BU CLA CS 530: Analysis of Algorithms
Spring 1999

Selected Solutions for Problem Set 8

Problem 1. (Exercise 30.2-3, page 709, in [CLR])

An easy solution but not the best, for which you nevertheless get full credit, is to reduce the problem of computing the OR of n boolean values in the input array $A[0 \,.\,.\, n-1]$ to the problem of finding the maximum in Section 30.2 : For this reduction you change the boolean value **false** and **true** to the integers 0 and 1, respectively. The resulting algorithm is a n^2-processor CRCW algorithm that runs in constant time. A better solution is to solve the problem directly, as follows:

FAST-OR(A)
$n \leftarrow \mathit{length}[A]$
$\mathit{result} \leftarrow$ **false**
for $i \leftarrow 0$ **to** $n-1$ in parallel
    do **if** $A[i] =$ **true** **then** $\mathit{result} \leftarrow$ **true**
return result

This is a n-processor CRCW (in fact ERCW) algorithm that runs in constant time.

Problem 2.

(a) Below is a CREW algorithm to multiply two $n \times n$ matrices A and B; the algorithm computes an intermediary matrix D of dimension $n \times n \times n$ before computing the output matrix C.

CREW-MATRIX-MULTIPLICATION(A, B, C)
    $\triangleright$ Compute intermediary matrix D from input matrices A and B
1. **for** $i \leftarrow 1$ **to** n in parallel
2.   **do** **for** $j \leftarrow 1$ **to** n in parallel
3.   **do** **for** $k \leftarrow 1$ **to** n in parallel
4.   **do** $D[i, j, k] \leftarrow A[i, j] \times B[j, k]$

    $\triangleright$ Compute output matrix C from D
5. **for** $i \leftarrow 1$ **to** n in parallel
6.   **do** **for** $k \leftarrow 1$ **to** n in parallel
7.   **do** parallel prefix computation of the summation
8. $C[i, k] \leftarrow D[i, 1, k] + \ldots + D[i, n, k]$
9. **return** C

The computation of matrix D (lines 1 to 4) runs in constant time with n^3 processors. Moreover, this first part of the algorithm works in CREW mode, as concurrent-writes must be made if we want to keep the running time within a constant.

The computation of matrix C (lines 5 to 8) runs in $\mathcal{O}(\log n)$ time with n^3 processors again: There are 2 nested parallel-**for** loops, corresponding to n^2 summations of the form $D[i, 1, k] + \cdots + D[i, n, k]$ to be carried out, but each such summation requires time $\mathcal{O}(\log n)$ with n processors (if we use parallel prefix computation). This second part of the algorithm works in EREW mode.

(b) The powers $A^1, A^2, \ldots, A^n$ can be obtained as the products of prefixes of the n-component sequence $(A, A, \ldots, A)$. This can be accomplished in $\mathcal{O}(\log^2 n)$ time by $\mathcal{O}(n^4)$ processors, using a parallel prefix computation (Section 30.1.2 in [CLR]). The associative operation here is multiplication of $n \times n$ matrices, which can be carried out by a CREW algorithm in $\mathcal{O}(\log n)$ times using n^3 processors, by part (a). The parallel-prefix computation part of the algorithm can be carried out in EREW mode.

Another way of carrying out parallel prefix computation, not using a linked list (as in Section 30.1.2) is as follows. This is a general method, not dependent on the fact that the input is an n-component sequence of matrices $(A, A, \ldots, A)$, and so we use instead the sequence $(x_1, x_2, \ldots, x_n)$ as input. This is a recursive algorithm. For simplicity, assume n is a power of 2.

PARALLEL-PREFIX$(x_1, x_2, \ldots, x_n)$
1 **if** $n = 1$
2 **then** $y_1 \leftarrow x_1$
3 **else** **for** $i \leftarrow 1$ **to** $n/2$ in parallel
4 **do** $y_i \leftarrow x_{2i-1} \times x_{2i}$
5 $(z_1, \ldots, z_{n/2}) \leftarrow$PARALLEL-PREFIX$(y_1, \ldots, y_{n/2})$
6 **for** even $i \in \{1, \ldots, n\}$ in parallel
7 **do** $y_i \leftarrow z_{i/2}$
8 **for** odd $i \in \{1, \ldots, n\}$ in parallel
9 **do** $y_i \leftarrow z_{(i-1)/2} \times x_i$
10 **return** $(y_1, \ldots, y_n)$

If "$\times$" is standard integer multiplication, in lines 4 and 9, this algorithm uses at most $n/2$ processors in parallel (in each of the three parallel-**for** loops), each performing at most 1 integer multiplication. (In fact, it uses exactly $n/2$ processors at its first recursive call, $n/4$ processors at its second recursive call, $n/8$ processors at its third recursive call, etc.) As PARALLEL-PREFIX is called recursively $\log n$ times, it uses $n/2$ processors, each performing at most $\log n$ times a single integer multiplication. If each integer multiplication takes one time unit, the algorithm runs in $\mathcal{O}(\log n)$ time using $n/2$ processors on a EREW PRAM.

If $x_1, x_2, \ldots, x_n$ are $n \times n$ matrices, then "$\times$" takes $\mathcal{O}(\log n)$ time using n^3 processors on a CREW PRAM, by part (a) above. Hence, the whole algorithm can therefore be executed in $\mathcal{O}(\log^2 n)$ time using $\mathcal{O}(n^4)$ processors on a CREW PRAM.

Problem 3.

This is the parallel adaptation of the algorithms in Section 26.1 in [CLR]. Standard matrix multiplication is replaced by (min/+) matrix multiplication, where min and + correspond to + and $\times$, respectively, in the standard matrix multiplication. Algorithm EXTEND-SHORTEST-PATHS (top page 554, [CLR]) carries out a single (min/+) multiplication; its parallel version can be carried out in $\mathcal{O}(\log n)$ times using n^3 processors on a CREW PRAM, according to part (a) of Problem 2 above. Algorithm SLOW-ALL-PAIRS-SHORTEST-PATHS (page 555) computes the powers $W^1, W^2, \ldots, W^{n-1}$, based on (min/+) matrix multiplication; its parallel version can be carried in $\mathcal{O}(\log^2 n)$ time using $\mathcal{O}(n^4)$ processors on a CREW PRAM, by part (b) of Problem 2 above.

We can save a factor of n processors, i.e. reduce the number of processors to n^3, by observing that we do not need to compute all the matrices $W^1, W^2, \ldots, W^{n-1}$. We are interested only in matrix W^{n-1}. Hence, we can use algorithm FASTER-ALL-PAIRS-SHORTEST-PATHS (page 556), in its sequential version (i.e. we do not attempt to parallelize it), but every time we call EXTEND-SHORTEST-PATHS (line 5, page 556) we execute the latter on a CREW PRAM in $\mathcal{O}(\log n)$ time using n^3 processors. The **while**-loop in FASTER-ALL-PAIRS-SHORTEST-PATHS is executed $\mathcal{O}(\log n)$ times, resulting in $\mathcal{O}(\log^2 n)$ run time for the entire algorithm, using still n^3 processors.

Problem 4.

Associate a processor with each vertex in a tree, for a total of n processors. The processor associated with vertex v will calculate a pointer $next(v)$ to the successor of v in the preorder traversal. If v is not a leaf, then $next(v)$ is its left child. If v is a leaf, then $next(v)$ is the right sibling of $d(v)$, where $d(v)$ is the lowest ancestor of v (possibly v itself) which is not a right child. If $d(v)$ has no right sibling, then v is the last vertex in preorder and has no successor.

To compute the *next* pointers of leaf vertices in logarithmic time, use *pointer jumping*. Initially, each vertex v sets

$$d(v) \leftarrow \begin{cases} parent(v), & \text{if } v \text{ is the right child of its parent,} \\ v, & \text{if } v \text{ is not the right child of its parent or if } v \text{ is the root.} \end{cases}$$

Each vertex v then iterates the operation $d(v) \leftarrow d(d(v))$ $\log n$ times. This is done in parallel, and involves "concurrent-reads". At the end of this computation, $d(v)$ is the lowest ancestor of v which is not a right child. This first part of the algorithm is carried out in CREW mode. (You can be a bit more careful and run it in EREW mode, because for all distinct leaf vertices v_1 and v_2 we have $d(v_1) \neq d(v_2)$.) We now have a list *next* of all vertices in the correct order. It remains to compute the preorder number of v for each v — this is the second part of the algorithm. This can be solved by *pointer jumping* (as in "list ranking" or in "parallel prefix addition", Section 30.1 in [CLR]), and carried out in $\mathcal{O}(\log n)$ time on a EREW PRAM.

In conclusion, this two-part algorithm can bee run on a CREW PRAM (or, more carefully, a EREW PRAM) in $\mathcal{O}(\log n)$ time using n processors.

Assaf Kfoury
Created: 99.03.17