$\mathcal{NC}$ and Dynamic Programming

Nick's Class

$\mathcal{NC}$, or Nick's Class, is named after Nick Pippenger, currently on the faculty of Harvey Mudd College. A language is $\mathcal{NC}$ if its membership problem can be solved by a parallel program using polynomially many processors in polylogarithmic time.
Many of the problems that you are familiar with are in the class $\mathcal{NC}$. For example, the 0/1 version of the shortest path problem is in $\mathcal{NC}$, and every context-free language is in the class $\mathcal{NC}$. Whether $\mathcal{NC} = \mathcal{P}$–TIME is an open question of enormous theoretical and practical importance.

$\mathcal{NC}$ Functions

Let $\Sigma = \{0,1\}$, the binary alphabet. Without loss of generality, all strings are *i.e.* over Σ. *i.e.* binary. We consider any mathematical function to be a function $f : \Sigma^* \to \Sigma^*$. The function f is defined to be $\mathcal{P}$–TIME if there exists a constant k such that, for any $w \in \Sigma^*$, $f(w)$ can be computed by a (single processor) machine in $O(n^k)$ steps, where $n = |w|$, while f is defined to be $\mathcal{NC}$ if there is a k such that, for any $w \in \Sigma^*$, $f(w)$ can be computed by $O(n^k)$ processors in $O(\log^k n)$ time.

The Circuit Value Problem, or the Boolean Circuit Problem

We say that a $\mathcal{P}$-TIME language (problem) is $\mathcal{P}$-complete if every $\mathcal{P}$-TIME language can be reduced to it by an $\mathcal{NC}$ function. We now give a $\mathcal{P}$-complete problem, namely the circuit value problem (CVP) which is a dynamic programming problem with Boolean variables. An instance of the CVP consists of a sequence of assignments, where

1. the left side of the i^{th} assignment is the Boolean variable x_i,
2. the right side of the i^{th} assignment is one of the following:
   (a) 0 (false),
   (b) 1 (true),
   (c) x_j for some $j < i$,
   (d) $!x_j$ for some $j < i$,
   (e) $x_j * x_k$ for some $j < i$ and $k < i$,
   (f) $x_j + x_k$ for some $j < i$ and $k < i$.
   (g) $!x_j$ for some $j < i$,

   We write $+$, $*$, ! for and, or, not. The answer is the value of the last variable, x_n.

The answer is the value of the last variable, x_n.
Trivially, CVP is in $\mathcal{P}$. Simply execute the n statements in order. In fact, the CVP is a dynamic programming problem. It is known that CVP is $\mathcal{P}$-complete, which implies that if CVP $\in \mathcal{NC}$ then $\mathcal{NC} = \mathcal{P}$–TIME.

Dynamic Programming Can be $\mathcal{NC}$

.
The CVP is clearly a dynamic programming problem Thus, in general, dynamic programming problems are not known to be a subclass of $\mathcal{NC}$. However, there are DP problems of importance that are $\mathcal{NC}$.

A Definition of Dynamic Programming

The usual definition of a DP instance $\mathcal{I}$ is an acyclic directed graph whose vertices are processes, which we usually call subproblems. Label these $P_1, \ldots P_t, \ldots P_n$. The output w_t of P_t is computed from initial inputs together with the outputs of all $P_{t'}$ such that there is an arc of $\mathcal{I}$ from $P_{t'}$ to P_t. Outputs and inputs are strings, which we can assume are binary. We define the *length* of $\mathcal{I}$ to be T. For convenience, we assume that P_n is a sink of $\mathcal{I}$, and the output of P_n is the output of $\mathcal{I}$.

Width of a DP Instance. For the rest of this section, we let $\mathcal{I}$ be a specific DP instance. We assume that the computation by each process is $\mathcal{NC}$, that is, P_t computes w_t using $O(m^k)$ processors in $O(\log^k m)$ time, where m is the number of input bits of P_t.
We define a *cut* $\mathcal{C}$ of $\mathcal{I}$ to be a partion of the processes into $\mathcal{I}_1$ and $\mathcal{I}_2$ such that there is no arc from any member of $\mathcal{P}_2$ to any member of $\mathcal{P}_1$. Let $\mathcal{L}$ be the set of all members of $\mathcal{I}_1$ which have an arc to some member of $\mathcal{I}_2$. We define *signature*$(\mathcal{C})$ to be the concatenation of the outputs of all members of $\mathcal{L}$, which we think of as the information that flows across the cut. Let $\mathcal{C}_t$ be the cut $(\{P_1, \ldots P_t\}, \{P_{t+1}, \ldots P_n\})$. Let $s_t = \textit{signature}(\mathcal{C}_t)$ We define $W(\mathcal{I})$, the *width* of $\mathcal{I}$ to be $\max_t \left\{2^{|s_t|}\right\}$.

Informally, we say that $\mathcal{I}$ is *thin* if $W(\mathcal{I})$ is a polynomial function of n. We will show that a thin DP problem can be solved by an $\mathcal{NC}$ computation.
For $0 \leq t \leq n$, let S_t be the set of all possible values of s_t. For example, S_0 is the set of all possible input strings. Since $\mathcal{I}$ is thin, each $|S_t|$ is a polynomial function of n. Let $G_t^{t+1} : S_t \to S_{t+1}$ be the function that computes s_{t+1} from s_t.
Remark: G_t^{t+1} is an $\mathcal{NC}$ function. $\mathcal{I}$ is illustrated by the following diagram.

$$S_0 \xrightarrow{G_0^1} S_1 \xrightarrow{G_1^2} S_2 \xrightarrow{G_2^3} \cdots \xrightarrow{G_{n-1}^n} S_n$$

Implementation of Composition. We store a function f as a set of ordered pairs, $\textit{pairs}(f) = \{(x, y) : f(x) = y\}$ For example, if $f(x) = x^2$ for $x \in \{-1, 0, 1, 2\}$, then $\textit{pairs}(f) = \{(-1, 1), (0, 0), (1, 1), (2, 4)\}$. If $f : A \to B$ and $g : B \to C$ are functions, let $gf : A \to C$ be the composition, where $gf(x) = g(f(x))$. We implement the composition using pairs:

$$\textit{pairs}(gf) = \{(x, z) \in A \times C : \exists y \in B : (x, y) \in \textit{pairs}(f), (y, z) \in \textit{pairs}(g)\}$$

This composition is an $\mathcal{NC}$ function with respect to the sizes of the sets $A \times B$ and $B \times C$.
For any $i < j$, we let $G_i^j : S_i \to S_j$ be the composition $G_{j-1}^j G_{j-2}^{j-1} \cdots G_{i+2}^{i+1} G_i^{i+1}$. Composing pairs of functions, we obtain

$$S_0 \xrightarrow{G_0^2} S_2 \xrightarrow{G_2^4} S_4 \xrightarrow{G_4^6} \cdots \xrightarrow{G_{n-2}^n} S_n$$

All these functions can be computed simultaneously in polylogarithmic time using polynomially many processors. Since the size of the domain of G_i^j is polynomial in n, $\textit{pairs}(G_i^j)$ has polynomial size.

Continuing, we obtain

$$S_0 \xrightarrow{G_0^4} S_4 \xrightarrow{G_4^8} S_8 \xrightarrow{G_8^{12}} \cdots \xrightarrow{G_{n-4}^n} S_n$$

(For convenience, we assume that n is a power of 2.) After $\log_2 n$ steps, we obtain the function $G_0^n : S_0 \to S_n$.

The solution to $\mathcal{I}$ is then $G_0^n(s_0)$. The entire computation is done in $O(\log^k n)$ time using $O(n^k)$ processors for some constant k, and hence is $\mathcal{NC}$.

Regular Languages are $\mathcal{NC}$

We will show that every regular language L is $\mathcal{NC}$. Let $M = (Q, \Sigma, \delta, q_{zero}, F)$ be a DFA which accepts M, where Q is the set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the start state, and $F \subseteq Q$ is the set of final states. without loss of generality, $L \subseteq \Sigma^*$ where $\Sigma = \{0, 1\}$, the binary alphabet. We give a dynamic program which decides whether a given string $w \in \Sigma^*$ is a member of L.

Let $n = |w|$, and let $w[i]$ be the ith symbol of w. For any $a \in \Sigma$, let $\delta_a : Q \to Q$ be the function defined by: $\delta_a(q) = \delta(a, q)$ for any $q \in Q$. Let $\mathcal{I}$ be the dynamic program given by the diagram:

$$Q \xrightarrow{\delta_{w_1}} Q \xrightarrow{\delta_{w_2}} Q \xrightarrow{\delta_{w_3}} \cdots \xrightarrow{\delta_{w_n}} Q$$

Putting $\mathcal{I}$ into the language of the previous section, we have $S_i = Q$ and $G_i^{i+1} = \delta_{w_i}$, while $s_i \in S_i = Q$ is the state of M after having read the first i symbols of w. We treat $|Q|$ as a constant, hence F_0^n can be computed in polylogarithmic time with polynomially many processors; $w \in L$ if and only if $G_0^n(q_0) \in F$. Thus L is $\mathcal{NC}$.

Adding Integers is $\mathcal{NC}$

The addition problem is, given binary numerals x, y of length n, compute the binary numeral z for the sum $x + y$. Note that z could have langth $n + 1$.

Let $\Sigma = \{0, 1\}$, the binary alphabet. Let x_i, y_i, and z_i be the i^{th} bits if x, y, and z, respectively. In the list below, we think of the bits as integers 0 or 1. We let c_i be the ith carry bit from the i^{th} place to the $(i + 1)^{\text{st}}$ place. We have:

1. $x_i = (x/2^i)\%2$
2. $y_i = (y/2^i)\%2$
3. $z_i = (z/2^i)\%2$
4. $c_{-1} = 0$;
5. $z_i = (x_i + y_i + c_{i-1})\%2$
6. $c_i = (x_i + y_i + c_{i-1})/2$

Addition of binary numerals is not a 0/1 problem, since we need to obtain $n + 1$ bits. However, it is still an $\mathcal{NC}$ function.

Since x and y are given, we can treat x_i and y_i as constants. We define the function $G_{i-1}^i : \Sigma \to \Sigma$ as follows.

$$G_i^{i+1}(b) = (b + x_i + y_i)/2$$

Then the following dynamic program computes the last carry bit, c_n.

$$\Sigma \xrightarrow{G_0^1} \Sigma \xrightarrow{G_1^2} \Sigma \xrightarrow{G_2^3} \cdots \xrightarrow{G_{n-1}^n} \Sigma$$

In terms of the earlier section, $S_i = \Sigma$ and $s_i = c_i$ for all i.

Finishing the Computation The above dynamic programming finds only the last carry bit, since not all carry bits appear in the computation. We rectify that problem by defining separate dynamic programs for all c_i and running them simultaneously. Once we have all c_i stored in an array, we compute all z_i simultaneously in constant time.