Introduction to Descriptive Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 600

Subsection 1

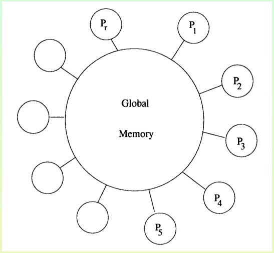
Concurrent Random Access Machines

Parallel Random Access Machines

- In a **parallel random access machine** (**PRAM**) every processor is connected with every other processor.
- A word of memory can be sent from any processor to any other processor in the time it takes to perform a single instruction.
- We define a precise model of PRAMs called Concurrent Random Access Machines.
  - This model is **synchronous**, i.e., the processors work in lock step.
  - It is also **concurrent**, i.e., at the same time step, several processors may:
    - Read from the same location;
    - Try to write the same location.

Concurrent Random Access Machines: Priority Write

- A **concurrent random access machine** (**CRAM**) consists of a large number of processors, all connected to a common, global memory.



- 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.

Concurrent Random Access Machines: Common Write

- In the “common write” model 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.

CRAM: Processor Registers

- Each processor has a finite set of registers, including the following:
  - Processor
    Contains the number between 1 and $p(n)$ of the processor;
  - Address
    Contains an address of global memory;
  - Contents
    Contains a word to be written or read from global memory;
  - ProgramCounter
    Contains the line number of the instruction to be executed next.

CRAM: Instruction Set

- 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 if R is less than zero.

Input and Output Conventions

- Initially, the contents of the first $|\text{bin}(\mathcal{A})|$ words of global memory contain one bit each of the input string $\text{bin}(\mathcal{A})$.
- But any other plausible setting of the input will work as well.
- 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.

Parallel Time Complexity

- 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$[t(n), p(n)]$.
- CRAM-PROC$[t(n), p(n)]$ 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 that the complexity class CRAM$[t(n)]$ is quite robust.
    - It is not affected by exactly how we place the input in the CRAM;
    - It is not affected by the global memory word size;
    - It is not affected by the size of the local registers;

  provided that the latter two are both polynomially bounded.

Subsection 2

Inductive Depth Equals Parallel Time

CRAM, Induction and Quantifiers

Theorem

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 uniform first-order induction whose depth, for structures of size n, is at most $t(n)$;
3. There exist:
   - A first-order quantifier-block $[QB]$;
   - A quantifier-free formula M_0;
   - A tuple $\overline{c}$ of constants,

   such that the query S for structures of size at most n is expressed as
   $$[QB]^{t(n)} M_0(\overline{c}/\overline{x}),$$
   i.e., the quantifier-block repeated $t(n)$ times followed by M_0.

CRAM, Induction and Quantifiers (Strategy)

Theorem (Cont'd)

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

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

- Note that the inclusion

$$\mathrm{IND}[t(n)] \subseteq \mathrm{FO}[t(n)]$$

  has already been proved.
- So we prove, first, that

$$\mathrm{CRAM}[t(n)] \subseteq \mathrm{IND}[t(n)].$$

- And, then, show that

$$\mathrm{FO}[t(n)] \subseteq \mathrm{CRAM}[t(n)].$$

CRAM and Induction

Lemma

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

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

- 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 $p(n)$.

  The following are all polynomially bounded:
  - The number of processors;
  - The time;
  - The memory word size.

  So 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.

CRAM and Induction (Cont'd)

- 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

  $$\text{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$$

  meaning that bit $\overline{x}$ in register r of processor $\overline{p}$ just after step t is equal to b.

  The base case is that if $\overline{t} = 0$, then memory is correctly loaded with $\text{bin}(\mathcal{A})$.

  This is first-order expressible.

  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 $\overline{p}$.

CRAM and Induction (Cont'd)

- The inductive definition of the relation $\text{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$ 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.

  Then we 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;
  - 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 $\text{bin}(\mathcal{A})$ if $i < |\text{bin}(\mathcal{A})|$, and zero otherwise.

CRAM and Induction (Cont'd)

- It remains to check that Addition, Subtraction, BLT and Shift are first-order expressible.

  Addition was handled in a previous proposition.

  Subtraction and Less Than may be expressed in a similar way.

  Relation BIT allows our first-order formulas to examine any of the $\log n$ bits of a domain variable. By a previous theorem, the addition relation on such variables is first-order expressible.

  Using addition, we can specify the Shift operation.

  Thus, we have sketched an inductive definition of relation VALUE, coding M's entire computation.

  Finally, note that one iteration of the definition occurs for each step of M.

Quantifiers and CRAM

Lemma

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

$$\mathrm{FO}[t(n)] \subseteq \mathrm{CRAM}[t(n)].$$

- Suppose the $\mathrm{FO}[t(n)]$ problem is determined by:
  - The quantifier free formulas $M_0, M_1, \ldots, M_k$;
  - The quantifier block $QB = (Q_1 x_1.M_1)\cdots(Q_k x_k.M_k)$;
  - The tuple of constants $\overline{c}$.

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

$$\varphi_n \equiv [QB]^{t(n)} M_0(\overline{c}/\overline{x}).$$

Quantifiers and CRAM (Cont'd)

- The CRAM will use n^k processors and n^{k-1} bits of global memory.

  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.

  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 $\varphi_n(\overline{c}/\overline{x})$.

  For $0 \leq r \leq t(n) \cdot k$ and i such that $r = k \cdot (q+1) + 1 - i$, let

  $$\varphi^r \equiv (Q_i x_i . M_i) \cdots (Q_k x_k . M_k) [QB]^q M_0.$$

  Denote the $(k-1)$-tuple resulting from $x_1 \ldots x_k$ by removing x_i by

  $$x_1 \ldots \widehat{x_i} \ldots x_k.$$

Quantifiers and CRAM (Cont'd)

- We give a program for the CRAM broken into rounds.

  Each round consists 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.

  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. $b := \text{loc}(a_1 \ldots \widehat{a}_{i+1} \ldots a_k)$;
    2. $\text{loc}(a_1 \ldots \widehat{a_i} \ldots a_k) := 0$;
    3. if $M_i(a_1, \ldots, a_k)$ and b then $\text{loc}(a_1 \ldots \widehat{a_i} \ldots a_k) := 1$;
  - $\{Q_i$ is $\forall\}$
    1. $b := \text{loc}(a_1 \ldots \widehat{a}_{i+1} \ldots a_k)$;
    2. $\text{loc}(a_1 \ldots \widehat{a_i} \ldots a_k) := 1$;
    3. if $M_i(a_1, \ldots, a_k)$ and $\neg b$ then $\text{loc}(a_1 \ldots \widehat{a_i} \ldots a_k) := 0$;

Quantifiers and CRAM (Cont'd)

- It is not hard to prove by induction that the displayed property holds.

  Thus, the CRAM simulates the formula.

  The bit fetched into b tells us whether $\mathcal{A}$ satisfies the formula

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

  The effect of lines 2 and 3 is that, in parallel, for all values of x_i, the truth of φ^r is tested and recorded.

  This completes the inductive step.

Quantifiers and CRAM (Cont'd)

- In the base case, at Step 1, processor $(a_1 \dots a_k)$ must set

$$b = 1 \quad \text{iff} \quad \mathcal{A} \models M_0(a_1, \dots, a_k).$$

  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 $\text{bin}(\mathcal{A})$.

  Furthermore, in constant time, it can compute the boolean combination of these bits indicated by M_0.

CRAM$[t(n)]$ and Format of the Input

Corollary

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 $O(\log n)$.

CRAM[$t(n)$] and Format of the Input (Cont'd)

Corollary (Cont'd)

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.

- One can show that the preceding lemmas still hold with any consistent set of these modifications.

Subsection 3

Number of Variables Versus Number of Processors

Introduction

- We 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 different pattern of interconnection between its processors and memory than the first-order inductive definition "model of parallelism".
- We analyze the proof of the preceding theorem to give processor-versus-variable bounds for translating between CRAM and IND.

Processors and Variables

Corollary

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,

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

- We prove these bounds using the following two lemmas.
- A previous lemma simulated a CRAM using an inductive definition.
- We defined relation VALUE, encoding the entire CRAM computation.

Lemma 1

Lemma

Let M be a CRAM-PROC$[t(n), n^{\pi}]$ machine, such that the maximum size of a register word and of $t(n)$ are both $o[\sqrt{n}]$. Then the inductive definition of VALUE may be written using $2\pi + 2$ variables.

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

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

  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.

  We must show each disjunct can be written using $2\pi + 2$ variables.

Lemma 1 (Cont'd)

- 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.

Lemma 1 (Cont'd)

- The last and most interesting case is R.
  Here we must express the following.
  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. $\text{Address}(\overline{p'}, t') = \text{Address}(\overline{p}, t)$;
     c. $\text{VALUE}(\overline{p'}, t', x, r, b)$;
     d. Processor $\overline{p'}$ wrote at time t';
     e. For all $\overline{p''} < \overline{p'}$, if $\overline{p''}$ wrote at time t', then $\text{Address}(\overline{p''}, t') \neq \text{Address}(\overline{p'}, t')$;
     f. For all t'' such that $t' < t'' < t$ and for all $\overline{p''}$, if $\overline{p''}$ wrote at time t'', then $\text{Address}(\overline{p''}, t'') \neq \text{Address}(\overline{p'}, t')$.

  On its face, this formula uses three $\overline{p'}$'s and three t's.

  However, we show that two copies of each suffice.

  Where we quantify $\overline{p''}$ in Lines 3e and 3f, we no longer need $\overline{p}$.

  So we may use these variables instead.

Lemma 1 (Cont'd)

- 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 $\overline{p}$ 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 $\text{Address}(\overline{p''}, t'') \neq \text{Address}(\overline{p'}, t')$, 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 $\text{Address}(\overline{p''}, t'')$;
  - $1 - b$ is the bit at position i of $\text{Address}(\overline{p'}, t')$.

  To help in expressing the first conjunct, we may use a variable from $\overline{p'}$.

  To help in the second conjunct, we may use a variable from $\overline{p}$.

  Thus $2\pi + 2$ variables suffice.

Lemma 2

Lemma

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.
The relation defined by φ is computable in $\text{CRAM-PROC}[d(n), O(n^k)]$.

- This is similar to the proof of $\text{FO}[t(n)] \subseteq \text{CRAM}[t(n)]$.

  Let T be the parse tree of φ.

  The CRAM will have $n^k|T|$ processors.

  There is 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 $\text{FO}[t(n)] \subseteq \text{CRAM}[t(n)]$, in rounds consisting of 3δ steps, the CRAM will evaluate an iteration of φ.

Lemma 2 (Cont'd)

- Let $r = \text{arity}(R) =$ 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 $R_t = \varphi^t(\varnothing)$.

  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.

The Factor 2

- Why is the number of variables needed to express a computation of n^π processors $2\pi + 2$, instead of π?
- 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;
  - We need to say that no lower numbered processor $\overline{p''}$ wrote into the same address as $\overline{p'}$.
- The factor of 2 would be eliminated if we adopted a weaker parallel machine model.
  - It would allow only common writes;
  - 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 Additive Factor 2

- 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 that we need only $O(\log \log n)$ bits to name a bit of a word.

Subsection 4

Circuit Complexity

Circuit Computation

- 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
$$\mathcal{C} = \{C_i : i = 1, 2, \ldots\},$$
where C_n is a circuit with n input bits and a single output bit r.
- For $w \in \{0,1\}^n$, $C_n(w)$ denotes 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 $w \in \{0,1\}^n$,
$$w \in S \quad \text{iff} \quad C_n(w) = 1.$$

Circuits

- Recall that 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.

A Normalization: Layered Circuits

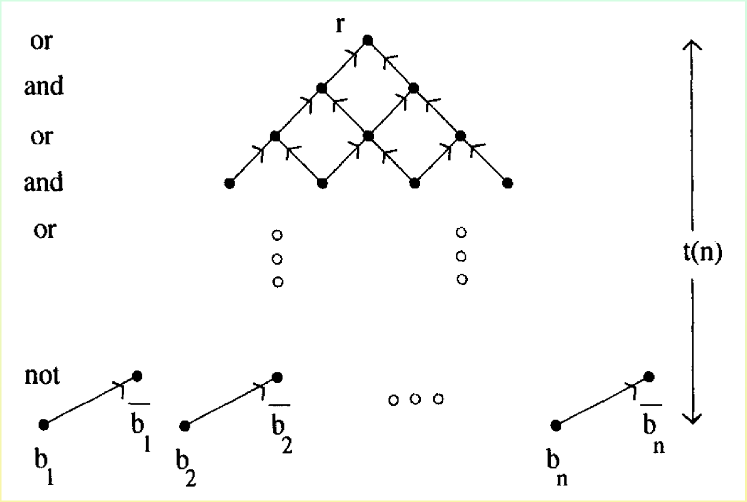
- It is convenient to assume that all the “not” gates in our circuits have been pushed down to the bottom.
- We can do this using the De Morgan Laws

$$\begin{array}{rcl}\neg(\alpha \wedge \beta) & \equiv & (\neg\alpha \vee \neg\beta);\\ \neg(\alpha \vee \beta) & \equiv & (\neg\alpha \wedge \neg\beta).\end{array}$$

- This transformation does not increase the depth, nor does it significantly increase the size of the circuit.
- Furthermore, we can assume that the levels alternate:
  - The top level consists of all “or” gates;
  - The next level consists of all “and” gates;

  $\vdots$
- Such a normalized circuit is called a **layered circuit**.

Drawing of a Layered Circuit



Vocabulary of Circuits

- We recall 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.
- The property that x is a leaf is expressed by
$$L(x) \equiv (\forall y)(\neg E(y,x)).$$
- The leaves need to be ordered $1, 2, \ldots$ so that we know where to place input bits $b_1, b_2, \ldots, b_n$.

Vocabulary of Circuits (Cont'd)

- 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:

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

- 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;
  - A not-gate if $G_\neg(w)$ holds.

Threshold Gates

- A **threshold gate** with **threshold value** i has output 1 iff at least i of its inputs have value 1.
- Note that threshold gates include as special cases:
  - "or" gates in which the threshold is 1;
  - "and" gates in which the threshold is equal to the number of inputs.
- We generalize the vocabulary of circuits to the **vocabulary of threshold circuits**,

$$\tau_{thc} = \tau_c \cup \{G_t^2\}.$$

- $G_t(g, v)$ means that g is a threshold gate with threshold value v.
- If $G_t(g, v)$ holds, the g takes value 1 in a circuit iff at least v of its inputs have value 1.

Accepting a Structure

- Let $\mathcal{A} \in \mathrm{STRUC}[\tau]$, with $n = \|\mathcal{A}\|$.
- A circuit C_n with $\widehat{n}_\tau(n)$ leaves can take $\mathcal{A}$ as input by placing the binary string $\mathrm{bin}(\mathcal{A})$ into its leaves.
- We write $C(w)$ to denote the output of circuit C on input w.
- That is, $C(w)$ denotes the value of the root node when:
  - w is placed at the leaves;
  - C is evaluated.
- We say that circuit C **accepts** structure $\mathcal{A}$ if and only if

$$C(\mathrm{bin}(\mathcal{A})) = 1.$$

Uniformity

Definition (Uniformity)

Let $\mathcal{C}$ be a sequence of circuits.
Let $\tau \in \{\tau_c, \tau_{thc}\}$ be the vocabulary of circuits or threshold circuits.
Let $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau]$ be a query such that, for all $n \in \mathbb{N}$,

$$I(0^n) = C_n.$$

That is, on input a string of n zero's, the query produces circuit n.

- If $I \in \mathrm{FO}$, then $\mathcal{C}$ is a **first-order uniform sequence** of circuits.
- If $I \in \mathrm{L}$, then $\mathcal{C}$ is a **logspace uniform sequence** of circuits.
- If $I \in \mathrm{P}$, then $\mathcal{C}$ is **polynomial-time uniform sequence** of circuits.

  $\vdots$

- Whether we use first-order, logspace or polynomial-time uniformity, any uniform sequence of circuits is polynomial-size.

Circuit Complexity

Definition (Circuit Complexity)

Let $t(n)$ be a polynomially bounded function.
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 $C_n = I(0^n)$, satisfying:

1. For all $\mathcal{A} \in \text{STRUC}[\tau]$, $\mathcal{A} \in S$ iff $C_{\|\mathcal{A}\|}$ accepts $\mathcal{A}$;
2. The depth of C_n is $O(t(n))$;
3. The gates of C_n consist, respectively, of binary "and" and "or" gates (NC), unbounded fan-in "and" and "or" gates (AC), unbounded fan-in threshold gates (ThC).

Circuit Complexity Classes

- For $i \in \mathbb{N}$, we define the following classes:

$$\begin{array}{rcl} \mathrm{NC}^i & = & \mathrm{NC}[(\log n)^i], \\ \mathrm{AC}^i & = & \mathrm{AC}[(\log n)^i], \\ \mathrm{ThC}^i & = & \mathrm{ThC}[(\log n)^i]. \end{array}$$

- Finally, let

$$\mathrm{NC} = \bigcup_{i=0}^{\infty} \mathrm{NC}^i.$$

Comments on the Three Classes

- The NC circuits correspond 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 the inclusions, proven rigorously later,

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

- We also see below that the unbounded fan-in gates in AC circuits correspond exactly to concurrent writing in the CRAM model.

Regular Languages and NC^1

- A **regular language** is a set of strings $S \subseteq \Sigma^*$ accepted by a finite automaton.
- A **finite automaton** is a Turing machine with no work tapes.

Proposition

Every regular language is in NC^1.

- We are given a deterministic finite automaton $D = \langle \Sigma, Q, \delta, s, F \rangle$. We must construct a first order query

$$I_D : \text{STRUC}[\tau_s] \to \text{STRUC}[\tau_c]$$

such that, letting $C_n = I_D(0^n)$, we have, for all strings $w \in \Sigma^*$,

$$w \in \mathcal{L}(D) \quad \text{iff} \quad C_{|w|} \text{ accepts } w.$$

Regular Languages and NC^1 (Cont'd)

- 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 $f_{\ell c}$ and f_{rc} of its left child and right child;
  - Computes their composition

    $$f_v = f_{rc} \circ f_{\ell c}.$$

Regular Languages and NC^1 (Cont'd)

- Inductively, the output of every node v is the function

  $$f_v = \delta^*(\cdot, w_v),$$

  where w_v is the subword of w that is sitting below v's subtree.

  In particular,

  $$w \in \mathcal{L}(D) \quad \text{iff} \quad f_r(s) \in F,$$

  where f_r is the mapping stored at the root.

  Now D is a fixed, finite state automaton.

  So 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 $O(\log n)$.

AC and FO

Theorem

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

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

- The equality of the first three classes has already been proven.
  We first show that $\text{FO}[t(n)] \subseteq \text{AC}[t(n)]$.
  Let S be a $\text{FO}[t(n)]$ boolean query given by:
  - The quantifier block $QB = [(Q_1x_1.M_1)\cdots(Q_kx_k.M_k)]$;
  - The initial formula M_0;
  - The tuple of constants $\overline{c}$.

  We must write a first-order query I, generating circuit $C_n = I(0^n)$, such that, for all $\mathcal{A} \in \text{STRUC}[\tau]$,

$$\mathcal{A} \vDash (QB^{t(\|\mathcal{A}\|)}M_0)(\overline{c}/\overline{x}) \quad \text{iff} \quad C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A}.$$

AC and FO (Cont'd)

- Initially the circuit evaluates the quantifier-free formulas
$$M_i, \quad i = 0, 1, \ldots, k.$$
The nodes $\langle M_i, b_1, \ldots, b_k \rangle$ will be the gates that have evaluated these formulas, i.e.,
$$\langle M_i, b_1, \ldots, b_k \rangle(\mathrm{bin}(\mathcal{A})) = 1 \quad \text{iff} \quad \mathcal{A} \vDash M_i(b_1, \ldots, b_k).$$
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
$$\langle 2r, b_1, \ldots, \widehat{b_i}, \ldots, b_k \rangle(\mathrm{bin}(\mathcal{A})) = 1 \quad \text{iff} \quad \mathcal{A} \vDash \varphi^r(b_1, \ldots, b_k).$$

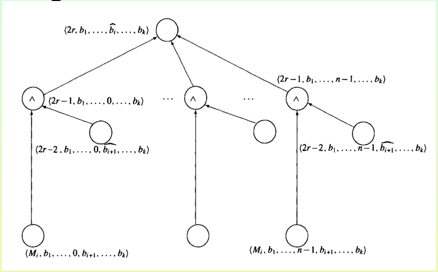
AC and FO (Cont'd)

- 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 depending on whether $Q_i = \forall$ 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 $|\mathcal{A}|$.


$\langle 2r, b_1, \ldots, \widehat{b_i}, \ldots, b_k \rangle$
$\langle 2r-1, b_1, \ldots, n-1, \ldots, b_k \rangle$
$\wedge$
$\langle 2r-1, b_1, \ldots, 0, \ldots, b_k \rangle$
$\cdots$
$\wedge$
$\cdots$
$\wedge$
$\langle 2r-2, b_1, \ldots, 0, \widehat{b_{i+1}}, \ldots, b_k \rangle$
$\langle 2r-2, b_1, \ldots, n-1, \widehat{b_{i+1}}, \ldots, b_k \rangle$
$\langle M_i, b_1, \ldots, 0, b_{i+1}, \ldots, b_k \rangle$
$\langle M_i, b_1, \ldots, n-1, b_{i+1}, \ldots, b_k \rangle$

  Each $\langle 2r-1, b_1, \ldots, b_i, \widehat{b_{i+1}}, \ldots, b_k \rangle$ is a binary "and" -gate whose inputs are $\langle M_i, b_1, \ldots, b_k \rangle$ and $\langle 2r-2, b_1, \ldots, b_i, \widehat{b_{i+1}}, \ldots, b_k \rangle$.

  The circuit we have described may be constructed via a first-order query I, and it satisfies the required equivalence.

AC and FO (Cont'd)

- We now show that $\mathrm{AC}[t(n)] \subseteq \mathrm{IND}[t(n)]$.

  Let $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau_c]$ be a first-order query, such that $C_n = I(0^n)$, $n = 1, 2, \ldots$, is a uniform sequence of $\mathrm{AC}[t(n)]$ circuits.

  We must write an inductive formula $\Phi \equiv (\mathrm{LFP}\varphi)(\overline{c})$, such that, for all $\mathcal{A} \in \mathrm{STRUC}[\tau]$,

  $$\mathcal{A} \vDash \Phi \quad \text{iff} \quad C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A}.$$

  Suppose $\mathcal{A}$ is given.

  Applying the query, we get

  $$C_{\|\mathcal{A}\|} = \langle E, G_\wedge, G_\vee, G_\neg, \mathrm{bin}(\mathcal{A}), r\rangle.$$

  The input string $I = \mathrm{bin}(\mathcal{A})$ is first-order describable from $\mathcal{A}$.

AC and FO (Cont'd)

- We construct a first-order inductive definition of the relation $V(x,b)$ meaning that gate x has boolean value b.
  We use:
  - DEFINED(x), meaning that x is ready to be defined,
    $$(\forall y)(\exists c)(E(y,x) \to V(y,c));$$
  - $C(x)$, saying that all of x's inputs are true,
    $$C(x) \equiv (\forall y)(E(y,x) \to V(y,1));$$
  - $D(x)$, saying that some of x's inputs are true,
    $$D(x) \equiv (\exists y)(E(y,x) \wedge V(y,1));$$
  - $N(x)$, saying that its input is false,
    $$N(x) \equiv (\exists! y)(E(y,x)) \wedge (\exists y)(E(y,x) \wedge V(y,0)).$$

AC and FO (Cont'd)

- Then we have

$$\begin{aligned} V(x,b) \quad \equiv \quad & \mathrm{DEFINED}(x) \wedge [L(x) \wedge (I(x) \leftrightarrow b) \vee \\ & G_\wedge(x) \wedge (C(x) \leftrightarrow b) \vee G_\vee(x) \wedge (D(x) \leftrightarrow b) \\ & \vee G_\neg(x) \wedge (N(x) \leftrightarrow b)]. \end{aligned}$$

The inductive definition of V closes in exactly the depth of C_n.

So it takes $O(t(n))$ iterations.

After the last iteraton, $\Phi \equiv V(r,1)$ expresses the acceptance condition in $\mathrm{IND}[t(n)]$.

So we have

$$\mathcal{A} \models \Phi \quad \text{iff} \quad C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A}.$$

Characterization of NC

- A corollary of the theorem is the following characterization of the class NC.

Corollary

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

Subsection 5

Alternating Complexity

The Logarithmic Time Hierarchy

- Let the **logarithmic time hierarchy** (LH) be defined by

$$\mathrm{LH} := \mathrm{ATIME\text{-}ALT}[\log n, O(1)].$$

- This is 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.

Theorem

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

Deterministic Logarithmic Time and FO

- The most delicate part of the proof is the following

Lemma

$\mathrm{DTIME}[\log n] \subseteq \mathrm{FO}$.

- Let T be a $\mathrm{DTIME}[\log n]$ machine.

  We must write a first-order sentence φ, such that, for all inputs $\mathcal{A}$,

$$T(\mathrm{bin}(\mathcal{A}))\downarrow \quad \text{iff} \quad \mathcal{A} \vDash \varphi.$$

  The sentence φ will begin with existential quantifiers,

$$\varphi \equiv (\exists x_1 \ldots x_c)\psi(\overline{x}).$$

  The variables $\overline{x}$ will code the $\log n$ steps of T's computation.

Deterministic Logarithmic Time and FO (Cont'd)

- For each step t of T's computation, the coding by $\overline{x}$ will include the values:
  - q_t, representing T's state;
  - w_t, representing the symbol it writes;
  - d_t, representing the direction its head moves;
  - I_t representing the value of the input being scanned by the index-tape-controlled input head.

  It is important to remember that:
  - Each variable is a $\lceil \log n \rceil$ bit number;
  - The numeric predicate BIT allows these bits to be specified.

Deterministic Logarithmic Time and FO (Cont'd)

- The formula ψ must now assert that the information in $\overline{x}$ meshes together to form a valid accepting computation of T.

  To accomplish this, we must define the first-order relations:
  - $C(p,t,a)$, the contents of cell p at time t is a;
  - $P(p,t)$, meaning that for the computation determined by $\overline{x}$, the work head is at position p at time t.

  Given C and P we can assert that $\overline{x}$ is self-consistent.

Deterministic Logarithmic Time and FO (Cont'd)

- Note, for example, that we can:
  - Guess the contents y of the index tape;
  - Then use C to verify that y is correct;
  - Next, using y, we can verify that the input symbol I is correct.

  Next, note that, using P, we can write C.

  This is 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 using BIT, by a previous result.

The Logarithmic Time Hierarchy (Cont'd)

- Now we show LH $\subseteq$ FO.

  Consider an alternating logarithmic-time machine.

  It may be assumed to:
  - Write its guesses on a work tape;
  - Then deterministically check for acceptance.

  By hypothesis, there are a bounded number of alternations.

  Moreover, the total time is $O(\log n)$.

  So these guesses may be simulated by a bounded number of first-order quantifiers.

  The remaining work is in DTIME$[\log n]$.

  Thus, by the lemma, it is in FO.

The Logarithmic Time Hierarchy (Cont'd)

- We must, finally, show that FO $\subseteq$ LH.

  Consider a first-order sentence
  $$\varphi \equiv (\exists x_1)(\forall x_2)\cdots(Q_k x_k)M(\overline{x}).$$
  We must show that, there exists an ATIME-ALT$[\log n, 0(1)]$ machine T, such that, for all input strings $\mathcal{A}$,
  $$T(\text{bin}(\mathcal{A}))\downarrow \quad \text{iff} \quad \mathcal{A} \vDash \varphi.$$
  Note that M is a constant size quantifier-free formula.

The Logarithmic Time Hierarchy (Cont'd)

- So 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
$$\mathcal{A} \vDash M(\overline{a}).$$
The most complicated part of this is to verify the BIT predicate.
This requires counting in binary up to $O(\log n)$ on a work tape, which is straightforward.
Thus, using $k-1$ alternations between existential and universal states, a Σ_k logarithmic-time machine can:
    - Guess $a_1, \ldots, a_k$;
    - Then deterministically verify $M(\overline{a})$.

Characterizations of FO

- Notice that by the theorems in this set, we now have three interesting characterizations of the class FO.

Corollary

$\mathrm{FO} = \mathrm{AC}^0 = \mathrm{CRAM}[1] = \mathrm{LH}$.

- The truth of the corollary depends on our choice of:
  - Including BIT as a numeric predicate;
  - Including the SHIFT operation in the CRAM;
  - Our definition of uniformity for AC^0.

AC and Alternation

Theorem

For $t(n) \geq \log n$,

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

- We have already seen that $\text{AC}[t(n)] = \text{FO}[t(n)]$.

  We now sketch $\text{AC}[t(n)] \subseteq \text{ASPACE-ALT}[\log n, t(n)]$.

  Let $t(n) \geq \log n$ and consider the same $\text{AC}[t(n)]$ boolean query I, as in the proof that $\text{AC}[t(n)] \subseteq \text{IND}[t(n)]$, with $I(0^n) = C_n$.

  Now $\text{ASPACE-ALT}[\log n, t(n)] \supseteq \text{ATIME-ALT}[\log n, 1] = \text{LH}$.

  So, by the preceding theorem, $\text{ASPACE-ALT}[\log n, t(n)] \supseteq \text{FO}$.

  Thus, the circuit C_n is available in $\text{ASPACE-ALT}[\log n, t(n)]$.

AC and Alternation (Cont'd)

- We can simulate an $\mathrm{AC}[t(n)]$ circuit via an $\mathrm{IND}[t(n)]$ definition.

  Looking at the simulation 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 $\mathrm{ASPACE\text{-}ALT}[\log n, t(n)]$ machine:
  - Each universal quantifier is simulated by $\log n$ universal moves;
  - 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.

AC and Alternation (Cont'd)

- Next, we sketch ASPACE-ALT$[\log n, t(n)] \subseteq$ AC$[t(n)]$.

  Let M be an ASPACE-ALT$[\log n, t(n)]$ machine.

  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$_M$(ID$_1$, ID$_2$) mean "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.

  EPATH and APATH are expressible in IND$[\log n]$.

  Thus, the following simultaneous induction has depth $O(\log n + t(n))$ and expresses the acceptance condition for M as desired:

$$\begin{array}{rcl}\mathrm{ACCEPT}_M(\mathrm{ID}_1) & \equiv & \mathrm{ID}_1 \text{ is the accept ID} \vee (\exists \mathrm{ID}_2)[\mathrm{ACCEPT}_M(\mathrm{ID}_2) \\ & & \wedge(\mathrm{EPATH}(\mathrm{ID}_1,\mathrm{ID}_2) \vee \mathrm{APATH}(\mathrm{ID}_1,\mathrm{ID}_2))].\end{array}$$

NC and Alternation (Cont'd)

- We state a similar characterization of $\mathrm{NC}[t(n)]$ without proof.

Theorem

For $t(n) \geq \log n$,

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

- The proof is similar to that of the preceding theorem.
- The difference is that the definitions of C and D in $V(x, b)$ now involve binary "and"s and "or"s rather than universal and existential quantifiers.

NC and AC

- For $i \geq 1$, the bound $\mathrm{NC}^i \subseteq \mathrm{AC}^i$ is not optimal.
- The following improvement is known to be optimal.
- This is because the NC^1 query PARITY requires depth $\frac{\log n}{\log \log n}$.

Theorem

For $t(n) \geq \log n$,

$$\mathrm{NC}[t(n)] \subseteq \mathrm{AC}\left[\frac{t(n)}{\log \log n}\right].$$

- We prove, equivalently,

$$\text{ASPACE-TIME}[\log n, t(n)] \subseteq \mathrm{IND}\left[\frac{t(n)}{\log \log n}\right].$$

NC and AC (Cont'd)

- Let M be an ASPACE-TIME$[\log n, t(n)]$ machine.

  We inductively define the acceptance condition ACCEPT$_M$ 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.

  We need to improve this simulation by a $\log\log n$ factor.

NC and AC (Cont'd)

- A list of which existential moves to make in the event of each possible sequence of $\frac{\log\log n}{2}$ 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 $\frac{\log\log n}{2}$ universal moves.

  Let a be such a sequence of universal moves.

  We can inductively define the relation

$$\mathrm{MOVES}_M(e, u, \mathrm{ID}_1, \mathrm{ID}_2)$$

  meaning "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 $\log\log n$.

NC and AC (Cont'd)

- Our definition of ACCEPT_M is then a simultaneous inductive definition with MOVES_M.

  Namely, we have

$$\begin{array}{rcl} \mathrm{ACCEPT}_M(\mathrm{ID}_1) & \equiv & \mathrm{ID}_1 \text{ is the accept ID} \\ & & \vee(\exists e)(\forall u)(\exists \mathrm{ID}_2) \\ & & (\mathrm{MOVES}_M(e,u,\mathrm{ID}_1,\mathrm{ID}_2) \\ & & \qquad \wedge \mathrm{ACCEPT}_M(\mathrm{ID}_2)). \end{array}$$

  The depth of this simultaneous induction is $\frac{t(n)}{\log\log n}$.