15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Sets and Sequences II **Date:** Thursday January 29, 2009
Scribe: Michael Rule

5.1 Efficient Function Composition

In the previous lecture we saw how to perform a prefix-sum operation using contraction. This procedure can be viewed as a special case of function composition. The general case seems inherently sequential, but can sometimes be converted to a parallel procedure if certain conditions are met.

Since every iteration generates the state for the next iteration, this seem like an inherently linear algorithm that is not trivial to parallelise like quicksort. Last class we talked about how prefix-sum could be parallelized by the contraction method. We started to generalise this to arbitrary function composition where a given element :

$$X_i = F_i(F_{i-1}(F_{i-2}(...F_0(X_0)...))) \tag{5.1.1}$$

That is, we first apply F_0, then F_1 then F_2, F_3 all the way up to F_{i-1} and finally F_i

We could calculate each X_i directly if we could generate this composition and apply it in parallel. In NESL these would be something like

$$A = [I, F_0 \circ I, F_1 \circ F_0 \circ I, F_2 \circ F_1 \circ F_0 \circ I, ..., F_{n-1} \circ ... \circ F0] \tag{5.1.2}$$

$$f(x_0) : finA \tag{5.1.3}$$

Examining the case of prefix sum for array $[3, 1, 7, 2, 6, \dots]$, our functions F_i are :

$$F_i = [+3, +1, +7, +2, +6, ...] \tag{5.1.4}$$

Composing Add is pretty trivial

$$A = [I, +3, +3+1, +3+1+7, +3+1+7+2, +3+1+7+2+6, ...] \tag{5.1.5}$$

$$A = [I, +3, +4, +11, +13, +19, ...] \tag{5.1.6}$$

This is similar to the prefix sum described in the previous class. In this case function composition is trivial. In general it may not be this easy. An inefficient method of performing function composition would just be to store the composition to be evaluated later. In order for us to be able to efficiently solve this problem in parallel, we must have a way to efficiently do function composition. We present three types of techniques for efficiently composing functions :

- State Machine Transitions
- Carry Propagation
- Linear Recurrences

5.1.1 State Machine Transitions

If we are able to represent a seemingly linear sequence of operations as a series of transitions on a state machine, we parallelize this linear procedure by composing the state machine transition functions.

Example : Say we have a state machine with 3 states [1, 2, 3] and two transition functions [a, b]. :

States : [1,2,3]

Transitions :
a : state $\rightarrow$ (state + 1) mod 3
b : state $\rightarrow$ state

It turns out that its easy to represent state machines as functions. We describe a state machine as a function from input state to output state.

$$F_a = 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1 \tag{5.1.7}$$

$$F_b = 1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3 \tag{5.1.8}$$

Consider a sequence of transitions, a string in [a,b], applied to this state machine from some initial state. We want to find the resultant state. We could do this linearly using function composition, but it is possible to compose the state transition functions efficiently.

Some state transition function composition examples :

$$F_a \circ F_a = 1 \rightarrow 3, 2 \rightarrow 1, 3 \rightarrow 2 \tag{5.1.9}$$

$$F_a \circ (F_a \circ F_a) = F_b \tag{5.1.10}$$

Notice that the description of the state transition function is compact, and can be calculated quite easily. The output is always some sort of permutation or map from states to states. If we have some constant number of K states, we can compose two function in K steps. This can be done with table lookup very efficiently. If you have a small number of possible transitions N, you can store a NxN table which stores the new transition formed by compositing two transitions.You can apply the same contraction method, discussed two lectures ago, to compute all the state transition compositions.

This procedure can be used to construct a parallel algorithm with linear work and log(n) depth which can figure out the state after any sequence of transitions. Carry propagation is one application of this technique.

5.1.2 Carry Propagation

For simplicity consider adding together two binary numbers. This procedure generalizes to any base, but binary provides a more clear example.

Consider adding two binary number A and B where

$$A = 1010011010 \tag{5.1.11}$$
$$B = 1101101000 \tag{5.1.12}$$

We break convention and write the number with the least significant byte first. This is so we can represent the algorithms as scanning form left to right, which is consistent with previous examples. A linear procedure for adding these two number would produce output like this :

$$Carry = 0100000100 \tag{5.1.13}$$
$$A = 1010011010 \tag{5.1.14}$$
$$B = 1101101000 \tag{5.1.15}$$
$$A + B = 0000001110 \tag{5.1.16}$$

Because of the dependence of higher order bits on carries from lower order bit, this seems to be an inherently sequential algorithm. However, it can be represented as a very simple state machine :
States (0 : no carry, 1 : carry)
Transitions

$$a: \text{ clear carry } (A_i = 0, B_i = 0) \tag{5.1.17}$$
$$b: \text{ preserve carry } (A_i = 0, B_i = 1 OR A_i = 1, B_i = 0) \tag{5.1.18}$$
$$c: \text{ create carry } (A_i = 1, B_i = 1) \tag{5.1.19}$$

$$statetransitions = cbbbbbcaba \tag{5.1.20}$$
$$A = 1010011010 \tag{5.1.21}$$
$$B = 1101101000 \tag{5.1.22}$$

State transition functions :

$$F_a = 0 \to 0, 1 \to 0 \tag{5.1.23}$$
$$F_b = 0 \to 0, 1 \to 1 \tag{5.1.24}$$
$$F_c = 0 \to 1, 1 \to 1 \tag{5.1.25}$$

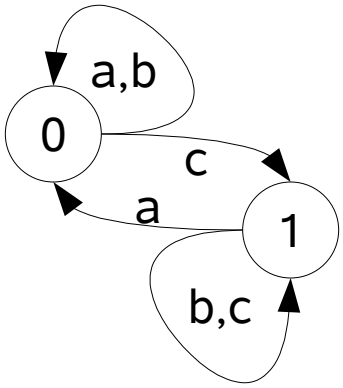

Figure 5.1.1: state machine for carry propagation

These can now be easily composed e.g.

$$F_b \circ F_b \circ F_b \circ Fb = F_b \tag{5.1.26}$$
$$F_b \circ F_b \circ F_a = F_a \tag{5.1.27}$$
$$F_b \circ F_b \circ F_c = F_c \tag{5.1.28}$$
$$F_c \circ F_b \circ F_b \circ F_b \circ F_a = F_c \tag{5.1.29}$$

We can therefore easily compose these things, and apply a prefix sum algorithm on the state transition function array. This means we can do carry propagation in linear work and in parallel.

$$W(n) = W(n/2) + O(n) = O(n) \tag{5.1.30}$$

Try to think about converting a linear procedure into a state machine. More generally, try to see if the functions in question can be composed efficiently. Linear Recurrences are an example of something that is hard to phrase as a state machine, but still has a efficient method of function composition which makes it parallelizable.

5.1.3 Linear Recurrences

Example consider a loop

```
for i = 0 ..  n-1
```
$X_{i+1} = a_i + b_i \cdot x_i$

This is an example of a linear recurrence. Since each X_{i+1} depends only on X_i this is a first order recurrence.

our function :

$$f_i(x) = a_i + b_i \cdot x \tag{5.1.31}$$

Efficient composition : composition itself is fast (constant time) complexity of resultant function does not grow. Linear functions can be easily composed :

$$f_1 = a_1 + b_1 \cdot x \tag{5.1.32}$$
$$f_2 = a_2 + b_2 \cdot x \tag{5.1.33}$$

$$f_2 \circ f_1(x) = a_2 + b_2 \cdot (a_1 + b_1 \cdot x) \tag{5.1.34}$$
$$f_2 \circ f_1(x) = a_2 + b_2 \cdot a_1 + b_2 \cdot b_1 \cdot x \tag{5.1.35}$$
$$f_2 \circ f_1(x) = (a_2 + b_2 \cdot a_1) + b_2 \cdot b_1 \cdot x \tag{5.1.36}$$

$$f_2 \circ f_1(x) = f'(x) = a' + b' \cdot x \tag{5.1.37}$$
$$a' = a_2 + b_2 \cdot a_1 \tag{5.1.38}$$
$$b' = b_2 \cdot b_1 \tag{5.1.39}$$

So, in one $+$ and two $\cdot$ we can compose two of these linear functions. we can use the prefix-sum algorithm to parallelize the loop for log(n) depth

We can generalize recurrences :

$+ \rightarrow$ any semi-associative operator

In general if you can create, in constant time, a representation of the composition of two function that is not larger than some bounded value, and who's evaluation is bounded, you can do this in linear work and logarithmic depth and linear work.

5.2 Prefix Sums on a List

So far : prefix sums on an array input (see two lectures ago)

[3 2 1 1 2 3 1]

[0 3 5 6 7 9 12]

But, what if we are given this input as a linked list ?

$$HEAD \rightarrow (3) \rightarrow (2) \rightarrow (1) \rightarrow (1) \rightarrow (2) \rightarrow (3) \rightarrow (1) \rightarrow NULL \tag{5.2.40}$$

Prefix sum (linear loop version)

- $x = 0$
- $p =$ head

- $while($ node ! = NULL)
  - $p \rightarrow$ val $= x$
  - $x = f(p \rightarrow$ value , $x)$
  - $p = p \rightarrow$ next

Result :

$$HEAD \rightarrow (0) \rightarrow (3) \rightarrow (5) \rightarrow (6) \rightarrow (7) \rightarrow (9) \rightarrow (12) \rightarrow NULL \tag{5.2.41}$$

This seems much trickier to do in parallel. It turns out that everything we could do in arrays we can do in Linked Lists. Caveat : in order to make this result work, we have to already have a handle to all list elements ahead of time. However, these handles do not need to be in order. When you do problem 3 in the homework assignment you will see what this means.

How can we calculate the prefix sum of a linked list, given that we have a handle to each list element, in linear work and logarithmic depth ? We are going to need a randomized algorithm, to break the symmetry, the fact that all list elements look the same. Use Contraction and Symmetry Breaking In the array version we paired up elements, applied the sum operation to generate a smaller list. We apply the symmetry breaking algorithm from 4 lectures ago to find, with high probability, a good sized maximally independent subset.

Random Mating :

- each node flips a coin
- if you are a head and you point to a tail then short-cut over the next node
- otherwise do nothing

short cut :

- if $(A : x) \rightarrow (B : y) \rightarrow (C : z)$, A short-cuts to C by performing two operations :
- $A \rightarrow value = A \rightarrow value + A \rightarrow next \rightarrow value$
- $A \rightarrow next = A \rightarrow next \rightarrow next$
- This alters the list configuration to :
- $(A : x + y) \rightarrow (C : z)$

Example :

$$(3) : h \rightarrow (2) : [h] \rightarrow (1) : t \rightarrow (1) : [h] \rightarrow (2) : t \rightarrow (3) : t \rightarrow (1) : h \rightarrow NULL \tag{5.2.42}$$

$$(3) \rightarrow (3) - - - - - - \rightarrow (3) - - - - - - - - \rightarrow (3) \rightarrow (1) \rightarrow NULL \tag{5.2.43}$$

$$(3) \rightarrow (3) \rightarrow (3) - (3) \rightarrow (1) \rightarrow NULL \tag{5.2.44}$$

Assume we recursively solve this to generate the following prefix solution :

$$(0) \rightarrow (3) \rightarrow (6) - (9) \rightarrow (12) \rightarrow NULL \tag{5.2.45}$$

How do we expand this recursive solution out to out desired answer ? Are the numbers written in the recursive solution the correct answers for those nodes ? Yes, so the only missing answers are those of the nodes that were short-cut out. Assuming that we still remember the original nexts (short-cut perhaps generated a new list, so that the old one remains intact) Splice : add the original node value back to the recursively solved prefix sum value to generate the prefix sum value for the next node. To analyze this in a handwavy manner (stricter analysis of randomized algorithms in later lecture)

probability of $H \rightarrow T$ is 0.25, so on average we expect to remove 1/4 of the elements on each iteration, so our work set is geometrically decreasing.

Informally :

$$E(W(n)) = E(W(3/4n)) + O(n) = O(n) \tag{5.2.46}$$

$$E(W(n)) = \text{ Expectation of the amount of work for N} \tag{5.2.47}$$

$$E(W(3/4n)) = \text{ Expectation of the amount of work from} \tag{5.2.48}$$

This approach will be useful in several homework problems. This solution uses linear work. There is an easier method that uses $O(n \cdot log(n))$ work using point jumping, which will be sufficient for several homework problems.

repeat ceil(log(n)) times :

- $if(p! = null)$
- – $p \rightarrow val = p \rightarrow val + (p \rightarrow next) \rightarrow val$
  – $p \rightarrow next = (p \rightarrow next) \rightarrow next$

Similar to previously mentioned algorithm, excerpt short-cut is unconditional and performed on every node. Repeat until no more non-nulls exist.