15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Propagating State **Date:** January 27, 2009
Scribe: Sameer Chopra

5.1 Parallel Prefix

Here is a pseudocode for the computing the prefix sum of array b in to a:

```
a[0] = 0;
for (i = 1; i < n; i++)
a[i] = a[i-1]+b[i-1]
```

For example, if b = [6, 7, 8, 9, 2, 0, 3], then the prefix a is [0, 6, 13, 21, 30, 32, 32, 35]
How can we parallelize this when each element is dependent on the previous element?
Use contraction! (see previous lecture) This takes logarithmic depth and linear work.
In this class, we will see how we can generalize this.
NOTE: Though we have be talking about prefix sum, these techniques can also be used even if the plus operator is replaced with any other binary associative operator.

5.2 Pack

The Pack operation can be described as follows: given an array of values and an array of flags telling us which values to keep, it outputs an array consisting of only those values whose corresponding flags have been set to true. For example:
input:

```
[a, b, c, d, e, f, g]
[t, f, f, t, f, t, t]
```

gives the output:

```
[a, d, f, g]
```

This is useful in quicksort since to find all elements less/greater than pivot, we compare all the elements to the pivot to generate the flags and then use pack to separate the array.

Claim: Pack can be done in log(n) depth and n work.
This is easy to do sequentially; we just walk down the list, and each time we see a true flag, move that element into the next position in the output array. How can we do this in parallel?

We really just want to figure out what position an input element will be in the output array. First, we can convert trues and falses as 1's and 0's respectively (this is implicit in languages like C). Then we can count up how many 1's come before any given position; that tells us how many elements are in front of it in the output array, which gives the position in the output array. These values can be obtained by running Parallel Prefix on the converted flag array (which now has 1's and 0's).
Lets look at our example above. The flag array is `[1, 0, 0, 1, 0, 1, 1]`; whose prefix sum is `P=[0, 1, 1, 1, 2, 2, 3]`.
We can also write to the output array in parallel, since we know where every element needs to go. We only bother to write element i to the output array if `P[i]==P[i+1]`, in which case we write the element to `P[i]`-th position in the output array. Since this makes sure that every output array location is written to only once, there are no race conditions.
The interesting work here is done by parallel prefix sum; the only other steps which are converting the flag array (already implicit in some languages and which can be done in constant depth and linear work otherwise) and parallelly writing the output (which is also done in constant depth and linear work).

5.3 Function Composition

Here is the pseudocode for function composition:
`for(i = 1; i < n; i++)`
a`[i] =` f_i`(`a`[i-1])`

This is a generalization of parallel prefix (which is just function composition with $f_i(x) = x + b$[i-1]).
Lets see what the elements of our output array are:
a[1] = $f_1(a[0])$
a[2] = $f_2(a[1]) = f_2(f_1(a[0]))$
a[3] = $f_3(f_2(f_1(a[0])))$
$= f_3 \circ f_2 \circ f_1(a[0])$.
So, if the composition of the functions can be generated somehow, that could just be applied to the initial value to get the value we want. But how can we compose efficiently in parallel? Well, function composition is associative, so we can use the parallel prefix sum algorithm! This would generate a list of all the composed functions, and we can just call each one on a[0].
However, there are 2 problems:

1. it is not obvious that the work and depth are constant time
2. this only works when the f_is have efficient composition

NEXT CLASS: We'll look at some examples where this works and where it doesn't work.