CRAM, ID = PT and Circuit Complexity

Vasilis Stamatis

MSc. programme in "Algorithms, Logic and Discrete Mathematics", NKUA & NTUA

Descriptive Complexity | Spring 2023

May 2023

Intro

1. Parallel computing refers to the process of breaking down larger problems into smaller, independent, often similar parts that can be executed simultaneously by multiple processors communicating via shared memory, the results of which are combined upon completion as part of an overall algorithm.
2. The primary goal of parallel computing is to increase available computation power for faster application processing and problem solving. Not for problems in NP, a problem there that takes 400 billion centuries to solve on a uniprocessor, would still take 400 centuries even if it can be perfectly parallelized over 1 billion processors.

Intro

1. Thus, in parallelism we can take advantage of many different processors or computers working simultaneously
2. For a long time, computers were sequential devices having a single processor and thus executing one instruction at a time. We have increased the ability to produce small, fast, inexpensive proccessors.
3. Over time it became possible to build large parallel computers as well as PCs that can interact.

Intro

1. 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 i.e. if $\phi \in FO(LFP)$ the min r such that $(\phi^{\mathcal{A}})^r(\emptyset) = (\phi^{\mathcal{A}})^{r+1}(\emptyset)$
2. Close relation between the amount of hardware used - memory and processors - and the number of variables in the inductive definition.
3. The query $(\exists x)S(x)$ can be executed using n processors in constant (parallel) time. The processor p_i, for every $i \in [n-1]$ checks wether the $S(i)$ holds. Any p_i for which $S(i)$ does hold should write $\mathbf{1}$ into a specific location in global memory that was originally $\mathbf{0}$.

Intro

1. Question: How to make the **use** of these machines efficient.
2. Numerous models of parallel computation have been developed. These models vary on how **tightly coupled** these processors are:
   - 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.
   - Distributed Computation where Many PCs or WSs are connected via a network, which might be fairly fast and local - or it might be the internet.
3. 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.

Parallel Random Access Machine (PRAM)

A PRAM is a model, which is considered for most of the parallel algorithms. Here, multiple processors are attached to a single block of memory. A PRAM model has the following properties:

- A set of similar type of processors.
- All the processors share a common memory unit. Processors can communicate among themselves through the shared memory only.
- A memory access unit (MAU) connects the processors with the single shared memory.

Here, n number of processors can perform independent operations on n number of data in a particular unit of time. This may result in simultaneous access of same memory location by different processors. Therefore, if there are n processors in a PRAM, then n number of independent operations can be performed in a particular unit of time. definition.

Parallel Random Access Machine (PRAM)

- The reason we care about PRAMs is that parallelism as on a PRAM 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.

Parallel Random Access Machine (PRAM)

Categories of PRAMs:

1. Exclusive read exclusive write (EREW): every memory cell can be read or written to by only one processor at a time
2. Concurrent read exclusive write (CREW): multiple processors can read a memory cell but only one can write at a time
3. **Concurrent read concurrent write (CRCW): multiple processors can read and write.**
    1. Common: all processors write the same value; otherwise is illegal
    2. Arbitrary: only one arbitrary attempt is successful, others retire
    3. **Priority: processor rank indicates who gets to write**

Parallel Random Access Machine (PRAM)

- Therefore, by using the **common write** model, the program guarantees that different values will never be written to the same location at the same time.
- And we find this to be the more natural model for logic : a formula such $(\forall x)\phi$ 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 $\mathbf{0}$ into a location in global memory that was initially $\mathbf{1}$.

Concurrent Random Access Machines

CRAMs are a special case of the concurrent read, concurrent write, parallel random access machine (CRCW-PRAM) . A 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 number processor succeeds (priority write) Properties:

- **Synchronous:** Processors work in lock step.
- **Concurrent:** Several processors may read from the same location at the same time and several processors may try to write the same location at the same time step

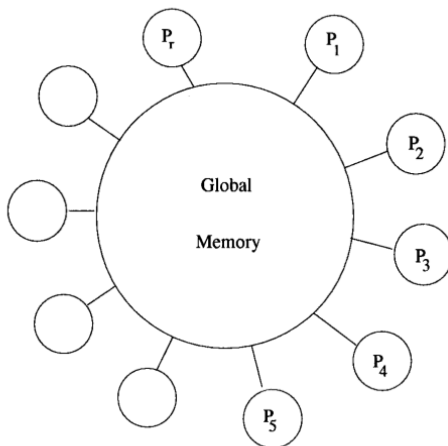

Figure 5.1: A concurrent random access machine (CRAM)

Example of the operation of a CRCW-PRAM

Example: Suppose we wish to add an array consisting of n numbers.

- We generally iterate through the array and use n steps to find the sum of the array. So, if the size of the array is n and for each step, let's assume the time taken to be 1 second. Therefore, it takes n seconds to complete the iteration.
- The same operation can be performed more efficiently using a CRCW model of a PRAM. Let there be $n/2$ parallel processors for an array of size n, then the time taken for the execution is 4 which is less than $n = 6$ seconds in the following illustration.

Title of the Slide

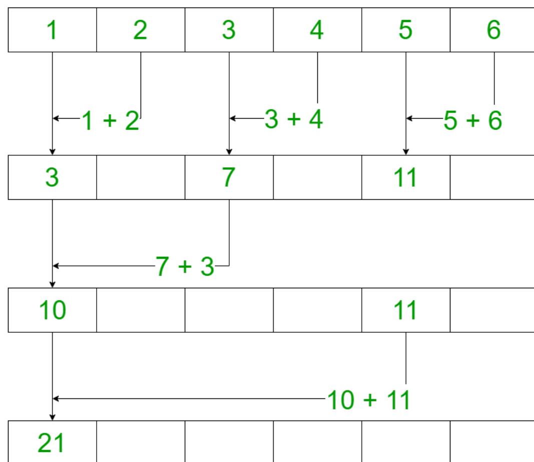

Figure: Example

Concurrent Random Access Machines

Description of the instruction set of CRAM : We need to describe this model in a level that is detailed enough so that we may prove its equivalence to the descriptive model. We can then write our parallel programs using FO logic.

Description of the instruction set of CRAM

Each RAM has a finite set of registers, including the following:

- Processor: containing the number between 1 and $p(n)$ of the RAM
- Address: containing an address of global memory
- Contents: containing a word to be written or read from global memory
- ProgramCounter: containing the line number of the instruction to be executed next.

All RAMs are **identical** except the Processor number. These registers provide the processor with access to the global memory and allow it to execute instructions on that memory. The finite number of registers limits the computational power of each processor, but the ability to share the global memory with other processors compensates for this limitation.

Description of the instruction set of CRAM

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 RL: Branch to line L if the contents of R is less than zero.

Description of the instruction set of CRAM

- We assume initially that the contents of the first $|bin(\mathcal{A})|$ words of global memory contain one bit each of the input string $bin(\mathcal{A})$.
- This is not necessary.
- We'll also 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.

Concurrent Random Access Machines

Description of the instruction set of CRAM :

- The Shift instruction for the CRAM, allows each bit of global memory to to be available to each processor in constant time. Without Shift, CRAM$[t(n)]$ would be too weak to simulate $FO[t(n)]$, $t(n) < logn$.
- The BIT instruction, denoted BIT(i, j), tests whether the j-th bit in the binary representation of i is 1. In first-order logic, this instruction can be added as a numeric predicate, enabling the expression of formulas that can access individual bits in a given variable.

Remember: $FO[t(n)]$ is the set of properties defined by quantifier blocks iterated $t(n)$ times

Concurrent Random Access Machines

The measure of parallel time complexity will be time on a CRAM:

Definition

$CRAM[t(n)]$ is defined 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 $CRAM - PROC[t(n), p(n)]$

Definition

$CRAM - PROC[t(n), p(n)]$ is defined to be the set of boolean queries computable by a CRAM using at most $p(n)$ processors and time $O(t(n))$.

Thus, $CRAM[t(n)] = CRAM - PROC[(t(n), n^{O(1)}]$

Inductive Depth = Parallel Time

As said earlier, the parallel time is identical to inductive depth of a FO induction.
In other words, a depth-optimal first-order inductive description of a query is a parallel-time-optimal algorithm to compute the query. are equal.
The following theorem states states this formally and also completes the circle and shows that number of quantifier-block iterations and inductive depth

Inductive Depth = Parallel Time

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 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 $[QB]^{t(n)} M_0(\bar{c}/\bar{x})$, i.e., the quantifier-block repeated $t(n)$ times followed by M_0 .*

In other words:

$$CRAM[t(n)] = IND[t(n)] = FO[t(n)]$$

Inductive Depth = Parallel Time (The Proof)

Already shown :

Theorem

(Euergetes et. al.) *For all $t(n)$ and all classes of finite structures*

$$IND[t(n)] \subseteq FO[t(n)]$$

Remains to prove:

1. $CRAM[t(n)] \subseteq IND[t(n)]$.
2. $FO[t(n)] \subseteq CRAM[t(n)]$.

$CRAM[t(n)] \subseteq IND[t(n)]$

Lemma

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

$$CRAM[t(n)] \subseteq IND[t(n)]$$

Proof.

- **We simulate a CRAM** M : On input $\mathcal{A}$, a structure of size n, M runs in $t(n)$ synchronous steps, using $p(n)(= poly(n))$ processors.
- Since the number of processors, the time and the memory word size are all polynomially bounded, we need only a constant number of variables $x_1, \dots, x_k$ each ranging over universe of $\mathcal{A}$ (where $|| \mathcal{A} ||= n$), 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.

$CRAM[t(n)] \subseteq IND[t(n)]$

Proof.

- We now write a first-order inductive definition for the relation $VALUE(\bar{p}, \bar{t}, \bar{x}, r, b)$ meaning that bit $\bar{x}$ in register r of processor $\bar{p}$ just after step $\bar{t}$ is equal to b.
- **Base case:** If $\bar{t} = 0$, then memory is correctly loaded with $bin(\mathcal{A})$. 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 $\bar{p}$.
- The inductive definition of the relation $VALUE(\bar{p}, \bar{t}, \bar{x}, r, b)$ is a disjunction depending on the value of $\bar{p}$'s program counter at time $\bar{t} - 1$.

$CRAM[t(n)] \subseteq IND[t(n)]$

Proof.

- It Remains to check that **Addition, Subtraction, BLT, and Shift** are first-order expressible.
- They are. (exc 2.3, prop. 1.9, Theorem 1.17 of Immerman)
- 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.

- 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.
- We've done that.
- Next.

Inductive Depth = Parallel Time (The Proof)

Lemma

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

$$FO[t(n)] \subseteq CRAM[t(n)]$$

Proof.

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

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

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

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

$FO[t(n)] \subseteq CRAM[t(n)]$

Proof.

- 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 $0 < a_i < n$.
- Using the Shift operation it can retrieve each of the a_i's in constant time (we can just let each processor break its processor number into k $\lceil logn \rceil$-tuples of bits. If any of these is greater than or equal to n, then the processor should do nothing during the entire computation).

$FO[t(n)] \subseteq CRAM[t(n)]$

Proof.

- The CRAM will evaluate ϕ from right to left, simultaneously for all values of the variables $x_1, \dots x_n$. At its final step, it will output the bit $\phi_n(\bar{c}/\bar{x})$.
- For $0 \leq r = k \cdot (q-1) - i + 1 \leq t(n) \cdot k$, let:

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

Then,

$$\phi^1 \equiv (Q_k x_k.M_k)M_0 \ ; \ \phi^2 \equiv (Q_{k-2} x_{k-1}.M_{k-1})(Q_k x_k.M_k)M_0$$

$$\phi^k \equiv [QB]M_0 \ ; \ \phi^{k+1} \equiv (Q_k x_k.M_k)[QB]M_0$$

$$\phi^{t(n)\cdot k} \equiv [QB]^{t(n)} M_0$$

$FO[t(n)] \subseteq CRAM[t(n)]$

Proof.

- Also 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 $a_1 \ldots \hat{a}_i \ldots a_k$ is 1 or 0 according as whether $\mathcal{A} \models \phi^r(a_1, \ldots, a_k)$ or not. Note that x_i does not occur free in ϕ^r.
- Equivalently , Each processor $a_1, \ldots a_k$ at step $r+1$ sets $b := 1$ iff $\mathcal{A} \models \phi^r$

$FO[t(n)] \subseteq CRAM[t(n)]$

Proof.

- Proof via induction to r: At step 1, processor $a_1 \dots a_k$ must set: $b = 1$ **iff** $\mathcal{A} \models M_0(a_1, \dots, a_k)$
- At round r, processor number $a_1 \dots a_k$ executes the following three instructions according to whether Q_i is $\exists$ or Q_i is $\forall$:
  - $\{Q_i \equiv \exists\}$
    1. $b := loc(a_1 \dots \hat{a}_{i+1} \dots a_k)$
    2. $loc(a_1 \dots \hat{a}_i \dots a_k) := 0$
    3. if $M_i(a_1, \dots, a_k)$ and b then $loc(a_1 \dots \hat{a}_i \dots a_k) := 1$
  - $\{Q_i \equiv \forall\}$
    1. $b := loc(a_1 \dots \hat{a}_{i+1} \dots a_k)$
    2. $loc(a_1 \dots \hat{a}_i \dots a_k) := 1$
    3. if $M_i(a_1, \dots, a_k)$ and then $loc(a_1 \dots \hat{a}_i \dots a_k) := 0$
- Just after round r, the contents of memory location $a_1 \dots \hat{a}_i \dots a_k$ is 1 or 0 according as whether $\mathcal{A} \models \phi^r(a_1, \dots a_k)$ or not and thus that the CRAM simulates the formula.

□

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.
- **Now, in this part of the presentation, we will** define the major circuit complexity classes, the related connections between circuits and the other models of parallel computation, i.e., CRAMs, alternating machines, and first-order inductive definitions.

Circuit Complexity

We begin by asking ourselves why we can we see a (boolean) circuit as a graph:

Definition

A **boolean circuit** is DAG:

$$C = (V, E, G_\wedge, G_\vee, G_\neg, I, r) \in STRUC[\tau_c]$$

Internal node w is an and-gate if $G_\wedge$ holds, an or-gate if $G_\vee$ holds, and a not-gate if $G_\neg$ holds. The nodes v with no edges entering them are called leaves, and the input relation $I(v)$ represents the fact that the leaf v is on. Often we will be given a circuit C, and separately we will be given its input relation I.

CVP: The computational problem of computing the output of a given Boolean circuit on a given input.

Circuit Complexity

- A circuit can be represented as 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 also convenient to assume that all the "not" gates in our circuits have been pushed down to the bottom and also 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**.

Layered Circuit

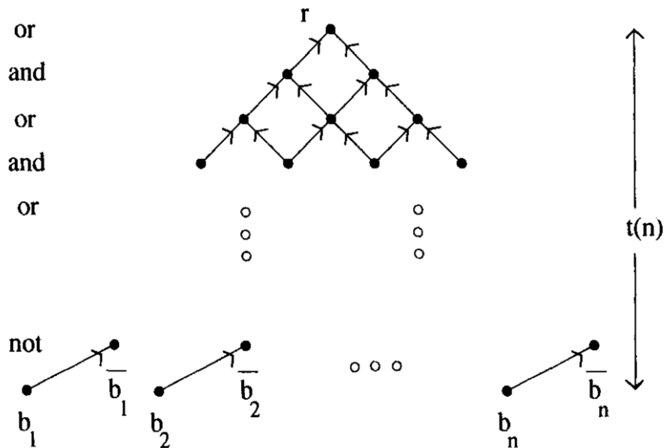

Figure 5.15: A layered circuit of depth $t(n)$.

Circuit Complexity

- We know that every query of a $STRUC[\tau]$ can be represented as a FO query of binary strings:

$$bin_\tau : STRUC[\tau] \to STRUC[\tau_s]$$

  Let $S \subseteq STRUC[\tau_s]$ be a boolean query on binary strings, let $||S|| = n$
- In circuit complexity, S would be computed by an infinite sequence of circuits $\mathcal{C} = \{C_i \mid i = 1, 2, \ldots\}$ where C_n is a circuit with n input bits and a single output bit r.
  Thus, C_n can take S as an input by placing $bin_\tau(S)$ into it's leaves.
- 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 $w \in \{0,1\}^n$,

$$w \in S \textbf{ iff } C_n(w) = 1$$

Circuit Complexity

- Below, we'll define the three families of circuit complexity classes which will concern us in this presentation.

Definition

- 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.
- Recall $\tau_c =< E^2, G^1_\wedge, G^1_\vee, G^1_\neg, I^1, r >$ the vocabulary of circuits. 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 generalize the vocabulary of circuits to the **vocabulary of threshold circuits**, $\tau_{thc} := \tau_c \cup \{G^2_t\}$, where $G_t(g, k)$ means that g is a threshold gate with threshold value k.

Circuit Complexity

Let $A \in STRUC[\tau]$ and let $n = ||\mathcal{A}||$. A circuit C_n with $\hat{n_\tau}$ leaves can take $\mathcal{A}$ as input by placing the binary string $bin(\mathcal{A})$ 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 C is then evaluated. We say that circuit C accepts structure $\mathcal{A}$ iff $C(bin(\mathcal{A})) = 1$

Circuit Complexity

- 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.
- 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 $I : STRUC[\tau_s] \rightarrow STRUC[\tau_c]$ with $C_n = I(0^n)$, $n = 1, 2, \ldots$. Here $0^n \in STRUC[\tau_s]$ is the string consisting of n zeros. Note that this uniformity condition implies that C_n has polynomially bounded size.

Circuit Complexity

Definition

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

Definition

Equivalent definition of uniformicity: Let U be a small uniform complexity class (like LOGSPACE) and let C be a circuit class. Then the class U-uniform C is defined to be the: set of languages recognized by a circuit family $\{C_n\}$ from C, and there is an algorithm A implementable in U such that $A(1^n)$ prints C_n as output.

Circuit Classes

We are now ready to 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 $p(n)$.

Circuit Complexity

Definition

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

1. For all $\mathcal{A} \in STRUC[\tau]$, $\mathcal{A} \in S$ **iff** $C_{||\mathcal{A}||}$ accepts $\mathcal{A}$
2. The depth of C_n is $O(t(n))$
3. The gates of C_n consist of binary "and" and "or" gates (NC), unbounded fan-in "and" and "or" gates (AC), and unbounded fan-in threshold gates (ThC), respectively.
   Let $NC^i = NC[(logn)^i]$, $AC^i = AC[(logn)^i]$ and $ThC^i = ThC[(logn)^i]$. Finally, let $NC = \bigcup_{i>0} NC^i$

Circuit Classes

- Let $\mathcal{B} = \{\forall n \mid f : \{0,1\}^n \rightarrow \{0,1\}$ be a set of Boolean functions, which we call a basis set. The **fan-in** of a function $g \in \mathcal{B}$ is the number of inputs that g takes. (Typical choices are fan-in 2, or unbounded fan-in, meaning that g can take any number of inputs.)
- In 1979, Niclaus Pippenger suggested that efficiently parallelizable problems in P might be defined as those problems that can be solved in a time period that is at most polylogarithmic in the problem size n, i.e., $O(log^k n)$ for some constant k, using no more than a polynomial number $O(n^k)$ of processors. This class of problems was later named Nick's Class (NC) in his honor. The class NC has been extensively studied and forms a foundation for parallel complexity theory.

Circuit Classes

- NC^k = languages computed in $O(log^k n)$ depth, polynomial size, bounded fan-in A particularly well-studied case is NC^1 , which is equivalent in power to polynomial size Boolean formulas.
- The NC circuits correspond reasonably well to standard silicon-based hardware.
- A weaker form of NC, known as the parallel computation thesis, is stated as follows: Anything that can be computed on a Turing machine using polynomially (polylogarithmically) bounded space in unlimited time can be computed on a parallel machine in polynomial (polylogarithmic) time using an unlimited number of processors, and vice versa.

Circuit Classes

- $AC^{k-1}[m]$ = languages computed in constant-depth, polynomial size, unbounded fan-in over the basis AND, OR, $MODm$, where a $MODm$ gate outputs 1 iff the sum of its input bits is divisible by m. Note that NOT can be simulated with MODm, but it is open whether AND (or OR) can be simulated with only $MODm$ s in constant depth.
- ThC^{k-1} = languages computed in $O(log^{k-1}n)$ depth, polynomial size, unbounded fan-in over MAJORITY gates (with NOTs for free). A MAJORITY gate outputs the most popular input; if there is a tie then it outputs 1.
- 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 $O(logn)$ time delay. On the other hand, once we have such a binary tree, we can also compute threshold functions.

This explains the following:

$$AC[t(n)] \subseteq ThC[t(n)] \subseteq NC[t(n)logn]$$

Circuit Complexity

Theorem

Every regular language is in NC^1

Proof.

Scheme

- Given a DFA We must construct a FO query $I_d : STRUC[\tau_s] \rightarrow STRUC[\tau_c]$ such that for all strings $w \in \Sigma^*$

$$w \in \mathcal{L}(D) \textbf{ iff } C_{|w|=n} = I_D(0^n) \textbf{ accepts } w$$

- Circuit C_n is a complete binary tree with n leaves. The input to leaf $L(i)$ is Wi, 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
- 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 $O(logn)$ as desired.

Circuit Complexity

Theorem

The boolean majority query MAJ is in NC^1

$$MAJ = \{\mathcal{A} \in STRUC[\tau_s] \mid \text{ string } \mathcal{A} \text{ contains more than } \frac{||\mathcal{A}||}{2} \text{ "1"'s } \}$$

Proof.

Hint: Build an NC^1 circuit for majority by adding the n input bits via a full binary tree of height $logn$, by using the ambiguous notation □

Circuit Complexity

Theorem

$\forall i \in \mathbb{N}$

$$NC^i \subseteq AC^i \subseteq ThC^i \subseteq NC^{i+1}$$

Proof.

- Remember, $NC^k =$ languages computed in $O(log^k n)$ depth, polynomial size, bounded fan-in. $AC^{k-1}[m] =$ languages computed in constant-depth, polynomial size, unbounded fan-in over the basis AND, OR, $MODm$ (we can stimulate negation with MODm). Thus, $NC^i \subseteq AC^i$.
- For the second part, $ThC^i \subseteq NC^{i+1}$: We can simulate any ThC-gate using a circuit of depth $logn$ recognising MAJ. Let threshold gate with threshold value k and boolean input w.
  - If $k \leq ||w||/2$ we are just checking if $w1^{||w||-2k} \in MAJ$
  - If $k > ||w||/2$ we are just checking if $w0^{2k-||w||} \in MAJ$
  - It is known that inclusion is strict for $i = 0$.

Circuit Complexity

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.

Circuit Complexity

Theorem

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

$$CRAM[t(n)] = IND[t(n)] = FO[t(n)] = AC[t(n)]$$

- NC Hierarchy:

$$NC^1 \subseteq NC^2 \subseteq \ldots \subseteq NC^i \subseteq NC$$

- $NC^1 \subseteq L \subseteq NL \subseteq AC^1 \subseteq NC^2 \subseteq P$
- Is NC Proper? : if $NC^i = NC^{i+1}$ for some i, then $NC^i = NC^j$ for all $j \geq i$, and as a result, $NC^i = NC$. This observation is known as NC-hierarchy collapse because even a single equality in the chain of containments.