Parallel Computation

Parallel computation means computation with any number of processors working simultaneously, possibly sharing data. The *work complexity* of a parallel computation is defined to be the time complexity times the number of processors. Number of processors is *worst case*; if the number of steps is T and we use p_t processors at step t, for each t, the number of processors for the computation is defined to be $\max\{p_t\}$.

Nick's Class. We say that a computation is $\mathcal{NC}$, in Nick's Class, if it takes polylogarithmic time with polynomially many processors.

Balancing. Suppose a computation consists of $\log n$ steps, and we use 2^t processors at the t^{th} step for each $1 \leq t \leq \log n$. We are taking logarithms to be base 2, so we use n processors at the last step, and no more than n at any step. The work complexity of the computation is thus $n \log n$. But if we let w_t be the work done at step t, $\sum w_t = \Theta(n)$. So, you would think it would be possible to have the total work of the computation be $\Theta(n)$, as well. This is possible by rearranging the computation.

General dynamic programming is not believed to be $\mathcal{NC}$. However, many practical dynamic programming problems are $\mathcal{NC}$.

Let $S_1, S_2, \ldots S_n$ be the subproblems of a dynamic program $\mathcal{P}$. We define $\mathcal{P}$ to be *proper* if the following conditions hold.

1. The input to S_0 is a string of $O(\log n)$ bits.
2. For $i < n$, the output of subprogram S_i is a string of $O(\log n)$ bits.
3. For $i > 0$, the input of subprogram S_1 is the output of subprogram S_{i-1}.
4. The computation of each subprogram takes polylogarithmic time and uses polynomially many processors.
5. The output of subprogram S_n is one bit.

Theorem 1 *If $\mathcal{P}$ is a proper dynamic program, then $\mathcal{P}$ is $\mathcal{NC}$.*

Proof: Let ℓ be the maximum length of an input string of a subprogram; by padding with zeros, we require that all strings have length ℓ. Let $\Sigma = \{0.1\}$

and $L = \Sigma^\ell$, the set of binary strings of length ℓ. Since $\ell = O(\log n)$, $|L| = 2^\ell$ is a polynomial function of n. We write L^L for the set of functions $L \to L$. Thinking of a function as a set of ordered pairs, any $F \in L^L$ is a subset of $L \times L$ of order 2^ℓ. We store eacn $F \in L^L$ as a table T_F with 2^ℓ rows, one for each $\sigma \in L$, and 2ℓ columns to store the ordered pair $(\sigma, F(\sigma))$ for each row. The composition of two such functions can be computed in polylogarithmic time with polynomially many processors, as follows: for any $F, G \in L^L$ and any $\sigma \in L$, $F \circ G(\sigma) = F(G(\sigma))$. Use one processor for each $\sigma \in L$, a total of 2^ℓ processors. To compute $F(G(\sigma))$, that processor fetches $\tau = F(\sigma)$ from T_F, then searches T_G for row τ, then fetches $G(\tau)$, then stores the ordered pair $(\sigma, G(\tau))$ in $T_{F \circ G}$.

Let σ_0 be the input string of $\mathcal{P}$, let $\sigma_i \in L$ be the output of S_i, and let F^i be the function computed by S_i, *i.e.*, $F^i(\sigma_{i-1}) = \sigma_i$. For $i < j$, let F_i^j be the composition $F^{j-1} \circ F^{j-2} \circ \cdots \circ F^i$; that is, $F_i^j(\sigma_i) = \sigma_j$ Note that $F_i^k = F_j^k \circ F_i^j$ for $i < j < k$.

Finally, we give an $\mathcal{NC}$ computation for $\mathcal{P}$. We can assume n is a power of 2.

1. Compute F_{t-1}^t‘ $= F^t$ for each $1 \le t \le n$.
2. Using composition, for each $p = 2^k \le n$, compute $F_{(t-1)p}^{tp}$ for $1 \le t \le n/p$. For example:
   $F_0^2 = F_1^2 \circ F_0^1$
   $F_2^4 = F_3^4 \circ F_2^3$
   $F_4^6 = F_5^6 \circ F_4^5$
   ... *etc.*
   $F_0^4 = F_2^4 \circ F_0^2$
   $F_4^8 = F_6^8 \circ F_4^6$
   $F_8^{12} = F_{10}^{12} \circ F_8^{10}$
   ... *etc.*
   $F_{80}^{96} = F_{88}^{96} \circ F_{80}^{88}$
   ... *etc.*
   Finally, $\sigma_n = F_0^n(\sigma_0)$

The computation consists of $\log n$ phases, each of which can be done using polynomially many processors in polylogarithmic time. The output is the first bit of σ_n. Thus $\mathcal{P}$ is $\mathcal{NC}$. ▮

Regular Languages

Lemma 1 *Every regular language is $\mathcal{NC}$.*

Proof: Let L be a regular language over an alphabet Σ. Let M be a DFA which decides L, with state set Q, transition function $\delta : Q \times \Sigma \to Q$, where the set of final states is $F \subseteq Q$.

Let w be a string over Σ of length n. Let $w[i]$ be the i^{th} symbol of w. Let $\mathcal{P}^w$ be the dynamic program with subprograms $S_1, \dots S_n$, where

1. The input of S_1 is the start state of M.
2. For $i > 1$, The input of S_i is the output of S_{i-1}. a member of Q.
3. S_i computes the function $f_i : Q \to Q$, where $f_i(q) = \delta(q, w[i])$, which is the output of S_i for $i < n$. The output of S_n is **1** if S_n computes a member of F, otherwise **0**.

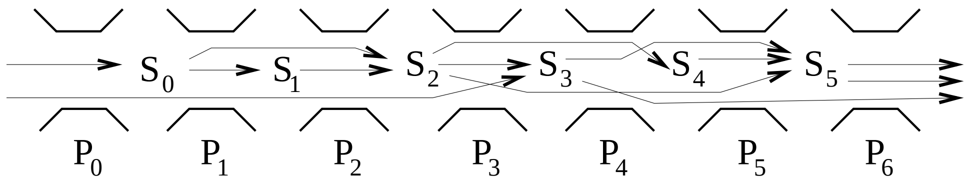
Thus, for $1 \leq i \leq n$, the input of S_i is the $(i-1)^{\text{st}}$ state in the computation of M with input w, and its output is the i^{th} state of that computation, unless $i = n$, in which case the output is Boolean: 1 if w is accepted by M, 0 if not.

Each output is a single bit or a member of Q, whose size is taken to be constant. By Theorem 1, L is $\mathcal{NC}$. ∎

0.1 Pipeline Analysis

In your future, as a professional programmer (perhaps), you will need to judge whether a sequential program can be efficiently parallelized. If so, Theorem 2 below will be the result to look at.

Let $\mathcal{DP}$ be a dynamic program with subproblems $S_0, \dots S_{n-1}$. There are p_0 bits of input. Each subproblem can read bits from any earlier subproblem. We define P_i to be the *pipeline* of information flowing between S_{i-1} and S_i. Let p_i be the number of bits in P_i. The bits of P_i could be input bits or could have been in the input, or have been sent by an subprogram S_j for $j < i$. Let P_n be the pipeline of bits of output of $\mathcal{DP}$, and p_n the number of bits of output, and that $p_i = O(\log n)$. We assume that the computation of each S_i takes polylogarithmic time and uses polynomially many processors.



In the example shown in the figure, $n = 7$, $p_0 = 2$, $p_1 = p_2 = p_6 = 3$, and $p_3 = p_4 = p_5 = 4$.

Theorem 2 *The computation of $\mathcal{DP}$ is emulated by an $\mathcal{NC}$ program.*

Proof: Note that P_i is a bitstring of length p_i. The goal is to compute the output string P_n from P_0, the input string.

For any $0 \leq i < j \leq n$, let F_i^j be the function which returns P_j given P_i, which can easily be computed in polynomial time using one processor. For some constant k, $p_i \leq k \log n$ for each i, and the computation time of each S_i is no greater than $\log^k n$.

By the same reasoning used in the proof of Theorem 1, each F_i^j is one of at most n^k functions, each stored as polynomially many bits. We can compute each F_{i-1}^i in polylogarithmic time using polynomially many processors, and we can compute F_i^j from F_i^ℓ and F_ℓ^j, for any $i < \ell < j$, in constant time with polynomially many processors.

Again, in the manner used in Theorem 1, we can compute F_0^n in $O(\log n)$ phases, each of which takes at most n processors and uses polylogarithmic time. Finally, $P_n = F_0^n(P_0)$. ∎