

FOUNDATIONS OF COMPLEXITY THEORY

Lecture 19: Circuit Complexity

David Carral
Knowledge-Based Systems

TU Dresden, January 1, 2021

Computing with Circuits

David Carral, January 1, 2021

Foundations of Complexity Theory

slide 2 of 47

Motivation

One might imagine that $P \neq NP$, but **SAT** is tractable in the following sense: for every ℓ there is a very short program that runs in time ℓ^2 and correctly treats all instances of size ℓ . – Karp and Lipton, 1982

Some questions:

- Even if it is hard to find a universal algorithm for solving all instances of a problem, couldn't it still be that there is a simple algorithm for every fixed problem size?
- What can complexity theory tell us about parallel computation?
- Are there any meaningful complexity classes below LogSpace? Do they contain relevant problems?

↪ circuit complexity provides some answers

Intuition: use circuits with logical gates to model computation

Boolean Circuits

Definition 19.1: A Boolean circuit is a finite, directed, acyclic graph where

- each node that has no predecessor is an **input node**
- each node that is not an input node is one of the following types of **logical gate**:
 - **AND** with two input wires
 - **OR** with two input wires
 - **NOT** with one input wire
- one or more nodes are designated **output nodes**

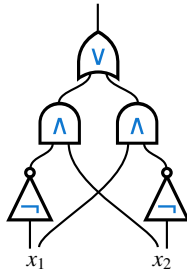
The outputs of a Boolean circuit are computed in the obvious way from the inputs.

↪ circuits with k inputs and ℓ outputs represent functions $\{0, 1\}^k \rightarrow \{0, 1\}^\ell$

We often consider circuits with only one output.

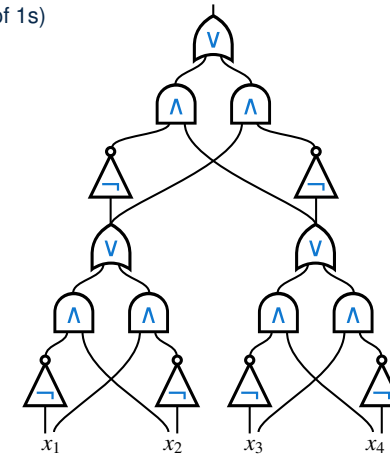
Example 1

XOR function:



Example 2

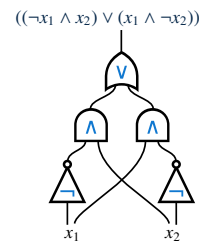
Parity function with four inputs:
(true for odd number of 1s)



Alternative Ways of Viewing Circuits (1)

Propositional formulae

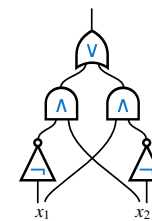
- propositional formulae are special circuits:
each non-input node has only one outgoing wire
- each variable corresponds to one input node
- each logical operator corresponds to a gate
- each sub-formula corresponds to a wire



Alternative Ways of Viewing Circuits (2)

Straight-line programs

- are programs without loops and branching (if, goto, for, while, etc.)
 - that only have Boolean variables
 - and where each line can only be an assignment with a single Boolean operator
- n -line programs correspond to n -gate circuits

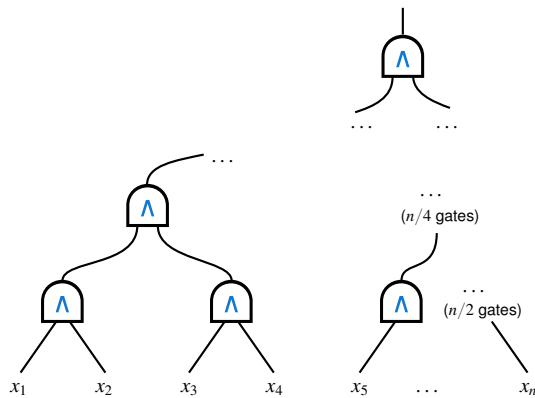


```

01 z1 := ¬x1
02 z2 := ¬x2
03 z3 := z1 ∧ x2
04 z4 := z2 ∧ x1
05 return z3 ∨ z4
    
```

Example: Generalised AND

The function that tests if all inputs are 1 can be encoded by combining binary AND gates:



- works similarly for OR gates
- number of gates: $n - 1$
- we can use n -way AND and OR (keeping the real size in mind)

Solving Problems with Circuits

Circuits are not universal: they have a fixed number of inputs!

How can they solve arbitrary problems?

Definition 19.2: A **circuit family** is an infinite list $C = C_1, C_2, C_3, \dots$ where each C_i is a Boolean circuit with i inputs and one output. We say that C **decides a language** L (over $\{0, 1\}$) if

$$w \in L \quad \text{if and only if} \quad C_n(w) = 1 \text{ for } n = |w|.$$

Example 19.3: The circuits we gave for generalised AND are a circuit family that decides the language $\{1^n \mid n \geq 1\}$.

Circuit Complexity

To measure difficulty of problems solved by circuits, we can count the number of gates needed:

Definition 19.4: The **size** of a circuit is its number of gates.

Let $f : \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. A circuit family C is **f -size bounded** if each of its circuits C_n is of size at most $f(n)$.

Size($f(n)$) is the class of all languages that can be decided by an $O(f(n))$ -size bounded circuit family.

Example 19.5: Our circuits for generalised AND show that $\{1^n \mid n \geq 1\} \in \text{Size}(n)$.

Examples

Many simple operations can be performed by circuits of polynomial size:

- Boolean functions such as **parity** (=sum modulo 2), **sum modulo n** , or **majority**
- Arithmetic operations such as **addition**, **subtraction**, **multiplication**, **division** (taking two fixed-arity binary numbers as inputs)
- Many **matrix operations**

See exercise for some more examples

Polynomial Circuits

Polynomial Circuits

A natural class of problems to consider are those that have polynomial circuit families:

Definition 19.6: $P_{\text{poly}} = \bigcup_{d \geq 1} \text{Size}(n^d)$.

Note: A language is in P_{poly} if it is solved by some polynomial-sized circuit family. There may not be a way to compute (or even finitely represent) this family.

How does P_{poly} relate to other classes?

Quadratic Circuits for Deterministic Time

Theorem 19.7: For $f(n) \geq n$, we have $\text{DTime}(f) \subseteq \text{Size}(f^2)$.

Proof sketch (see also Sipser, Theorem 9.30)

- We can represent the DTime computation as in the proof of Theorem 16.10: as a list of configurations encoded as words

$$* \sigma_1 \cdots \sigma_{i-1} \langle q, \sigma_i \rangle \sigma_{i+1} \cdots \sigma_m *$$

of symbols from the set $\Omega = \{*\} \cup \Gamma \cup (Q \times \Gamma)$.

\leadsto Tableau (i.e., grid) with $O(f^2)$ cells.

- We can describe each cell with a list of bits (wires in a circuit).
- We can compute one configuration from its predecessor by $O(f)$ circuits (idea: compute the value of each cell from its three upper neighbours as in Theorem 16.10)
- Acceptance can be checked by assuming that the TM returns to a unique configuration position/state when accepting □

From Polynomial Time to Polynomial Size

From $\text{DTime}(f) \subseteq \text{Size}(f^2)$ we get:

Corollary 19.8: $P \subseteq P_{\text{poly}}$.

This suggests another way of approaching the P vs. NP question:

If any language in NP is not in P_{poly} , then $P \neq \text{NP}$.

(but nobody has found any such language yet)

CIRCUIT-SAT

Input: A Boolean Circuit C with one output.
Problem: Is there any input for which C returns 1?

Theorem 19.9: CIRCUIT-SAT is NP-complete.

Proof: Inclusion in NP is easy (just guess the input).

For NP-hardness, we use that NP problems are those with a P-verifier:

- The DTM simulation of Theorem 19.7 can be used to implement a verifier (input: $(w\#c)$ in binary)
- We can hard-wire the w -inputs to use a fixed word instead (remaining inputs: c)
- The circuit is satisfiable iff there is a certificate for which the verifier accepts w □

Note: It would also be easy to reduce SAT to CIRCUIT-SAT, but the above yields a proof from first principles.

The Power of Circuits

A New Proof for Cook-Levin

Theorem 19.10: 3SAT is NP-complete.

Proof: Membership in NP is again easy (as before).

For NP-hardness, we express the circuit that was used to implement the verifier in Theorem 19.9 as propositional logic formula in 3-CNF:

- Create a propositional variable X for every wire in the circuit
- Add clauses to relate input wires to output wires, e.g., for AND gate with inputs X_1 and X_2 and output X_3 , we encode $(X_1 \wedge X_2) \leftrightarrow X_3$ as:

$$(\neg X_1 \vee \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \wedge (X_2 \vee \neg X_3)$$

- Fixed number of clauses per gate = constant factor size increase
- Add a clause (X) for the output wire X □

Is $P = P_{\text{poly}}$?

We showed $P \subseteq P_{\text{poly}}$. Does the converse also hold?

No!

Theorem 19.11: P_{poly} contains undecidable problems.

Proof: We define the unary Halting problem as the (undecidable) language:

$$\mathbf{UHALT} := \{1^n \mid \text{the binary encoding of } n \text{ encodes a pair } \langle \mathcal{M}, w \rangle \\ \text{where } \mathcal{M} \text{ is a TM that halts on word } w\}$$

For a number $1^n \in \mathbf{UHALT}$, let C_n be the circuit that computes a generalised AND of all inputs. For all other numbers, let C_n be a circuit that always returns 0. The circuit family C_1, C_2, C_3, \dots accepts \mathbf{UHALT} . □

Uniform Circuit Families

$P_{/poly}$ is too powerful, since we do not require the circuits to be computable. We can add this requirement:

Definition 19.12: A circuit family C_1, C_2, C_3, \dots is **log-space-uniform** if there is a log-space computable function that maps words 1^n to (an encoding of) C_n .

Note: We could also define similar notions of uniformity for other complexity classes.

Theorem 19.13: The class of all languages that are accepted by a log-space-uniform circuit family of polynomial size is exactly P.

Proof sketch: A detailed analysis shows that our earlier reduction of polytime DTMs to circuits is log-space-uniform.

Conversely, a polynomial-time procedure can be obtained by first computing a suitable circuit (in log-space) and then evaluating it (in polynomial time). \square

The Power of Circuits

Turing Machines That Take Advice

One can also describe $P_{/poly}$ using TMs that take “advice”:

Definition 19.14: Consider a function $a : \mathbb{N} \rightarrow \mathbb{N}$. A language L is accepted by a Turing Machine M with a bits of advice if there is a sequence of advice strings $\alpha_0, \alpha_1, \alpha_2, \dots$ of length $|\alpha_i| = a(i)$ and M accepts inputs of the form $(w\#\alpha_{|w|})$ if and only if $w \in L$.

$P_{/poly}$ is equivalent to the class of problems that can be solved by a PTime TM that takes a polynomial amount of “advice” (where the advice can be a description of a suitable circuit).

(This is where the notation $P_{/poly}$ comes from.)

$P_{/poly}$ and NP

We showed $P \subseteq P_{/poly}$. Does $NP \subseteq P_{/poly}$ also hold? Nobody knows.

Theorem 19.15 (Karp-Lipton Theorem): If $NP \subseteq P_{/poly}$ then $PH = \Sigma_2^P$.

Proof sketch (see Arora/Barak Theorem 6.19):

- if $NP \subseteq P_{/poly}$ then there is a polysize circuit family solving Sat
- Using this, one can argue that there is also a polysize circuit family that computes the lexicographically first satisfying assignment (k output bits for k variables)
- A Π_2 -QBF formula $\forall \vec{X}. \exists \vec{Y}. \varphi$ is true if, for all values of \vec{X} , $\varphi(\vec{X})$ is satisfiable.
- In Σ_2^P , we can: (1) guess the polysize circuit for SAT, (2) check for all values of \vec{X} if its output is really a satisfying assignment (to verify the guess)
- This solves Π_2^P -hard problems in Σ_2^P
- But then the Polynomial Hierarchy collapses at Σ_2^P , as claimed. \square

$P_{/poly}$ and ExpTime

We showed $P \subseteq P_{/poly}$. Does $\text{ExpTime} \subseteq P_{/poly}$ also hold?
Nobody knows.

Theorem 19.16 (Meyer's Theorem):

If $\text{ExpTime} \subseteq P_{/poly}$ then $\text{ExpTime} = \text{PH} = \Sigma_2^P$.

See [Arora/Barak, Theorem 6.20] for a proof sketch.

Corollary 19.17: If $\text{ExpTime} \subseteq P_{/poly}$ then $P \neq \text{NP}$.

Proof: If $\text{ExpTime} \subseteq P_{/poly}$ then $\text{ExpTime} = \Sigma_2^P$ (Meyer's Theorem).
By the Time Hierarchy Theorem, $P \neq \text{ExpTime}$, so $P \neq \Sigma_2^P$.
So the Polynomial Hierarchy doesn't collapse completely, and $P \neq \text{NP}$. \square

Modelling Parallelism With Circuits

How Big a Circuit Could We Need?

We should not be surprised that $P_{/poly}$ is so powerful:
exponential circuit families are already enough to accept any language

Exercise: show that every Boolean function over n variables can be expressed by a circuit of size $\leq n2^n$.

It turns out that these exponential circuits are really needed:

Theorem 19.18 (Shannon 1949 (!)): For every n , there is a function $\{0, 1\}^n \rightarrow \{0, 1\}$ that cannot be computed by any circuit of size $2^n/(10n)$.

In fact, one can even show: almost every Boolean function requires circuits of size $> 2^n/(10n)$ – and is therefore not in $P_{/poly}$

Is any of these functions in NP? Or at least in Exp? Or at least in NExp?
Nobody knows.

What is Efficiently Parallelisable?

Experience suggests:

Some problems can be solved efficiently in parallel, while others can not.

How could this be shown?

Intuitive definition:

A problem has an **efficient parallel algorithm** if it can be solved for inputs of size n

- in **polylogarithmic time**, i.e., in time $O(\log^k n)$ for some $k \geq 0$,
- using a computer with a **polynomial number of parallel processors**, i.e., $O(n^d)$ processors for some $d \geq 0$.

Note: Using $O(n^d)$ processors efficiently requires a massively parallel algorithm. However, one could always use fewer processors (each taking on more work), possibly leading to a proportional increase in time.

The hard bit in parallelisation is to utilise many processors effectively – reducing to fewer processors is easy.

Modelling Parallel Computation

What kind of “parallel computer” do we mean here?

- (1) How do processors communicate?
- (2) What can a processor do in one step?
- (3) How are processors synchronised?

Detailed answer: define **Parallel Random Access Machine (PRAM)**

Our answer:

Details are not critical as long as we can make some general assumptions:

- (1) Every processor can send a message to any other processor in $O(\log n)$ time
- (2) In one step, each processors can perform one Boolean operation on “a few” bits, say $O(\log n)$
- (3) Processor steps are synched with a global clock

Modelling Parallel Computation in Circuits

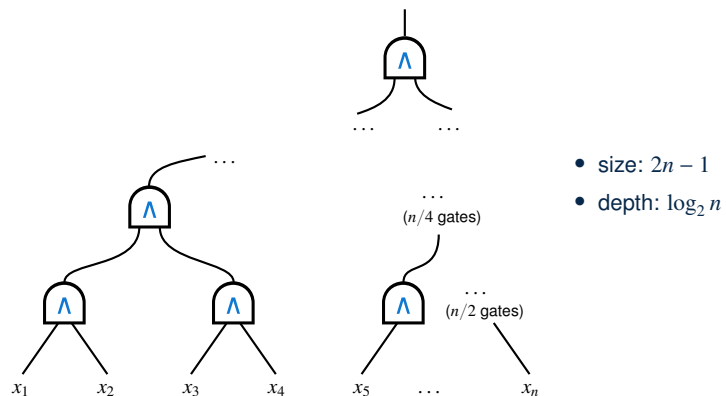
Simple PRAM computations can be mapped to Boolean circuits (with some extra circuitry for executing more operations or for modelling message passing)

Circuits as models for parallel computation:

- circuit gates can operate in parallel – they only depend on their inputs
- the time needed to evaluate a circuit depends on its **depth**, not size (depth = longest distance from an input to an output node)

Example: Generalised AND

The function that tests if all inputs are 1 can be encoded by combining binary AND gates:



Small-Depth Circuits

Small depth = short (parallel) time

However: Every Boolean function can be computed by depth $O(\log n)$ circuits using $O(n^{2^n})$ gates (exercise)

Hence, to capture “efficient parallel computation”, we also restrict the size:

Definition 19.19: For $k \geq 0$, we define NC^k to be the class of all problems that can be solved by a circuit family $C = C_1, C_2, C_3, \dots$ such that

- the depth of C_n is bounded by $O(\log^k n)$, and
- there is some $d \geq 0$ so that the size of C_n is bounded by $O(n^d)$ (in other words: $\text{NC}^k \subseteq \text{P}_{\text{poly}}$).

(NC is for “Nick’s class”, named in honour of Nicholas Pippenger, who studied such circuits, by Stephen Cook.)

Alternating Circuits

Different complexity classes are obtained when allowing generalised Boolean gates with many inputs:

Definition 19.20: An **AND gate with unbounded fan-in** is a gate that computes a generalised AND function over an arbitrary number $n \geq 2$ of inputs. **OR gates with unbounded fan-in** are defined similarly.

For $k \geq 0$, we define AC^k exactly like NC^k but allowing circuits to use gates with unbounded fan-in.

Example 19.21: Generalised AND is in NC^1 and in AC^0 .

Uniform vs. Non-uniform

Recall: a circuit family is uniform if it can be computed by a (restricted form of) Turing machine

- Our definitions of NC^k and AC^k do not require uniformity
- It is common to define **uniform NC^k** and **uniform AC^k** using logspace-uniformity (or even more restricted forms of uniformity)
- Clearly: **uniform $NC^k \subseteq NC^k$** and **uniform $AC^k \subseteq AC^k$**

Convention: For the rest of this lecture, we restrict to (logspace) uniform versions of NC^k and AC^k .

The NC Hierarchy

The classes NC^k and AC^k form a hierarchy:

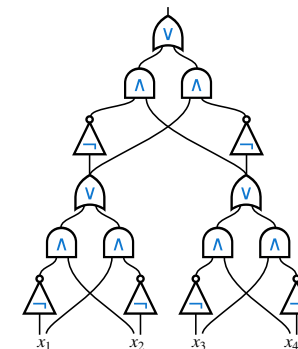
- if $i \leq j$ then $NC^i \subseteq NC^j$ (obvious)
- if $i \leq j$ then $AC^i \subseteq AC^j$ (obvious)
- $NC^i \subseteq AC^i$ (obvious)
- $AC^i \subseteq NC^{i+1}$ (since generalised AND and OR can be replaced with $O(\log n)$ bounded fan-in gates as in our example)

The limit of this hierarchy is defined as $NC = \bigcup_{k \geq 0} NC^k$ so we get:

$$AC^0 \subseteq NC^1 \subseteq AC^1 \subseteq \dots \subseteq NC^k \subseteq AC^k \subseteq NC^{k+1} \subseteq \dots \subseteq NC$$

Note: NC^0 is not a very useful class, as those circuits cannot process the whole input

Example: Parity is in NC^1



However, we also have the following major result (without proof):

Theorem 19.22 (see Arora/Barak, Chapter 14): **PARITY** is not in AC^0 , and therefore $AC^0 \subsetneq NC^1$.

Example: FOL model checking

FOL MODEL CHECKING

Input: First-order sentence φ ; finite first-order structure \mathcal{I}
Problem: Is φ satisfied by \mathcal{I} ?

We showed that this problem is PSpace-complete.
It turns out that this complexity is caused by the formula, not by the model:

FOL MODEL CHECKING FOR φ

Input: A finite first-order structure \mathcal{I} .
Problem: Is φ satisfied by \mathcal{I} ?

Theorem 19.23 (see course Database Theory, Summer 2020, TU Dresden):
For any first-order sentence φ , **FOL MODEL CHECKING FOR φ** is in AC^0 .

Relationships to Other Complexity Classes (1)

Using the assumption of uniformity, we can solve circuit complexity problems by (1) computing the circuit and (2) evaluating it.

The following are not hard to show:

Theorem 19.24 (Sipser, Theorem 10.41): $NC \subseteq P$

Theorem 19.25 (Sipser, Theorem 10.39): $NC^1 \subseteq L$

Relationships to Other Complexity Classes (2)

Conversely, some known classes are also subsumed by NC:

Theorem 19.26: $NL \subseteq AC^1$

Proof notes:

General proof idea: (1) construct a "generalised" configuration graph for an NL machine (a graph that describes all possible configuration graphs for inputs of a given length, using transitions that depend on the actual input that is given); (2) check reachability of the goal state in this graph (basically by repeated matrix multiplication in the reachability matrix).

We do not give a proof here. Sipser (Theorem 10.40) sketches the proof for $NL \subseteq NC^2$; the proof for $NL \subseteq AC^1$ is the same but also uses that the depth is only logarithmic if we can use unbounded fan-in gates. \square

We therefore obtain the following picture:

$$AC^0 \subset NC^1 \subseteq L \subseteq NL \subseteq AC^1 \subseteq NC^2 \subseteq \dots \subseteq NC \subseteq P$$

P-Completeness

The Limits of Parallel Computation

NC defines a hierarchy of efficiently parallelisable problems in P

Are all problems in P efficiently parallelisable?
Nobody knows.

State of the art:

- It is not known if $NC \neq P$ or not
- It is not even known if $NC^1 \neq PH$ or not
- It is clear that $AC^0 \neq P$ (since $AC^0 \subset NC^1$)
- It is clear that $NC \neq PSpace$ (exercise: why?)

"Most experts believe that" $NC \neq P$

→ if this is true, then some problems in P cannot be parallelised efficiently

P-Complete Problems

Recall the definition from Lecture 11:

Definition 11.7: A problem $L \in P$ is **complete for P** if every other language in P is log-space reducible to L.

If $NC \neq P$ then P-complete problems are tractable but **not efficiently parallelisable** and therefore **inherently serial**.

Circuit Evaluation is P-complete

CIRCUIT VALUE

Input: A Boolean Circuit C with one output, and an input word $w \in \{0, 1\}^n$

Problem: Does C return 1 on this input?

Theorem 19.27: CIRCUIT VALUE is P-complete.

Proof: Membership is easy. For completeness, we reduce the word problem of polynomially time-bounded Turing machines. A circuit for this problem was constructed earlier for Theorem 18.12. This circuit family is logspace-uniform (as already remarked in Theorem 18.18), so we get a logspace-reduction. \square

Propositional Horn Logic

A problem that is closer to artificial intelligence:

- A **propositional fact** is a formula consisting of a single propositional variable X
- A **propositional Horn rule** is a formula of the form $X_1 \wedge X_2 \rightarrow X_3$
- A **propositional Horn theory** is a set of propositional Horn rules and facts

The semantics of propositional Horn theories is defined as usual for propositional logic.

PROP HORN ENTAILMENT

Input: A propositional Horn theory T and a propositional variable X

Problem: Does T entail X to be true?

Propositional Horn Logic is P-Complete

Theorem 19.28: PROP HORN ENTAILMENT is P-complete.

Proof sketch: One can give a direct Turing machine encoding:

- We use propositional variables to represent configurations as for Cook-Levin
- We encode TM behaviour directly, e.g., for transitions $\langle q, \sigma \rangle \mapsto \langle q', \sigma', d \rangle$ we can use rules like $Q_{q,t} \wedge P_{i,t} \wedge S_{i,\sigma,t} \rightarrow Q_{q',t+1} \wedge P_{i+d,t+1} \wedge S_{i,\sigma',t+1}$ (for all times t and positions i)
- We do not need rules that forbid inconsistent configurations (two states at once etc.): Horn logic has a least model, and we don't need to worry about other models when checking entailment
- Disjunctive acceptance conditions ("accepts if there is some time point at which it reaches an accepting state") can be encoded by many implications (one for each case) without "real" disjunctions

For details, see Theorem 4.2 in Dantsin, Eiter, Gottlob, Voronkov: [Complexity and expressive power of logic programming](#) (link). ACM Computing Surveys, 2001. \square

Complexity vs. Runtime

Horn logic is P-complete:

- One of the hardest problems in P
- Inherently non-parallelisable

However:

- **PROP HORN ENTAILMENT** can be decided in [linear time](#) [Dowling/Gallier, 1984]
- This does not imply that all problems in P have linear time algorithms

Summary and Outlook

Circuits provide an alternative model of computation

CIRCUIT-SAT is NP-complete.

$P_{/poly}$ is very powerful – uniform circuit families help to restrict it

[Small-depth circuits](#) can be used to model efficient parallel computation

NC defines a hierarchy of problems below P:

$$AC^0 \subset NC^1 \subseteq L \subseteq NL \subseteq AC^1 \subseteq NC^2 \subseteq \dots \subseteq NC \subseteq P$$

P-complete problems, such as Horn logic entailment, are believed not to be efficiently parallelisable.

What's next?

- Randomness