cdot x_1 \cdot x_2 \cdots x_i \; .$$

Consider the following circuit with n input gates and n output gates. The ith input gate receives x_i and the ith 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

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

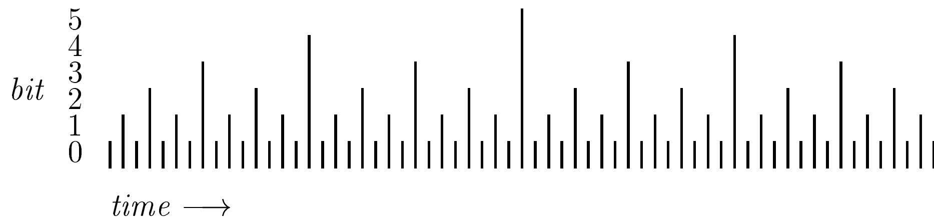
the corresponding sequence of strings in the binary representation is obtained by starting from $0 \ldots 0$ and successively adding 1 in binary. For example, for $n = 4$ we get the sequence

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

In the Gray representation, the sequence is

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

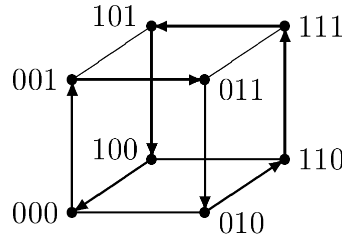
Each element is obtained from the last by flipping one bit. If we graph the sequence of bits that are flipped, the picture looks similar to an English ruler with demarcations for inches, half inches, quarter inches, and so forth.



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 jth bit of g_i (counting from the left and starting at 0) is the exclusive-or of the jth and $j - 1^{\text{st}}$ bits of b_i, and the jth bit of b_i is obtained from g_i by taking the exclusive-or of the jth 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.