Chapter 5

Parallelism

Descriptive complexity is inherently parallel in nature. This is a particularly delightful dividend of applying this form of logic to computer science. The time to compute a query on a certain parallel computer corresponds exactly to the depth of a first-order induction needed to describe the query. There is also a close relationship between the amount of hardware used — memory and processors — and the number of variables in the inductive definition.

Quantification is a parallel operation. The query $(\exists x)S(x)$ can be executed using n processors in constant parallel time. Processor p_i checks whether $S(i)$ holds for $i = 0, 1, \ldots, n-1$. Any p_i for which $S(i)$ does hold should write a one into a specified location in global memory that was originally zero.

The real world is inherently parallel. There are many atoms, molecules, cells, organisms, computers, factories, towns, countries, all working on their own. For a long time, however, computers were sequential devices having a single processor and thus executing one instruction at a time. Over recent years, we have vastly increased our ability to produce small, fast, inexpensive processors. It is thus possible to build large parallel computers as well as a large number of personal computers that can interact. One of the fundamental problems in computer science is how to make efficient and effective use of this dramatic proliferation of computing power, including many processors that we may use at once.

As researchers and practitioners struggle with the question of how to use many processors at once, numerous models of parallel computation have been developed. The main axis along which these models vary is how tightly coupled the processors are. One extreme is the parallel random access machine (PRAM) in which the

inter-connection pattern is essentially a complete graph. In this model, a word of memory can be sent from any processor to any other processor in the time it takes to perform a single instruction. The other extreme is distributed computation, in which many personal computers or work stations are connected via a network, which might be fairly fast and local — or it might be the Internet.

Both of these models are important, and neither is well enough understood. For general applications, it is still very difficult to effectively use a tightly coupled parallel computer or a distributed network of computers and gain a large speed up compared to doing the computation at a single uni-processor. Of course there have been great successes. Some problems, such as linear algebra, are very easy to parallelize. Other problems may be inherently sequential, but this remains to be seen.

In this chapter we study highly coupled parallelism, that is, parallelism as on a PRAM. We show that this model corresponds very closely and nicely with descriptive complexity. We see in particular that the optimal depth of inductive definitions of a query corresponds exactly to the optimal parallel time needed to compute the query on a PRAM. There is also a close relationship between the number of processors needed by the PRAM and the number of variables used in the inductive definition.

This connection between parallelism and descriptive complexity sheds a great deal of light on the parallel nature of computation. It is very natural to describe our queries via inductive definitions. It is enlightening that inductive depth corresponds exactly to parallel time.

Later in this chapter we also study other models of parallel computation namely circuit complexity and alternating Turing machines. We tie these other two parallel models together with PRAMs and descriptive complexity.

The net result of this approach is that we can see what is basic about parallel computation and what is merely model dependent; and, if we choose, we can understand the main issues via the quantifier depth and the number of variables needed for first-order descriptions of the queries of interest.

5.1 Concurrent Random Access Machines

In this section we define a precise model of PRAMs called Concurrent Random Access Machines. This model is *synchronous*, that is the processors work in lock step, and it is *concurrent*, that is, several processors may read from the same loca-

tion at the same time step and several processors may try to write the same location at the same time step.

A *concurrent random access machine* (CRAM) consists of a large number of processors, all connected to a common, global memory. See Figure 5.1. The processors are identical except that they each contain a unique processor number. At each step, any number of processors may read or write any word of global memory. If several processors try to write the same word at the same time, then the lowest numbered processor succeeds.

This is the "priority write" model. The results in this chapter remain true if instead we use the "common write" model, in which the program guarantees that different values will never be written to the same location at the same time. The common write model is the more natural model for logic: a formula such as $(\forall x)\varphi$ specifies a parallel program using n processors — one for each possible value of x. Any processor finding that φ is false for its value of x will write a zero into a location in global memory that was initially one. See Corollary 5.8.

The CRAM is a special case of the concurrent read, concurrent write, parallel random access machine (CRCW-PRAM). We now describe the precise instruction set for the CRAM. This model will be familiar to anyone who has ever programmed a computer using assembly language, and it may seem strange to anyone who has not. Whether or not it seems strange, we must describe our model in a level that is detailed enough so that we may prove its equivalence to the descriptive model. Once we have done this, we can from then on write our parallel programs using first-order logic.

In addition to assignments, the CRAM instruction set includes addition, subtraction, and branch-on-less. The CRAM instruction set also includes a Shift instruction. Shift(x, y) causes the word x to be shifted y bits to the right.

Some of our choices for the instruction set correspond to the choices we have made earlier concerning which numeric relations to include in our first-order language. The Shift operation for the CRAM allows each bit of global memory to be available to every processor in constant time. Without Shift, CRAM$[t(n)]$ would be too weak to simulate FO$[t(n)]$ for $t(n) < \log n$. It was in working to prove Theorem 5.2 that we originally realized that BIT should be added as a numeric predicate to first-order logic. Without BIT, a first-order formula cannot access the individual bits in a given variable. In more powerful logics such as FO$($LFP$)$, or even IND$[\log n]$, BIT is definable from ordering and does not need to be explicitly added (Exercise 4.18).

Each processor has a finite set of registers, including the following, Processor: containing the number between 1 and $p(n)$ of the processor; Address: containing

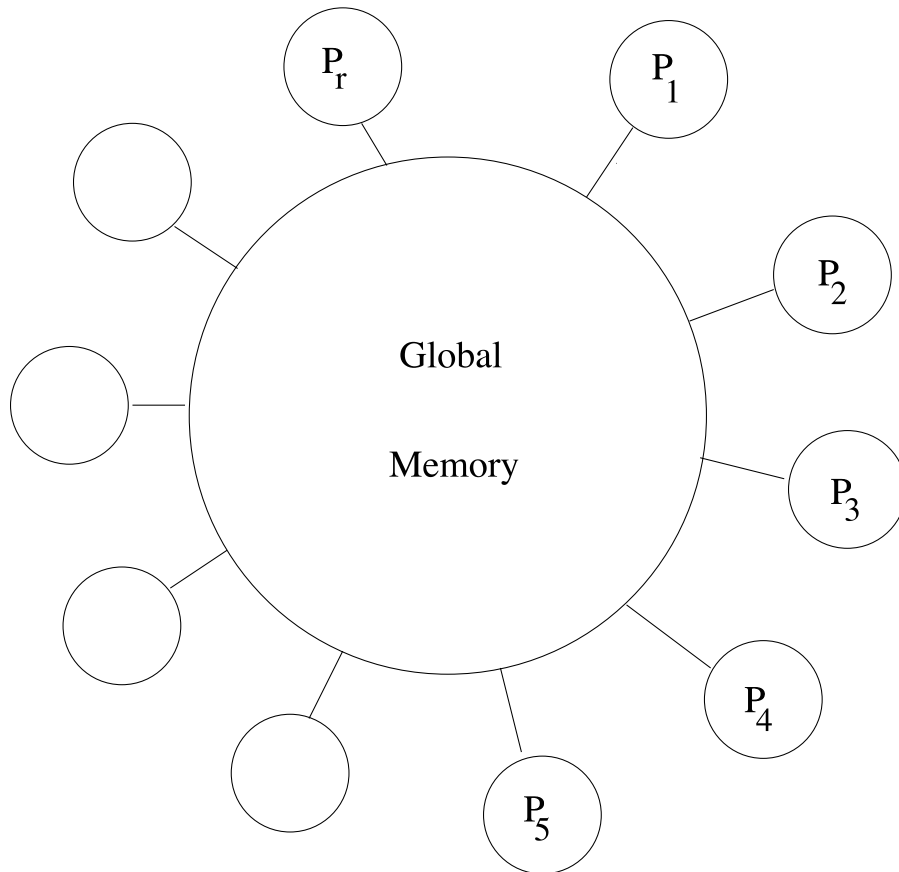

Figure 5.1: A concurrent random access machine (CRAM)

an address of global memory; Contents: containing a word to be written or read from global memory; and ProgramCounter: containing the line number of the instruction to be executed next. The instructions of a CRAM consist of the following:

READ: Read the word of global memory specified by Address into Contents.

WRITE: Write the Contents register into the global memory location specified by Address.

OP R_a R_b: Perform OP on R_a and R_b and leave the result in R_b. Here OP may be Add, Subtract, or Shift.

MOVE R_a R_b: Move R_a to R_b.

BLT $R\,L$: Branch to line L if the contents of R is less than zero.

The above instructions each increment the ProgramCounter, with the exception of BLT which replaces it by L, when R is less than 0.

We assume initially that the contents of the first $|\text{bin}(\mathcal{A})|$ words of global memory contain one bit each of the input string $\text{bin}(\mathcal{A})$. We will see in Corollary 5.8 that any other plausible setting of the input will work as well. We assume that a section of global memory is specified as the output. One of the bits of the output may serve as a flag indicating that the output is available.

Our measure of parallel time complexity will be time on a CRAM. Define CRAM$[t(n)]$ to be the set of boolean queries computable in parallel time $t(n)$ on a CRAM that has at most polynomially many processors. When we want to measure how many processors are needed, we use the complexity classes CRAM-PROC$[t(n), p(n)]$. This is the set of boolean queries computable by a CRAM using at most $p(n)$ processors and time $O(t(n))$. Thus, CRAM$[t(n)]$ = CRAM-PROC$[t(n), n^{O(1)}]$.

We will see in Theorem 5.2 and Corollary 5.8 that the complexity class CRAM$[t(n)]$ is quite robust. In particular, it is not affected by exactly how we place the input in the CRAM, by the size of the global memory word size, or by the size of the local registers, as long as these are both polynomially bounded.

5.2 Inductive Depth Equals Parallel Time

The following theorem says that parallel time is identical to inductive depth. In other words, a depth-optimal first-order inductive description of a query is a parallel-

time-optimal algorithm to compute the query. The theorem also completes the circle and shows that number of quantifier-block iterations and inductive depth are equal.

This is a very basic theorem, and its proof is not difficult. There are, however, many details to combine in showing that these rather different looking models are identical. For this reason, the proof is longer than it is deep.

Theorem 5.2 *Let S be a boolean query. For all polynomially bounded, parallel time constructible $t(n)$, the following are equivalent:*

1. *S is computable by a* CRAM *in parallel time $t(n)$ using polynomially many processors and registers of polynomially bounded word size.*

2. *S is definable as a first-order induction whose depth, for structures of size n, is at most $t(n)$.*

3. *There exists a first-order quantifier-block* [QB]*, a quantifier-free formula M_0 and a tuple $\bar{c}$ of constants such that the query S for structures of size at most n is expressed as* $[\text{QB}]^{t(n)} M_0(\bar{c}/\bar{x})$*, i.e., the quantifier-block repeated $t(n)$ times followed by M_0.*

In symbols, the equivalence of these three conditions can be written,

$$\text{CRAM}[t(n)] \quad = \quad \text{IND}[t(n)] \quad = \quad \text{FO}[t(n)]$$

Theorem 5.2 follows immediately from three containments: Lemmas 4.25, 5.3 and 5.4. The first of these — $\text{IND}[t(n)] \subseteq \text{FO}[t(n)]$ — has already been proved. We state and prove the remaining two now.

Lemma 5.3 *For any polynomially bounded $t(n)$ we have,*

$$\text{CRAM}[t(n)] \quad \subseteq \quad \text{IND}[t(n)]$$

Proof We want to simulate the computation of a CRAM M. On input $\mathcal{A}$, a structure of size n, M runs in $t(n)$ synchronous steps, using $p(n)$ processors, for some polynomial $p(n)$. Since the number of processors, the time and the memory word size are all polynomially bounded, we need only a constant number of variables $x_1, \ldots, x_k$, each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation. We can thus define the contents of all the relevant registers for any processor of M by induction on the time step.

We now write a first-order inductive definition for the relation VALUE$(\overline{p}, \overline{t}, \overline{x}, r, b)$ meaning that bit $\overline{x}$ in register r of processor $\overline{p}$ just after step $\overline{t}$ is equal to b.

The base case is that if $\overline{t} = 0$, then memory is correctly loaded with bin$(\mathcal{A})$. This is first-order expressible (Exercise 2.3). We also need to say that the initial contents of each processor's register Processor is its processor number. This is easy, since we are given the processor number as the argument $\overline{p}$.

The inductive definition of the relation VALUE$(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of $\overline{p}$'s program counter at time $\overline{t} - 1$. The most interesting case is when the instruction to be executed is READ. Here we simply find the most recent time $\overline{t'} < \overline{t}$ at which the word specified by $\overline{p}$'s register Address at time $\overline{t}$ was written into, and the lowest numbered processor $\overline{p'}$ that wrote into this address at time $\overline{t'}$. In this way we can access the answer, namely bit $\overline{x}$ of $\overline{p'}$ s register Contents at time $\overline{t'}$. If there exists no such time $\overline{t'}$ then this memory location contains its input value. This is bit i of the input bin$(\mathcal{A})$ if $i < |\text{bin}(\mathcal{A})|$, and zero otherwise.

It remains to check that Addition, Subtraction, BLT, and Shift are first-order expressible. Addition was handled in Proposition 1.9. In a similar way we can express Subtraction and Less Than.

Relation BIT allows our first-order formulas to examine any of the $\log n$ bits of a domain variable. It follows that the addition relation on such variables is first-order expressible (Theorem 1.17). Using addition, we can specify the Shift operation.

Thus we have described an inductive definition of relation VALUE, coding M's entire computation. Furthermore, one iteration of the definition occurs for each step of M. $\square$

Notice that the above proof is simple because first-order inductive definitions are very general and easy to use. Notice also that we did not write out the definition in its entirety. This would be a complete formal definition of the CRAM which we do not need. All that we needed to show is that the contents of all the bits of all the registers at time $t + 1$ is first-order definable from this same information at time t or earlier.

Lemma 5.4 *For polynomially bounded and parallel time constructible $t(n)$,*

$$\text{FO}[t(n)] \quad \subseteq \quad \text{CRAM}[t(n)]$$

Proof Let the FO$[t(n)]$ problem be determined by the following quantifier free formulas, quantifier block, and tuple of constants,

$$M_0, M_1, \ldots, M_k, \quad \text{QB} = (Q_1 x_1.M_1) \ldots (Q_k x_k.M_k), \quad \bar{c}, \ .$$

Our CRAM must test whether an input structure $\mathcal{A}$ satisfies the sentence,

$$\varphi_n \quad \equiv \quad [\text{QB}]^{t(n)} M_0(\bar{c}/\bar{x})$$

where $n = \|\mathcal{A}\|$. The CRAM will use n^k processors and n^{k-1} bits of global memory. Note that each processor has a number $a_1 \ldots a_k$ with $0 \leq a_i < n$. Using the Shift operation it can retrieve each of the a_i's in constant time.[1]

The CRAM will evaluate φ_n from right to left, simultaneously for all values of the variables $x_1, \ldots, x_k$. At its final step, it will output the bit $\varphi_n(\bar{c}/\bar{x})$.

For $0 \leq r \leq t(n) \cdot k$, let,

$$\varphi^r \quad \equiv \quad (Q_i x_i.M_i) \ldots (Q_k x_k.M_k)[\text{QB}]^q M_0 \tag{5.5}$$

where $r = k \cdot (q+1) + 1 - i$. Let $x_1 \ldots \hat{x}_i \ldots x_k$ be the $k-1$-tuple resulting from $x_1 \ldots x_k$ by removing x_i. We will now give a program for the CRAM which is broken into rounds each consisting of three processor steps such that:

Just after round r, the contents of memory location $a_1 \ldots \widehat{a_i} \ldots a_k$ is 1 or 0 according as whether $\mathcal{A} \models \varphi^r(a_1, \ldots, a_k)$ or not. (5.6)

Note that x_i does not occur free in φ^r. At round r, processor number $a_1 \ldots a_k$ executes the following three instructions according to whether Q_i is $\exists$ or Q_i is $\forall$:

$\{Q_i$ is $\exists\}$

1. $b := loc(a_1 \ldots \hat{a}_{i+1} \ldots a_k)$;
2. $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 0$;
3. **if** $M_i(a_1, \ldots, a_k)$ and b **then** $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 1$;

$\{Q_i$ is $\forall\}$

1. $b := loc(a_1 \ldots \hat{a}_{i+1} \ldots a_k)$;
2. $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 1$;
3. **if** $M_i(a_1, \ldots, a_k)$ and $\neg b$ **then** $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 0$;

[1] This is obvious if n is a power of 2. If not, we can just let each processor break its processor number into k $\lceil \log n \rceil$-tuples of bits. If any of these is greater than or equal to n, then the processor should do nothing during the entire computation.

It is not hard to prove by induction that Equation (5.6) holds and thus that the CRAM simulates the formula. The bit fetched into b tells us whether $\mathcal{A}$ satisfies the formula,

$$\varphi^{r-1} \quad \equiv \quad (Q_{i+1}x_{i+1}.M_{i+1}) \cdots [\text{QB}]^q M_0 \ .$$

The effect of lines 2 and 3 is that in parallel for all values of x_i, the truth of φ^r (Equation (5.5)) is tested and recorded. This completes the inductive step.

In the base case, at step 1, processor $(a_1 \dots a_k)$ must set $b = 1$ iff $\mathcal{A} \models M_0(a_1, \dots, a_k)$. Note that $M_0(x_1, \dots, x_k)$ is a quantifier-free formula. Observe that in constant time using its processor number, the shift operation, and addition, processor $(a_1 \dots a_k)$ can access the appropriate bits of bin$(\mathcal{A})$, for example the bit corresponding to $R_3(a_2, a_1)$, cf. Exercise 2.3. Furthermore, in constant time it can compute the boolean combination of these bits indicated by M_0. □

Remark 5.7 *The proof of Lemma 5.4 provides a very simple network for simulating an* FO$[t(n)]$ *property. The network has n^{k-1} bits of global memory and kn^k gates, where k is the number of distinct variables in the quantifier block. Each gate of the network is connected to two bits of global memory in a simple connection pattern. The blowup of processors going from* CRAM *to* FO *to* CRAM *seems large (cf. Corollary 5.10). However, it is plausible to build first-order networks with billions of processing elements, i.e. gates, thus accommodating fairly large n. It is crucial that k is kept small.*

An immediate corollary of Theorem 5.2 is that the complexity class CRAM$[t(n)]$ is not affected by minor changes in how the input is arranged, nor in the global memory word size, nor even by a change in the convention on how write conflicts are resolved.

Corollary 5.8 *For any function $t(n)$, the complexity class* CRAM$[t(n)]$ *is not changed if the definition of a* CRAM *is modified in any consistent combination of the following ways. (By consistent, we mean that input words larger than the global word size or larger than the allowable length of applications of Shift are not allowed.)*

1. *Change the input distribution so that either*
   - *(a) The entire input is placed in the first word of global memory.*
   - *(b) The $I_\tau(n)$ bits of input are placed $\log n$ bits at a time in the first $I_\tau(n)/\log n$ words of global memory.*

2. *Change the global memory word size so that either*

   (a) *The global word size is 1, i.e. words are single bits. (Local registers do not have this restriction so that the processor's number may be stored and manipulated.)*

   (b) *The global word size is bounded by $O(\log n)$.*

3. *Modify the Shift operation so that shifts are limited to the maximum of the input word size and of the log base two of the number of processors.*

4. *Remove the polynomial bound on the number of memory locations, thus allowing an unbounded global memory.*

5. *Instead of the priority rule for the resolution of write conflicts, adopt the "common write" rule in which different processors never write different values into the same memory location at a given time step.*

Proof The proof is that Lemmas 5.3 and 5.4 still hold with any consistent set of these modifications. This is immediate for Lemma 5.3. For Lemma 5.4, we must only show that processor number $a_1 \ldots a_k$ still has the power to evaluate the quantifier free formula $M_i(a_1, \ldots, a_k)$ and to name the global memory location $a_1 \ldots \hat{a}_i \ldots a_k$, for $1 \leq i \leq k$, in constant time. Recall that we are assuming that the input structure $\mathcal{A} = \langle \{0, 1, \ldots, n-1\}, R_1^{\mathcal{A}} \ldots R_p^{\mathcal{A}}, c_1^{\mathcal{A}} \ldots c_q^{\mathcal{A}} \rangle$ is coded as a bit string of length $I_\tau(n) = n^{r_1} + \cdots + n^{r_p} + q\lceil \log n \rceil$. It is clear that all of the consistent modifications above allow processor $a_1 \ldots a_k$ to test in constant time whether the relation $R(t_1, \ldots, t_r)$ holds, where R is an input or numeric relation, and $t_j \in \{a_1, \ldots, a_k\} \cup \{c_j | 1 \leq j \leq q\}$. □

Exercise 5.9 Show that Corollary 4.11 holds for IND$[t(n)]$, for any $t(n)$. That is, show that any problem $S \in$ IND$[t(n)]$ may be expressed as a single, depth $t(n)$ first-order induction. [Hint: Use Theorem 5.2.] □

5.3 Number of Variables versus Number of Processors

We now show that the number of variables in an inductive definition determines the number of processors needed in the corresponding CRAM computation. The intuitive idea is that using k $\log n$-bit variables, we can name approximately n^k different parts of the CRAM. Thus, very roughly, k variables corresponds to n^k

processors. The correspondence is not exact because the CRAM has a somewhat different pattern of interconnection between its processors and memory than the first-order inductive definition "model of parallelism". Later, we prove a tight relationship between number of variables and deterministic space (Theorem 10.16).

We now carefully analyze the proof of Theorem 5.2 to give processor-versus-variable bounds for translating between CRAM and IND. The proofs in this section consist of rather detailed variable counting. This whole section may be omitted.

Corollary 5.10 *Let* CRAM-PROC$[t(n), p(n)]$ *be the complexity class* CRAM$[t(n)]$ *restricted to machines using at most* $O(p(n))$ *processors. Let* IND-VAR$[t(n), v(n)]$ *be the complexity class* IND$[t(n)]$ *restricted to inductive definitions using at most* $v(n)$ *distinct variables. Assume for simplicity that the maximum size of a register word and* $t(n)$ *are both* $o[\sqrt{n}]$ *and that* $\pi \geq 1$ *is a natural number. Then,*

$$\begin{aligned} &\text{CRAM-PROC}[t(n), n^{\pi}] \\ &\qquad \subseteq \quad \text{IND-VAR}[t(n), 2\pi + 2] \\ &\qquad\qquad \subseteq \quad \text{CRAM-PROC}[t(n), n^{2\pi+2}] \end{aligned}$$

Proof We prove these bounds using the following two lemmas. For the first lemma, recall that Lemma 5.3 simulated a CRAM using an inductive definition. We inductively defined relation VALUE, which encoded the entire CRAM computation.

Lemma 5.11 *If the maximum size of a register word and of* $t(n)$ *are both* $o[\sqrt{n}]$*, and if* M *is a* CRAM-PROC$[t(n), n^{\pi}]$ *machine, then the inductive definition of* VALUE *may be written using* $2\pi + 2$ *variables.*

Proof We write out the inductive definition of VALUE in enough detail to count the number of variables used:

$$\text{VALUE}(\overline{p}, t, x, r, b) \quad \equiv \quad Z \vee W \vee S \vee R \vee M \vee B \vee A \ ,$$

where the disjuncts have the following intuitive meanings:

Z: $t = 0$ and the initial value of r is correct.

W: $t \neq 0$, the instruction just executed is WRITE, and the value of r is correct, i.e., unchanged, unless r is Program-Counter.

S, R, M, B, A: Similarly for SHIFT, READ, MOVE, BLT, and, ADD or SUBTRACT, respectively.

It suffices to show that each disjunct can be written using the number of variables claimed. First we consider the disjunct Z. The only interesting part of Z is the case where r is "Processor". In this case we use relation BIT to say that $b = 1$ iff bit x of $\overline{p}$ is 1. No extra variables are needed. Note that the number of free variables in the relation is $\pi + 1$ because the values t, x, r, and b may be combined into a single variable.

Next we consider the case of Addition. Recall that the main work is to express the carry bit:

$$C[A, B](x) \equiv (\exists y < x)[A(y) \wedge B(y) \wedge (\forall z.y < z < x)A(z) \vee B(z)] \ .$$

This definition uses two extra variables. Thus $\pi + 3 \leq 2\pi + 2$ variables certainly suffice. The cases S, M, and B are simpler.

The last and most interesting case is R. Here we must say,

1. The instruction just executed is READ,
2. Register r is register Contents,
3. There exists a processor $\overline{p'}$ and a time t' such that:
   (a) $t' < t$,
   (b) Address$(\overline{p'}, t')$ =Address$(\overline{p}, t)$,
   (c) VALUE$(\overline{p'}, t', x, r, b)$,
   (d) Processor $\overline{p'}$ wrote at time t',
   (e) For all $\overline{p''} < \overline{p'}$, if $\overline{p''}$ wrote at time t', then Address$(\overline{p''}, t') \neq$Address$(\overline{p'}, t')$,
   (f) For all t'' such that $t' < t'' < t$ and for all $\overline{p''}$, if $\overline{p''}$ wrote at time t'', then Address$(\overline{p''}, t'') \neq$Address$(\overline{p'}, t')$.

We count variables. On its face, this formula uses three $\overline{p}$'s and three t's. However, we show that two copies of each suffice. Observe that where we quantify $\overline{p''}$ in lines 3e and 3f, we no longer need $\overline{p}$, so we may use these variables instead.

The most subtle case is 3f. We use the fact that t is $o[\sqrt{n}]$, so t' and t'' can be coded into a single variable. We use a variable from $\bar{p}$ to encode t and t'. Then we can use t to universally quantify $t = \langle t', t'' \rangle$. Now we can universally quantify $\overline{p}$ to act as $\overline{p''}$. To say that Address$(\overline{p''}, t'') \neq$Address$(\overline{p'}, t')$, we use the extra variable

(t') to assert that there exists a bit position i and a bit b such that b is the bit at position i of Address$(\overline{p''}, t'')$, and $1 - b$ is the bit at position i of Address$(\overline{p'}, t')$. To help with expressing the first conjunct, we may use a variable from $\overline{p'}$ and to help in the second conjunct, we may use a variable from $\overline{p}$.

Thus $2\pi + 2$ variables suffice as claimed. □

The second lemma we need (Lemma 5.12) is a refinement of Lemma 5.4.

Lemma 5.12 *Let $\varphi(R, \overline{x})$ be an inductive definition of depth $d(n)$. Let k be the number of distinct variables, including $\overline{x}$, occurring in φ. Then the relation defined by φ is also computable in* CRAM-PROC$[d(n), O(n^k)]$.

Proof This is very similar to the proof of Lemma 5.4. Let T be the parse tree of φ. The CRAM will have $n^k|T|$ processors: one for each value of the k variables and each node in T. Let δ be the depth of T. As in the proof of Lemma 5.4, in rounds consisting of 3δ steps, the CRAM will evaluate an iteration of φ. Let r = arity(R) = the number of variables in $\overline{x}$; so $r \leq k$. The CRAM will have n^r bits of global memory to hold the truth value of $R_t = \varphi^t(\emptyset)$. It will use an additional $n^k|T|$ bits of memory to store the truth values corresponding to nodes of T. Thus $R_{d(n)}$, the least fixed point of φ, is computed in time $O(d(n))$ using $O(n^k)$ processors, as claimed. □

This completes the proof of Corollary 5.10. □

The above proofs give us some information concerning the efficiency of our simulation of CRAM s with first-order inductive definitions. After these results, the questions is, "Why is the number of variables needed to express a computation of n^π processors $2\pi + 2$, instead of π?" We discuss the multiplicative factor of two, and the additional two variables, respectively in the next two paragraphs.

We need the term 2π for two reasons: we must specify $\overline{p}$ and $\overline{p'}$ at the same time in order to say that their address registers are equal; and we need to say that no lower numbered processor $\overline{p''}$ wrote into the same address as $\overline{p'}$. This term points out a difference between the CRAM model and the network described in Remark 5.7 that was used to simulate a FO$[t(n)]$ property.

The factor of two would be eliminated if we adopted a weaker parallel machine model allowing only common writes and such that the memory location accessed by a processor at a given time could be determined by a very simple computation on the processor number and the time.

The additional two variables arise for various bookkeeping reasons. This term can be reduced if we make the following two changes:

1. Rather than keeping track of all previous times, we can assume that every bit of global memory is written into at least every T time steps for some constant T.

2. The register size can be restricted to $O(\log n)$, so we need only $O(\log \log n)$ bits to name a bit of a word.

Remark 5.13 The above observations show that the relation between the number of variables needed to give an inductive definition of a relation and the logarithm to the base n of the number of processors needed to quickly compute the relation are nearly identical. The cost of programming with first-order inductive definitions rather than CRAMs is theoretically very small.

The number of variables needed to describe a query is not a perfect measure of the amount of hardware needed in the parallel computation of the query. This is due to the difference in connection patterns of the parallel models CRAM$[t(n)]$ and FO$[t(n)]$. For example, in the proof of Lemma 5.4, processor $\bar{a}$ at round t accesses only a fixed pair of bits of global memory: bits $loc(a_1 \ldots \hat{a}_i \ldots a_k)$ and $loc(a_1 \ldots \hat{a}_{i+1} \ldots a_k)$. Thus a "first-order parallel machine" has a more restrictive pattern of connections between processors and global memory than a CRAM (cf. Remark 5.7).

Compare this to Theorem 10.16 where the set of queries describable using $k+1$ variables is proved identical to the set of queries computable using deterministic space n^k.

5.4 Circuit Complexity

Real computers are built from many copies of small and simple components. Circuit complexity is the branch of computational complexity that uses circuits of boolean logic gates as its model of computation. The circuits that we consider are directed acyclic graphs, in which inputs are placed at the leaves and signals proceed up the circuit toward the root r. Thus, in this idealized model, a gate is never reused during a computation.

This simple and basic model admits many beautiful and deep combinatorial arguments, some of which we will see in Chapter 13. In this section, we define the major circuit complexity classes. It should be intuitively clear that the depth

of a circuit, that is, the length of a longest path from root to leaf, corresponds to parallel time. We demonstrate this and the related connections between circuits and the other models of parallel computation, i.e., CRAMs, alternating machines, and first-order inductive definitions.

Let $S \subseteq \mathrm{STRUC}[\tau_s]$ be a boolean query on binary strings. In circuit complexity, S would be computed by an infinite sequence of circuits

$$\mathcal{C} \quad = \quad \{C_i \,|\, i = 1, 2, \ldots\} \;, \tag{5.14}$$

where C_n is a circuit with n input bits and a single output bit r. For $w \in \{0,1\}^n$, let $C_n(w)$ be the value at C_n's output gate, when the bits of w are placed in its n input gates. We say that $\mathcal{C}$ *computes* S iff for all n and for all $w \in \{0,1\}^n$,

$$w \in S \qquad \Leftrightarrow \qquad C_n(w) = 1 \;.$$

In this section we present an introduction to circuit complexity and relate complexity classes defined via uniform sequences of circuits to descriptive complexity. We also derive a completely syntactic definition for circuit uniformity. This definition is equivalent to the usual Turing machine-based definition in the range where the latter exists.

As seen in Definition 2.27, a circuit is a directed, acyclic graph. The leaves of the circuit are the input nodes. Every other vertex is an "and", "or", or "not" gate. The edges of the circuit indicate connections between nodes. Edge (a, b) would indicate that the output of gate a is an input to gate b.

It is convenient to assume that all the "not" gates in our circuits have been pushed down to the bottom. We can do this using De Morgan laws ($\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$; $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$) without increasing the depth and without significantly increasing the size of the circuit.

Furthermore, we can assume that the levels alternate, with the top level being all "or" gates, the next level all "and" gates and so on. Such a normalized circuit is called a *layered circuit*. See Figure 5.15, in which a layered circuit of depth $t(n)$ is drawn.

Below, we define three families of circuit complexity classes. They vary depending on whether all gates have bounded fan-in (NC), the "and" and "or" gates may have unbounded fan-in (AC), or there are threshold gates (ThC). A threshold gate with threshold value i has output one iff at least i of its inputs have value one. Note that threshold gates include as special cases "or" gates in which the threshold is one and "and" gates in which the threshold is equal to the number of inputs.

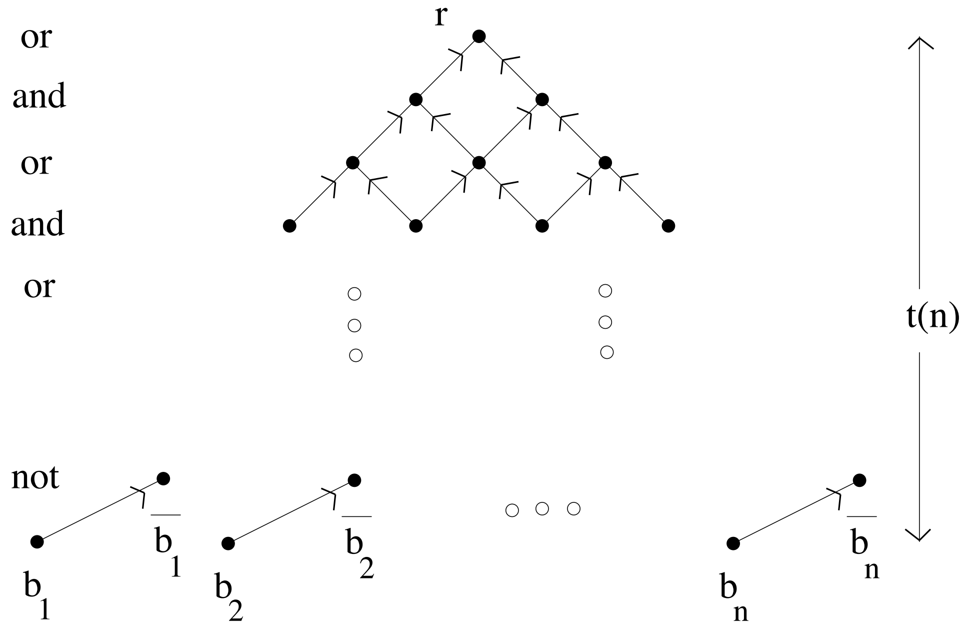

Figure 5.15: A layered circuit of depth $t(n)$.

Recall from Definition 2.27 the vocabulary of circuits, $\tau_c = \langle E^2, G^1_\wedge, G^1_\vee, G^1_\neg, I^1, r\rangle$. Constant r refers to the root node, or output of the circuit. The gates that have no incoming edges are the leaves of the circuit. We use the following abbreviation,

$$L(x) \quad \equiv \quad (\forall y)(\neg E(y, x))$$

The leaves need to be ordered $1, 2, \ldots$ so that we know where to place input bits $b_1, b_2, \ldots, b_n$. We assume for simplicity that the leaves of a circuit are the initial elements of the universe of a circuit. That is, we assume that every circuit $\mathcal{C}$ satisfies the following formula:

$$\text{Leaves-Come-First} \quad \equiv \quad (\forall xy)(L(x) \wedge \neg L(y) \;\rightarrow\; x < y)$$

Input relation $I(v)$ represents the fact that leaf v contains value 1. Internal node w is an and-gate if $G_\wedge(w)$ holds, an or-gate if $G_\vee(w)$ holds, and a not-gate if $G_\neg(w)$ holds.

We generalize the vocabulary of circuits to the vocabulary of *threshold circuits*, $\tau_{thc} = \tau_c \cup \{G^2_t\}$. Relation $G_t(g, v)$ means that g is a threshold gate with threshold value v. Thus, g would take value 1 in a circuit iff at least v of its inputs have value one.

Let $\mathcal{A} \in \text{STRUC}[\tau]$ and let $n = \|\mathcal{A}\|$. A circuit C_n with $\hat{n}_\tau(n)$ leaves can take $\mathcal{A}$ as input by placing the binary string $\text{bin}(\mathcal{A})$ into its leaves. We write $C(w)$ to denote the output of circuit C on input w, i.e., the value of the root node when w is placed at the leaves and $\mathcal{C}$ is then evaluated. We say that circuit C *accepts* structure $\mathcal{A}$ iff $C(\text{bin}(\mathcal{A})) = 1$.

In proving lower bounds on circuit complexity, one considers the size and structure of the circuits C_n, but one rarely needs to consider how the sequence of circuits relate for different values of n. To relate circuit complexity to machine or descriptive complexity, however, we must explain where these infinitely many circuits come from. The idea is that the circuits all come from unwindings of a particular program and architecture.

Formally we assume that there is a query of low complexity that on input 0^n produces C_n. We insist upon first-order uniformity. This means that there is a first-order query $I : \text{STRUC}[\tau_s] \to \text{STRUC}[\tau_c]$ with $C_n = I(0^n)$, $n = 1, 2, \ldots$. Here $0^n \in \text{STRUC}[\tau_s]$ is the string consisting of n zeros. Note that this uniformity condition implies that C_n has polynomially bounded size.

Definition 5.16 (Uniform) Let $\mathcal{C}$ be a sequence of circuits as in Equation (5.14). Let $\tau \in \{\tau_c, \tau_{thc}\}$ be the vocabulary of circuits or threshold circuits. Let I : $\text{STRUC}[\tau_s] \to \text{STRUC}[\tau]$ be a query such that for all $n \in \mathbf{N}$, $I(0^n) = C_n$. That is, on input a string of n zero's the query produces circuit n. If $I \in \text{FO}$, then $\mathcal{C}$ is a *first-order uniform* sequence of circuits. Similarly, if $I \in \text{L}$, then $\mathcal{C}$ is *logspace uniform*. If $I \in \text{P}$, then $\mathcal{C}$ is *polynomial-time uniform*, and so on. □

We now define the standard circuit complexity classes. The notion of uniformity that we use is first-order uniformity. Observe that whether we use first-order, logspace, or polynomial-time uniformity, any uniform sequence of circuits is polynomial-size. That is, there is a function $p(n)$ such that circuit C_n has size at most $p(n)$.

Definition 5.17 (Circuit Complexity) Let $t(n)$ be a polynomially bounded function and let $S \subseteq \text{STRUC}[\tau]$ be a boolean query. Then S is in the (first-order uniform) circuit complexity class $\text{NC}[t(n)]$, $\text{AC}[t(n)]$, $\text{ThC}[t(n)]$, respectively iff there exists a first-order query $I : \text{STRUC}[\tau_s] \to \text{STRUC}[\tau_{thc}]$ defining a uniform class of circuits $C_n = I(0^n)$ with the following properties:

1. For all $\mathcal{A} \in \text{STRUC}[\tau]$, $\quad \mathcal{A} \in S \quad \Leftrightarrow \quad C_{\|\mathcal{A}\|}$ accepts $\mathcal{A}$.
2. The depth of C_n is $O(t(n))$.

3. The gates of C_n consist of binary "and" and "or" gates (NC), unbounded fan-in "and" and "or" gates (AC), and unbounded fan-in threshold gates (ThC), respectively.

For $i \in \mathbf{N}$, let $\mathrm{NC}^i = \mathrm{NC}[(\log n)^i]$, $\mathrm{AC}^i = \mathrm{AC}[(\log n)^i]$, and $\mathrm{ThC}^i = \mathrm{ThC}[(\log n)^i]$. Finally, let

$$\mathrm{NC} \quad = \quad \bigcup_{i=0}^{\infty} \mathrm{NC}^i \qquad \square$$

The NC circuits correspond reasonably well to standard silicon-based hardware. The AC circuits are idealized hardware in that it is not known how to connect n inputs to a single gate with constant delay time. The practical way to do this is to connect them in a binary tree, causing an $O(\log n)$ time delay. On the other hand, once we have such a binary tree, we can also compute threshold functions. This explains what we rigorously prove below, namely:

$$\mathrm{AC}[t(n)] \quad \subseteq \quad \mathrm{ThC}[t(n)] \quad \subseteq \quad \mathrm{NC}[t(n)\log n] \ .$$

We also see below that the unbounded fan-in gates in AC circuits correspond exactly to concurrent writing in the CRAM model. Similarly, threshold gates correspond to the tree connections in the NYU ultracomputer and to the "scan" operation on the connection machine, cf. Theorem 5.27, Exercises 5.28, 5.29.

Recall that a *regular language* is a set of strings $S \subseteq \Sigma^\star$ accepted by a *finite automaton*. A finite automaton is essentially a Turing machine with no work tapes. See [LP81] or [HU79] for details. As an example of computing with circuits, we prove the following,

Proposition 5.18 *Every regular language is in* NC^1.

Proof We are given a deterministic finite automaton $D = \langle \Sigma, Q, \delta, s, F \rangle$. We must construct a first-order query $I_D : \mathrm{STRUC}[\tau_s] \rightarrow \mathrm{STRUC}[\tau_c]$ such that, letting $C_n = I_D(0^n)$, for all strings $w \in \Sigma^\star$,

$$w \in \mathcal{L}(D) \qquad \Leftrightarrow \qquad C_{|w|} \text{ accepts } w$$

Circuit C_n is a complete binary tree with n leaves. The input to leaf $L(i)$ is w_i, character i of the input string. Each such leaf contains the finite hardware to

produce as output the transition function of D on reading input symbol w_i. That is, we store a table for $f_{L(i)} = \delta(\cdot, w_i) : Q \to Q$.

Each internal node v of the tree takes as input the transition functions f_{lc} and f_{rc} of its left child and right child, and computes their composition $f_v = f_{rc} \circ f_{lc}$. Thus, inductively, the output of every node v is the function $f_v = \delta^\star(\cdot, w_v)$ where w_v is the subword of w that is sitting below v's subtree. In particular, w is in $L(D)$ iff $f_r(s) \in F$, where f_r is the mapping stored at the root.

Since D is a fixed, finite state automaton, the hardware at the leaves and at each internal node is a fixed, bounded size NC circuit. The first-order query I_D need only describe a complete binary tree with n leaves with these two fixed circuits placed at each leaf and each internal node, respectively. The height of the resulting circuits is $O(\log n)$ as desired. $\square$

Exercise 5.19 Prove that the boolean majority query MAJ is in NC^1.

$$\text{MAJ} \quad = \quad \left\{\mathcal{A} \in \text{STRUC}[\tau_s] \;\middle|\; \text{string } \mathcal{A} \text{ contains more than } \|\mathcal{A}\|/2 \text{ “1”s}\right\}$$

[Hint: The obvious way to try to build an NC^1 circuit for majority is to add the n input bits via a full binary tree of height $\log n$. The problem with this is that while the sums being added have more and more bits, they must be added in constant depth.

A solution to this problem uses ambiguous arithmetic notation. Consider a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example 3213 and 3221 are different representations of the decimal number 37 in this ambiguous notation,

$$3213 \;=\; 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 \;=\; 37 \;=\; 3221 \;=\; 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0 \;.$$

Show that adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit, i.e., with bounded depth.

See the following sample addition problem.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

This is doable in NC^0 because the carry from column i can be computed by looking only at columns i and $i+1$.

Translating from ambiguous notation back to binary, which must be done only once at the end, is just an addition problem. This is first-order, and thus AC^0, and thus NC^1.] □

Exercise 5.20 A good way to become familiar with circuit complexity classes is to prove the following containments. For all $i \in \mathbf{N}$,

$$\mathrm{NC}^i \quad \subseteq \quad \mathrm{AC}^i \quad \subseteq \quad \mathrm{ThC}^i \quad \subseteq \quad \mathrm{NC}^{i+1} \tag{5.21}$$

[Hint: the only subtle containment is the the last. For this, you should use Exercise 5.19.] □

The following theorem summarizes the relationships between all the parallel models that we have seen. Note that the equivalence of FO$[t(n)]$ and AC$[t(n)]$ shows that the uniformity of AC circuits can be defined in a completely syntactic way: circuit C_n is constructed by writing down a quantifier block $t(n)$ times.

Theorem 5.22 *For all polynomially bounded and first-order constructible $t(n)$, the following classes are equal:*

$$\mathrm{CRAM}[t(n)] \;=\; \mathrm{IND}[t(n)] \;=\; \mathrm{FO}[t(n)] \;=\; \mathrm{AC}[t(n)]$$

Proof The equality of the first three classes was already proved in Theorem 5.2. The proof of FO$[t(n)] \subseteq$ AC$[t(n)]$ is similar to the proof of Lemma 5.4. Let S be a FO$[t(n)]$ boolean query given by the quantifier block, QB $= [(Q_1x_1.M_1)\ldots(Q_kx_k.M_k)]$, initial formula, M_0, and tuple of constants, $\bar{c}$. We must write a first-order query, I, to generate circuit $C_n = I(0^n)$, so that for all $\mathcal{A} \in \mathrm{STRUC}[\tau]$,

$$\mathcal{A} \models (\mathrm{QB}^{t(\|\mathcal{A}\|)}M_0)(\bar{c}/\bar{x}) \qquad \Leftrightarrow \qquad C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A} \tag{5.23}$$

Initially the circuit evaluates the quantifier-free formulas M_i, $i = 0, 1, \ldots, k$. The nodes $\langle M_i, b_1, \ldots, b_k\rangle$ will be the gates that have evaluated these formulas, i.e.,

$$\langle M_i, b_1, \ldots, b_k\rangle(\mathrm{bin}(\mathcal{A})) = 1 \qquad \Leftrightarrow \qquad \mathcal{A} \models M_i(b_1, \ldots, b_k)$$

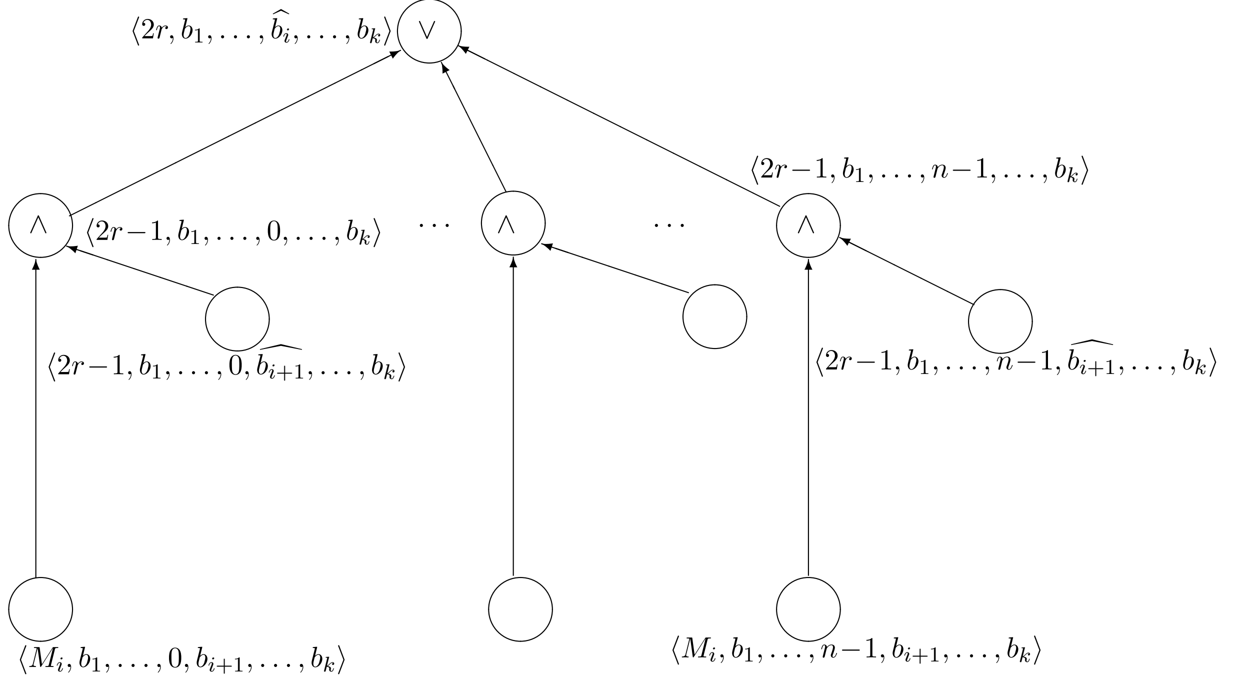

Figure 5.24: Two rows of AC$[t(n)]$ circuit simulating FO$[t(n)]$ formula $Q_i = \exists$.

As in Equation (5.5), let φ^r be the inside r quantifiers of QB$^{t(\|\mathcal{A}\|)}M_0$. The first of these quantifiers is Q_i, where $i \equiv 1 - r \pmod{k}$.

We construct the gate $\langle 2r, b_1, \ldots, \widehat{b_i}, \ldots, b_k\rangle$ so that

$$\langle 2r, b_1, \ldots, \widehat{b_i}, \ldots, b_k\rangle(\text{bin}(\mathcal{A})) = 1 \qquad \Leftrightarrow \qquad \mathcal{A} \models \varphi^r(b_1, \ldots, b_k)$$

This is achieved inductively by letting gate $\langle 2r, b_1, \ldots, \widehat{b_i}, \ldots, b_k\rangle$ be an "and"-gate, or "or"-gate according as $Q_i = \forall$ or $\exists$. This gate has inputs gates $\langle 2r - 1, b_1, \ldots, b_i, \widehat{b_{i+1}}, \ldots, b_k\rangle$ for b_i ranging over $|\mathcal{A}|$. Each $\langle 2r - 1, b_1, \ldots, b_i, \widehat{b_{i+1}}, \ldots, b_k\rangle$ is a binary "and"-gate whoses inputs are $\langle M_i, b_1, \ldots, b_k\rangle$ and $\langle 2r - 2, b_1, \ldots, b_i, \widehat{b_{i+1}}, \ldots, b_k\rangle$. See Figure 5.24 for a diagram of this construction.

The circuit we have described may be constructed via a first-order query I, and it satisfies Equation (5.23).

To prove that AC$[t(n)] \subseteq$ IND$[t(n)]$, let $C_n = I(0^n)$, $n = 1, 2, \ldots$, be a uniform sequence of AC$[t(n)]$ circuits, with $I : \text{STRUC}[\tau_s] \to \text{STRUC}[\tau_c]$ a first-

order query. We must write an inductive formula,

$$\Phi \quad \equiv \quad (\text{LFP}\varphi)(\bar{c})$$

so that for all $\mathcal{A} \in \text{STRUC}[\tau]$,

$$\mathcal{A} \models \Phi \qquad \Leftrightarrow \qquad C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A} \ .$$

From $\mathcal{A}$ we can get the circuit $C_{\|\mathcal{A}\|} = \langle E, G_\wedge, G_\vee, G_\neg, \text{bin}(\mathcal{A}), r\rangle$ via the first-order query I. Note that, the input string $I = \text{bin}(\mathcal{A})$ is first-order describable from $\mathcal{A}$ (Exercise 2.3). The following is a first-order inductive definition of the relation $V(x, b)$ meaning that gate x has boolean value b,

$$\begin{array}{rcl} V(x,b) & \equiv & \text{DEFINED}(x) \ \wedge \ \big[\ L(x) \wedge (I(x) \leftrightarrow b) \quad \vee \\ & & \quad G_\wedge(x) \wedge (C(x) \leftrightarrow b) \quad \vee \quad G_\vee(x) \wedge (D(x) \leftrightarrow b) \\ & & \quad \vee \quad G_\neg(x) \wedge (N(x) \leftrightarrow b)\big] \end{array} \tag{5.25}$$

Here, DEFINED(x), meaning that x is ready to be defined, is an abbreviation for $(\forall y)(\exists c)(E(y, x) \rightarrow V(y, c))$.

Predicates $C(x)$ says that all of x's inputs are true, $D(x)$ says that some of x's inputs are true, and $N(x)$ says that its input is false:

$$\begin{array}{rcl} C(x) & \equiv & (\forall y)(E(y,x) \rightarrow V(y,1)) \\ D(x) & \equiv & (\exists y)(E(y,x) \wedge V(y,1)) \\ N(x) & \equiv & (\exists! y)(E(y,x)) \ \wedge \ (\exists y)(E(y,x) \wedge V(y,0)) \end{array}$$

The inductive definition of V closes in exactly the depth of C_n, which is $O(t(n))$ iterations. Once it closes, $\Phi \equiv V(r, 1)$ expresses the acceptance condition in IND$[t(n)]$, as desired. ☐

A corollary of Theorem 5.22 is the following characterization of the class NC,

Corollary 5.26

$$\text{NC} \quad = \quad \bigcup_{k=1}^{\infty} \text{FO}[(\log n)^k] \quad = \quad \bigcup_{k=1}^{\infty} \text{CRAM}[(\log n)^k]$$

In the NYU ultracomputer, processors are connected using a complete binary tree. At each node v of this tree, there is enough logic to compute the sum of the

words at v's children and send this value to v's parent. Thus, very quickly — and it is convenient to call this unit time — we can compute the "or" or the sum of n locations.

Thus, an ultracomputer is an extension of a CRAM in which when several processors try to write into the same location of global memory at the same time, that location contains at the next time step the sum of all the values written. Define the complexity class ULTRA$[t(n)]$ to be the set of boolean queries computable on an ultracomputer using polynomially much hardware and parallel time $t(n)$. The Connection Machine has an operation called "scan", which is equivalent to this n-ary sum. Thus, the Connection Machine is another example of an ultracomputer.

The following generalization of Theorem 5.22 is not difficult to prove. The main subtlety is in working with the first-order analogue of the circuit class ThC^0. Define the majority quantifier $(Mx)\varphi(x)$ to mean that more than half of the elements of the universe satisfy φ. Let FO(M), be first-order logic extended by the majority quantifier. It is shown in [BIS88] that FO(M) $=$ ThC^0. Later, we define FO(COUNT) — a robust way to add counting to first-order logic (Definition 12.11). It is also shown in [BIS88] that FO(M) $=$ FO(COUNT).

Theorem 5.27 *For all polynomially bounded and constructible $t(n)$, the following classes are equal:*

$$\text{ULTRA}[t(n)] \;=\; \text{FO}(M)[t(n)] \;=\; \text{ThC}[t(n)]$$

The following exercises give some sense of the complexity class ThC^0 = FO(M) = FO(COUNT).

Exercise 5.28 Let $\mathcal{A}$ be an ordered structure with $n = \|\mathcal{A}\|$. As usual, we can think of any element $i \in |\mathcal{A}|$ as being a number from 0 to $n-1$. Let the formula $C_\varphi(i)$ mean that the number of $a \in |\mathcal{A}|$ such that $\mathcal{A} \models \varphi(a)$ is at least i. Show that if $\varphi(x)$ is expressible in FO(M), then so is C_φ.

[Hint: use the ordering relation. Split the problem into showing that there are j a's no greater than $n/2$ satisfying φ and that there are $i - j$ a's greater than $n/2$. These statements can be written with one majority quantifier each.] □

Exercise 5.29 The class ThC^0 $=$ FO(M) is a rather interesting complexity class. Show that the following arithmetic operations are computable in ThC^0:

1. The sum of n-bit natural numbers.

2. Multiplication of two n-bit integers.

3. Multiplication of two $n \times n$ integer matrices, each of whose entries is an integer of at most n bits.

[Hint: (3) follows easily from (2) which follows easily from (1). To do (1), let $S = \sum_{i=0}^{n-1} A_i$ be the sum we are trying to compute. Observe that we can count the number of 1s in each column. Thus, we just have to add the relevant column-counts, i.e., we have to add n $\log n$-bit numbers. To make this easier still, split each A_i into two parts: $A_i = B_i + C_i$, where the bits of B_i and C_i are split into blocks of size $\log n$, and B_i is zero on the even blocks and C_i is zero on the odd blocks. Thus $S = \sum B_i + \sum C_i$, and we have reduced the problem to computing $\sum B_i$ and $\sum C_i$. You have to figure out why this is easier than the original problem! (For a similar argument, see the part of the proof of Theorem 1.17 in which it is shown that TIMES is first-order definable using BIT.)] □

5.5 Alternating Complexity

Alternating Turing machines are very closely tied with quantifiers. In this section we establish the precise relationship between descriptive and alternating complexity. We begin by examining the relationship at the lowest level. Let the *logarithmic-time hierarchy* (LH) be ATIME-ALT$[\log n, O(1)]$, i.e., the set of boolean queries computed by alternating Turing machines in $O[\log n]$ time, making a bounded number of alternations. The following theorem says that LH $=$ FO.[2]

Theorem 5.30 *The logarithmic-time hierarchy is exactly the set of first-order expressible boolean queries.*

Proof The most delicate part of the proof is the following:

Lemma 5.31 DTIME$[\log n]$ $\subseteq$ FO .

Proof Let T be a DTIME$[\log n]$ machine. We must write a first-order sentence φ such that for all inputs $\mathcal{A}$,

$$T(\text{bin}(\mathcal{A}))\downarrow \quad \Leftrightarrow \quad \mathcal{A} \models \varphi$$

[2]This is analogous to a result we will see later: The polynomial-time hierarchy (PH) is equal to the set of boolean queries expressible in second-order logic (SO), (Corollary 7.22).

The sentence φ will begin with existential quantifiers, $\varphi \equiv (\exists x_1 \ldots x_c)\psi(\bar{x})$. The variables $\bar{x}$ will code the $\log n$ steps of T's computation including, for each time step t, the values q_t, w_t, d_t, I_t representing T's state, the symbol it writes, the direction its head moves, and the value of the input being scanned by the index-tape-controlled input head at time t, respectively. (It is important to remember that each variable is a $\lceil \log n \rceil$ bit number and that the numeric predicate BIT allows these bits to be specified.)

The formula ψ must now assert that the information in $\bar{x}$ meshes together to form a valid accepting computation of T. The work we must do to accomplish this is to define the first-order relations $C(p, t, a)$ and $P(p, t)$ meaning that for the computation determined by $\bar{x}$, the contents of cell p at time t is a; and the work head is at position p at time t. Given C and P we can assert that $\bar{x}$ is self-consistent. Note, for example, that we can guess the contents y of the index tape, and then use C to verify that y is correct. Next, using y we can verify that the input symbol I_t is correct.

Next, note that using P we can write C because the contents of cell p at time t is just w_{t_1} where t_1 is the most recent time that the head was at position p.

Finally observe that to write the relation P it suffices to take the sum of $O[\log n]$ values each of which is either -1, or 1. We can do this in FO by Lemma 1.18. □

To prove LH $\subseteq$ FO, we need only note that an alternating logarithmic-time machine may be assumed to write its guesses on a work tape and then deterministically check for acceptance. Since there are a bounded number of alternations and the total time is $O(\log n)$, these guesses may be simulated by a bounded number of first-order quantifiers. The remaining work is in DTIME$[\log n]$ and thus in FO by Lemma 5.31.

The other direction of Theorem 5.30 is fairly easy. We have to show that for every first-order sentence,

$$\varphi \;\equiv\; (\exists x_1)(\forall x_2)\ldots(Q_k x_k)M(\bar{x})$$

there exists an ATIME-ALT$[\log n, O(1)]$ machine T such that for all input strings $\mathcal{A}$,

$$T(\text{bin}(\mathcal{A}))\downarrow \quad \Leftrightarrow \quad \mathcal{A} \models \varphi$$

Since M is a constant size quantifier-free formula, it is easy to build a DTIME$[\log n]$ Turing machine which on input $\mathcal{A}$ and with values $a_1, \ldots, a_k$ on its tape, tests whether or not $\mathcal{A} \models M(\bar{a})$. (The most complicated part of this is

to verify the BIT predicate, which requires counting in binary up to $O(\log n)$ on a work tape — this is straightforward.) Thus using $k-1$ alternations between existential and universal states, a Σ_k logarithmic-time machine can guess $a_1, \ldots, a_k$ and then deterministically verify $M(\bar{a})$.

□

Notice that from Theorems 5.2, 5.22 and 5.30, we now have three interesting characterizations of the class FO.

Corollary 5.32

$$\text{FO} \quad = \quad \text{AC}^0 \quad = \quad \text{CRAM}[1] \quad = \quad \text{LH}$$

Notice that the truth of Corollary 5.32 depends on our choice of including BIT as a numeric predicate, and the SHIFT operation in the CRAM, and our definition of uniformity for AC^0. In Chapter 11 we will obtain a non-uniform version of Corollary 5.32 in which we allow arbitrary numeric relations (Proposition 11.19).

5.5.1 Alternation as Parallelism

AC and NC circuits also have elegant characterizations via alternating machines.

Theorem 5.33 *For* $t(n) \geq \log n$,

$$\text{ASPACE-ALT}[\log n, t(n)] \quad = \quad \text{AC}[t(n)] \quad = \quad \text{FO}[t(n)] \; .$$

Proof We have already seen the second equality in Theorem 5.22.

(ASPACE-ALT$[\log n, t(n)] \supseteq$ AC$[t(n)]$): Let $t(n) \geq \log n$ and consider the same AC$[t(n)]$ boolean query as in the proof that AC$[t(n)] \subseteq$ IND$[t(n)]$ in Theorem 5.22. Since ASPACE-ALT$[\log n, t(n)] \supseteq$ ATIME-ALT$[\log n, 1]$ = LH, we know from Theorem 5.30 that ASPACE-ALT$[\log n, t(n)] \supseteq$ FO. Thus, the circuit C_n is available in ASPACE-ALT$[\log n, t(n)]$.

Recall that Equation (5.25) simulates an AC$[t(n)]$ circuit via an IND$[t(n)]$ definition. Looking at Equation (5.25) we see that it makes at most $O(t(n))$ alternations between existential and universal quantifiers. This inductive definition can thus be directly simulated by an ASPACE-ALT$[\log n, t(n)]$ machine: Each universal quantifier is simulated by $\log n$ universal moves and each existential quantifier

is simulated by $\log n$ existential moves. The space needed to hold the variables is $O(\log n)$. Furthermore, there are only a bounded number of alternations per iteration of the inductive definition.

(ASPACE-ALT$[\log n, t(n)] \subseteq$ AC$[t(n)]$): Conversely, let M be an ASPACE-ALT$[\log n, t(n)]$ machine. As in Theorem 3.16, an ID of M can be coded using a bounded number of variables. The acceptance condition of M can then be expressed via an inductive definition of depth $\log n + t(n)$ as follows. Let EPATH$_M(\mathrm{ID}_1, \mathrm{ID}_2)$ mean that there is a computation path of M from ID_1 to ID_2 all of whose states except perhaps the last is existential. Let APATH mean the same thing for universal paths. It is easy to see that EPATH and APATH are expressible in IND$[\log n]$ (cf. Proposition 4.17). Thus, the following simultaneous induction has depth $O(\log n + t(n))$ and expresses the acceptance condition for M as desired.

$$
\begin{aligned}
\mathrm{ACCEPT}_M(\mathrm{ID}_1) \quad \equiv \quad & \mathrm{ID}_1 \text{ is the accept ID } \vee \ [\exists \mathrm{ID}_2 \ \mathrm{ACCEPT}_M(\mathrm{ID}_2) \ \wedge \\
& (\mathrm{EPATH}(\mathrm{ID}_1, \mathrm{ID}_2) \ \vee \ \mathrm{APATH}(\mathrm{ID}_1, \mathrm{ID}_2))] \ \wedge \\
& (\mathrm{ID}_1 \text{ is universal } \ \rightarrow \ (\forall \mathrm{ID}_2)(\mathrm{APATH}(\mathrm{ID}_1, \mathrm{ID}_2)) \rightarrow \\
& \qquad\qquad\qquad\qquad \mathrm{ACCEPT}_M(\mathrm{ID}_2))
\end{aligned}
$$

□

We leave a similar characterization of NC$[t(n)]$ to the reader:

Exercise 5.34 Prove that for $t(n) \geq \log n$,

$$\mathrm{NC}[t(n)] \quad = \quad \text{ASPACE-TIME}[\log n, t(n)]$$

[Hint: this is similar to the proof of Theorem 5.33, the difference being that the definitions of C and D in Equation (5.25) now involve binary "and"s and "or"s rather than universal and existential quantifiers.] □

For $i \geq 1$, the bound NC$^i \subseteq$ AC^i in Equation (5.21) is not optimal. The following improvement is known to be optimal because the NC^1 query PARITY requires depth $\log n / \log\log n$ (Corollary 13.8).

Theorem 5.35 *For $t(n) \geq \log n$, the following containment holds,*

$$\mathrm{NC}[t(n)] \quad \subseteq \quad \mathrm{AC}[t(n)/\log\log n]$$

Proof We prove the equivalent containment,

$$\text{ASPACE-TIME}[\log n, t(n)] \quad \subseteq \quad \text{IND}[t(n)/\log\log n]$$

Suppose that we are given an ASPACE-TIME$[\log n, t(n)]$ machine M. As in Theorem 5.33, we can write an inductive definition for ACCEPT$_M$ — the acceptance condition of M. The straightforward way to do this is in IND$[t(n)]$ with one alternation of quantifiers per move of M.

As usual, we assume that M alternates at each step between existential and universal states. To improve this simulation by a $\log\log n$ factor, observe that a list of which existential moves to make in the event of each possible sequence of $(\log\log n)/2$ universal moves can be given in $\log n$ bits.

Let e be such a $\log n$-bit table of which existential move to make in the event of any sequence of $(\log\log n)/2$ universal moves, and let a be such a sequence of universal moves. Then we can inductively define the relation MOVES$_M(e, u, \text{ID}_1, \text{ID}_2)$ meaning that ID$_2$ follows from ID$_1$ in the $\log\log n$ moves of M determined by the universal moves u and the existential moves given by e indexed by u. It is easy to write such an inductive definition in depth $\log\log n$.

Our definition of ACCEPT$_M$ is then a simultaneous inductive definition with MOVES$_M$,

$$\begin{aligned}\text{ACCEPT}_M(\text{ID}_1) \quad \equiv \quad & \text{ID}_1 \text{ is the accept ID} \ \lor \quad (\exists e)(\forall u)(\exists \text{ID}_2) \\ & (\text{MOVES}_M(e, u, \text{ID}_1, \text{ID}_2) \ \land \ \text{ACCEPT}_M(\text{ID}_2))\end{aligned}$$

The depth of this simultaneous induction is $t(n)/\log\log n$ as desired. □

Historical Notes and Suggestions for Further Reading

Cook has written a useful survey of parallel complexity classes and problems, [Coo85]. Stockmeyer and Vishkin proved the equality of non-uniform versions of CRAM$[t(n)]$ and AC$[t(n)]$ in [SV84]. Most of the uniform results in this chapter appeared first in [I89a]. There was much previous work comparing different versions of PRAM's. See for example [FW78] for an early treatment of PRAM's and [FRW84] for the first proof that a common write machine can simulate a CRAM with a linear increase in time and a squaring of the number of processors.

Theorem 5.33 is due to Ruzzo and Tompa in [SV84]. The equality NC$[t(n)] =$ ASPACE-TIME$[\log n, t(n)]$ in Exercise 5.34 is due to Ruzzo [Ruz81].

Lindell has made some interesting observations about the definition FO $=$ uniform AC^0, [L92]. The non-uniform version of Theorem 5.35 is due to Chandra, Stockmeyer, and Vishkin, [CSV84]. The uniform version appears in [I89a].

The ambiguous arithmetic notation used in Exercise 5.19 is from Borodin, Cook, and Pippenger, [BCP83].

More information about the NYU Ultracomputer and the Connection Machine may be found in [AG94, Hil85].