Technische Universität München (I7) **Summer term 2019**
Prof. J. Křetínský / Dr. M.Raskin **09.06.2019**

Solution

Computational Complexity – Homework 11

Discussed on 14.06.2019.

Definition 1. A language L is in $\mathbf{P_{/poly}}$ if there exist a family $\{C_n\}$ of Boolean circuits of size polynomial in n such that for all $x \in \{0,1\}^n$

$$x \in L \text{ iff } C_n(x) = 1.$$

A family of Boolean circuits $\{C_n \mid n \in \mathbb{N}\}$ is *logspace uniform* if there is a deterministic Turing machine M running in logarithmic space which on input 1^n outputs a description of C_n. Similarly for *polytime uniform* we require M run in polynomial time.

(Note that the definition of **NC** requires the logspace uniformity together with polynomial size and polylog depth.)

Exercise 11.1

Show that $\mathbf{BPP} \subseteq \mathbf{P_{/poly}}$.

Remark: Use one of the results on **BPP** which have already been shown in the lecture.

Solution: We have seen in the lecture that if $L \in \mathbf{BPP}$ is decided by some TM $M(x,u)$ then for every $n \in \mathbb{N}$ there exists a $u_n \in \{0,1\}^{p(n)}$ s.t.

$$\forall x \in \{0,1\}^n : x \in L \text{ iff } M(x, u_n) = 1.$$

We therefore can first transform $M(x,u)$ into a family of circuits of size polynomial in x (recall that $|u| \leq p(|x|)$) and then hardwire u_n into the circuits.

Exercise 11.2

(a) Show that for every polynomial p the following language is in **coNP**:

$$L_p := \left\{ \langle C_1, C_2, \ldots, C_n \rangle \;\middle|\; \begin{array}{l} C_i \text{ is a circuit of size at most } p(i) \text{ which decides } \textsc{Sat} \text{ for every formula} \\ \text{of length exactly } i \end{array} \right\}.$$

Remark: Assume w.l.o.g. that every formula has length at least one with 0 (false) and 1 (true) the two formulae of length 1. Now, use the circuits $C_1, \ldots, C_i$ ($i \geq 0$) to check the correctness of circuit C_{i+1}. (Recall the so-called self-reducibility of Sat.)

(b) Show that **PH** collapses to the second level if $\mathbf{NP} \subseteq \mathbf{P_{/poly}}$, i.e. if there is a sequnce of polynomial sized circuits for SAT.

Remark: It suffices to show that $\Pi_2\text{SAT} \in \mathbf{\Sigma}_2^p$.

(c) What happens if there is a sequnce of polynomial sized circuits for SAT that is moreover logspace uniform?
What if it is polytime uniform?

Solution:

(a) The TM first checks that $|C_i| \leq p(i)$ and that C_i indeed encodes a Boolean circuit. This takes time $\mathcal{O}(n \cdot p(n))$ (assuming that $p(n)$ grows monotonically). Next, the TM chooses some Boolean formula ϕ of length at most n. If $|\phi| = 1$, the TM directly checks if $C_1(\phi) = \phi$. Otherwise, it choose (deterministically) some variable x from ϕ and checks again in polynomial time that

$$C_{|\phi|}(\phi) = \bigvee_{b=0,1} C_{|\phi[x:=b]|}(\phi[x := b]).$$

By definition of **coNP** the TM accepts $\langle C_1, \ldots, C_n \rangle$ only if it does not find any formula of length at most n for which this test fails.

(b) If $\mathbf{NP} \subseteq \mathbf{P_{/poly}}$, then there exists some polynomial p and a circuit sequence $(C_n)_{n\in\mathbb{N}}$ with $|C_n| \leq p(n)$ which decides SAT. In particular, for this p every (encoding of a) prefix $\langle C_1, \ldots, C_n \rangle$ is in L_p.

Now, Π_2SAT, i.e.,

$$\text{decide if } \forall u \exists v\ : \phi(u, v) \text{ is true (with } \phi \text{ a Boolean expression)}$$

is $\mathbf{\Pi}_2^p$-complete where u and v are bounded by the length of ϕ.

We now have

$$\forall u \exists v\ : \phi(u, v) \text{ iff } \exists \langle C_1, \ldots, C_{|\phi|} \rangle \in L_p\ : \forall u\ : C_{|\phi(u,\cdot)|}(\phi(u, \cdot))$$

where the latter describes a computation in $\mathbf{\Sigma}_2^p$. So, $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$ which implies $\mathbf{PH} = \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$.

(c) In both cases, we have $\mathbf{P} = \mathbf{NP}$. It suffices to show that SAT is then in **P**. Given a formula ϕ of length $n = |\phi|$, we construct in log-space from the input 1^n the circuit C_n which decides SAT for all formulae of length exactly n. As $\mathbf{L} \subseteq \mathbf{P}$, this can be done in time polynomial in $n = |\phi|$. We then evaluate C_n on ϕ. This can again be done in time polynomial in the size of C_n which by definition is polynomial in the size of $n = |\phi|$.

Exercise 11.3

Prove that for $n \geq 100$, most of the boolean functions on n variables require circuits of size at least $2^n/n$.

Solution:

THEOREM 6.15

For $n \geq 100$, almost all boolean functions on n variables require circuits of size at least $2^n/(10n)$.

PROOF: We use a simple counting argument. There are at most s^{3s} circuits of size s (just count the number of labeled directed graphs, where each node has indegree at most 2). Hence this is an upperbound on the number of functions on n variables with circuits of size s. For $s = 2^n/(10n)$, this number is at most $2^{2^n/10}$, which is miniscule compared 2^{2^n}, the number of boolean functions on n variables. Hence most Boolean functions do not have such small circuits. ■

Exercise 11.4

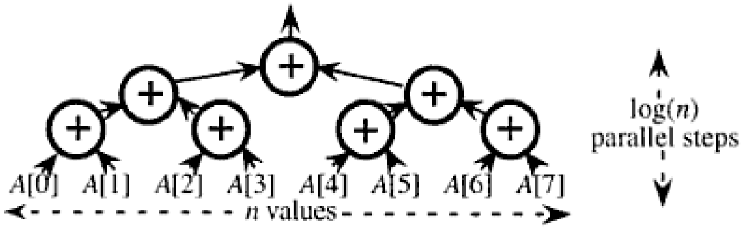
(a) Design a circuit family for the parity problem and describe it formally. Prove that there is a logspace uniform one.

(b) Let $A[0..n]$ be an array of integers. Design a PRAM for summing numbers in an array, i.e. compute $\sum_{i=0}^{n} A[i]$.
Can you compute the array-suffix-sum, i.e. $\sum_{i=j}^{n} A[i]$ for all $0 \leq j \leq n$, with the same complexity?

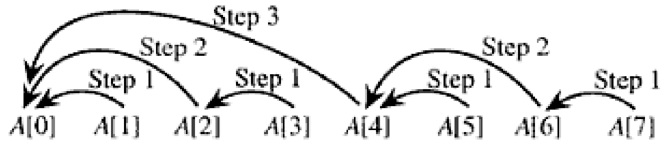
Solution:

Example: Summing an Array

Idea: To add up the items in an array $A[0]...A[n-1]$, in one step items one apart are added to cut the problem in half, in a second step items two apart are added to again cut the size of the problem in half, etc. This approach can be viewed as simulating a tree network with the array A stored in the leaves and the sum coming out of the root:



In practice, we do not really need all of the processors; we can keep overwriting A. Concurrent reads are not used, and the algorithm works in the EREW model.



The summing algorithm:

    function erewSUM($A[0]...A[n-1]$)
        for k=0 **to** $\lceil \log_2(n) \rceil -1$ **do**
            for $0 \le i < n-2^{k+1}$ **in parallel do**
                if i is a multiple of 2^{k+1} **then** $A[i] := A[i]+A[i+2^k]$
        return $A[0]$
        end

        Note: The test if i is a multiple of 2^{k+1} is not necessary (see the exercises).

An equivalent way to express the summing algorithm:

    function erewSUM($A[0]...A[n-1]$)
        $k := 1$
        while $k<n$ **do begin**
            for $0 \le i < n-2k$ **in parallel do**
                if i is a multiple of $2k$ **then** $A[i] := A[i]+A[i+k]$
            $k := k*2$
            end
        return $A[0]$
        end

Complexity: Each of the $\lceil \log_2(n) \rceil$ iterations uses $O(1)$ time (since $O(1)$ time is used for the body of the parallel *for* loop); hence the algorithm is $O(\log(n))$ time. $O(1)$ space is used in addition to the space used by A. The number of processors used is $n/2$; however, we shall see later (Brent's Lemma) that $O(n/\log(n))$ processors suffice.

Copyrighted material

Example: List Prefix-Sum / List Ranking

Notation: L is a singly-linked list represented by $A[0]...A[n-1]$, $0 \le first < n$ the index of the first item, and the array $NEXT[0]...NEXT[n-1]$ such that by starting with $i := A[first]$ and repeatedly doing $i :=$ NEXT[i], we visit all positions and end up at a position i such that $NEXT[i]=nil$; for simplicity assume items are ≥ 0 and $nil = -1$.

List suffix-sum: We wish to compute for each $0 \le i < n$ the sum of all positions from position i through the end of the list. We can use the same distance-doubling idea as for array sum, emanating from every vertex; only the EREW PRAM model is needed:

```
procedure erewListSuffixSum(L)
    while NEXT[first]≠nil do
        for 0 ≤ i < n in parallel do if NEXT[i]≠nil then begin
            A[i] := A[i] + A[NEXT[i]]
            NEXT[i] := NEXT[NEXT[i]]
            end
    end
```

List prefix-sum: We wish to compute for each $0 \le i < n$ the sum of all positions from position i through the start of the list. We can reverse the list (in parallel do NEXT[NEXT[i]] := i, set *first* to what used to be the last position, and set the NEXT field of what used to be the first position to *nil*) and then do a suffix sum (see the exercises).

List ranking: The special case of prefix-sum where all values of A are 1 (there could be additional data associated with each vertex), and we compute the position of each vertex.

Suffix-sum / prefix-sum / list ranking on an array: For the special case of a list in sequential positions of an array, define $NEXT[i]=i+1$, $0 \le i < n-1$, and $NEXT[n-1]=nil$. For example, suppose the array $A[0]...A[9]$ initially contained 1 in each location and consider the successive iterations of the *while* loop of erewListSuffixSum:

array *position*	0	1	2	3	4	5	6	7	8	9
starting values	1	1	1	1	1	1	1	1	1	1
values after *first* iteration	2	2	2	2	2	2	2	2	2	1
values after *second* iteration	4	4	4	4	4	4	4	3	2	1
values after *third* iteration	8	8	8	7	6	5	4	3	2	1
values after *fourth* iteration	10	9	8	7	6	5	4	3	2	1

Complexity: $O(\log(n))$ time since each iteration of the outer *while* loop for suffix-sum doubles the distance over which sums are taken, and prefix-sum adds only $O(1)$ additional time. $O(n)$ space in addition to the space for L (or $O(n)$ additional space if we cannot overwrite A and $NEXT$ and must first make copies). $O(n)$ processors are used.

Copyrighted material