Theory of Parallel Hardware — March 4, 2004
Massachusetts Institute of Technology — 6.896
Charles Leiserson, Michael Bender, Bradley Kuszmaul — Handout 7

Solution Set 3

Due: In class on Wednesday, February 25. Starred problems are optional.

Problem 3-1. Show how to divide an N-bit number by the constant 3 in $O(\lg N)$ time using $O(N)$ hardware.

Solution: We are asked to divide an N-bit number by the constant 3 in $O(\lg N)$ time using $O(N)$ hardware. In class, we showed how to compute the quotient u/y in $O(\lg N)$ time, but not in $O(N)$ hardware. Since we already know the divisor y, however, we can precompute $1/y$ using $O(N)$ hardware, even if it takes more than $O(\lg N)$ time. Thus, we assume that we have precomputed the value $1/3$ for this problem; now all we have to do is multiply this value by the N-bit number.

We don't know how to perform $O(\lg N)$ multiplication using only $O(N)$ hardware, so we have to do something more clever. We observe that the binary representation of $1/3$ is the following:

$$\begin{aligned} & 0.01010101\ldots \\ = \; & 0.01 + 0.0001 + 0.0000001\ldots \end{aligned} \tag{1}$$

Thus, multiplying an $N - bit$ number by $1/3$ is equivalent to repeatedly adding the number to itself after right-shifting it by 2 bits. Given an $N - bit$ number $X = x_1x_2 \ldots x_N$, the value $X/3$ can be expressed as follows:

$$\begin{array}{ll} & x_1x_2 \ldots x_{N-2}.x_{N-1}x_N \\ + & x_1x_2 \ldots x_{N-4}.x_{N-3}x_{N-2}x_{N-1}x_N \\ + & x_1x_2 \ldots x_{N-6}.x_{N-5}x_{N-4}x_{N-3}x_{N-2}x_{N-1}x_N \\ \ldots & \end{array}$$

Note that we only need to carry out this sum to the desired precision. We perform the sum in $O(\lg N)$ time using carry-save addition in a Wallace tree configuration. However, we avoid using $O(N^2)$ hardware by recognizing that the sums of separate subtrees are identical in our case, since we are essentially adding the same number (shifted by a certain number of bits) each time. We thus use $O(N)$ hardware by looping back our result for each subsequent subtree calculation. Figure **??** shows a high-level view of how this might work.

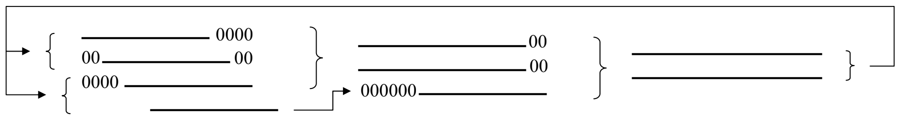

Figure 1: An $O(N)$ Wallace tree circuit for dividing an N-bit number by the constant 3.

The carry-save adders used in the above figure can be made to be $2N$-bits long to simplify the zero-padding. We basically start with lines 1 to 4 of the addition, reduce that to two lines (after two iterations), and then loop back two copies of the result to the beginning. Note that the second copy of the result has to be shifted from the first by 2 bits, since it represents the subtree computation of lines 5 to 8 of the addition. The resulting circuit has a constant number of carry-save adders, and some logic to perform the bit-shifting, so we use a total of $O(N)$ hardware. The computation is just a modified Wallace tree addition, so it completes in $O(\lg N)$ time (as shown in class).

Problem 3-2. A spanning tree of a graph is the tree formed by a subset of the edges of the graph such that all vertices in the graph are contained in the tree. A minimum spanning tree of an edge-weighted graph is a spanning tree of minimum weight, where the weight of the tree is defined to be the sum of the weights of the edges in the tree. Argue that a minimum spanning tree for a graph with N vertices can be computed in $O(N)$ time on an $N \times N$ mesh. You may assume that the graph is given as an $N \times N$ adjacency matrix of edge weights, and that all edge weights are distinct.

Solution: I will assume that all edges have distinct costs (tie-breaking is in principle easy to achieve, by associating a unique label based on the row and column to each weight). Finding a MST can be reduced to computing the shortest paths between any pair of verices, using $\max(\cdot, \cdot)$ as the path composition operation. That is, we want to find the path between any (u, v) which minimizes the maximum weight along the path. Assume the maximum weight along such a path is w_m. If there is an edge from u to v, this edge is in the MST iff its weight w is equal to w_m (that is, the direct edge is the path minimizing the maximum weight). It is easy to prove this fact:

- clearly, w cannot be smaller than w_m because the direct edge is a possible path.
- if $w = w_m$, it means any other path contains an edge with weight greater than w. Assume now that some MST did not contain the edge (u, v). This spanning tree must contain a path from u to v, and this path must contain an edge with weight greater than w. Then, we can replace this edge with (u, v) and get a better MST (removing that edge, and adding (u, v) leaves $n - 1$ edges and no cycles, so we still have a spanning tree).
- finally, assume $w > w_m$. Then there is a path from u to v where all edges have weight less than w. Assume a MST contained the edge (u, v). Then one edge from the optimal path $u \rightsquigarrow v$ must not be in the tree (otherwise, there would be a cycle). Add that edge, and remove (u, v), to obtain a better MST.

So we only have to compote all-pairs shortest paths under the max composition, and then the MST can be determined by just examining the local data in each processor (the weight of the shortest path, and the weight of the direct edge). But this is virtually identical to the transitive closure algorithm from class – at step k, we shortcut node k, by computing:

$$(\forall) i, j : \quad a_{ij}^{(k)} \leftarrow \min\left(a_{ij}^{(k-1)}, \max(a_{ik}^{k-1}, a_{kj}^{k-1})\right)$$

The proof that this works is by induction, virtually identical to the transitive closure case, and appears also in CLRS.