TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

COMPLEXITY THEORY

Lecture 19: Circuits and Parallel Computation

Markus Krötzsch, Stephan Mennicke, Lukas Gerlach

Knowledge-Based Systems

TU Dresden, 18th Dec 2023

More recent versions of this slide deck might be available.
For the most current version of this course, see
https://iccl.inf.tu-dresden.de/web/Complexity_Theory/en

The Power of Circuits

$\mathrm{P}_{/\mathrm{poly}}$ and NP

We showed $\mathrm{P} \subseteq \mathrm{P}_{/\mathrm{poly}}$. Does $\mathrm{NP} \subseteq \mathrm{P}_{/\mathrm{poly}}$ also hold?

$P_{/poly}$ and NP

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

Theorem 19.1 (Karp-Lipton Theorem): If $NP \subseteq P_{/poly}$ then $PH = \Sigma_2^p$.

$P_{/poly}$ and NP

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

Theorem 19.1 (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. □

$\mathsf{P}_{/\mathrm{poly}}$ and ExpTime

We showed $\mathsf{P} \subseteq \mathsf{P}_{/\mathrm{poly}}$. Does $\mathsf{ExpTime} \subseteq \mathsf{P}_{/\mathrm{poly}}$ also hold?

$P_{/poly}$ and ExpTime

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

Theorem 19.2 (Meyer's Theorem):
If $ExpTime \subseteq P_{/poly}$ then $ExpTime = PH = \Sigma_2^p$.

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

$P_{/poly}$ and ExpTime

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

Theorem 19.2 (Meyer's Theorem):
If $ExpTime \subseteq P_{/poly}$ then $ExpTime = PH = \Sigma_2^p$.

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

Corollary 19.3: If $ExpTime \subseteq P_{/poly}$ then $P \neq NP$.

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

How Big a Circuit Could We Need?

We should not be surprised that $\mathsf{P}_{/\mathrm{poly}}$ 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 $\leq n2^n$.

How Big a Circuit Could We Need?

We should not be surprised that $\mathsf{P}_{/\mathrm{poly}}$ 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 19.4 (Shannon 1949 (!)): For every n, there is a function $\{0,1\}^n \to \{0,1\}$ 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 $\mathsf{P}_{/\mathrm{poly}}$

How Big a Circuit Could We Need?

We should not be surprised that $\mathsf{P}_{/\mathrm{poly}}$ 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 19.4 (Shannon 1949 (!)): For every n, there is a function $\{0,1\}^n \to \{0,1\}$ 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 $\mathsf{P}_{/\mathrm{poly}}$

Is any of these functions in NP? Or at least in Exp? Or at least in NExp?

How Big a Circuit Could We Need?

We should not be surprised that $\mathsf{P}_{/\mathrm{poly}}$ 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 19.4 (Shannon 1949 (!)): For every n, there is a function $\{0,1\}^n \to \{0,1\}$ 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 $\mathsf{P}_{/\mathrm{poly}}$

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

Modelling Parallelism With Circuits

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 $d \geq 0$.

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)

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

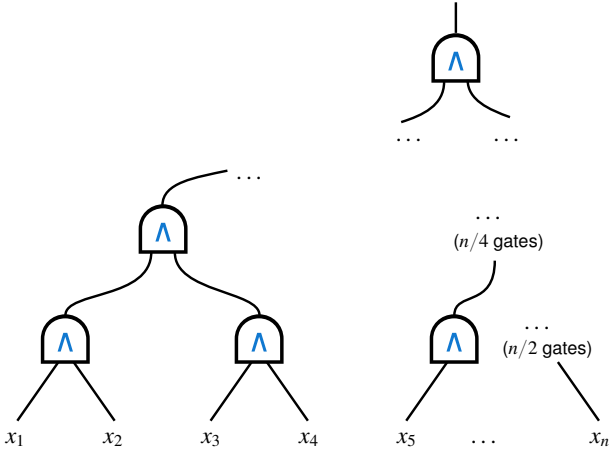
Simple PRAM computations can be mapped to Boolean circuits
(with some extra circuitry for executing more operations or for modelling message passing)

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:



- size: $2n - 1$
- depth: $\log_2 n$

Small-Depth Circuits

Small depth = short (parallel) time

However: Every Boolean function can be computed by depth $O(\log n)$ circuits using $O(n2^n)$ gates (exercise)

Small-Depth Circuits

Small depth = short (parallel) time

However: Every Boolean function can be computed by depth $O(\log n)$ circuits using $O(n2^n)$ gates (exercise)

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

Definition 19.5: For $k \geq 0$, we define NC^k to be the class of all problems that can be solved by a circuit family $C = C_1, C_2, C_3, \ldots$ 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: $\mathsf{NC}^k \subseteq \mathsf{P}_{/\mathrm{poly}}$).

(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 19.6: 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 19.7: Generalised AND is in NC^1 and in AC^0.

The NC Hierarchy

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

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

The NC Hierarchy

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

- if $i \leq j$ then $\mathsf{NC}^i \subseteq \mathsf{NC}^j$ (obvious)
- if $i \leq j$ then $\mathsf{AC}^i \subseteq \mathsf{AC}^j$ (obvious)
- $\mathsf{NC}^i \subseteq \mathsf{AC}^i$ (obvious)
- $\mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$ (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 $\mathsf{NC} = \bigcup_{k \geq 0} \mathsf{NC}^k$ so we get:

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

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

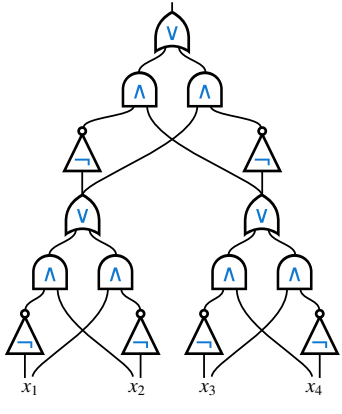
Uniform vs. Non-uniform

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

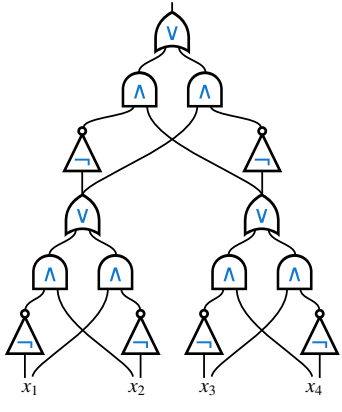
- Our definitions of NC^k and AC^k do not require uniformity
- It is common to define uniform NC^k and uniform AC^k using logspace-uniformity (or even more restricted forms of uniformity)
- Clearly: uniform $\mathsf{NC}^k \subseteq \mathsf{NC}^k$ and uniform $\mathsf{AC}^k \subseteq \mathsf{AC}^k$

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

Example: Parity is in NC^1



Example: Parity is in NC^1



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

Theorem 19.8 (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.

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 19.9 (see course Database Theory, Summer 2022, 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 19.10 (Sipser, Theorem 10.41): $\mathrm{NC} \subseteq \mathrm{P}$

Theorem 19.11 (Sipser, Theorem 10.39): $\mathrm{NC}^1 \subseteq \mathrm{L}$

Relationships to Other Complexity Classes (2)

Conversely, some known classes are also subsumed by NC:

Theorem 19.12: $\mathrm{NL} \subseteq \mathrm{AC}^1$

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 $\mathrm{NL} \subseteq \mathrm{NC}^2$; the proof for $\mathrm{NL} \subseteq \mathrm{AC}^1$ is the same but also uses that the depth is only logarithmic if we can use unbounded fan-in gates. □

Relationships to Other Complexity Classes (2)

Conversely, some known classes are also subsumed by NC:

Theorem 19.12: $\mathrm{NL} \subseteq \mathrm{AC}^1$

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 $\mathrm{NL} \subseteq \mathrm{NC}^2$; the proof for $\mathrm{NL} \subseteq \mathrm{AC}^1$ is the same but also uses that the depth is only logarithmic if we can use unbounded fan-in gates. □

We therefore obtain the following picture:

$$\mathrm{AC}^0 \subset \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{AC}^1 \subseteq \mathrm{NC}^2 \subseteq \cdots \mathrm{NC} \subseteq \mathrm{P}$$

P-Completeness

The Limits of Parallel Computation

NC defines a hierarchy of efficiently parallelisable problems in P

Are all problems in P efficiently parallelisable?

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
$\rightsquigarrow$ 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 $\mathbf{L} \in \mathrm{P}$ is complete for P if every other language in P is log-space reducible to $\mathbf{L}$.

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 19.13: **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. □

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 19.14: 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 is 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. □

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

Small-depth circuits can be used to model efficient parallel computation

NC defines a hierarchy of problems below P:

$$\mathrm{AC}^0 \subset \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{AC}^1 \subseteq \mathrm{NC}^2 \subseteq \cdots \mathrm{NC} \subseteq \mathrm{P}$$

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

What's next?

- Randomness
- Quantum Computing
- Examinations