3 Circuits and Parallel Computation

3.1 Circuit complexity

Combinational circuits have been widely studied as a universal model of computation which is in some sense closer to reality than the Turing machine. A *Boolean circuit* is a directed acyclic graph. Nodes with in-degree 0 are called *input nodes*, and there is a distinguished node with out-degree 0 called the *output node*. Each non-input node has in-degree 2 or 1 and is labelled with an element of the set $\{\wedge, \vee, \neg\}$. We allow arbitrary fan-out and the constants $\{\text{True}, \text{False}\}$ as inputs. Associating True with 1 and False with 0, we can view a Boolean circuit C as computing a function $f_C : \{0,1\}^n \to \{0,1\}$ in the obvious way, where n is the number of (non-constant) input nodes and $f_C(w_1 \dots w_n) = 1$ iff assigning values $w_1, \dots, w_n \in \{0,1\}$ to the input nodes causes the output node to have value 1. The *language accepted by* C is the set of strings $w \in \{0,1\}^n$ for which $f_C(w) = 1$. The *size* of a circuit is the number of non-input nodes (or *gates*), and the *depth* is the length of a longest path from an input node to the output node.

For a language $L \subseteq \{0,1\}^*$, define $L^{(n)} = L \cap \{0,1\}^n$. A family $\{C_n : n \in \mathbb{N}\}$ of circuits accepts L iff C_n accepts $L^{(n)}$ for each n. The *circuit complexity* of L is a function $g : \mathbb{N} \to \mathbb{N}$ such that, for each n, the size of a smallest circuit accepting $L^{(n)}$ is $g(n)$. L is said to have *polynomial size circuits* if its circuit complexity is bounded above by a polynomial.

Exercise 3.1. Design a family of circuits which accepts the language of palindromes over $\{0,1\}$; express the size and depth of your circuits as functions of the input size n.

By allowing multiple output nodes, circuits may be used to compute more general functions $f_C : \{0,1\}^n \to \{0,1\}^k$, and circuit families functions $f : \{0,1\}^* \to \{0,1\}^*$. This is analogous to viewing a Turing machine as a transducer rather than as a simple language recogniser.

Exercise 3.2. Design circuit families which compute (i) the scalar product of two Boolean n-vectors (presented as a string of $2n$ bits), in size $\mathrm{O}(n)$ and depth $\mathrm{O}(\log n)$; (ii) the product of two $n \times n$ Boolean matrices (presented as a string of $2n^2$ bits), in size $\mathrm{O}(n^3)$ and depth $\mathrm{O}(\log n)$; (iii) the transitive closure of an $n \times n$ Boolean matrix, in size $\mathrm{O}(n^3 \log n)$ and depth $\mathrm{O}((\log n)^2)$. [NOTE: In the Boolean domain, multiplication is taken to mean $\wedge$, and addition to mean $\vee$. Thus the scalar product of two n-vectors $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_n)$ is given by $\bigvee_i (x_i \wedge y_i)$. Also, the transitive closure of an $n \times n$ Boolean matrix A can be written $I + A + A^2 + \cdots + A^{n-1}$, where I denotes the identity matrix.]

In the remainder of this section, we explore the relationship between circuit size and Turing machine time.

Theorem 3.1. *If $L \subseteq \{0,1\}^*$ belongs to* P *then L has polynomial size circuits.*

Proof. Let M be a polynomial time-bounded one-tape deterministic TM with input alphabet $\{0,1\}$ accepting L, and let Q, Γ denote the sets of states and tape symbols respectively of M. For each n, we will construct a small circuit C_n which simulates M on inputs of size n.

The design of the circuit is based on the familiar idea of a tableau for M. It is convenient here to alter slightly our definition of a configuration by encoding the state together with the contents of the scanned square in a composite symbol. Thus the configuration previously written $\alpha q \beta$ now becomes $\langle\alpha_1, \sqcup\rangle \dots \langle\alpha_r, \sqcup\rangle\langle\beta_1, q\rangle\langle\beta_2, \sqcup\rangle \dots \langle\beta_s, \sqcup\rangle$, where $\alpha_i, \beta_i \in \Gamma$ are the symbols of α and β, $q \in Q$ is the state, and "$\sqcup$" denotes absence of the head at a tape square. A tableau for M on an input $w \in \{0,1\}^n$ is a table whose $p(n)+1$ rows, each of length $p(n)+1$, correspond to successive configurations of M on w, starting with the initial configuration. Note that halting configurations are assumed to repeat, so that all computations of M on w can be taken to have length exactly $p(n)$.

The circuit C_n is composed of three segments: a *computation segment*, an *input segment* and an *output segment*. (See Figure 1.) The computation segment is constructed from multiple copies of a basic block of gates, one for each square of the tableau except those in the initial row. The basic block is identical for all values of n. It takes as inputs a binary encoding of three adjacent squares $(i, j-1)$, (i, j) and $(i, j+1)$ of a tableau, and produces as output a binary encoding of the single square $(i+1, j)$ of the next row. (For squares at the ends of rows, a similar block with only two input squares is used.) This can clearly be implemented by a fixed piece of circuitry since it just involves hard-wiring the transition function of M. (Note that the symbol in the output square $(i+1, j)$ is uniquely determined by those in the three input squares.) Since the basic block has size $\mathrm{O}(1)$, the size of the whole computation segment is $\mathrm{O}(p(n)^2)$.

The input segment of the circuit converts the input (n binary digits transmitted through the non-constant input nodes) into a binary encoding of the initial row of the tableau. This can be done for all inputs of size n by $p(n)+1$ fixed blocks of gates of total size $\mathrm{O}(p(n))$. Finally, the output segment is composed of two parts: first, for each square in the final row of the tableau there is another fixed block of gates which takes as input the binary encoding of the square contents, and produces the output 1 if the square contains the head and the state is accepting, and 0 otherwise. Second, the "or" of these outputs is computed by a tree of $\vee$-gates of size $\mathrm{O}(p(n))$, whose single output is the output of the entire circuit. The entire output segment again requires $\mathrm{O}(p(n))$ gates.

From the construction of the circuit C_n, it is clear that its output on input $w \in \{0,1\}^n$ is 1 iff M accepts w. Hence the family $\{C_n : n \in \mathbb{N}\}$ accepts L. Since the size of C_n is $\mathrm{O}(p(n)^2)$, we see that L has polynomial size circuits. $\square$

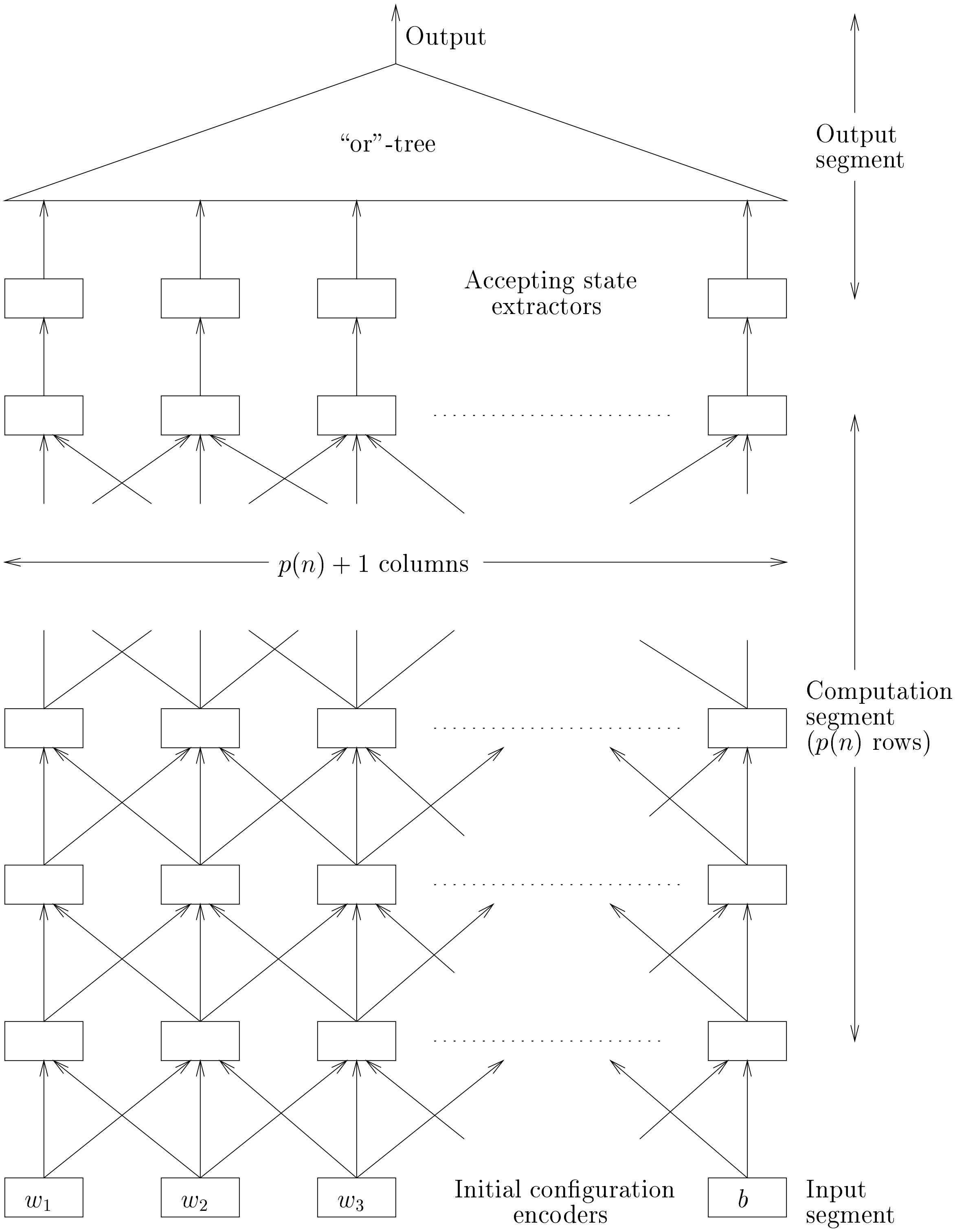

Figure 1: Circuit simulating a deterministic Turing machine

Note that the restriction to the alphabet $\{0,1\}$ is made for convenience only. We can easily generalise Theorem 3.1, and our other results concerning circuits, to languages over an arbitrary alphabet by fixing a suitable binary encoding of its symbols.

Exercise 3.3. (a) Sketch the construction of the fixed blocks of gates used in the above proof which implement (i) the transition function of M, and (ii) the test for accepting state.

(b) For a given $T(n)$ time-bounded deterministic TM M, the above proof shows how to construct a circuit family of size $\mathrm{O}(T(n)^2)$ which simulates M. Show how to construct a family of size only $\mathrm{O}(T(n)\log T(n))$ with the same property. (HINT: Use Lemma 1.5 to get an *oblivious* two-tape TM, then simulate it by a circuit using one block of gates per move.)

Is the converse of Theorem 3.1 true, i.e., does every language with polynomial size circuits lie in P? Actually, this fails even for the trivial reason that *all* languages over the unary alphabet $\{1\}$, including non-recursive languages, have polynomial size circuits. However, small circuit complexity does correspond in some sense to tractability, since the amount of hardware involved in the computation does not grow too fast with the input size. The essential difference between circuits and other models, such as Turing machines, is that the former are *non-uniform*, i.e., potentially radically different devices are required to process inputs of different sizes. Rather than measuring the amount of resources consumed by a *fixed* algorithm accepting an infinite language as a function of input size, circuit complexity measures the growth in the *size* of algorithm required to recognise finite portions of the language. Thus we may view the class of languages which have polynomial size circuits as a non-uniform analogue of P. Note that the fact that circuits can recognise non-recursive languages does not contradict the Church-Turing Thesis, since non-uniform models of computation cannot be regarded as "reasonable". In the next section, we will consider what happens when we impose a uniformity constraint on the circuit model.

It is worth observing that there is a definite limit to the circuit complexity of any language. Any Boolean function of n variables can be written in Disjunctive Normal Form (DNF), which can then be directly translated into a circuit of size $\mathrm{O}(n2^n)$. Thus the circuit complexity of any language cannot exceed $\mathrm{O}(n2^n)$. This is in direct contrast to the situation in respect of the Turing machine model, where the time hierarchy theorem assures us that complexity classes such as $\mathrm{DTIME}(2^{2^n})$ make perfect sense.

Among other things, circuit complexity has been investigated as a possible vehicle for settling the $\mathrm{P} =? \mathrm{NP}$ question. The idea is to find a language in NP which does not have polynomial size circuits, thus implying by Theorem 3.1 that $\mathrm{P} \neq \mathrm{NP}$. Since circuits are "more concrete" than Turing machines, such lower bounds may be easier to come by. Currently, however, no language in NP has been proved to require more than $3n$ gates.[8] Note also that a proof that SAT has polynomial size circuits would *not* imply that P = NP. It would, however, have a number of rather surprising consequences.

3.2 Uniform circuits

We noted above that the issue of uniformity meant we could not hope to prove a converse of Theorem 3.1. However, it turns out that uniformity is the *only* obstacle here. In order to make this claim precise, we could either add non-uniformity to the Turing machine model by supplying extra information (or "advice") which is a function of the input size, or impose uniformity on the Boolean circuit model. We shall adopt the latter approach. The first step is to introduce an encoding of Boolean circuits as words over a certain (fixed) alphabet. A straightforward technique is to encode an m-node circuit as a sequence of quadruples $\big(\langle s_i, g_i, l_i, r_i\rangle : 1 \le i \le m\big)$, each quadruple describing a different node in the circuit. The elements of each quadruple are binary integers with the following interpretations: s_i is the sequence number of the ith gate, g_i is the gate type (input, output, $\wedge$, $\vee$, or $\neg$), and l_i, r_i are the sequence numbers of the left and right operands (where applicable).

We say that a family $\{C_n : n \in \mathbb{N}\}$ of circuits is *uniform* if there is a log-space Turing machine transducer with input alphabet $\{1\}$ which, on input 1^n, writes to its output tape the encoding of the nth circuit C_n.

With this definition, we can turn Theorem 3.1 into a two-way correspondence.

Theorem 3.2. *Let L be a language over the alphabet $\{0,1\}$. Then $L \in \mathrm{P}$ if and only if L has polynomial-size uniform circuits.*

Proof. For the "only if" direction we use the construction of Theorem 3.1. The only fact we must check is that the circuits C_n—which simulate the operation of the TM on inputs of size n—form a *uniform* family. To make this claim plausible, note that each of the blocks that make up the computation segment of C_n is of fixed *shape*, with only the *sequence numbers* of gates varying between blocks. Thus it is possible to generate a description of the entire computation segment of C_n by taking a certain fixed length sequence of quadruples, corresponding to a single block of gates, and replicating that sequence as many times as is necessary. The sequence numbers of corresponding gates in different blocks of the circuit differ by a fixed offset. The description of the circuit C_n can thus be written out by a TM M_C which has, embedded in its finite control, a template for the building blocks of the circuit. The only bookkeeping which M_C need perform is involved in the computation of offsets. Since the number of blocks in C_n is bounded by a polynomial in n, the offsets (represented as binary integers) require only logarithmic space.

[8] Optimists take heart from a number of results from the mid-80s (the first by Razborov) proving exponential lower bounds for *restricted* circuit models.

The "if" direction is straightforward. Let M_C be a log-space Turing machine which generates descriptions of the circuits C_n, and modify M_C so that it writes the encoded circuit onto a worktape instead of an output tape. Since M_C is log-space bounded this operation takes polynomial time. The encoding describes a directed acyclic graph; a consistent order for the evaluation of the circuit can be determined by performing a topological sort of this graph: again, this can be done in polynomial time. Finally, the output from each gate can be computed in the sequence prescribed by the topological sort. □

Theorem 3.2 confirms that uniform circuits are a reasonable model of computation, when circuit size is used as a measure of (sequential) time. In the next section we look at uniform circuits as a model of *parallel* computation.

3.3 Parallel computation and the class NC

In the study of sequential computation, the choice of which model of computation to employ is not crucial. In this course, we chose to define the class P by reference to the Turing machine model; however, if we had chosen any other "reasonable" model of computation we would have been led to exactly the same class. In contrast, the choice of model in the study of parallel computation requires much greater care. It seems unlikely that the Turing machine can be extended in any clean way to provide an acceptable model for parallel computation. However, the uniform circuit model *does* provide a convenient and convincing foundation for the study of parallelism.

What should be the analogue of the class P in the world of parallel computation? Expressed in the language of circuits, one requirement is clear: the *depth* of circuits should grow slowly as a function of input size. (It is the depth of a circuit which determines the time taken for signals to propagate from inputs to outputs.) The second requirement is slightly more subtle. Consider, for example, the problem of determining whether an undirected graph on n vertices has a clique of size $k = \lfloor \log n \rfloor$. This problem can be solved by circuits of depth $\mathrm{O}((\log n)^2)$ by simply testing, for all possible k-subsets U of the vertex set of a graph, whether U is a clique. (The total number of subsets U to be tested is bounded above by $n^{\log n}$, so a balanced tree of $\vee$-gates of depth $\mathrm{O}((\log n)^2)$ suffices to combine the subresults of the computation.) However the *size* of the resulting circuits is super-polynomial, being $n^{\Omega(\log n)}$. We take the view that such an extravagant use of hardware does not constitute an efficient parallel solution to the problem. Therefore, we must bound the circuit size, as well as depth.

With these considerations in mind, we proceed to define the class NC of languages which can be recognised efficiently in parallel. (NC stands for "Nick's Class," the Nick in question being Nick Pippenger.) First define NC^k, for $k \in \mathbb{N}$, to be the class of all languages which are accepted by uniform circuits of polynomial size and depth $\mathrm{O}((\log n)^k)$. Then let $\mathrm{NC} = \bigcup_{k \geq 0} \mathrm{NC}^k$. Thus NC is the class of problems which can be solved in "super-fast" (i.e., poly-logarithmic) parallel time using a reasonable (i.e., polynomial) amount of hardware. There is a wide consensus that NC should play the same role in parallel computation as P does in sequential computation.

It is immediate from Theorem 3.2 that NC $\subseteq$ P. Presumably this inclusion is proper, since we do not believe that all problems which can be solved efficiently on a sequential machine also admit an efficient parallel solution. We shall return to this issue in Section 3.5.

3.4 Small depth circuits for addition and multiplication

In practice, the classes which assume the greatest importance in the theory of parallel computation are NC^1, NC^2, and NC itself. It is not too difficult to check (see Exercise 3.2) that multiplication of $n \times n$ Boolean matrices is in NC^1, and that transitive closure of an $n \times n$ Boolean matrix is in NC^2.[9] In this section, we study the parallel complexity of elementary arithmetic operations on binary integers.

Consider the problem of adding two n-bit binary integers to yield an $(n+1)$-bit result. The usual sequential algorithm for addition does not immediately provide an efficient parallel algorithm, since a "carry" may be propagated as many as n positions. (A worst-case example is the addition of the two n-bit numbers $1000\cdots001$ and $0111\cdots111$.) However, binary addition is within the lowest parallel complexity class NC^1, as we shall now demonstrate.

Suppose that the numbers $x_{n-1}\cdots x_2x_1x_0$ and $y_{n-1}\cdots y_2y_1y_0$ are to be added to produce the result $z_nz_{n-1}\cdots z_2z_1z_0$. For $0 \le i \le n-1$, let c_i denote the carry bit generated at the ith position, and define $g_i = x_i \wedge y_i$ and $p_i = x_i \vee y_i$. We can interpret g_i as meaning that a carry bit is generated at position i whether or not there is a carry into that position, and p_i as meaning that a carry into position i will be propagated. Thus $c_i = g_i \vee (c_{i-1} \wedge p_i)$ and hence

$$c_i = g_i \vee (g_{i-1} \wedge p_i) \vee (g_{i-2} \wedge p_{i-1} \wedge p_i) \vee \cdots \vee (g_0 \wedge p_1 \wedge p_2 \wedge \cdots \wedge p_i). \qquad (*)$$

The result $z_nz_{n-1}\cdots z_2z_1z_0$ can be computed by a log-depth circuit using the following strategy:

(1) compute $g_i = x_i \wedge y_i$ and $p_i = x_i \vee y_i$ for all $0 \le i \le n-1$;

(2) compute all conjunctions of the form $g_i \wedge p_{i+1} \wedge p_{i+2} \wedge \cdots \wedge p_j$, where $0 \le i < j \le n-1$;

[9] Note that this is an abuse of notation, since these problems involve the computation of a function from $\{0,1\}^*$ to $\{0,1\}^*$, while the classes NC^k, in common with all the other classes in this course, are defined in terms of language recognition. However, in this section only we will deliberately blur this distinction and assume that the circuits in the definition of NC^k have been extended in the obvious way to have multiple output nodes.

(3) for all $0 \le i \le n-1$, compute c_i using identity $(*)$;

(4) for all $0 \le i \le n$, compute z_i using $z_i = x_i \oplus y_i \oplus c_{i-1} = (x_i \wedge y_i \wedge c_{i-1}) \vee (x_i \wedge \neg y_i \wedge \neg c_{i-1}) \vee (\neg x_i \wedge y_i \wedge \neg c_{i-1}) \vee (\neg x_i \wedge \neg y_i \wedge c_{i-1})$.

(In the final step we are assuming that $c_{-1} = 0$.) Steps (1) and (4) can be done in parallel by constant depth circuitry, while steps (2) and (3) can be done in logarithmic depth using balanced trees of $\wedge$- and $\vee$-gates. The total number of gates is clearly polynomially bounded; moreover, the circuit is quite regular and can be generated by a log-space bounded Turing machine. Thus, addition of binary integers is in NC^1.

Now consider the multiplication of two n-bit binary integers, $x_{n-1} \cdots x_2 x_1 x_0$ and $y_{n-1} \cdots y_2 y_1 y_0$ to produce a $2n$-bit result $z_{2n-1} \cdots z_2 z_1 z_0$. In ordinary long multiplication, this task is reduced to the addition of n integers, each of $2n-1$ bits. Arranging these n additions in a balanced binary tree of depth $\log n$, and expanding each addition into the logarithmic-depth circuit described above, we see that binary multiplication is in NC^2. It is, however, possible to do better.

Suppose $\mathbf{p} = p_{m-1} \cdots p_2 p_1 p_0$, $\mathbf{q} = q_{m-1} \cdots q_2 q_1 q_0$, and $\mathbf{r} = r_{m-1} \cdots r_2 r_1 r_0$ are arbitrary m-bit numbers. Let $\mathbf{s} = s_{m-1} \cdots s_2 s_1 s_0$ be the result of adding $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ but ignoring any carries, and let $\mathbf{c} = c_{m-1} c_{m-2} \ldots c_2 c_1 0$ be the sequence of generated carry bits displaced one place left. Formally, $s_i = p_i \oplus q_i \oplus r_i$ and $c_i = (p_i \wedge q_i) \vee (p_i \wedge r_i) \vee (q_i \wedge r_i)$. Observe that $\mathbf{p} + \mathbf{q} + \mathbf{r} = \mathbf{s} + \mathbf{c}$, and that $\mathbf{s}$ and $\mathbf{c}$ can be computed from $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{r}$ by a constant depth circuit. This is the "2-for-3 trick": in constant parallel time, the addition of three binary integers is to reduced the addition of two.

Now return to long multiplication, which requires the addition of n binary integers. Applying the 2-for-3 trick in parallel to triples of integers, we reduce the number of binary integers to be added to $n - \lfloor \frac{1}{3} n \rfloor = \lceil \frac{2}{3} n \rceil$. Iterating the 2-for-3 trick $\mathrm{O}(\log n)$, times, the original list of n numbers is reduced to just two. The remaining pair can be dealt with using the logarithmic-depth addition algorithm described earlier. Since each iteration of the 2-for-3 trick requires just constant depth, the overall depth of the resulting multiplication circuit is $\mathrm{O}(\log n)$. Again, the number of gates employed is polynomial, and the circuit can be constructed in logarithmic space. Hence, multiplication of n-bit integers is in NC^1.

3.5 P-completeness

We have observed that $\mathrm{NC} \subseteq \mathrm{P}$. Since there is no reason to suppose that equality holds, it is interesting to identify problems in $\mathrm{P} \setminus \mathrm{NC}$ under the assumption that this class is non-empty. One way to do this is to demonstrate that a language L is complete for P with respect to log-space reduction. Then, as we shall see in Theorem 3.4 below, we will know that $L \notin \mathrm{NC}$ unless $\mathrm{NC} = \mathrm{P}$. In other words, under the reasonable assumption that not all languages in P have super-fast parallel solutions, L is a language which can be recognised by a fast sequential algorithm but not by a super-fast parallel algorithm.

In order to justify the above remarks, we need the fact that log-space reduction does not take us outside the class NC.

Theorem 3.3. *The class* NC *is closed under log-space reductions. That is:* $L \in$ NC *and* $L' \leq_{\log} L$ *implies* $L' \in$ NC.

Exercise 3.4. (a) Prove that LOG $\subseteq$ NC^2, using the following outline. Suppose M is a log-space bounded TM, and consider the computation of M on some input x of length n. The potential configurations of M on input x can be thought of as the vertices of a directed graph, in which there is an edge from configuration γ to γ' iff M can make a move from γ to γ' with x as input. Now appeal to Exercise 3.2. Note that you have in fact proved the stronger result that NLOG $\subseteq$ NC^2.

(b) Prove Theorem 3.3. Essentially, this involves extending part (a), to show that a log-space transducer can be simulated by NC^2 circuits (with multiple outputs). Note that the composition of two NC circuit families is an NC circuit family.

With the aid of Theorem 3.3, it is now easy to see how P-completeness can be interpreted as evidence that a problem cannot be solved by any super-fast parallel algorithm.

Theorem 3.4. *Let L be a language that is complete for* P *with respect to log-space reduction. Then L cannot be in* NC *unless* NC = P.

Perhaps the simplest example of a problem which is P-complete is the circuit value problem:

$$\begin{aligned}\text{CVP} = \{\langle C, w\rangle : C &\text{ is a Boolean circuit, } w \text{ is an input for } C,\\ &\text{and the output of } C \text{ on input } w \text{ is True}\}.\end{aligned}$$

Theorem 3.5. *The circuit value problem* CVP *is* P*-complete w.r.t. log-space reduction.*

Proof. This follows almost immediately from the proof of Theorem 3.2. First, to see that any language $L \in$ P can be reduced to CVP, let M be a polynomial time-bounded deterministic TM accepting L and consider a transducer which, when given an input w for M, outputs the pair $\langle C_n, w\rangle$, where n is the length of w and C_n is (an encoding of) the circuit C_n simulating M constructed in Theorem 3.1. As observed in the proof of Theorem 3.2, C_n can be computed in log-space. Obviously, $\langle C_n, w\rangle \in$ CVP iff M accepts w, so we have a log-space reduction.

The fact that CVP $\in$ P follows from the fact that a TM can simulate the operation of a circuit on a given input in polynomial time by first performing a topological sort, as in the proof of Theorem 3.2. $\square$

It is debatable whether CVP is a "natural" problem, referring directly as it does to a model of computation. However, many natural problems are known to be P-complete. Here are two examples.

$$\text{CFLNE} = \big\{G : \text{ the language generated by context-free grammar } G \text{ is non-empty}\big\}.$$

and

$$\textsc{LinProg} = \big\{\langle A, b\rangle : \ A \in \mathbb{Z}^{m\times n},\ b \in \mathbb{Z}^m, \text{ and there exists } x \in \mathbb{Z}^n \text{ s.t. } Ax \le b\big\}.$$

(For a reminder about context-free languages and grammars, see Chapter 4 of Hopcroft & Ullman. Note that LINPROG is asking whether an linear programming problem has a feasible solution.)

Theorem 3.6. CFLNE *and* LINPROG *are both* P*-complete w.r.t. log-space reduction.*