Lecture 4 Notes (February 18, 2025)

Sam Westrick

`s.westrick@nyu.edu`

Filtering

In last lecture, we discussed `scan`, a primitive for parallel prefix sums. An immediate application of this function is to efficiently filter a sequence in parallel, removing some elements which do not satisfy a predicate.

`filter: ('a -> bool) -> 'a seq -> 'a seq`

(For example:) `filter is_even` $\langle 1,1,2,3,3,4,4,5,6,7\rangle = \langle 2,4,4,6\rangle$

The challenge is how to implement `filter` with $O(n)$ work and $O(\text{polylog}(n))$ span for an input of length n, and assuming the predicate costs $O(1)$.

A solution in terms of `scan` is as follows. The idea is to compute, for every element, how many elements to the left of that element satisfy the predicate. This is the element's offset in the output. We can compute these offsets by mapping each element to 1 or 0 (depending on whether or not it satisfied the predicate) and then perform a plus-scan. We can then do one more pass over the data and write out the elements which satisfy the predicate.

```
fun filter p s =
  let
    val n = Seq.length s
    val offsets = Parallel.scan (op+) 0 (0, n) (fn i =>
      if p (Seq.nth s i) then 1 else 0)
    val output = ForkJoin.alloc (Seq.nth offsets n)
  in
    Parallel.parfor (0, n) (fn i =>
      if Seq.nth offsets i <> Seq.nth offsets (i+1) then
        (* If two adjacent offset values differ, then we know
         * this element must have satisfied the predicate.
         * Alternatively, we could just evaluate the predicate
         * a second time.
         *)
        Array.update (output, Seq.nth offsets i, Seq.nth s i)
      else
        ());
    Seq.fromArray output
  end
```

On the previous example, `filter is_even` $\langle 1,1,2,3,3,4,4,5,6,7\rangle$, we would have the following. Note in particular, if we look only at the offsets that line up with elements satisfying the predicate, we see the values $0, 1, 2, 3, \ldots$ These are the indices of those elements in the output.

input	1	1	**2**	3	3	**4**	**4**	5	**6**	7	
offsets	0	0	**0**	1	1	**1**	**2**	3	**3**	4	4
output			**2**			**4**	**4**		**6**		

Flattening

A similar problem that often arises in practice is the need to concatenate many sequences together into one "flat" result.

```
flatten: 'a seq seq -> 'a seq
```

(For example:) `flatten` $\langle\langle 1,2\rangle, \langle\rangle, \langle 3\rangle, \langle 4,5,6\rangle, \langle 7\rangle\rangle = \langle 1,2,3,4,5,6,7\rangle$

Using a very similar approach to filtering with a scan, we can perform a flatten with a scan to compute the starting offset of every inner sequence.

```
fun flatten s =
  let
    val n = Seq.length s
    val offsets = Parallel.scan (op+) 0 (0, n) (fn i =>
      Seq.length (Seq.nth s i))
    val output = ForkJoin.alloc (Seq.nth offsets n)
  in
    Parallel.parfor (0, n) (fn i =>
      let
        val si = Seq.nth s i
        val offset = Seq.nth offsets i
      in
        Parallel.parfor (0, Seq.length si) (fn j =>
          Array.update (output, offset + i, Seq.nth si j))
      end);
    Seq.fromArray output
  end
```

For example, on the input considered above:

input	$\langle 1,2\rangle$	$\langle\rangle$	$\langle 3\rangle$	$\langle 4,5,6\rangle$	$\langle 7\rangle$	
offsets	0	2	2	3	6	7

This solution has linear work and polylogarithmic span. In particular, on an input s:

Work: $O(|s| + \sum_i |s[i]|)$

Span: $O(\log |s| + \max_i \log |s[i]|)$

A nice example: filtering with flattening

The implementations of `filter` and `flatten` are remarkably similar. The connection between the two becomes especially clear if we consider implementing `filter` in terms of `flatten`:

```
(* alternative implementation of filter, in terms of flatten *)
fun filter p s =
  flatten (Parallel.tabulate (0, Seq.length s) (fn i =>
    if p (Seq.nth s i) then
      Seq.singleton (Seq.nth s i)
    else
      Seq.empty()))
```

This solution has the same asymptotic cost bounds as before (although it will not be as efficient in practice, due to the need to construct intermediate sequences.)

Parallel Quicksort with Filter

The classic quicksort algorithm is traditionally presented as a sequential algorithm. However, it can easily be made parallel by making two observations: (1) the pivoting step is essentially just filtering the input into three subsequences, and (2) the two recursive calls can proceed in parallel. An implementation along these lines is as follows.

```
(* The argument `cmp` is a comparison function for elements of
 * type 'a.
 *   cmp(x,y) returns one of: LESS, EQUAL, GREATER
 * This allows our quicksort implementation to be polymorphic
 * for any element type.
 *)
fun quicksort (cmp: 'a * 'a -> order) (s: 'a seq) : 'a seq =
  if Seq.length s <= 1 then
    s
  else
    let
      (* pick a random pivot element *)
      val pivot = Seq.nth s (random_int (0, Seq.length s))
      val L = Seq.filter (fn x => cmp (x, pivot) = LESS) s
      val E = Seq.filter (fn x => cmp (x, pivot) = EQUAL) s
      val G = Seq.filter (fn x => cmp (x, pivot) = GREATER) s
      val (SL, SR) =
        ForkJoin.par (fn () => quicksort L, fn () => quicksort R)
    in
      Seq.append (Seq.append (SL, E), SR)
    end
```

Tree Flattening

We can generalize the notion of flattening to any tree data structure. In particular, suppose you had a binary tree with elements at the leaves, and you want to “flatten” this tree by storing the in-order traversal into an output sequence.

We can do so by *augmenting* each node of the tree with its size, i.e., the number of elements in that subtree. An appropriate data structure is as follows. We allow for leaves to be “fat” in the sense that they can store more than one element if desired. Each `Node(n,l,r)` records its size `n` and its two child subtrees, `l` and `r`.

```
datatype 'a tree =
  Leaf of 'a seq
| Node of int * 'a tree * 'a tree

fun size(t) =
  case t of
    Leaf s => Seq.length s
  | Node (n, _, _) => n
```

We will be careful to ensure that the sizes of nodes are always properly recorded by never constructing `Node`s directly, and instead always using the following `make_node` function.

```
fun make_node(l, r) =
  Node(size(l) + size(r), l, r)
```

Given such a tree, we can flatten it; we'll call this function `tflatten`. It takes a tree as input and returns a flat sequence. The idea is to walk the tree, keeping track of the starting offset of every subtree. Descending into a left subtree does not affect the offset. Descending into a right subtree shifts the offset by the size of the left subtree.

```
fun tflatten(t: 'a tree) : 'a seq =
  let
    val output = ForkJoin.alloc (size t)
    fun walk(c: 'a tree, offset: int) =
      case c of
        Leaf s =>
          Parallel.parfor (0, Seq.length s) (fn i =>
            Array.update (output, offset + i, Seq.nth s i))
      | Node (_, l, r) =>
          ( ForkJoin.par (fn () => walk(l, offset),
                          fn () => walk(r, offset + size(l)))
          ; () )
  in
    walk(t, 0);
    Seq.fromArray output
  end
```

We can think of `tflatten` as a pure function from trees to sequences. This function will be an essential building block for writing purely functional parallel code without needing to directly write to an output array, instead relying on `tflatten` to perform this duty for us, with a guarantee of no off-by-one errors or data races. The next section develops a particular example.

Parallel Divide-and-Conquer Merge

We have considered previously a parallel `mergesort`:

```
fun mergesort (cmp: 'a * 'a -> order) (s: 'a seq) : 'a seq =
  if Seq.length s <= 1 then
    s
  else
    let
      (* split the sequence in half *)
      val half = Seq.length s div 2
      val (l, r) = (Seq.take s half, Seq.drop s half)
      val (sl, sr) = ForkJoin.par (fn () => mergesort cmp l,
                                   fn () => mergesort cmp r)
    in
      merge cmp (sl, sr)
    end
```

The question is how to implement the `merge` function. Sequentially, assuming comparisons are $O(1)$, merging requires a linear amount of work. It turns out we can achieve the same using a parallel divide-and-conquer approach that yields linear work and polylogarithmic span.

The key idea is to split one of the sequences in half, and use a binary search on the other sequence to find an appropriate split point. Here, we're taking advantage of the sortedness of the two input sequences. (A requirement for `merge` is that the two input sequences are already sorted.)

The code is as follows. We first build a tree by recursively performing the binary-search-based split, and then finally flatten that tree.

```
fun merge_tree cmp (s1: 'a seq, s2: 'a seq) : 'a tree =
  if Seq.length s1 = 0 then
    Leaf s2
  else if Seq.length s2 = 0 then
    Leaf s1
  else
    let
      val n1 = Seq.length s1
      val n2 = Seq.length s2
      val mid1 = n1 div 2
      (* select the median element of s1 *)
      val pivot = Seq.nth s1 mid1

      (* search for the index of the pivot in s2 *)
      val mid2 = binary_search cmp (s2, pivot)

      (* split the two sequences by the two midpoints
       * mid1 and mid2. We can then perform two recursive
       * merges. Note that it is convenient exclude the
       * pivot element from the recursive calls, and
       * instead put it directly into the output. This
       * avoids the need for a separate base case handling
       * the case where |s1| = 1.
       *)
      val l1 = Seq.take s1 mid1
      val r1 = Seq.drop s1 (mid1 + 1)
      val l2 = Seq.take s2 mid2
      val r2 = Seq.drop s2 mid2
      val middle = Leaf (Seq.singleton pivot)
      val (l, r) =
        ForkJoin.par (fn () => merge_tree cmp (l1, l2),
                      fn () => merge_tree cmp (r1, r2))
    in
      make_node (make_node (left, middle), right)
    end

fun merge cmp (s1: 'a seq, s2: 'a seq) : 'a seq =
  tflatten (merge_tree cmp (s1, s2))
```

Work analysis

The work of this function on inputs of length n and m can succinctly be described with the following recurrence. Here we use a variable k in the range $0 \leq k \leq m$ to describe the (unknown) midpoint of the second sequence, which is discovered by a binary search, costing $O(\log m)$. The size n of the first sequence always is cut in half, but the size of the second sequence is split into two sequences of sizes k and $m - k$, respectively.

$$W(n, m) = W(n/2, k) + W(n/2, m - k) + \log m \quad \text{(where } 0 \leq k \leq m)$$

$$W(0, m) = m \quad \text{(base case)}$$

$$W(n, 0) = n \quad \text{(base case)}$$

An interesting challenge (to be considered in a later lecture) is to prove that this recurrence can be upper bounded as $W(n, m) \in O(n + m)$.