15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Sets and Sequences I **Date:** September 11, 2007
Scribe: Michael Sanphy

5.1 Prefix Sums / Scan

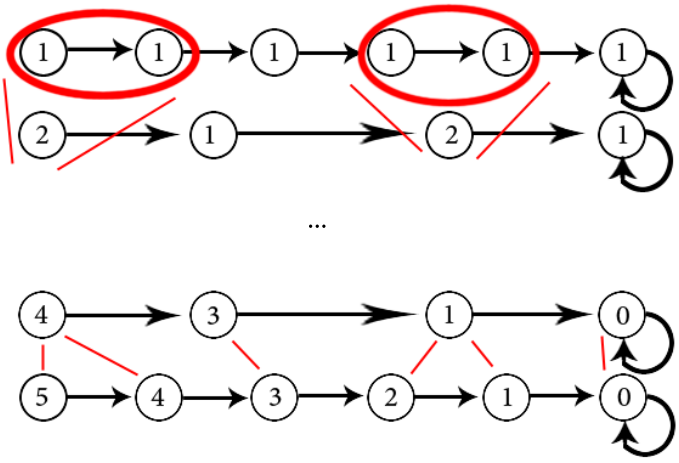
For $a_1, a_2, \ldots, a_n$ and a binary associative operator $\oplus$ with identity I, calculate the sequence: $I, a_1, a_1 \oplus a_2, a_1 \oplus a_2 \oplus a_3, \ldots, a_1 \oplus \cdots \oplus a_{n-1}$

As we saw last lecture, this can be performed on an array with Contraction, by pairing up elements to be combined and then recursively solving. This solution is then used to expand out to the full solution.

However, this method runs into problems when used on a linked list. There is no internal ordering (like the indices of the array) to differentiate the "odd" and "even" elements of the list. Without this, we can't pair them up to do combining.

We're looking for an algorithm to do this in linear work (we saw pointer jumping could accomplish this, but not in linear work).

- We would like to combine neighbors, but the symmetry of the list prevents this.
- The solution here is symmetry breaking—"random mate"



- Flip a coin for each node.
- Follow the rule "if you are heads and you're pointing to a tail, then combine with what you're pointing to" for each node.

- Recursively solve on resultant list.
- Project this solution back to the original list as in the array version.

This algorithm accomplishes Scan in linear work. Without a formal argument, $\frac{1}{2}$ chance of a node being heads $\times$ $\frac{1}{2}$ chance of the next node being tails $= \frac{1}{4}$ chance of 'removal' for each node. This means, in expectation, $\frac{n}{4}$ nodes are removed per iteration.

It can also be shown that this has with high probability $D(n) = O(\log n)$.

$W(n) = W(\frac{3}{4}n) + cn = O(n)$

5.2 Pack operation

Input: arrays A and B
$A = [a, b, c, d, e, f, g]$
$B = [1, 0, 1, 1, 0, 1, 0]$

Output: $[a, c, d, f]$

The Pack operation takes two arrays—one of some data type, the other of true/false values. The output is an array made from the initial array that only contains the elements which correspond to true values.

We can use a plus Scan to implement the Pack operation.

plusScan $(B) = S = [0, 1, 1, 2, 3, 3, 4]$

$A = [a, b, c, d, e, f, g]$
$B = [1, 0, 1, 1, 0, 1, 0]$
$S = [0, 1, 1, 2, 3, 3, 4]$
$R = [a, c, d, f]$

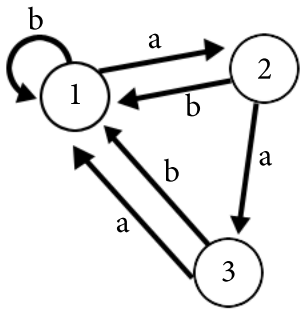
For each element that corresponds to true, the plusScan gives us its index in the output array. This allows us to perform the Pack in parallel.

5.3 Finite State Machine

Input: $[a, a, a, b, a, a, b, b, a, a]$
Output: $[2, 3, 1, 1, 2, 3, 1, 1, 2, 3]$

Here we want to output a list of the states the FSM is in after each element of the input string is entered. Once again we can use Scan to parallelize this computation. We start by forming the transition matrices for a and b.

a:
$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$



b:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

All we need to do to compute the list of states is multiply the transition matrices that correspond to the elements in the input string as our binary operator for Scan. This gives us an array of matrices, and to produce the final output we simply look up the start node in each matrix to get the end state for each element.

5.4 Linear Recurrences

$x_i = a_i x_{i-1} + b_i$

Linear recurrences such as this can be their own operator, allowing you to generate a whole sequence in parallel.

Scan can be used to make many "sequential" operations parallel by phrasing them as some type of binary associative operator.

5.5 Removing Duplicates Using Hashing

Input: $[a, b, c, a, d, a, b]$
Output: $[a, b, c, d]$

Here we assume our model is that of Concurrent Writes (leading to arbitrary combining). We also assume that we have a hash function and equality function over the data type in the array.

- All elements are hashed and written to the hash table at once.
- They then read back if their value is in the place they wrote.
- If so, resolve multiple collisions of the same value by writing their array indices at the same time and reading that value back. Only one element wins, and that stays in the hash table.

- Continue this process in rounds until all elements have been written to the hash table.
- Read back the elements, of which there are now no duplicates.