CS855: Parallel complexity theory

Marc Moreno Maza

February 19, 2008

- Basic notions of sequential complexity theory.
- Highly parallel feasibility and NC algorithms.
- Highly parallel reduction.
- P-completness.

Complexity class. A collection of problems of some characteristic worst cases difficulty. Some problems in a complexity class may be easier than the others, but all of them can be solved within the same resource bounds associated with the class. **Example.** Given a non-negative integer n, and two integers x, y with $|x|, |y| \leq n$, computing the product $x\,y$ or the GCD $\gcd(x, y)$ are two problems in the same complexity class.

Abstract problem. A binary relation between a set of **problem instances** and a set of the **problem solutions**. The **size** of an instance is the number of symbols (digits, letters, delimiters, etc) in the full problem specification. **Example. SPP = Shortest Path Problem.** Given graph G and two vertices u, v, find in G a shortest path between u and v. The set of instances is the set of all triples (G, u, v) and a given instance may have zero, one, or more solutions (paths from u to v).

Decision problem. A problem with solutions **Yes/No**. Optimization problems can be formulated as decision problems, typically by imposing bounds on the values to be optimized **Example. PATH = the decision problem for SPP:** Given G, u, v and an integer k, does a path exist in G between u and v whose length is at most k ?

Remark 1 *An easy optimization problem makes the corresponding decision problem easy. Hence, if a decision problem is hard, so is the related optimization problem. Therefore, complexity theory tends to deal with decision problems rather than optimization problems.*

Compact encoding. Any abstract notions, such as numbers, graphs, functions, polygons, programs, can be encoded using strings over some finite alphabet $\mathbb{Z}$. An encoding is **compact** if the length of any encoded instance is of the "same order" as the size of the abstract instance. This usually translates into the requirement that any abstract instance of size n must be encoded with $O(\log_m n)$ symbols of $\mathbb{Z}$, where $m = |\mathbb{Z}|$. **Example.**

Binary expansion for encoding numbers.

Remark 2 *Other obvious requirements on an encoding scheme are that it is must be easy to produce the encoding of any instance of the problem and that it may not produce more computational hints than any other encoding.*

Concrete problem. An abstract problem encoded for a computing device using some compact encoding, typically a binary one (such as ASCII). **Example.** Computing the GCD of two integers encoded by their binary expansions.

Input size. The length of a binary encoding of a problem instance, denoted by $|x|$ for input x. **Example** A graph G on n vertices can be encoded by an adjacency matrix of n^2 bits, hence the size of G is $|G| = n^2$.

Sequential time. A sequential algorithm solves a concrete problem in time $T(n)$ if it produces a solution in time $O(T(|x|))$ for any input x.

Problems solvable in polynomial time. The set of all concrete problems with $T(n) = O(n^{O(1)})$.

Proposition 1 *If an abstract problem can be solved by a polynomial time sequential algorithm for some compact encoding, then for all compact encodinga this abstract problem can be solved by a polynomial time sequential algorithm.*

Proof. Indeed, any compact encoding can be converted into a binary one in polynomial time. □

Remark 3 *That is why the complexity theory deals only with binary encoding, what we should be doing from now on.*

Binary language. A binary language is a subset of $\{0, 1\}^*$. Any decision problem Q can be viewed as a language $L_Q = \{x \in \{0, 1\}^* \mid Q(x) = 1\}$ of "yes" instances. **Example:** Let L_{PATH} be the set of all quadruples (G, u, v, k) where G is a graph, with u, v

vertices of G and there exists a path between u and v in G whose length is at most k.

Acceptability. We say that Algorithm A accepts (rejects) the binary string x if A outputs 1 (0) for input x. We say that the binary language L is accepted in time $T(n)$ if any $x \in L$ is accepted in time $T(n)$.

Remark 4 *In order to accept a language, the algorithm needs only to worry about strings in the language. If x is not accepted, then it is either rejected or A may loop forever for input x.*

Decidability. A *characteristic function* of the binary language L is a function $f_L : \{0, 1\}^* \longrightarrow \{0, 1\}$ such that $f_L(x) = 1$ if $x \in L$ and $f_L(x) = 0$ otherwise. The Language L is *decidable* in sequential time $T(n)$ iff f_L is computable in $O(T(n))$ time. That is, if there is an algorithm which accepts every string $x \in L$ and rejects every $x \notin L$ in time $O(T(|x|))$. There are problems such as *Turing Halting Problem* for which there is an accepting algorithm, but there is no deciding algorithm.

Polynomial class P: the set of all languages decidable in sequential polynomial time.

Verification algorithm. An algorithm A which

- takes as arguments x and y where x is an ordinary input set and y is a *certificate*,
- verifies whether $A(x, y) = 1$ holds or not.

is called a *verification algorithm.* The language verified by a verification algorithm A is

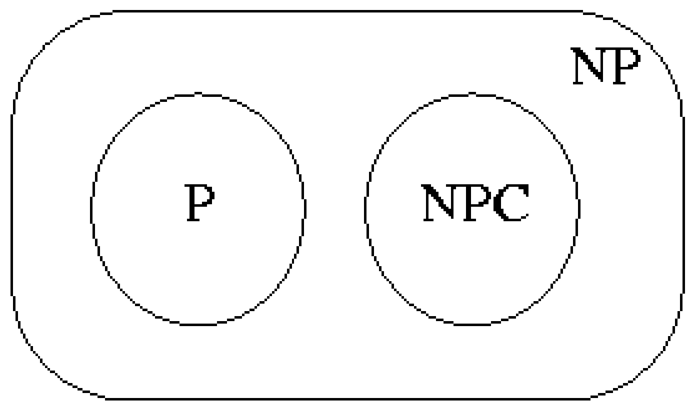
$$L_A = \{x \in \{0,1\}^* \;\mid\; \exists y \in \{0,1\}^* \;\mid\; A(x,y) = 1\}.$$

Complexity class NP: the class of all languages that can be verified by a polynomial-time verification algorithms. More precisely, the binary language L is in NP iff there is a verification algorithm A such that

$$L = \{x \in \{0,1\}^* \ \mid \ \exists y \in \{0,1\}^* \ \mid \ |y| = O(|x|^{O(1)}) \ \text{and} \ A(x,y) = 1\}.$$

and $A(x,y)$ can be computed in $O(|x|^{O(1)})$ time, for all $|y| = O(|x|^{O(1)})$. Historically, NP stands for nondeterministic polynomial. **Example:** Hamiltonian-cycle problem (HC): Given graph G, does it have a Hamiltonian cycle. No polynomial-time algorithm is known to decide HC. However, HC is in NP, since given a graph G and a permutation of its vertices, we can verify in polynomial time whether the permutation is a Hamiltonian cycle of G.

Obviously, we have $P \subseteq NP$, but it is not known whether $P = NP$ holds or not. However, it is conjectured that $NP \setminus P$ contains another subclass of problems called *NP-complete.* This subclass is denoted by NPC. To define NP-completeness, we need the notion of reducibility.



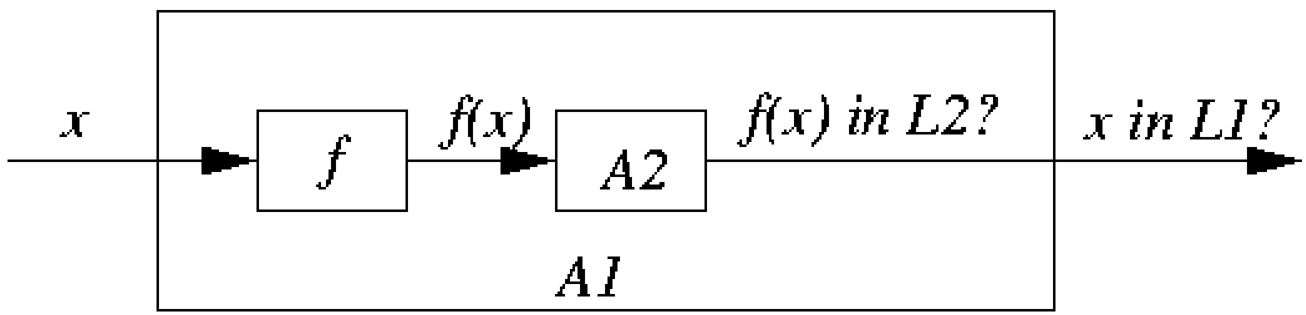
Definition 1 *Let $L_1, l_2 \subseteq \{0,1\}^*$ be two binary languages. The language L_1 is polynomial-time reducible to L_2, and we write $L_1 \leq_P L_2$, if there exists a polynomial-time function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$ we have*

$$x \in L_1 \iff f(x) \in L_2.$$

Lemma 1 *The class P is closed under the relation of P-reduction. More precisely, for all binary languages $L_1, L_2 \subseteq \{0,1\}^*$ we have*

$$L_1 \leq_P L_2 \text{ and } L_2 \in P \implies L_1 \in P.$$

PROOF.

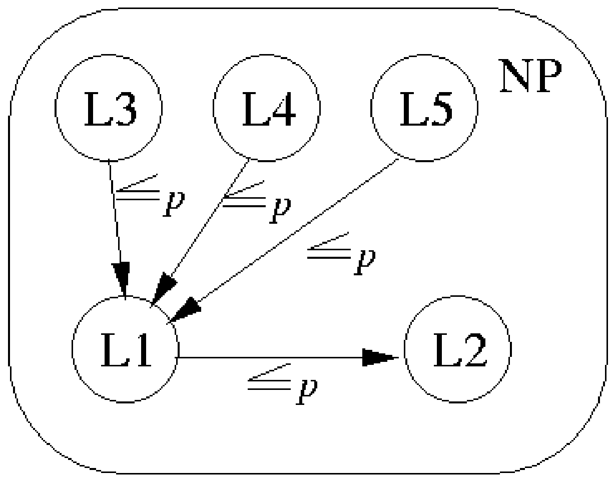


Definition 2 *A binary language L is* NP-complete *if L is in NP and for all L' in NP we have $L' \leq_P L$.*

Lemma 2 *For all binary languages $L_1, L_2 \subseteq \{0,1\}^*$ we have*

$$L_1 \leq_P L_2 \text{ and } L_1 \in \mathrm{NP}-\mathrm{complete} \text{ and } L_2 \in \mathrm{NP} \quad \Longrightarrow \quad L_2 \in \mathrm{NP}-\mathrm{complete}.$$

Proof.



Corollary 1 *We have:*

- *If there exists an NP-complete problem in P, the P = NP holds.*
- *If there exists an NP-complete problem which is proved not to be in P, then no NP-complete problem is in P.*

Remark 5 *To show that a language L is NP-complete, we have to show that L is in NP (which is usually easy) and then there are two choices:*

- *We show that any NP language can be reduced to L in polynomial time. This was necessary for the first NP-complete problems, such as SATISFIABILITY: "Given a Boolean combinatorial circuit composed of AND, OR, and NOT gates, with input* $x_1, x_2, ..., x_n$ *and designated output y, is there an input set such that y is true?"*
- *If there exist already some NP-complete problems, then we:*
  1. *select a suitable NP-complete* L'*,*
  2. *give a polynomial time algorithm that computes a function f mapping instances of* L' *into instances of L,*
  3. *prove that* $x \in L' \iff f(x) \in L$ *holds for all* $x \in \{0,1\}^*$*.*

In parallel computing world, we have another valuable resource, the number of processors. Except for this difference, all the notions of the classical complexity theory apply here. By default, we assume PRAM computation.

Definition 3 *A language L is decided in parallel time $T(n,p)$ with $p = p(n)$ processors iff f_L is computable in parallel time $T(n,p(n))$ with $p(n)$ processors.*

Remark 6 *The parallel feasibility usually means decidability in polynomial time with polynomial number of processors. So a problem is parallel feasible iff it is sequentially feasible and parallel complexity theory works only with complexity class P.*

Lemma 3 *A binary language L is decidable in sequential time $n^{O(1)}$ iff it is decidable in parallel time $n^{O(1)}$ with $n^{O(1)}$ processors.*

PROOF. This is an immediate consequence of Proposition 1 from the Chapter *Performance and scalability of parallel algorithms, PRAM models.* □

Remark 7 *However, the goal of parallel computation is to achieve a highly parallel feasibility: to reduce the sequential polynomial complexity of problems to subpolynomial parallel time!!!! For that, we must use polynomial number of processors, since we have*

$$p(n)T(n, p(n)) \geq SU(n).$$

There are two important practical considerations:

- *Which polynomial number of processors is acceptable/affordable? In practice, the polynomial number of processors is meant to be linear or close.*
- *Which subpolynomial time is achievable? It turned out that almost all highly parallel feasible problems can achieve polylogarithmic parallel time.*

Therefore, the traditional class of highly parallel feasible algorithms is the following.

Definition 4 *The class NC is the set of languages decidable in parallel time* $T(n, p(n)) = O(\log^{O(1)} n)$ *with* $p(n) = O(n^{O(1)})$ *processors.*

Proposition 2 *If an algorithm is NC with a PRAM submodel, then it is NC with any other PRAM submodel.*

PROOF. This is an immediate consequence of Theorem 1 from the Chapter *PRAM model.* □

Remark 8 *The class NC includes:*

- *parallel prefix (scan) computation, parallel sorting and selection*
- *matrix operations (matrix multiplication, solving special cases of systems of linear equations)*
- *parallel string matching algorithms, etc.*

Many NC algorithms are cost optimal. For example, they have $T(n, p(n)) = O(\log n)$ *with* $p(n) = n/\log(n)$ *if* $SU(n) = O(n)$ *(for example parallel scan). On the other hand, there are many problems for which no NC algorithms are known; to be discussed later.*

- The class NC may include some algorithms which are not efficiently parallelizable. The most infamous example is the *parallel binary search* of an intem x in a sorted aggregate of n items. The trouble is that this problem has polylogarithmic time complexity even for $p(n) = 1$. If we apply $p > 1$ processors, we get $T(n, p) = \log_{p+1}(n)$ and therefore the cost $C(n, p) = \log(n)p/\log(p + 1)$. With $SU(n) = \log(n)$, we obtain a speedup $S(n, p) = \log(p + 1)$ which is far from what could be considered as a highly parallel solution. Any sequential algorithm with logarithmic time is in NC regardless of its parallel feasibility!!!

- NC theory assumes situations where a huge machine (polynomial number of processors, e.g., millions) solves very quickly (polylogarithmic time, e.g., seconds) moderately sized problems (e.g., hundred of thousands input items). In practice, however, moderately sized machines (hundreds, at most thousands of processors) are used to solve large problems (e.g., millions of input items). So that the number of processors tends to be subpolynomial, even sublinear.

We have seen that NC is subset of P, but similarly to the NP-completeness theory, the problem whether P=NC is open and is likely equally difficult as its famous predecessor P=NP. The situation is very similar and also the techniques to deal with the problem are similar. Let us demonstrate these similarities in more detail.

Before defining parallel reduction, let us start with an informal example. Consider the following 4 problems:

Maximum Flow Value Problem (MFV): (functional optimization problem)

Input: Graph G, whose each edge e is labeled by capacity $c(e) >= 0$, and two vertices s, t.

Problem: Compute the maximum flow $f(s, t)$ from source s to goal t in G.

Maximum Flow Bit Problem (MFB(i)): (decision problem)

Input: Graph G labeled by capacities, vertices s, t, and integer i.

Problem: Is the i-th bit of the maximum flow $f(s, t)$ equal to 1?

Maximum Flow Threshold Problem (MFT(k)): (decision problem)

Input: Graph G labeled by capacities, vertices s, t, and integer k.

Problem: Is the maximum flow $f(s, t)$ less than k?

Maximum Flow Pattern Problem (MFP): (relational optimization problem)

Input: Graph G, edges labeled by capacities, and vertices s, t.

Problem: Compute a flow pattern in G achieving the maximum flow $f(s, t)$.

Parallel reductions allow to compare these problems from the viewpoint of highly parallel feasibility.

MFB(i) to MFV: trivial and highly parallel: solve MFV and look at the i-th bit.

MFV to MFB(i) : highly parallel:

- Compute an upper bound M on the value of $f(s,t)$, for example, by summing the capacities of all edges incident to s and t. This is a highly parallel computation. Let $m = \log M$.
- Solve m instances of MFB(i), for all $i = 1, \ldots, m$ in parallel.
- Combine the output of the MFB(i)'s into an integer value. Also can be done in parallel.

It follows that MFB(i) is highly parallel iff MFV is highly parallel. This approach can be generalized.

Lemma 4 *The computation of any function $g : \{0,1\}^* \longrightarrow \{0,1\}^*$ induces associated bitwise decision problems g_i: "Is bit i of $g(x)$ equal to 1?" If $g(x)$ is bounded, then both problems are of the same parallel complexity.*

However, MFT(k) and MFV likely differ in parallel complexity.

MFT(k) to MFV: trivial and highly parallel: solve MFV and compare the result with k.

MFV to MFT(k): cannot be likely done in a highly parallel way.

Assume that $f(s,t)$ is again bounded by M and let $m = \log_2 M$. We need a sequential binary search performed as a sequence of recursive calls:

- Call MFT(2^{m-1}). If MFT(2^{m-1}) answers NO,
- then m-th bit is 1 and call recursively MFT($2^{m-1} + 2^{m-2}$)
- else m-th bit is 0 and call recursively MFT(2^{m-2}),
- etc.

This works fine sequentially, but these m calls cannot be done in parallel. Indeed, each of them depends on the result of the preceding one. Consequently:

- even if MFT(k) had a highly parallel solution, MFV (and therefore MFB(i) as well) reduction on MFT(k) is inherently sequential.
- This could be an indication that MFT(k) itself is inherently sequential.

Definition 5 *A language L_1 in P is NC-reducible to a language L_2 in P, and we write $L_1 \leq_{NC} L_2$, if there is there exists a function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$ we have*

$$x \in L_1 \iff f(x) \in L_2.$$

and f can be computed by an NC algorithm.

Hence, if we are able to show that $L_1 \leq_{NC} L_2$ holds, we have shown that

- either if L_2 in NC, then L_1 in NC,
- or if $L_1 \notin$ NC, then $L_2 \notin$ NC.

Therefore, we can

- either prove an NC upper bound for L_2 which gives an NC upper bound for L_1,
- or prove a polynomial lower bound for L_1 which gives a polynomial lower bound for L_2.

Of course, if both $L_1 \leq_{NC} L_2$ and $L_2 \leq_{NC} L_1$ hold, then L_1 and L_2 are of the same parallel feasibility. And this is a basis for defining the class of P-complete languages/problems.

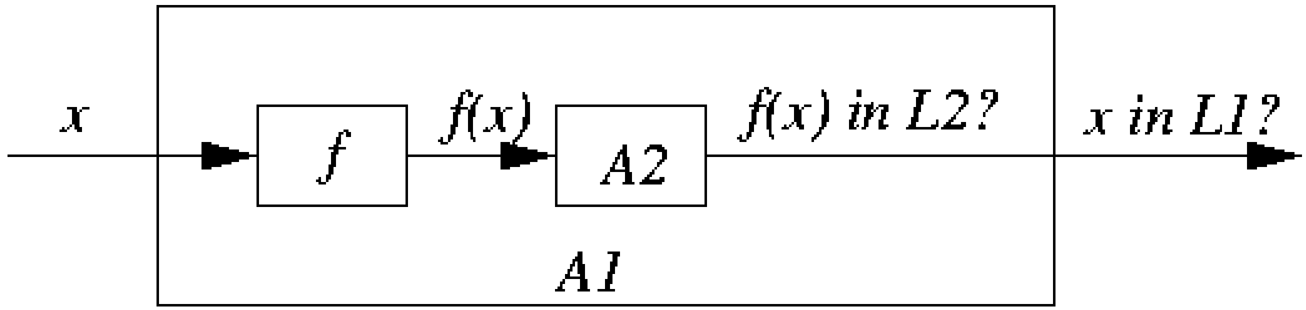
Definition 6 *A language L is* P-hard *under NC-reducibility if for all L' in P we have $L' \leq_{NC} L$.*

Informally, a P-hard language L is as hard to decide in parallel as any problem in P.

Definition 7 *A language L is* P-complete *if it is in P and it is P-hard.*

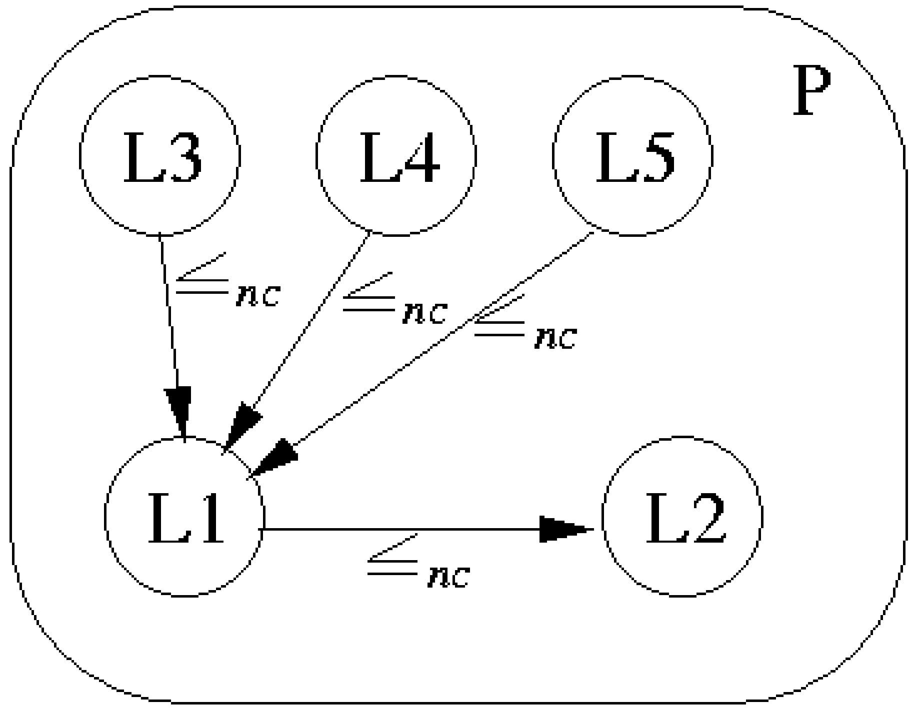
Lemma 5 *The class NC is closed under the relation of NC-reduction. More procesile, for any two binary languages L_1, L_2 we have: if $L_1 \leq_{NC} L_2$ holds and L_2 is in NC, then L_1 is in NC too.*

Proof.



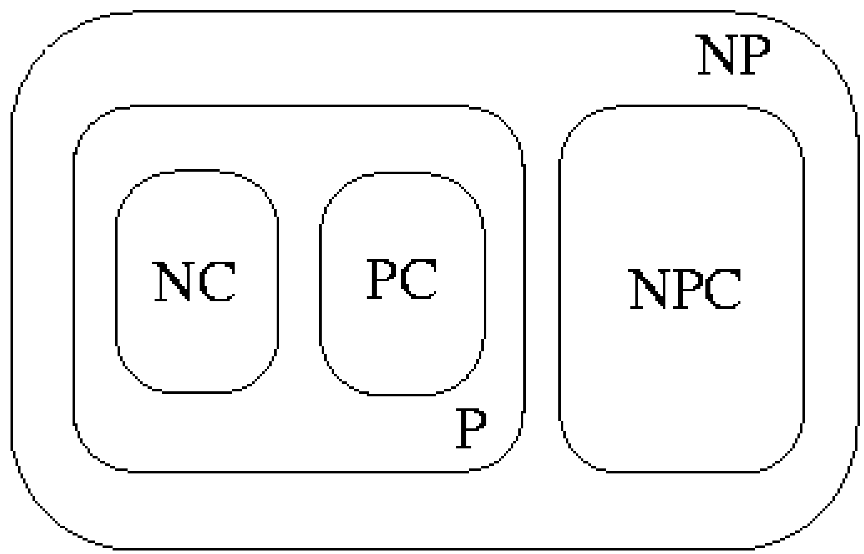
Lemma 6 *The class of P-complete problems is closed under the relation of NC-reduction: More precisely, for any two binary languages L_1, L_2 in P we have: if $L_1 \leq_{NC} L_2$ holds and L_1 is P-complete, then L_2 is P-complete.*

PROOF.



Corollary 2 *If any P-complete problem is in NC, then NC=P. If any P-complete problem is provably not to have an NC algorithm, then no P-complete problem is in NC.*

There is a strong belief that the second case is true, even though no proof was given up to now.



To show that a langauge L is P-complete, we have to show that L is in P, and then we have again two choices.

- We have to show that any language in P can be NC-reduced to L. This work had to be done for the first P-complete problem.

- We can select a suitable known P-complete L' and must
  - describe an algorithm computing a function f mapping instances of L' into instances of L,
  - prove that x in L' iff $f(x)$ in L for all x in $\{0,1\}^*$,
  - prove that f can be computed by an NC algorithm.

The reason behind P-completeness of a problem is usually an unavoidable data dependency within the problem solution, so that the processors cannot work on many parts of the problem simultaneously, and that is why these problems are called inherently sequential.

The basic P-complete problem One of the historically first P-complete problems, which plays the same role in parallel complexity theory as the SATISFIABILITY problem in the sequential NP-completeness theory, is the Circuit Value Problem.

Circuit Value Problem (CVP):

Input: A binary encoding of a Boolean combinatorial circuit C composed of AND, OR, and NOT gates, inputs x1, x2, ..., xn, and a designated output y.

Problem: Is y=true on inputs x1, x2, ..., xn?

Theorem 1 (Ladner, 1975) *CVP is P-complete under NC-reduction.*

Most of the material of these course notes are inspired by the following.

- Parallel Algorithms (WISM 459, 2005/2006) by Rob Bisseling.
  `http://www.math.uu.nl/people/bisselin/pas1.html`
- Topics in Parallel Computing (CS 838, Univ. of Wisconsin-Madison, Spring 1999) by Pavel Tvrdik.
  `http://www.cs.wisc.edu/ tvrdik/cs838.html`
- Parallel Algorithms (CS 662, San Diego Univ., Spring 1996) by Roger Whitney.
  `http://www.eli.sdsu.edu/courses/spring96/cs662/notes/`