Chapter 6 : Parallel computation

That certain problems are solvable in polynomial time does not seem to satisfy us. Obviously, a useful problem should consult its entire input, leading to a linear lower time bound. We would still like to solve problems in sub-linear time, of course, not using conventional computers, but using parallel versions of them. A parallel computer can process different instructions or operations (including reading the input) simultaneously and may bring down the (parallel) running time of a problem to sub-linear, say poly-log. Informally, a problem which has some such algorithms is called *massively parallelizable*, whereas a problem which parallelization is not known to help much is called *inherently sequential*. This chapter is an introduction to the theory of classification of problems based on their parallelizability potentials.

6.1 Boolean circuits

To start with we require a good model of parallel computers. A *parallel random access machine* (PRAM) is typically the most widely used model. A PRAM has many simple processors communicating through a shared memory. For our purpose, however, these machines are too complicated to deal with mathematically. So we stick to another model known as Boolean circuits and built of logic gates. Most modern processors have CPUs consisting of a vast number of such gates and so our model is not unrealistic. But programming at the gate level is clumsy. Let us accept this drawback of Boolean circuits to promote simpler reasoning.

Boolean circuits have AND, OR and NOT gates (capable of computing Boolean AND, OR and NOT functions of individual bits) connected by wires and having no loops (feedbacks). Input bits are given from an external source. Gates receiving these bits compute the desired outputs and pass the computed bit values on to the next level of gates. Eventually, computation ends at a specific *output gate*. The bit value computed by this output gate is designated as the value computed by the circuit on the given input.

6.1 Example Figure 6.1 describes the computation of the three-bit parity function by a Boolean circuit. The figure also describes the propagation of the bit values through the gates for the input setting $x_1x_2x_3 = 010$.

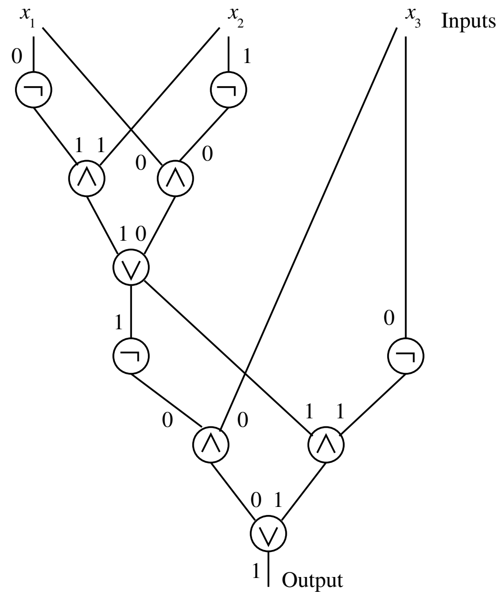
A Boolean circuit C on n inputs $x_1, \ldots, x_n$ computes a function $f_C : \{0,1\}^n \to \{0,1\}$ in the sense that if C outputs the bit value b for the input setting $a_1, \ldots, a_n$, we say $f_C(a_1, \ldots, a_n) = b$. We may often go lazier to say the same as $C(a_1, \ldots, a_n) = b$. We also conceive of Boolean circuits computing functions $\{0,1\}^n \to \{0,1\}^k$ using k output gates.

We now want to employ Boolean circuits to recognize languages. A language may have strings of any length, whereas a particular Boolean circuit has a fixed number of input variables. So we use a family $(C_0, C_1, C_2, \ldots)$ of Boolean circuits, where for each n the circuit C_n has provision for exactly n input bits. The family recognizes the language L in the sense that $\alpha \in L$ if and only if $C_n(\alpha) = 1$, where $n = |\alpha|$ and where the bits of α are naturally treated as the settings of the input variables of C_n.[6.1]

6.2 Definition The size of a Boolean circuit C is the number of gates in it. The depth of C is the maximum number of gates in a path from an input variable to the output. Two circuits C and C' on the same input variables are called equivalent, if they compute the same function, i.e., output the same bit values for the same input settings. C is called size minimal (resp. depth minimal), if no circuit of smaller size (resp. depth) is equivalent to it. The size complexity (resp. depth complexity) of a

[6.1] Every language can be encoded in binary. Unless otherwise stated, we assume throughout this chapter that all languages are subsets of $\{0,1\}^*$. Binary encoding changes the length of a string over an arbitrary alphabet only by a constant factor and so does not affect complexity results.

Figure 6.1: A Boolean circuit computing the three-bit parity function



circuit family $(C_0, C_1, C_2, \ldots)$ is the function $f : \mathbb{N}_0 \to \mathbb{N}_0$ such that $f(n)$ is the size (resp. depth) of C_n. The family is called size minimal (resp. depth minimal), if each C_n is size minimal (resp. depth minimal). The circuit size complexity (resp. circuit depth complexity) of a *language* $L \subseteq \{0,1\}^*$ is the size complexity (resp. depth complexity) of a size minimal (resp. depth minimal) circuit family that realizes L.

6.3 Example As a continuation of the Example 6.1 let us realize a circuit family for the language

$$\mathrm{PARITY} := \{\alpha \in \{0,1\}^* \mid \text{The number of } 1 \text{ bits in } \alpha \text{ is odd}\}.$$

Figure 6.1 describes how two NOT gates, two AND gates and one OR gate can be used to build the two-bit parity or the XOR function. We denote this circuit by an XOR gate $\oplus$. Using $n-1$ such XOR gates one can realize the n-bit parity circuit C_n as described in Figure 6.2. This circuit C_n has $n-1$ XOR gates, i.e., $5(n-1)$ basic (AND, OR, and NOT) gates, and so its size is $\mathrm{O}(n)$. Its depth is also $\mathrm{O}(n)$ — the longest path is from x_1 to output. Thus the size complexity of PARITY is $\mathrm{O}(n)$ and its depth complexity is also $\mathrm{O}(n)$. But these circuits for PARITY are *not* depth minimal. One can, in fact, construct a height-balanced binary tree on n leaves. Such a tree can also realize the n-bit parity function (with the inputs at the leaves and the output at the root), but with a depth of only $\mathrm{O}(\lg n)$. A realization of the eight-bit parity function with logarithmic depth is shown in Figure 6.3. It follows that PARITY has a minimal depth complexity $\mathrm{O}(\lg n)$. It is also clear that this height-balanced circuit family continues to have a size complexity of $\mathrm{O}(n)$.

Though circuit families handle individual languages, we are still not happy to take any arbitrary family as a representative of a language. The reason is that a typical computer program should have a *generic* way to handle inputs of different lengths. One simply should not stand an infinitely long program with branching for all possible values of $n \in \mathbb{N}_0$. In other words, the different circuits C_n in a family must have a *uniform description* independent of n. Moreover, this description should make it easy to manufacture the circuit C_n, after n is obtained from the input. We require this manufacturing doable by an algorithm.

6.4 Definition A circuit family $(C_0, C_1, C_2, \ldots)$ is called uniform, if a log-space transducer exists, that terminates with $\langle C_n \rangle$ on its work-tape, when started with 1^n as input (on its read-only input tape).

Figure 6.2: A Boolean circuit computing the n-bit parity function

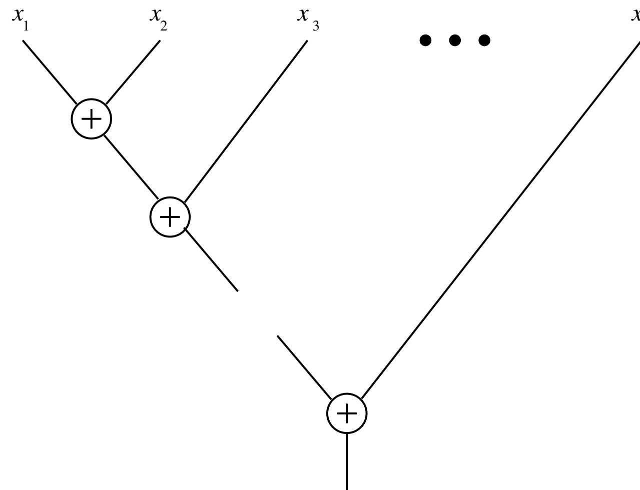
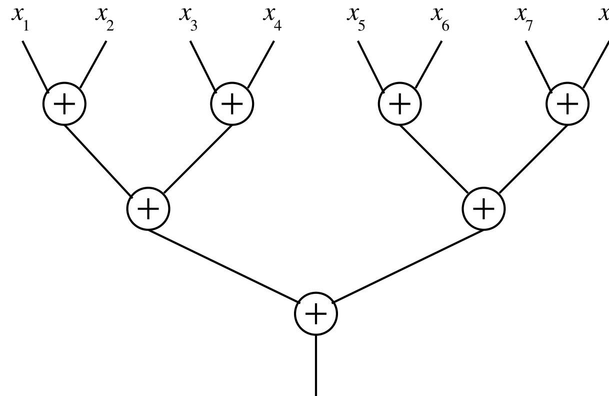


Figure 6.3: A Boolean circuit computing the 8-bit parity function



A uniform Boolean circuit family is henceforth treated as a model of a parallel computer for the language it recognizes. The size complexity of the family is a measure on the total number of elementary operations needed by the algorithm, whereas the depth complexity of the family reflects the *parallel running time* of the algorithm. This formalism is a departure from our previous efforts of modifying the basic Turing machine model to suit our different specific needs. However, as long as we confine ourselves inside the complexity class P, this deviation is not a serious issue, as the following theorem indicates.

6.5 Theorem A language $L \in \mathrm{TIME}(f(n))$ with $f(n) \geqslant n$ has a circuit size complexity of $\mathrm{O}(f^2(n))$.

Circuits may play an important role in settling the $\mathrm{P} = \mathrm{NP}$? issue. A Boolean circuit is called satisfiable, if some setting of the input variables leads C to output 1. It can be proved independently that the language

$$\text{CIRCUIT-SAT} := \{\langle C\rangle \mid C \text{ is a satisfiable Boolean circuit}\}$$

is NP-complete. As a corollary to this result, one obtains an alternate proof of the Cook-Levin theorem (the NP-completeness of SAT and 3SAT). These developments are not surprising, since our real computers are typically built from Boolean circuits and so circuits provide a significantly similar (to TMs) way to look at computation. We will not study these topics further in this course, but come back to parallelization issues.

6.6 Definition A language L is said to have a (simultaneous) size-depth complexity of $(f(n), g(n))$, if a uniform circuit family with size complexity $f(n)$ and depth complexity $g(n)$ realizes L.

By Example 6.3 the language PARITY has a size-depth complexity of $(O(n), O(\lg n))$. Thus PARITY is highly parallelizable in the sense that its parallel running time is logarithmic in the input size.

6.7 Example A circuit family can also *compute* a function instead of recognizing languages. Let $A = (a_{ij})$ and $B = (b_{jk})$ be two $m \times m$ Boolean matrices. Their product is defined to be the $m \times m$ matrix $C = (c_{ik})$, where $c_{ik} = \bigvee_{j=1}^{m} (a_{ij} \wedge b_{jk})$. The input size is $n = 2m^2$. A circuit may first compute all the quantities $d_{ijk} := a_{ij} \wedge b_{jk}$ in parallel using m^3 OR gates in only one unit of parallel running time. Then for each pair (i, k) of indices in parallel, the circuit computes the m-ary OR $c_{ik} = \bigvee_{j=1}^{m} d_{ijk}$. Like the m-ary XOR, the computation of each c_{ik} can be achieved using $O(m)$ size and $O(\lg m)$ depth. Thus Boolean matrix multiplication has a size-depth complexity of $(O(n^{3/2}), O(\lg n))$.

6.8 Example Let again $A = (a_{ij})$ be an $m \times m$ Boolean matrix. The transitive closure of A is the Boolean matrix $B := A \vee A^2 \vee \cdots \vee A^m$, where OR of two matrices means elementwise Boolean OR. By the previous example we know how to multiply two $m \times m$ matrices in parallel — call this circuit M_m. Now consider a circuit that computes for each $i \in \{1, 2, \ldots, m\}$ in parallel the matrix A^i using $O(i)$ invocations of M_m and in $O(\log i)$ depth of these invocations. Since M_m has a complexity of $(O(m^3), O(\lg m))$, computation of all of $A, A^2, \ldots, A^m$ can be achieved in $O(m^5)$ size and $O(\lg^2 m)$ depth. Next we have to logically OR the matrices. We run for each pair (i, j) of indices in parallel a circuit that computes the m-ary OR of the (i, j)-th elements of the intermediate products $A, A^2, \ldots, A^m$. Each sub-circuit for (i, j) requires $O(m)$ size and $O(\log m)$ depth, yielding a total size of $O(m^3)$ and depth of $O(\log m)$ for this OR stage. Combining the multiplication and OR stages implies that the computation of the transitive closure has a size-depth complexity of $(O(n^{5/2}), O(\lg^2 n))$.

Note that if A is the adjacency matrix of a directed graph G and B the transitive closure of A, then $b_{ij} = 1$ if and only if there is a path from vertex i to vertex j in G.

6.2 Nick's complexity classes

So far we have seen many examples of parallel computations in $O(n^i)$ size and $O(\lg^j n)$ depth. We give a special name to such computations.

6.9 Definition A language L is said to be in the class NC^j, if a uniform Boolean circuit family of size-depth complexity $(O(n^i), O(\lg^j n))$ realizes L. Here the choice of i is left arbitrary and the j in NC^j refers to the exponent in the depth complexity. We also define

$$\mathrm{NC} := \bigcup_{j \in \mathbb{N}} \mathrm{NC}^j.$$

NC is an abbreviation for Nick's complexity class. Nicholas Pippenger was the first to study these classes. Languages in NC are *highly parallelizable*. $\mathrm{NC}^1 \subseteq \mathrm{NC}^2 \subseteq \mathrm{NC}^3 \subseteq \cdots$ form a hierarchy of languages with union equal to NC. For $L \in \mathrm{NC}$ a sequential simulation of an NC circuit for L can be evidently done in polynomial time. We can also prove relations of NC classes with the log-space classes.

6.10 Theorem $\mathrm{NC} \subseteq \mathrm{P}$.

6.11 Theorem $\mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{NC}^2$.

6.3 P-completeness

We have $\mathrm{NC} \subseteq \mathrm{P}$. But are these two classes equal? The answer is not known. $\mathrm{NC} = \mathrm{P}$ implies that every poly-time solvable problem is massively parallelizable. This is a bit surprising, since many problems appear to be inherently sequential. With our usual attempt of separating NC from P we define P-complete problems. Doing that requires the closure of the class NC under log-space reduction.

6.12 Theorem Let $A \leqslant_{\mathrm{L}} B$ and $B \in \mathrm{NC}$. Then $A \in \mathrm{NC}$ too.

Proof Let f be a log-space reduction from A to B. For an input instance $\alpha = a_1 \ldots a_n$ of A we first compute the string $f(\alpha) = b_1 \ldots b_l$ for B. Since f is a log-space reduction, the language that contains $\langle \alpha, j \rangle$ if and only if $b_j = 1$ is in L and so in NL and so in NC^2. Thus we can compute all the bits $b_1, \ldots, b_l$ of $f(\alpha)$ in parallel using NC^2 circuits. We finally feed the bits $b_1, \ldots, b_l$ to an NC circuit for B. ◀

6.13 Corollary Let $A \leqslant_{\mathrm{L}} B$ and $B \in \mathrm{NC}^j$ for $j \geqslant 2$. Then $A \in \mathrm{NC}^j$ too.

6.14 Definition A language L is called P-complete, if it satisfies the following two conditions:

(1) $L \in \mathrm{P}$.

(2) $A \leqslant_{\mathrm{L}} L$ for every $A \in \mathrm{P}$.

6.15 Corollary If a P-complete problem is in NC, then $\mathrm{NC} = \mathrm{P}$.

6.16 Example The following languages are P-complete:

$$\begin{aligned} \text{CIRCUIT-VALUE} &:= \{\langle C, \alpha \rangle \mid C \text{ is a Boolean circuit and } C(\alpha) = 1\}. \\ \text{ODD-MAX-FLOW} &:= \{\langle G, s, t, c \rangle \mid \text{ The maximum flow from } s \text{ to } t \text{ in the network } G \text{ is odd}\}. \end{aligned}$$

In the second example c stands for the capacity matrix for G and consists of positive integral entries.

6.4 Randomized parallel classes

Many interesting problems have been identified as P-complete.[6.2] With the reasonable belief that these problems cannot be effectively parallelized, it is noteworthy to look at their approximate and/or heuristic parallel versions. Randomization also helps here. Like the passage from P to RP we define a randomized version of the class NC. Certain problems that do not have known good deterministic parallel algorithms can now be parallelized using randomization. Some probability of error, that can be made vanishingly small, is all the price we have to pay.

6.17 Definition A language L is said to be in the class RNC if there is a uniform family $(C_0, C_1, C_2, \ldots)$ of Boolean circuits with the following properties: The circuit C_n that is meant for input strings α of length n takes as input a string $\alpha\beta$ with $|\alpha| = n$ and $|\beta| = p(n)$, where $p(n)$ is some fixed polynomial function of n. The string β supplies the random bits needed for the computation of C_n. If $\alpha \in L$, at least half of

[6.2] For a good collection look at the book 'Limits to parallel computation: P-completeness theory' by Greenlaw, Hoover and Ruzzo, Oxford University Press, 1995. The book is available for free download from the following web-sites:

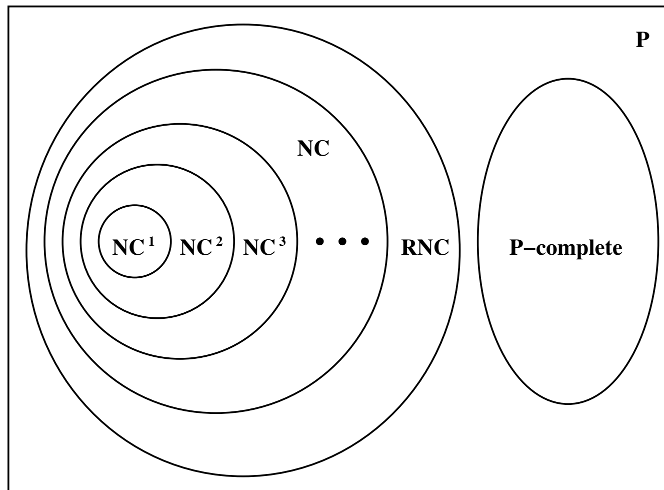
http://www.cs.armstrong.edu/greenlaw/research/PARALLEL/
http://www.cs.ualberta.ca/~hoover/P-complete/
http://www.cs.washington.edu/homes/ruzzo/

the $2^{p(n)}$ possible settings for β lead the circuit C_n to output 1, whereas if $\alpha \notin L$, no setting for β lets the circuit output 1.

The RP algorithm for SYMBOLIC-DET, that we described in Chapter 5, can be implemented by an NC circuit family. So SYMBOLIC-DET $\in$ RNC.

The following figure explains the relations among the parallel complexity classes introduced in this chapter. All containments are conjectured to be proper.

Figure 6.4: Parallel complexity classes



Exercises for Chapter 6

1. Show that the 2-bit parity function can be realized using only four basic (AND, OR and NOT) gates, but cannot be realized by only three basic gates.

2. Show that if $\mathrm{NC}^{j+1} = \mathrm{NC}^j$ for some $j \geqslant 1$, then $\mathrm{NC} = \mathrm{NC}^j$.

*** 3.** A Boolean circuit is called *monotone*, if it does not contain NOT gates. Use reduction from CIRCUIT-VALUE to show that the language

$$\text{MONOTONE-CIRCUIT-VALUE} := \{\langle C, \alpha\rangle \mid C \text{ is a monotone Boolean circuit and } C(\alpha) = 1\}$$

is P-complete.

4. In the text we have considered OR and AND gates receiving only two input bits. Now think of OR and AND gates that can take any number of inputs and compute the logical OR or AND of all the input bits in one unit of time. Such gates are said to have *unbounded fan-in*. Define the complexity class AC^j, $j \geqslant 0$, to comprise those languages that have uniform families of Boolean circuits built from gates of unbounded fan-in and having size-depth complexity $(\mathrm{O}(n^i), \mathrm{O}(\lg^j n))$. Also define $\mathrm{AC} := \bigcup_{j \in \mathbb{N}_0} \mathrm{AC}^j$. Prove the following assertions:

(a) $\mathrm{AC}^j \subseteq \mathrm{NC}^{j+1} \subseteq \mathrm{AC}^{j+1}$ for every $j \geqslant 0$.

(b) $\mathrm{AC} = \mathrm{NC}$.

(c) If $\mathrm{AC}^{j+1} = \mathrm{AC}^j$ for some $j \geqslant 0$, then $\mathrm{AC} = \mathrm{AC}^j$.

*** 5.** [*Binary addition*] Design an AC^0 circuit that, given $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}$ as input, *computes* the bits $z_0, \ldots, z_n$, where $(z_n z_{n-1} \ldots z_1 z_0)_2 = (x_{n-1} \ldots x_1 x_0)_2 + (y_{n-1} \ldots y_1 y_0)_2$. In particular, binary addition is in NC^1. (**Hint:** Compute all the carries in parallel using an AC^0 circuit.)