Parallel Computations: Nick's Class

An algorithm is in class $\mathcal{NC}$ (Nick Pippenger's Class) if it runs in $O(\log^{O(1)})$ time using $O(n^{O(1)})$ processors. We discuss $\mathcal{NC}$ algorithms for a number of problems of practical importance, such as addition of n-bit binary numerals, and the regular language membership problem.

Dynamic Progrmming

In general, dynamic programming is polynomial time, and some DP problems such as CVP are known to be $\mathcal{P}$-complete. In this manuscript we concentrate on $\mathcal{NC}$ dynamic programming.

Linear Dynamic Programming Our general problem is that we are given a linear array of data of length n and a linear dynamic program with those data as input.

Here is our model. We are given an array of inputs $x_1, \ldots x_n$ and s_0, and a dynamic program $\mathcal{D}$ with outputs $s_1, s_2 \ldots s_n$, where For each $i \in \{1 \ldots n\}$, $\mathcal{D}$ computes s_i, using as inputs only s_{i-1} and x_i, time t_i. We write $s_i = \mathcal{D}(s_{i-1,x_i}$ The time for $\mathcal{D}$ to compute all outputs is $\sum_{i=1}^n t_i$. In each of our examples, t_i is a polynomial function of n, hence $\mathcal{D}$ is $\mathcal{P}$–TIME.

Here are some examples.

1. Find the sum, or product, of an array of numbers.
2. Find the maximum (or minimum) of an array of numbers, or members of some ordered set.
3. Compute the product of an array of matrices.
4. Compute the sum (or difference) of binary integers, each represented as an array of bits, or whether an integer u is less than an integer v.
5. Given a language $L \subset \Sigma^*$ and an NFA which accepts L, deterine whether a string $w \in \Sigma^*$, which we think of as an array of elements of Σ, is a member of L.
6. $\#$ is an associative operation on a set X, (*i.e.*,, $(X, \#)$ is a semigroup) $x_i \in X$, and

$$s_i = \#_{j=1}^{i} x_i$$

We use a "tournament" paradigm for $\mathcal{D}$. Here are some examples.

Sum of Integers

$6 \quad -3 \quad 5 \quad 1 \quad -8 \quad 2 \quad 0 \quad 7 \quad -1 \quad 5 \quad 4 \quad -3 \quad -6 \quad 7 \quad -8 \quad 1$

$3 \qquad 6 \qquad -6 \qquad 7 \qquad 4 \qquad 1 \qquad 1 \qquad -7$

$9 \qquad\qquad 1 \qquad\qquad 5 \qquad\qquad -6$

$10 \qquad\qquad\qquad -1$

9

Deciding a Regular Language

Let L be a regular language over an alphabet Σ, accepted by

an NFA $M = (\Sigma, Q, \Delta, q_0, F)$. Recall $\Delta : \Sigma \times Q \to 2^Q$. Let $k = |Q|$. For simplicity, we do not allow λ-transitions, There is no loss of generality, since λ-transtions can always be removed without increasing the number of states.

Let $w \in \Sigma^*$, a string of n symbols of Σ. Let $x_i = w_i$, be the i^{th} symbol of w. We let S be the set of logical vectors of length k, Let $s_0 = (1, 0, \ldots 0)$, the vector with 1 (true) in position 0 and all other terms 0 (false), indicating that after 0 steps of a computation, the state of M must be q_0. In general, s_t is true in position i if and only if it is possible for the state of M to be q_i after t steps of the computation, that is, reading the first t symbols of w. The computation accepts w if and only if, for some

$q_j \in F$, position j of s_n is 1.

Logical Matrices

A logical matrix is a matrix whose entries are of Boolean type. We write 1 for true and 0 for false. Matrix addition and multiplication is defined in the usual manner for logical matrices, except that disjunction replaces addition and conjunction replaces multiplication.

For example, $\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

For each $a \in \Sigma$ we define a $k \times k$ logical matrix T_a. The rows and columns of T_a are indexed from 0 to $k-1$. For $0 \le i, j < k$:

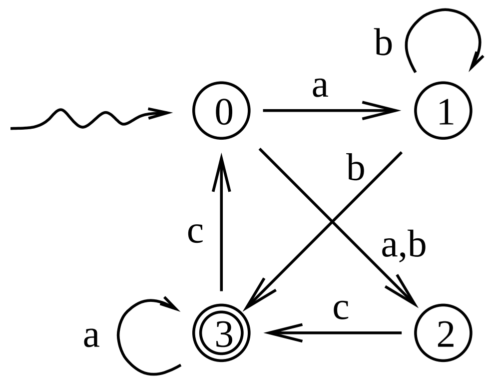
$$T_a[i,j] = \begin{cases} 1 \text{ if } q_j \in \Delta(a, q_i) \\ 0 \text{ otherwise} \end{cases}$$

Finally, $\mathcal{D}$ is computed using transition matrices: if $w_i = a \in \Sigma$, then $s_{i-1} T_a = s_i$.

The definition of a transition matrix can be extended to all strings over Σ, by the rule that $T_{uv} = T_u T_v$ for any strings $u, v \in \Sigma^*$. Thus $T_w = T_{w_1} T_{w_2} \cdots T_{w_n}$, and $s_0 T_w = s_n$, hence $w \in L$.

Example

Let $\Sigma = \{a, b, c\}$ and $L = L(M)$, where M is the followinhg NFA. Let $w = acacabba$.



We compute transition matrices of elementary strings, then copy to the 8 leaves of our computation tree. Each matrix in rows 2–4 is the product of the two above it. Then $s_0 T_w = s_n = (0\,0\,0\,1)$ and $q_3 \in F$.

$$\underset{T_\lambda}{\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}} \quad \underset{T_a}{\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}} \quad \underset{T_b}{\begin{bmatrix}0&0&1&0\\0&1&0&1\\0&0&0&0\\0&0&0&0\end{bmatrix}} \quad \underset{T_c}{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}}$$

$$\underset{T_a}{\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}} \; \underset{T_c}{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}} \; \underset{T_a}{\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}} \; \underset{T_c}{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}} \; \underset{T_a}{\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}} \; \underset{T_b}{\begin{bmatrix}0&0&1&0\\0&1&0&1\\0&0&0&0\\0&0&0&0\end{bmatrix}} \; \underset{T_b}{\begin{bmatrix}0&0&1&0\\0&1&0&1\\0&0&0&0\\0&0&0&0\end{bmatrix}} \; \underset{T_a}{\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}}$$

$$\underset{T_{ac}}{\begin{bmatrix}0&0&0&1\\0&0&0&0\\0&0&0&0\\1&0&0&0\end{bmatrix}} \quad \underset{T_{ac}}{\begin{bmatrix}0&0&0&1\\0&0&0&0\\0&0&0&0\\1&0&0&0\end{bmatrix}} \quad \underset{T_{ab}}{\begin{bmatrix}0&1&0&1\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}} \quad \underset{T_{ba}}{\begin{bmatrix}0&0&0&0\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{bmatrix}}$$

$$\underset{T_{acac}}{\begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}} \qquad \underset{T_{abba}}{\begin{bmatrix}0&0&0&1\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}$$

$$\underset{T_{acacabba} = T_w}{\begin{bmatrix}0&0&0&1\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}$$

Adding Binary Numerals

In this example, we finally use an unconvential semigroup, which not commutative. Let $X = \{0, 1, 2\}$ We will add binary numerals of length n for integers u and v. let $u[i]$ and $v[i]$ be the ith binary digits of u and v; that is, $u = \sum_{i=0}^{n} 2^i$. By convention we write those digits from right to left. Using the standard "ripple" algorithm for addition. Let $w = u + v \bmod 2^n$. The ripple algorithm computes w as follows:

$c_{-1} = 0$
  for i from 1 to $n - 1$
$x_i = u[i] + v[i]$
$w_i = (x_2 + c_{i-1}) \bmod 2$
$c_i = \lfloor x_2 + c_{i-1} \rfloor$

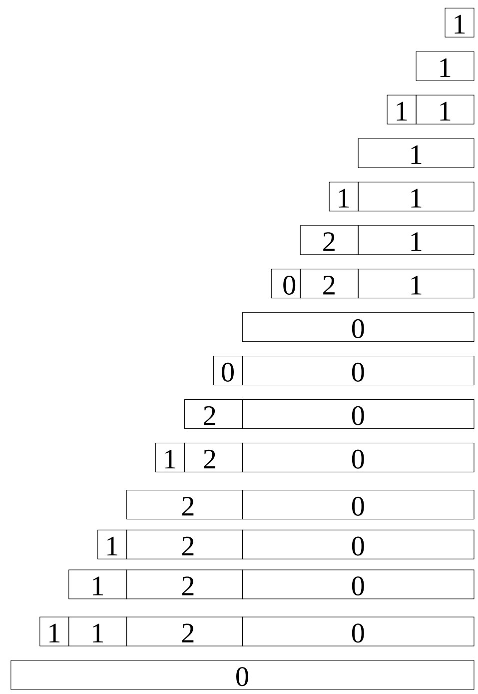
Let $C = \{0, 1\}$ the We think of each $x \in X$ as a function $x : C \to C$, making a carry bit to a later carry bit. In fact, x_i maps c_{i-1} to c_i. and let $\#$ be the operation defined by the following table:

$0\#0 = 0$	$0\#1 = 0$	$0\#2 = 0$
$1\#0 = 0$	$1\#1 = 1$	$1\#2 = 2$
$2\#0 = 2$	$2\#1 = 2$	$2\#2 = 2$

The operation $\#$ composes those functions. For example $x_7\#x_6\#x_5\#x_4$ maps c_3 to c_7.

0 1 1 0 0 1 1 0 0 0 1 0 1 1 0 1 u
0 0 0 1 1 0 1 0 1 0 1 1 0 0 1 0 v

0 1 1 1 1 1 2 0 1 0 2 1 1 1 1 1 x
0 1 1 2 0 2 1 1
0 2 0 1
0 0
0



0 2 2 2 2 2 2 0 0 0 2 1 1 1 1 1

0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 carry
0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 carry
0 1 1 0 0 1 1 0 0 0 1 0 1 1 0 1 u
0 0 0 1 1 0 1 0 1 0 1 1 0 0 1 0 v
1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 u+v

c