Lecture 3 Notes (February 10, 2025)

Sam Westrick

s.westrick@nyu.edu

Sequences

A *sequence* is an ordered collection of elements indexed by natural numbers between 0 and $n-1$ where n is the length. We can define a few useful notational conveniences and a cost model. All work and span quantities in the following table are asymptotic, e.g., 1 means $O(1)$ and n means $O(n)$.

Syntactic Sugar	Code	Work W	Span S
$\|s\|$	`length s`	1	1
$s[i]$	`nth s i`	1	1
$s[i..j]$	`subseq s (i, j-i)`	1	1
$\langle\rangle$	`empty()`	1	1
$\langle x\rangle$	`singleton x`	1	1
$\langle e_i : \texttt{lo} \le i < \texttt{hi}\rangle$	`tabulate (lo, hi) (fn i => e_i)`	$\sum_{i=\texttt{lo}}^{\texttt{hi}} W(e_i)$	$\log(\texttt{hi-lo})+$ $\max_i S(e_i)$
$\langle f(x) : x \in s\rangle$	`map f s`	$\sum_x W(f(x))$	$\log\|s\| +$ $\max_x S(f(x))$
$s_1 \bowtie s_2$	`append (s1, s2)`	$\|s_1\| + \|s_2\|$	$\log(\|s_1\| +$ $\|s_2\|)$

This particular cost model assumes sequences that are represented as arrays. More generally, it is helpful to think of a sequence as an *abstract data type* which could have many different representations. For example, another common representation is as a (balanced) tree, with elements stored at nodes, where the in-order traversal of the tree implicitly represents the ordering of the elements in the sequence.

Parallel Prefix "Sums"

Perhaps the single most important primitive in parallel computing is an algorithm for computing **prefix sums** in parallel. This core building block can be used to parallelized computations that may at first appear to be sequential.

Given a sequence of integers s, suppose we wanted to know the sum of every prefix of s. On input of length n, the output will be length $n+1$, as there are $n+1$ prefixes (including the initial zero-length prefix).

s	prefix sums of s
$\langle 1, 2, 3, 4 \rangle$	$\langle 0, 1, 3, 6, 10 \rangle$
$\langle 1, 0, 0, 1, 0, 1 \rangle$	$\langle 0, 1, 1, 1, 2, 2, 3 \rangle$
$\langle \rangle$	$\langle 0 \rangle$

Sequentially, we could clearly compute all prefix sums by walking left-to-right across the input and keeping track of a running sum. At each element, we write down the current running sum to the output, and then update the running sum with the next element. This has $O(n)$ work but is entirely sequential.

It turns out that it is possible to compute all prefix sums in parallel, with only $O(n)$ work and $O(\log n)$ span. Before we get there, let's see a few examples of how we could compute prefix sums in parallel, but less efficiently.

Work-inefficient Parallel Prefix Sums #1

The most obvious way we could compute prefix sums in parallel would be to perform one sum for every prefix:

$$\left\langle \texttt{reduce (op+)}\ 0\ s[0..i] : 0 \leq i \leq |s| \right\rangle$$

This solution has $O(n^2)$ work and $O(\log n)$ span for an input of size n. This is a significant amount of parallelism, but it is **work-inefficient**: this approach requires asymptotically more work than the best sequential solution.

To see that the solution above requires $O(n^2)$ work, consider that we have pay $O(i)$ work for each prefix of length i, which comes out to $O(\sum_i i) = O(n^2)$ work in total.

Work-inefficient Parallel Prefix Sums #2

Another possibility would be to compute prefix sums with a divide-and-conquer approach.

```
fun dc_prefix_sums (s: int Seq.t) =
  if Seq.length s = 0 then
    Seq.singleton 0
  else if Seq.length s = 1 then
    Seq.fromList [0, Seq.nth s 0]
  else
    let
      val n1 = Seq.length s div 2
      val n2 = Seq.length s - n1
      val s1 = Seq.subseq s (0, n1)
      val s2 = Seq.subseq s (n1, n2)
      val (p1, p2) = ForkJoin.par (fn () => dc_prefix_sums s1,
                                   fn () => dc_prefix_sums s2)
```

```
    val t1 = Seq.nth p1 n1 (* total sum of the left-hand side *)
  in
    Seq.append (Seq.subseq p1 (0, n1), Seq.map (fn x => t1 + x) p2)
  end
```

This divide-and-conquer solution admits the following work and span recurrences.

$$W(n) = 2W(n/2) + O(n) = O(n \log n)$$

$$S(n) = S(n/2) + O(\log n) = O(\log^2 n)$$

We see that this approach is closer to work-efficient, but still off by a logarithmic factor. The problem is that, after solving the recursive instances, we still have to pay a linear cost to update the results of the right-hand side (by adding the total from the left-hand side).

Work-efficient Parallel Prefix Sums: Upsweep-Downsweep

To achieve $O(n)$ work, we can use the following idea. Notice that in the divide-and-conquer solution above, to generate the right-hand-side results, we needed to know the total sum on the left. So, why not just precompute all of these total sums?

The first phase of our work-efficient algorithm will do just this. We call this an *upsweep*. It is similar to a parallel divide-and-conquer summation, but we also write down all of the intermediate results. These intermediate results can be organized into a balanced binary tree, mirroring the structure of the divide-and-conquer recursion.

The second phase of the algorithm is called the *downsweep*. We walk down the tree in parallel, maintaining an "accumulator" value (initially 0 at the root). Every time we walk to a left child, the accumulator stays the same. Every time we walk to a right child, we update the accumulator by adding the total sum of the left child.

When this traversal hits a leaf, the current value of the accumulator will be the prefix sum at that position.

Here is code that implements this solution. In the downsweep, we write directly to an array, and keep track of our current position (`offset`) and the number of elements that need to be written (`n`).

```
datatype tree = Leaf of int | Node of int * tree * tree
fun sum_of(t) = case t of Leaf x => x | Node (x, _, _) => x

fun upsweep(s: int Seq.t) =
  if Seq.length s = 1 then Leaf (Seq.nth s 0) else
  let val half = Seq.length s div 2
      val (t1, t2) = ForkJoin.par (fn () => upsweep (Seq.take s half),
                                   fn () => upsweep (Seq.drop s half))
```

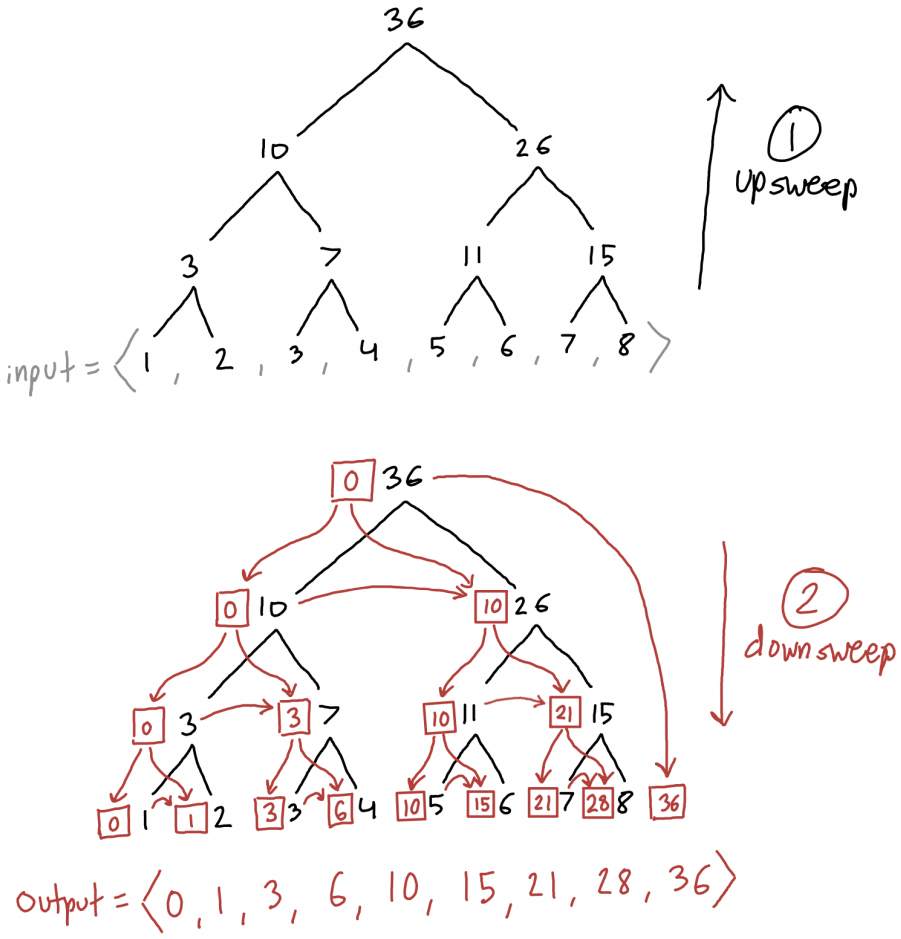

Figure 1: Upsweep-downsweep prefix sums

```
  in Node (sum_of t1 + sum_of t2, t1, t2)
  end

fun downsweep(t, acc, output, offset, n) =
  case t of
    Leaf _ => Array.update(output, offset, acc)
  | Node(_, t1, t2) =>
      let val half = n div 2
      in ForkJoin.par (fn () => downsweep(t1, acc, output,
                                          offset, half),
                       fn () => downsweep(t2, sum_of t1 + acc, output,
                                          offset + half, n - half));
         ()
      end

fun up_down_prefix_sums(s: int Seq.t) =
  if Seq.length s = 0 then Seq.singleton 0 else
  let
    val n = Seq.length s
    val t = upsweep s
    val output = ForkJoin.alloc(n + 1)
  in
    downsweep(t, 0, output, 0, n);
    Array.update(output, n, sum_of t);

    (* In actual MaPLe, this converts an array to a sequence; at this
     * point, the programmer needs to be careful to never mutate the
     * underlying array.
     *)
    ArraySlice.full(output)
  end
```

This solution has two phases (the upsweep, and the downsweep), so the overall work and span is just the sum of the two phases:

$$W(n) = W_{\text{upsweep}}(n) + W_{\text{downsweep}}(n)$$

$$S(n) = S_{\text{upsweep}}(n) + S_{\text{downsweep}}(n)$$

Each phase has a simple work and span recurrence:

$$W_{\text{upsweep}}(n) = 2W_{\text{upsweep}}(n/2) + O(1) = O(n)$$

$$S_{\text{upsweep}}(n) = S_{\text{upsweep}}(n/2) + O(1) = O(\log n)$$

$$W_{\text{downsweep}}(n) = 2W_{\text{downsweep}}(n/2) + O(1) = O(n)$$

$$S_{\text{downsweep}}(n) = S_{\text{downsweep}}(n/2) + O(1) = O(\log n)$$

Overall, therefore, we have $O(n)$ work and $O(\log n)$ span. This algorithm is

work-efficient (it asymptotically matches the best sequential solution), yet still has ample parallelism.

Another work-efficient Prefix Sum Algorithm: Contraction

Another idea would be to allow for the tree of partial results to be implicit by working level-by-level instead of constructing the tree explicitly.

The initial level is the input sequence, and the next level in the tree can be obtained by **contracting**: combining adjacent elements, pairwise, to produce a sequence of half the length. We can then recursively compute the prefix sums across the contracted level. Finally, the overall prefix sums can be computed by **expanding** the result (filling in the missing elements). This sometimes also called a **three-phase scan**.

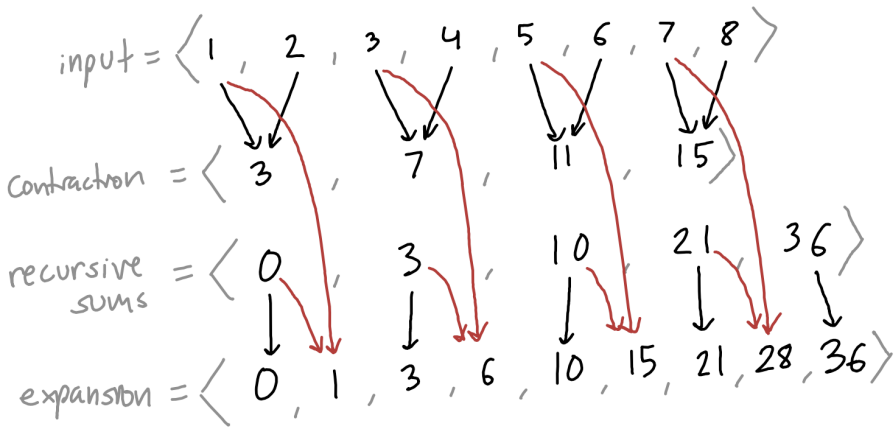

Figure 2: Contraction-based Prefix Sums

Code for this solution is as follows. Here, for simplicity, we assume the input length is a power of 2.

```
(* Note: assumes the input is power-of-two length *)
fun contraction_prefix_sums(s: int Seq.t) =
  if Seq.length s = 0 then Seq.singleton 0
  else if Seq.length s = 1 then
    Seq.fromList [0, Seq.nth s 0]
  else
    let
      val n = Seq.length s
      val contracted =
        Seq.tabulate (fn i => Seq.nth s (2*i) + Seq.nth s (2*i + 1)) (n div 2)
      val recursive_sums =
        contraction_prefix_sums(contracted)
```

```
  fun expanded_elem(i) =
    if i mod 2 = 0 then Seq.nth recursive_sums (i div 2)
    else Seq.nth recursive_sums (i div 2) + Seq.nth s (i-1)
in
  Seq.tabulate expanded_elem (n+1)
end
```

Note that this approach only has a single recursive call, yielding the following work and span recurrences:

$$W(n) = W(n/2) + O(n)$$

$$S(n) = S(n/2) + O(\log n)$$

The span recurrence we have seen before; this solves to $O(\log^2 n)$. The work recurrence can be solved by noticing that it forms a geometrically decreasing series. We can use the fact that $\sum_{i=0}^{\infty} 1/2^i = 2$ to see that this comes out to a total of linear work.

$$W(n) = O(n + n/2 + n/4 + \ldots) = O(n \sum_i 1/2^i) = O(n)$$

Generalization: Scan

Above, we have focused on prefix sums, specifically for performing a summation (with addition of integers).

But, similar to `reduce`, we can generalize the algorithm to compute a "sum" of any *associative binary operation*. Anywhere above where we combine two numbers with addition, we can instead compute an arbitrary function, `g`.

The general form is written `Seq.scan g z s` for computing all the prefix "sums" (w.r.t. function `g`) of the sequence `s`. Alternatively, in MaPLe, we can write `Parallel.scan g z (lo, hi) f` to compute prefix sums over the implicit sequence `f(lo), ..., f(hi-1)`.

From this we can compute a variety of interesting functions. Here are two especially interesting and powerful examples:

- Solutions to linear recurrences of the form $r(i) = a_i \cdot r(i-1) + b_i$ can be computed using the associative operation $g((a_1, b_1), (a_2, b_2)) = (a_1 \cdot a_2, b_1 \cdot a_2 + b_2)$ and corresponding identity $z = (1, 0)$, over the elements (a_i, b_i). (Note that this function g is associative!)
- We can compute the nearest non-zero value to the left of any position in a sequence using the function `copy(a,b) = if b > 0 then b else a` and corresponding identity value `z = 0`. (Note that `copy` is associative!) We call this a **copy-scan**, and it is a very powerful primitive in parallel computing. For example, `scan copy 0` $\langle 0, 1, 0, 2, 0, 0, 0, 3, 0 \rangle = \langle 0, 0, 1, 1, 2, 2, 2, 2, 3, 3 \rangle$.

A Practical Consideration: Block-based Three-phase Scan

Commonly, in practice, the preferred implementation of `scan` is a three-phase (contraction-based) algorithm with a small modification: rather than combine adjacent elements pairwise, we instead break up the input in larger blocks and contract into a much smaller recursive problem size.

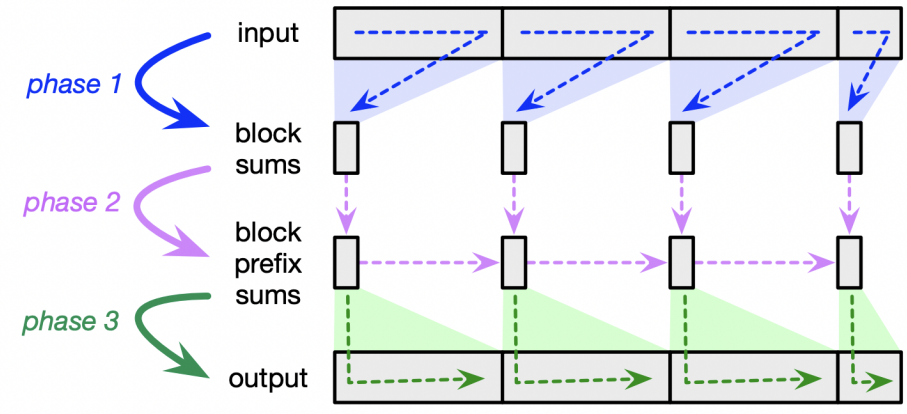

Figure 3: Three-phase Blocked Scan

Note that in this approach, we sacrifice some parallelism, because the expansion step has to walk sequentially within each block (but is parallel across the blocks).

For a block-size B, this approach admits the following work and span recurrences.

$$W(n) = W(n/B) + O(n) = O(n)$$

$$S(n) = S(n/B) + O(B + \log n) = O(B + \log^2 n)$$

As the block size increases, this approach gets more and more sequential. But as long as $n \gg B$, this solution still has ample parallelism.

But, a question we haven't answered: why? Why would you want to increase the block size and sacrifice parallelism?

The reason is **work efficiency**. Let's consider the following recurrence, u, which is similar to the work recurrence above but is no longer asymptotic. This function is counting (approximately) the number of array updates performed within the block-based scan, which is a good estimate of its performance in practice. The question is, how does this function behave as we change the block size?

$$u(n) = u(n/B) + n + n/B = u(n/B) + n\tfrac{B+1}{B}$$

We can upper bound this recurrence by a geometric series:

$$u(n) \leq n\tfrac{B+1}{B} \textstyle\sum_{i=0}^{\infty} \frac{1}{B^i} = n\frac{B+1}{B}\frac{B}{B-1} = n\frac{B+1}{B-1}$$

And actually, for sufficiently large n, we will have $u(n) \approx n\frac{B+1}{B-1}$.

For $B = 2$, we have $u(n) \approx 3n$, and as we increase B, this function decreases; at the limit as $B \to \infty$, we have that $u(n)$ approximately approaches n.

In other words, **as we increase the block size, the work-efficiency of the algorithm improves**.

In practice, we typically use a block size such as $B = 100$ or $B = 1000$ which guarantees that $u(n)$ is within a few percent of n while still exposing ample parallelism for large n.