Contraction and Scan

1 Contraction

Recall that *divide-and-conquer* is a technique for solving a problem by reducing it to *multiple* smaller versions of itself, solving those recursively, then combining the solutions to the smaller problems to solve the bigger problem. Divide-and-conquer is highly attractive for parallel algorithms, because the recursive calls can usually be done in parallel. However, divide-and-conquer algorithms tend to do a lot of work if the combine step is not constant time.

Contraction is a similar technique to divide-and-conquer that can sometimes achieve better cost bounds. Instead of breaking a problem into multiple smaller versions of itself, contraction *shrinks* a problem into **one** smaller version of itself, solves that recursively, then uses that solution to figure out the solution to the original (bigger) problem. Because it only performs one recursive call instead of many, contraction can sometimes perform less work than divide-and-conquer on the same problem.

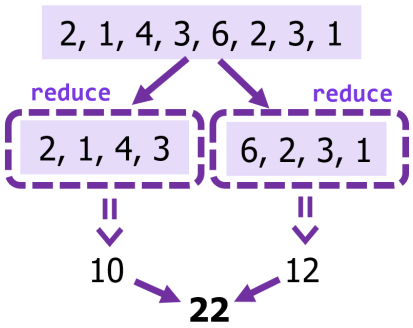
As a warm up, lets solve the familiar *reduce* problem, which we already solved with divide-and-conquer, using contraction.

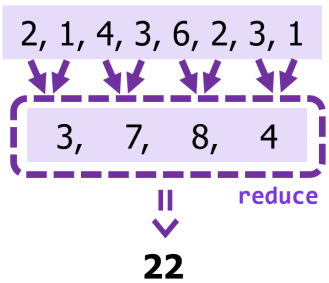
1.1 Reduce via Contraction

The idea of contraction is to shrink a problem to a smaller problem that still contains enough information to help us solve the bigger problem. Given a sequence of n elements that we want to reduce, can you think of a sequence of $n/2$ elements that has the same answer?

Here's a simple idea: pair up adjacent elements and reduce them (add them together). This shrinks the sequence to one that is half as big and has the same total answer as the original. Now we just recursively reduce this smaller sequence and we are done.

This figure shows the difference between the divide-and-conquer algorithm for reduce (left) and the contraction algorithm for reduce (right).





Algorithm: Reduce via Contraction

```
fun reduce(f : (T, T) -> T, I : T, S : sequence<T>) -> T:
  match length(S) with:
    case 0: return I
    case 1: return S[0]
    case _:
      B = parallel [f(S[2*i], S[2*i+1]) for i in 0...|S|/2-1]
          + [S[|S|-1]] if |S|%2 == 1 else []
      return reduce(f, I, B)
```

Since there is only one recursive call, we no longer get any parallelism from the recursion. Instead, the contraction step itself (the step which combines pairs of elements) is parallel.

Theorem: Cost of Contraction-Based Reduce

Assuming that f can be evaluated in constant time, reduce implemented with contraction also costs $O(|S|)$ work and $O(\log|S|)$ span.

Proof. Let $n = |S|$. At the root, the contraction step performs $O(n)$ work. Each recursive call halves the size of the sequence, so we have

$$W(n) = W\left(\frac{n}{2}\right) + O(n).$$

This is root dominated, so $W(n) = O(n)$. Similarly, the span at the root is $O(1)$ since it just costs a tabulate, so the span recurrence is

$$S(n) = S\left(\frac{n}{2}\right) + O(1).$$

This recurrence has $O(\log n)$ levels so the span is $S(n) = O(\log n)$. □

2 Scan

In the last lecture, we studied the **reduce** problem and its applications to parallel algorithm design. We saw that `reduce` is not only useful for its obvious application of computing sums, but that it can be customized with nontrivial associative functions to compute interesting results, like the MCSS, or the number of excess parenthesis from the Parenthesis Matching problem.

Now, we study the **scan** problem, which generalizes reduce to compute not just the total, but the reduction of *every prefix* of a sequence with respect to a given associative function. With scan, we can solve even more interesting problems.

Definition: Scan

Given a sequence S, an associative function and an identity, scan computes the exclusive prefix sums and the total sum of S with respect to the associative function and identity.

The output of scan is equivalent to the following brute-force implementation. Of course we will aim to implement something much more efficient.

```
fun scan(f : (T, T) -> T, I : T, S : sequence<T>) -> (sequence<T>,T):
  return tabulate(fn i => reduce(f, I, subseq(S, 0, i)), |S|),
           reduce(f, I, S)
```

Note that scan actually returns two values. The first is the *exclusive* prefix sum of S (meaning that at position i, the sum consists of all elements that occur *strictly before*, not including, position i). The second return value is the total sum since it is not included as part of the first value, since the final value of the sequence is excluded.

For example, given the sequence $[2,1,4,3,6,2,3,1]$, using addition with the identity 0, the exclusive prefix sums are $[0,2,3,7,10,16,18,21]$ and the total is 22.

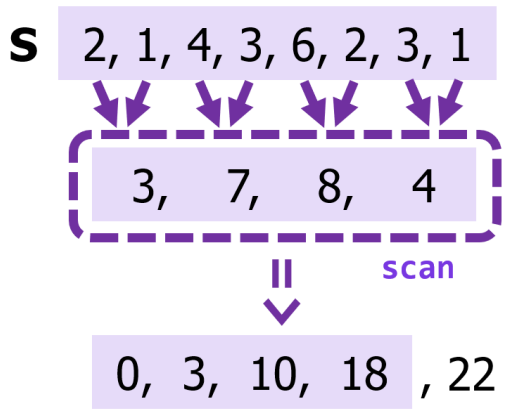
Remark: Scan via Divide-and-Conquer

Scan can also be implemented via divide-and-conquer, with a cost of $O(n \log n)$ work and $O(\log n)$ span. As an exercise, try to figure out how. This is much better than brute force, but we can still do better.

2.1 Scan via Contraction

The implementation of the scan function follows a similar pattern to reduce at the beginning. Given a sequence S, we get a second sequence B by pairing up adjacent elements and applying the supplied function f, which combines two elements. We then make a recursive call to scan on B.

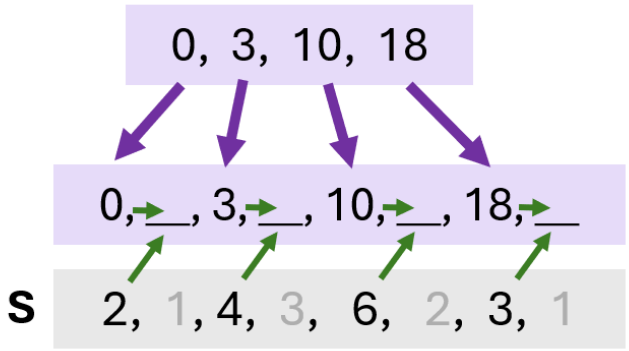
The tricky part is that the result of the recursive call is no longer the entire answer to the problem like it was for reduce. Instead, we need to perform an **expansion** step, where we use the result of the recursive call to figure out the solution to the original problem. To figure this out, lets look at an example of what the recursive call returns and see what information it offers us.



The recursive call gave us back the sequence $[0,3,10,18]$ and the total value. What useful information does it contain? Clearly the total is useful, but what about the prefix sums? Remember

that the sequence we need to produce is $[\underline{0},2,\underline{3},7,\underline{10},16,\underline{18},21]$. Notice that the recursive result contains the **even-indexed elements** of the answer!

To fill in the odd-indexed elements, we just observe that an element at odd index i is just the sum (with respect to f) of its preceding prefix sum and the element at position $i-1$, so we can fill them all in in parallel.



This gives us the following efficient parallel implementation of scan.

Algorithm: Scan

```
fun scan(f : (T,T) -> T, I : T, S : sequence<T>) -> (sequence<T>,T):
  match length(S) with:
    case 0: return [], I
    case 1: return [I], S[0]
    case _:
      B = parallel [f(S[2*i], S[2*i+1]) for i in 0...|S|/2-1]
          + ([S[|S|-1]] if |S|%2 == 1 else [])
    R, total = scan(f, I, B)
    return tabulate(fn i => R[i/2] if i%2==0
                      else f(R[i/2], S[i-1]), |S|), total
```

Theorem: Cost of Scan

Assuming f can be evaluated in $O(1)$ time, scan costs $O(|S|)$ work and $O(\log|S|)$ span.

Proof. Let $n = |S|$. The contraction step pairs up and sums adjacent elements, which costs $O(n)$ work and $O(1)$ span. The expansion step performs a tabulate of length n to produce the answer, which also costs $O(n)$ work and $O(1)$ span. The recursion step recursively calls `scan` on a sequence of half the length, so it has work

$$W(n) = W\left(\frac{n}{2}\right) + O(n),$$

which is root dominated and solves to $O(n)$. Similarly, the span is

$$S(n) = S\left(\frac{n}{2}\right) + O(1),$$

which unrolls to $O(\log n)$. □

Note that this assumes f can be evaluted in constant time—if f has a different cost bounds, then we need to do a separate analysis with the new costs to get the work and span for that particular call to the `scan` function.

3 Application of Scan: MCSS Revisited

Let's consider (yet again) the problem of Maximum Contiguous Subsequence Sum (MCSS).

Example: Brute-Force MCSS

In RefreshLab, we implemented a *brute-force* solution to MCSS by trying *all contiguous subsequences* and computing their sum using `reduce`, then taking the maximum of those sums (with another `reduce`).

```
fun mcss(S : sequence<int>) -> int:
  fun sum(i : int, k : int): return reduce(plus, 0 subseq(S, i, k))
  sums = parallel [sum(i,k) for i in 0...|S|-1 for k in 0...|S|-i]
  return reduce(max, -∞, sums)
```

There are $O(n^2)$ contiguous subsequences, and taking the sum of one such subsequence using reduce has $O(n)$ work. The work is therefore $O(n^3)$.

Despite the high work, the span is very good since all $O(n^2)$ subsequences can be computed in parallel, so the span is dominated by the reduce calls which have $O(\log n)$ span. The overall span is therefore $O(\log n)$.

In this section we will show that we can use `scan` to reduce the work of this solution, first down to $O(n^2)$, then with a further optimization to bring it all the way down to $O(n)$, which is optimal and as good as our divide-and-conquer- and reduce-based solutions!

3.1 Optimizing Interval Sum Calculations

The low-hanging fruit to address is the redundant computations of subsequence sums. Every subsequence gets its sum computed with a separate `reduce` which costs $O(n)$. Instead, we can use **prefix sums**, as computed by `scan` to substantially optimize this.

The key is in the following critical observation:

$$\texttt{sum(S[i...j))} = \texttt{sum(S[0...j))} - \texttt{sum(S[0...i))}$$

That is, the sum of any interval (inclusive on the left and exclusive on the right) is just the difference between two prefix sums! Since scan computes all of the prefix sums, i.e., it computes `sum(S[0...i))` for all values of i, we have all the information needed to compute the sum of any interval in *constant time*. We could therefore improve our brute-force solution like so:

Algorithm: Improved Brute-Force MCSS

```
fun mcss(S : sequence<int>) -> int:
  splus, total = scan(plus, 0, S)
  prefix_sum = splus + [total]

  fun sum(i : int, k : int): return prefix_sum[i+k] - prefix_sum[i]
  sums = parallel [sum(i,k) for i in 0...|S|-1 for k in 0...|S|-i]
  return reduce(max, -∞, sums)
```

In this improved algorithm, notice that we have left the "brute force" part untouched: it still tries all $O(n^2)$ contiguous subsequences then picks the largest sum. The optimization makes `sum` run in $O(1)$ after preprocessing the prefix sums using `scan` which costs $O(n)$ work and $O(\log n)$ span. The cost of the improved algorithm is therefore $O(n^2)$ work and $O(\log n)$ span.

3.2 A Further Optimization: Prefix Minimums

The second optimization is more subtle, but is another clever application of `scan` that will bring the work all the way down to $O(n)$.

Of course, the algorithm as it stands is still brute-force—it computes every contiguous subsequence's sum individually. These subsequences have a lot in common, so there is redundancy in here to remove. In particular, consider the interval $[i, j)$ that corresponds to the maximum sum. We know that this sum, in terms of prefix sums is just

```
prefix_sum[j] - prefix_sum[i]
```

Furthermore, its the maximum such value for any choice of j and i. So, if we fix a particular value of j, we end up brute-forcing over all possible values of $i \leq j$ and computing this formula for each of them. But notice that there's a very clear choice for which i to pick. Since we want to *maximize* this difference (`prefix_sum[j] - prefix_sum[i]`), we must pick the **minimum possible value** of `prefix_sum[i]` for all values of $i \leq j$.

Now remember that *minimum* is an associative function, and computing the value of an associative function on *every possible prefix* is **exactly what `scan` does**! So, by computing a `scan` with the minimum function *over the prefix sums sequence*, we will find, for each position j, the minimum value of `prefix_sum[i]` for all $i \leq j$. Say we call this quantity `min_prefix[j]`.

This gives us the following solution to the MCSS problem. For any fixed index j, the maximum sum of any interval of the form $[i, j)$ is given by

```
prefix_sum[j] - min_prefix[j]
```

With both of these quantities precomputed, we can therefore evaluate the maximum such sum for each individual value of j in constant time. Finally we take the maximum over all values of j and obtain the answer. Putting it all together, we get something like this:

Algorithm: Optimal MCSS Using Scan

```
fun mcss(S : sequence<int>) -> int:
  splus, total = scan(plus, 0, S)
  prefix_sum = splus + [total]
  min_prefix, _ = scan(min, ∞, prefix_sum)
  mcss_j = parallel [prefix_sum[j] - min_prefix[j] for j in 0...|S|]
  return reduce(max, -∞, mcss_j)
```

This algorithm performs two scans, one append, one tabulate, and one reduce, all of which cost $O(n)$ work and at most $O(\log n)$ span. This algorithm therefore runs in $O(n)$ work and $O(\log n)$ span, which is as good as our previous best algorithms!

4 Scan With Custom Associative Functions

Recall the requirements of the `reduce` function from last lecture. The function f must be *associative* and I must be an *identity value* for the function/type.

Formally, this means that:

- for all values x, y, z, we must have $f(f(x,y),z) = f(x,f(y,z))$,
- for all values of x, we require $f(I,x) = f(x,I) = x$.

Although we already saw interesting nontrivial uses of scan with `plus` and `min` to optimize the MCSS problem, we can solve even more problems by writing custom associative functions and reducing/scanning with those.

Here is one interesting such problem we can solve. What makes it really interesting is that by definition, it sounds very sequential. It is easy to come up with a sequential algorithm, but seemingly difficult to think of a parallel algorithm.

Problem: Previous SOME

Given a sequence S of `optional<T>`, for every position $0 \le i < |S|$. compute the rightmost `SOME` (i.e., non-`NONE`) value that occurs before position i (i.e., the most recent value seen if going left-to-right).

For example, given the sequence

$$[\textbf{None}, \textbf{Some}(5), \textbf{None}, \textbf{Some}(3), \textbf{None}, \textbf{None}, \textbf{Some}(2), \textbf{None}, \textbf{Some}(3), \textbf{Some}(1)],$$

the desired output is

$$[\textbf{None}, \textbf{None}, \textbf{Some}(5), \textbf{Some}(5), \textbf{Some}(3), \textbf{Some}(3), \textbf{Some}(3), \textbf{Some}(2), \textbf{Some}(2), \textbf{Some}(3)].$$

Notice that what happens is that the **Some**s "propagate" to the right and overwrite the **None**s.

A simple sequential solution would be to fold left-to-right and just take the previous value if the current value is `NONE`, otherwise take the current value. Essentially, a fold over the following function would give us the answer sequentially.

```
fun take_right_some(a : optional<T>, b : optional<T>) -> optional<T>:
  match b with:
    case SOME(_): return b
    case _: return a
```

The amazing thing, despite not looking like it, this function is actually associative!

Theorem: Associativity of `take_right_some`

The function `take_right_some` is associative, with `NONE` as an identity value.

Proof. There are eight possible cases for x, y, z, each of them can either be **SOME** or **NONE**. The proof simply (tediously) enumerates all such cases. Let $f =$ `take_right_some`.

x	y	z	$f(x,y)$	$f(y,z)$	$f(f(x,y),z)$	$f(x,f(y,z))$
SOME(x)	SOME(y)	SOME(z)	SOME(y)	SOME(z)	SOME(z)	SOME(z)
SOME(x)	SOME(y)	NONE	SOME(y)	SOME(y)	SOME(y)	SOME(y)
SOME(x)	NONE	SOME(z)	SOME(x)	SOME(z)	SOME(z)	SOME(z)
SOME(x)	NONE	NONE	SOME(x)	NONE	SOME(x)	SOME(x)
NONE	SOME(y)	SOME(z)	SOME(y)	SOME(z)	SOME(z)	SOME(z)
NONE	SOME(y)	NONE	SOME(y)	SOME(y)	SOME(y)	SOME(y)
NONE	NONE	SOME(z)	NONE	SOME(z)	SOME(z)	SOME(z)
NONE	NONE	NONE	NONE	NONE	NONE	NONE

Furthermore, note that $f(\texttt{NONE}, x) = x$ and $f(x, \texttt{NONE}) = x$, so `NONE` is an identity value. □

Since this function is associative, we can use it with `scan` to solve the Previous SOME problem efficiently in parallel.

Algorithm: Previous SOME

```
fun previous_some(S : sequence<optional<T>>) -> sequence<optional<T>>:
  propagated, _ = scan(take_right_sum, NONE, S)
  return propagated
```

This gives us an $O(n)$ work and $O(\log n)$ span solution to find the previous non-`NONE` option at every position in the sequence. This may seem contrived, but this technique of propagating information left-to-right down a sequence has useful applications in several parallel algorithms. You will see one in your next lab!