Parallel Computation

Parallel computation is the use of machines that allow for the execution of multiple computation steps simultaneously.

We have encountered some form of parallelism already in the form of nondeterministic Turing machines. However, this parallelism is limited since there is no communication allowed between the computational branches.

Here we investigate a model of parallelism that is more realistic: deterministic processing elements and communication between the elements.

Since our processing elements are deterministic we will see that class of decision problems amenable to parallelization is (most likely) a subset of P.

Matrix Multiplication

We start our investigation by examining a problem that can be readily executed in parallel: matrix multiplication.

Given two $m \times m$ matrices A and B, the product $C = A \times B$ is defined as

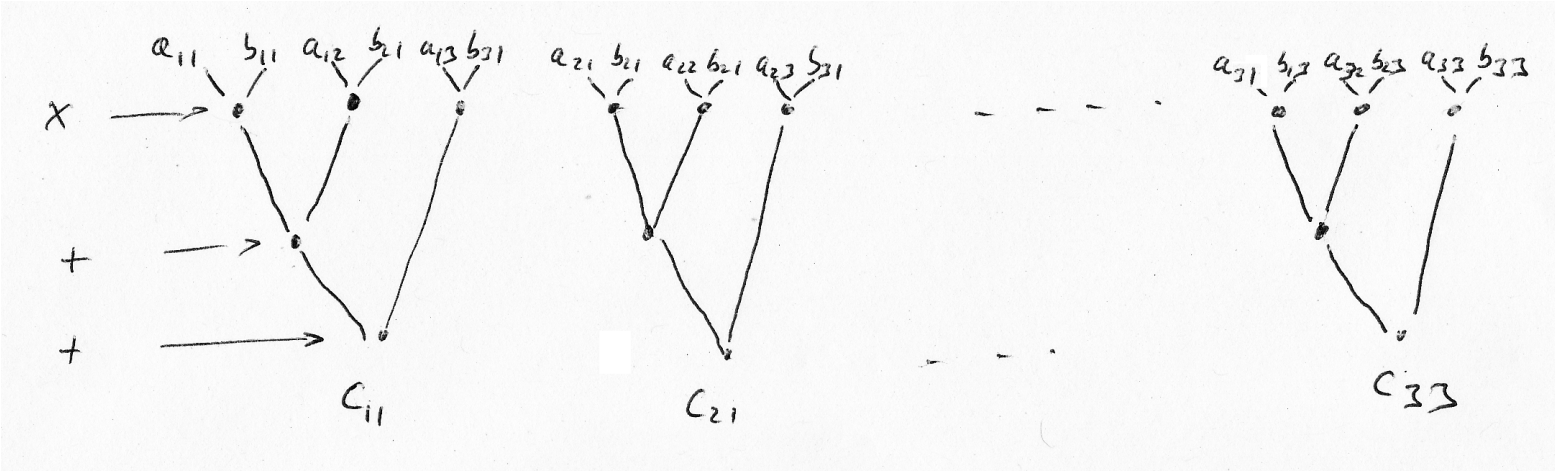
$$c_{ij} = \sum_{k=1}^{m} a_{ik} \times b_{kj}, \text{ for } i, j = 1, \ldots, m.$$

Let $n = 2m^2$, then it is easy to see that a straightforward sequential implementation of this algorithm will execute in $O(m^3) = O((\sqrt{\frac{n}{2}})^3) = O(n^{3/2})$ time.

The question is, if we had just the right parallel machine, what would be the execution time then?

Parallel Matrix Multiplication

Assume that we had just the right number of processors, then we could embed the computation on the parallel machine as follows, assuming a 3×3 multiplication problem:



Note that we assume $(3+2)m^2 = m^3 + 2m^2$ processors (with $m = 3$ for our example). In other words, we assume $O(m^3) = O(n^{3/2})$ processors. Observe that our total execution time characterized by the depth of the layers of processors drops to $O(\log m) = O(\log n^{3/2}) = O(\frac{1}{2}(\log n - \log 2)) = O(\log n)$.

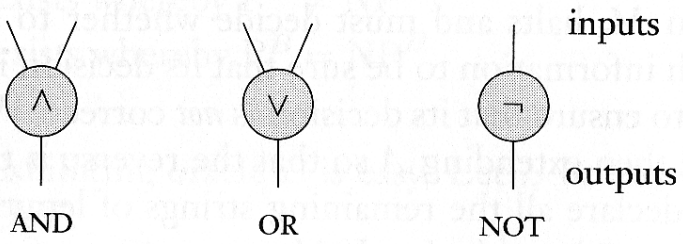
This means, given just the right machine we can go from polynomial time to logarithmic time.

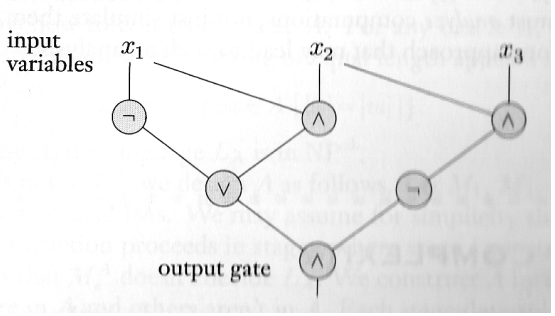
Circuits

The idea of 'just the right machine' gives us a way to characterize parallizability of problems.

We do this via the notion of ***circuits***. Circuits are constructed from gates and we view each gate as an extremely simple processor. By connecting these processors we obtain a ***parallel machine*** since each of the processors can perform work independently as soon as input is available. Formally,

Definition: A ***Boolean circuit*** is a collection of ***gates*** and ***inputs*** connected by ***wires***. Cycles are not permitted. Gates can take three forms: AND, OR, and NOT gates.



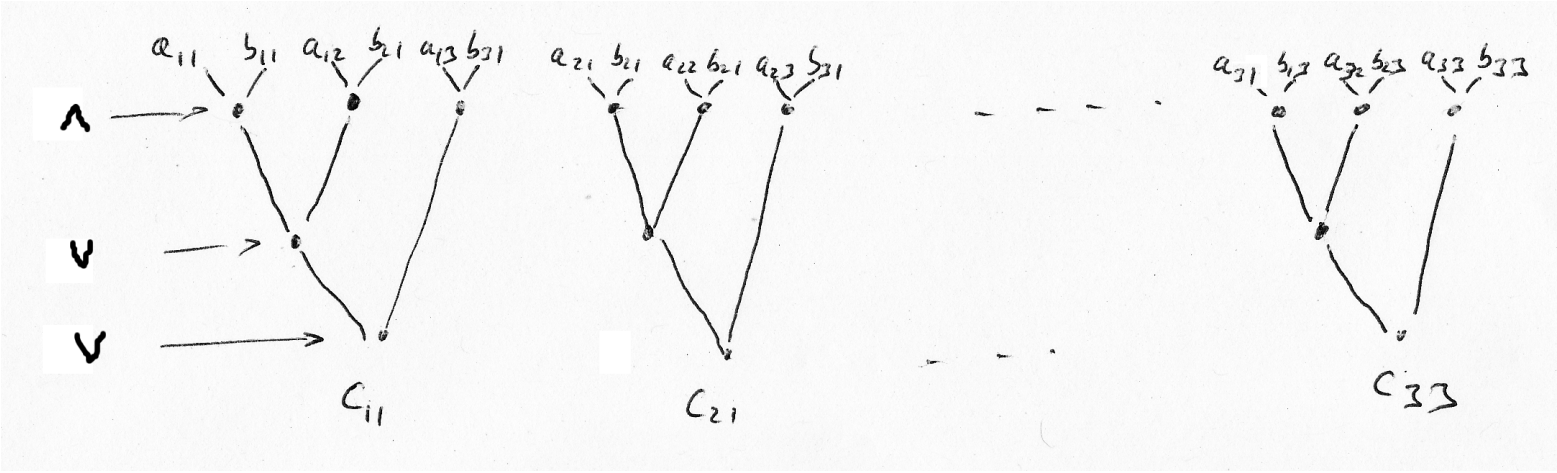


Circuit Matrix Multiply

Given two binary $m \times m$ matrices A and B, the binary product $C = A \wedge B$ is defined as

$$c_{ij} = \bigvee_{k=1}^{m} a_{ik} \wedge b_{kj}, \text{ for } i, j = 1, \ldots, m.$$

Our 3×3 example is then,



Note, all the prior analysis still holds, the only difference is that we now have only very simple processors.

Circuit Families

Note, in general we can use circuits as deciders:

$$f_C : \{0,1\}^n \rightarrow \{0,1\}.$$

That is, given a string $w \in \{0,1\}^n$, the circuit implementing the function f_C can answer the question $w \stackrel{?}{\in} L(C)$.

This means we can use circuits as general representations of algorithms (Perhaps not surprising since computers are made up of circuits).

One of our assumption was that we have always the exact circuit available for our problem, or in other words, there is always a circuit available whose n matches the string we want to decide.

Definition: A ***circuit family*** C is an infinite list of circuits $(C_0, C_1, \ldots)$, where C_n has n input variables. We say that C decides a language A over $\{0,1\}$ if, for every string w,

$$w \in A \text{ iff } C_n(w) = 1,$$

where $|w| = n$.

Parallel Complexity

Definition: We define the ***parallel processor complexity*** of a Boolean circuit to be its *size*. We define the ***parallel time complexity*** of a Boolean circuit to be its *depth*, or the longest distance from an input variable to an output gate.

The complexity of a parallel algorithm is always given as a pair

$$\langle \textit{processor complexity}, \textit{time complexity} \rangle .$$

For our Boolean matrix multiplication example we have a parallel complexity of

$$\langle O(n^{3/2}), O(\log n) \rangle .$$

The Class L

The class L is the class of languages that is decidable in $log(n)$-space,

$$L = \{A | A \text{ is decidable a in } log(n)\text{-space deterministic TM}\}$$

Observation: $L \subseteq P$

The Class NC

Before we can define the class of all parallel algorithms we need one more technical definition,

Definition: A family of circuits C_k, $k = 1, 2, \ldots$, is ***uniform*** if some log space transducer T outputs $\langle C_n \rangle$ when T's input is 1^n.

This definition says that given a specification of a circuit, the circuit can be constructed easily.

This avoids the problem that we can assume that we can construct circuits that solve the TSP problem easily.

Definition: For $i \geq 1$ let NC^i be the class of languages that can be decided by a uniform family of circuits with polynomial size and $O(\log^i n)$ depth. Let

$$NC = \bigcup_i NC^i, \text{ for } i \geq 1.$$

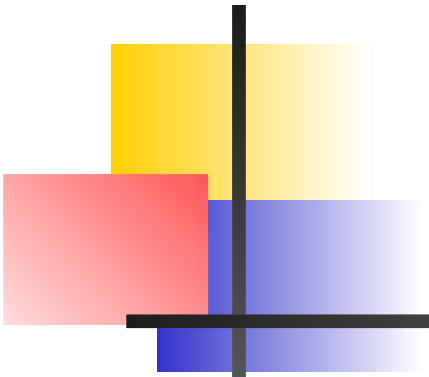
The Class NC

Theorem:

$$NC \subseteq P$$

Proof: A polynomial time algorithm can run the log space transducer to generate circuit C_n and then simulate it on an input of length n. $\square$

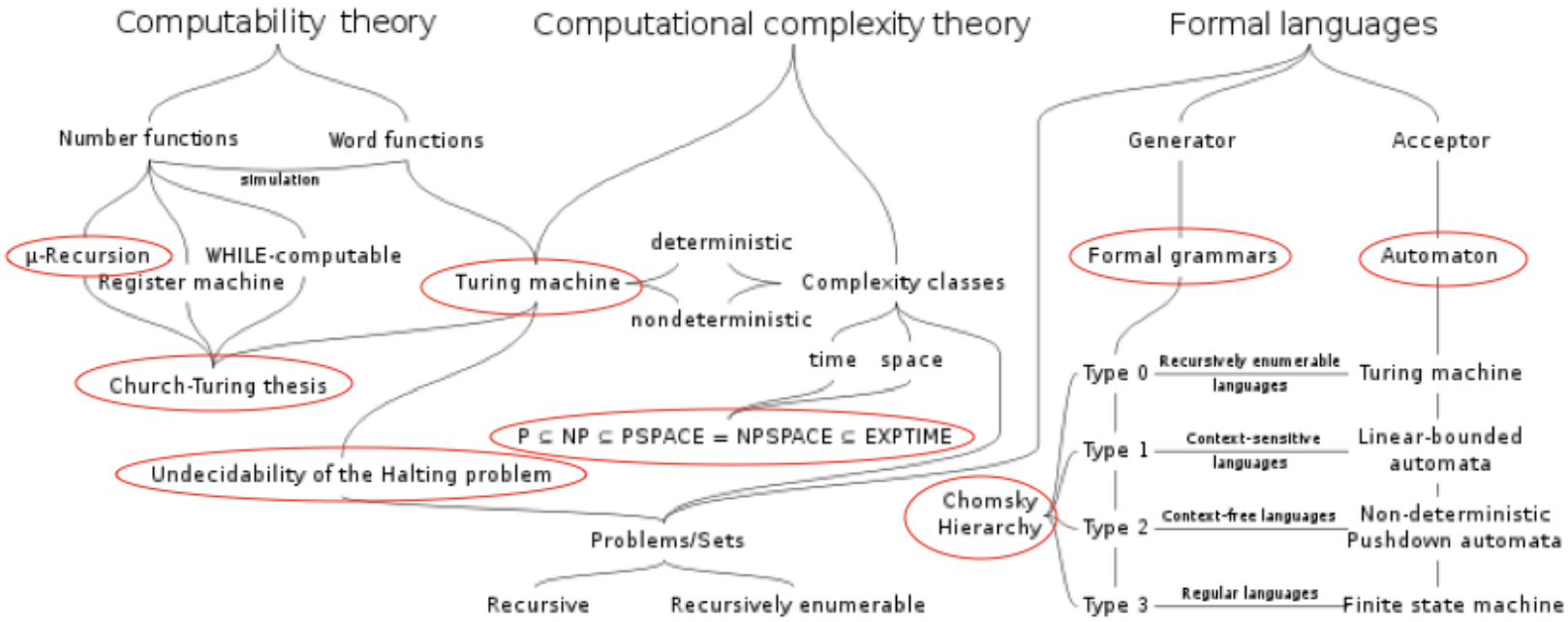
Now, is $NC = P$? Probably not, since P contains a number of inherently sequential problems.



Finally!

Observation: $L \subseteq NC \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$

Overview


Computability theory
Computational complexity theory
Formal languages
Number functions
Word functions
simulation
μ-Recursion
WHILE-computable
Register machine
Turing machine
deterministic
nondeterministic
Complexity classes
time
space
Church-Turing thesis
P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME
Undecidability of the Halting problem
Problems/Sets
Recursive
Recursively enumerable
Generator
Acceptor
Formal grammars
Automaton
Chomsky Hierarchy
Type 0
Recursively enumerable languages
Turing machine
Type 1
Context-sensitive languages
Linear-bounded automata
Type 2
Context-free languages
Non-deterministic Pushdown automata
Type 3
Regular languages
Finite state machine