Recitation 3 — More Scan

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

1st February 2012

1 Announcements

- HW1 will hopefully be graded and returned after tomorrow's lecture.
- HW2 has been out. It's due at 23:59 EST on Tuesday 7th February, with late days as stated in the policy.
- Questions from lecture, homework, or life?

2 Scan

We covered the implimentation of the function `scan` in lecture yesterday. We'll spend most of today revistiting this implimentation and showing off some unexpected applications of it.

2.1 Definition

Recall:

`scan` has type

$$\texttt{scan} : (\alpha * \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq} * \alpha)$$

Informally, scan computes both the reduction of the sequence using the supplied operator and the sequence of all the partial results that were computed along the way.

`scan`, like `reduce`, is highly parallel. `reduce` and `scan` both have work in $O(n)$ and span in $O(\lg n)$.

While the work of `scan` may seem obvious, getting the span to be $O(\lg n)$ is a little harder to discern.

2.2 Implementation

Recall the implementation from yesterday's lecture:

```
% implements: the Scan problem on sequences that have a power of 2 lenngth
fun scanPow2 $f$ $i$ $s$ =
   case $|s|$ of
      $0 \Rightarrow (\langle\rangle, i)$
    | $1 \Rightarrow (\langle i\rangle, f(i, s[0]))$
    | $n \Rightarrow$
        let
          val $s' = \langle f(s[2i], s[2i+1]) : 0 \le i < n/2\rangle$
          val $(r, t) = \texttt{scanPow2}\ f\ i\ s'$
        in
          $(\langle p_i : 0 \le i < n\rangle, t)$, where $p_i = \begin{cases} r[i/2] & \textbf{if } \texttt{even}(i) \\ f(r[i/2], s[i-1]) & \textbf{otherwise}. \end{cases}$
        end
```

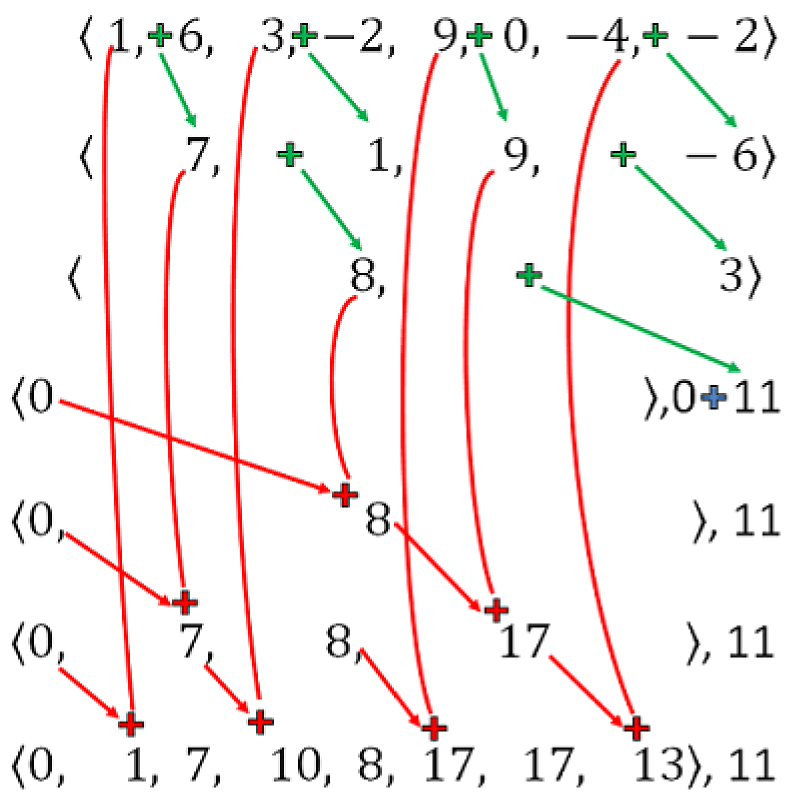
Rememeber that the crux of this problem is that we can pairwise apply the function to the elements in the sequence to generate a sequence consisting of every other element of the final output. Performing this procedure recursivelly and using the results of the subcalls allows us to generate the final output.

We will now see a well done animation demonstrating this contraction algorithm.

Below is a diagram illustrates this animation:

If s $= \langle 1, 6, 3, -2, 9, 0, -4\rangle$, then

`scan op+ 0 s` yields the following.



We can write recurrences for the above implimentation to check the work and span.

$$\begin{aligned} W(n) &= W(n/2) + O(n) \\ S(n) &= W(n/2) + O(1) \end{aligned}$$

Using the brick method from last week, we see the work at each level is $O(n/2^i)$. Since work is geometrically decreasing at each level, the recurrence is root dominated. Thus $W(n)$ solves to $O(n)$. For the span, we see that there are $\lg n$ levels each with constant cost. Thus $S(n)$ solves to $O(\lg n)$.

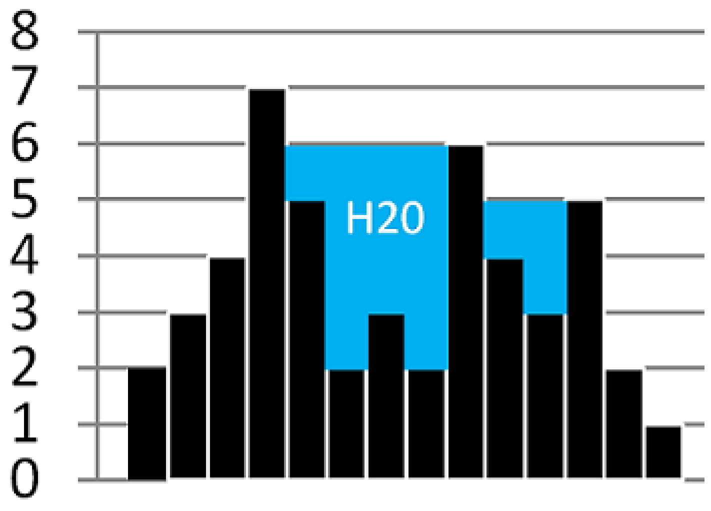
2.3 Example Uses of Scan

At first glance, `scan` seems not to offer much that isn't already available through `reduce`. With clever choices of associative functions, though, `scan` can be used to compute some surprising things efficiently in parallel.

2.3.1 Histogram

Consider the following problem:

Given a histogram, if we were to pour water over it, how much water (in terms of area) would it hold? For simplicity we will represent a histogram as a sequence of integers. For example the histogram shown below is represenented by the sequence $\mathtt{s} = \langle 2, 3, 4, 7, 5, 2, 3, 2, 6, 4, 3, 5, 2, 1 \rangle$, and holds 15 units of water.



Any ideas on how we might solve this problem?

The idea is to single out one bar b_i. If we know the maximum of the bar heights to the left of b_i ($maxl$) and the maximum of the bar heights to the right of b_i ($maxr$), given that $maxl > height(b_i)$ and $maxr > height(b_i)$ then the water b_i will hold above it is $MIN(maxl, maxl) - height(b_i)$.

Do we know of any functions that could be useful for generating these sequences of max-bar-heights? How about `scan`. Using a few `scan`'s, a `map` and a `reduce`, this problem becomes very simple.

```
open Seq

(* W = O(n),  S = O(1) *)
fun reverseSeq (s : 'a seq) =
    let val len = length hist
    in tabulate (fn i => nth (len - i - 1) s) len end

(* W = O(n),  S = O(log n) *)
fun collectwaterhist (hist : int seq) =
    let
        val len = length hist
        val SOME MIN = Int.minInt

        (* computes sequence of max value to the left of index i *)
        val (leftmaxseq, _) = scan Int.max MIN hist

        (* computes sequence of max value to the right of index i *)
        val (rmseq_rev, _) = scan Int.max MIN (reverseSeq hist)
        val rightmaxseq = reverseSeq rmseq_rev

        (* sequence of (maxvalue to the left,
                        bar height,
                        max value to the right) *)
        val lvrseq = tabulate (fn i => (nth i leftmaxseq,
                                        nth i hist,
                                        nth i rightmaxseq))
                              len

        (* mapping function: returns the amount that
           a specific bar would collect above it*)
        fun collect (maxl, barheight, maxr) =
            Int.max (Int.min (maxl, maxr) - barheight, 0)
    in
        reduce op+ 0 (map collect lvrsq)
    end
```

2.3.2 Computing Fibbonacci Numbers

With a carefully chosen matrix, we can use `scan` to compute the Fibonnaci numbers. In the extremely unlikely event that you've forgotten, the Fibbonacci numbers are defined as follows:

Definition: The Fibbonacci numbers are an integer sequence given by the following recurrence[1]

- $F_{-1} = 1$
- $F_0 = 0$
- $F_1 = 1$

[1]It is slightly contrived, but harmless, to define the -1^{st} element of the Fibbonacci sequence. The other base cases are such that the recursive case will never use it, so this could be any constant and produce the same sequence of integers. This one happens to make the proof work, though.

- $F_n = F_{n-1} + F_{n-2}$

We make the following claim about this definition, which we will prove by induction:

Claim:

For all natural numbers n,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Proof: We'll prove this by induction on n.
Base Case: $n = 0$

Any $n \times n$ matrix to the zero power is the $n \times n$ identity matrix, so

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} F_1 & F_0 \\ F_0 & F_{-1} \end{pmatrix}$$

which is exactly as desired.

Inductive Case:

Assume that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

We want to show that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

It suffices to show that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

Recall matrix multiplication, specifically in the case of taking the product of two 2×2 matrices:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Therefore,

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} &= \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} \\ &= \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \\ &= \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \end{aligned}$$

This is exactly as desired and concludes the proof.

Remember that matrix multiplication is an associative operation on square matrices. We'll only need 2×2 `int` matrices, so for simplicity let's represent them as values of type `int * int * int * int`.

The above proof means that we can compute the Fibbonacci numbers by applying scan to a matrix multiplication function:

```
functor FibboScan (S : SEQUENCE) : FIBBO =
struct
  structure Seq = S
  open Seq

  (* very simple representation of 2x2 matrices *)
  fun mmult ((a,b,c,d),(e,f,g,h)) = (a*e + b*g, c*e + d*g,
                                     a*f + b*h, c*f + d*h)

  (* returns the first n fibbonacci numbers *)
  fun fib n =
      let
        val s = tabulate (fn _ => (1,1,1,0)) (n+1)
        val (ans,_) = scan mmult (1,0,0,1) s
      in
        map (fn (_,_,_,x) => x) (drop(ans,1))
      end
end
```

Note that we have to produce $n+1$ terms of our version of the Fibbonacci sequence and discard the first. This exactly corresponds to the choice we made to make the base case is correct.

Since the matrices are of a constant 2×2 size, the matrix multiplication is actually a constant time operation—we're really just doing a handful of integer additions and multiplications. That means that we can compute n Fibbonacci numbers with total work in $O(n)$ and span $O(\lg n)$.

3 Leaps and Bounds

Last week, we saw the recurrence of $W(n) = 2W(n/2) + \Theta(n)$ with $W(1) = 1$ This means $W(n) = 2W(n/2) + O(n)$ *and* $W(n) = 2W(n/2) + \Omega(n)$, so by definition, we know that there are constants α and β such that

$$W(n) \le \begin{cases} 2W(n/2) + \alpha n & n > 1 \\ 1 & n = 1 \end{cases} \qquad \text{and} \qquad W(n) \ge \begin{cases} 2W(n/2) + \beta n & n > 1 \\ 1 & n = 1. \end{cases}$$

We then used to the substitution method to show that $W(n) = O(n \lg n)$. What if we want want to show a Θ bound? Today we will show this recurrence solves to $W(n) = \Theta(n \lg n)$ by proving the lower bound Ω. The proof generally follows the same structure as showing big-O, but we need to reverse the direction of inequalities and work with a different set of constants.

Theorem: if

$$W(n) \ge \begin{cases} 2W(n/2) + \beta n & n > 1 \\ 1 & n = 1 \end{cases}$$

then for some k_1 and k_2, $W(n) \ge k_1 n \lg n + k_2$.

Proof

Base Case ($n = 1$): for the inequality to hold, the only restrictions on the constants is that $k_2 \geq 1$. So let's let $k_1 = \beta$ and $k_2 = 1$.

IH: The theorem holds for length less than n.

Inductive Case: We want to show $W(n) \geq k_1 n \lg n + k_2$, but we know:

$$W(n) \geq 2W(n/2) + \beta n \tag{1}$$

$$W(n/2) \geq k_1(n/2)\lg(n/2) + k_2 \tag{2}$$

Substitute (2) into (1):

$$\begin{aligned} W(n) &\geq 2\Big(k_1(\tfrac{n}{2})\lg(\tfrac{n}{2}) + k_2\Big) + \beta n \\ &= nk_1(\lg n - 1) + 2k_2 + \beta n \\ &= (k_1 n \lg n + k_2) + \underbrace{\beta n - k_1 n + k_2}_{\geq 0} \end{aligned}$$

The leftover $\beta n - k_1 n + k_2$ has to be greater than or equal to 0. This will be true with the constants we picked above. Check to make sure you know why. Thus the theorem holds and $W(n) = \Omega(n \lg n)$.

Because we proved $W(n) = O(n \lg n)$ and $W(n) = \Omega(n \lg n)$, we can conclude $W(n) = \Theta(n \lg n)$.