Parallel And Sequential Data Structures and Algorithms

Contraction and Scan

Learning Objectives

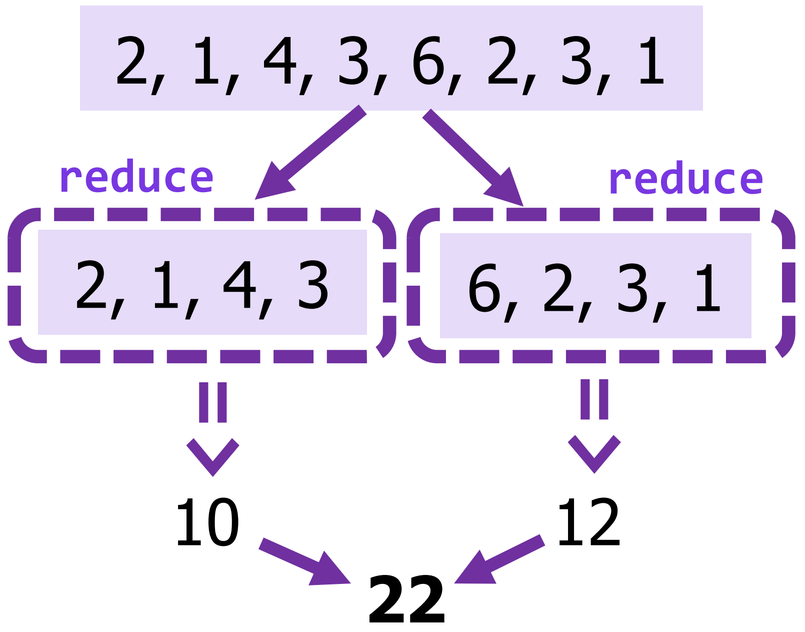
- Understand **contraction** as a technique for designing efficient parallel algorithms for sequences
- Understand the **scan** problem on sequences, and an efficient algorithm for it via contraction
- Practice **applications** of scan for precomputing prefix sums and prefix minimums

Contraction

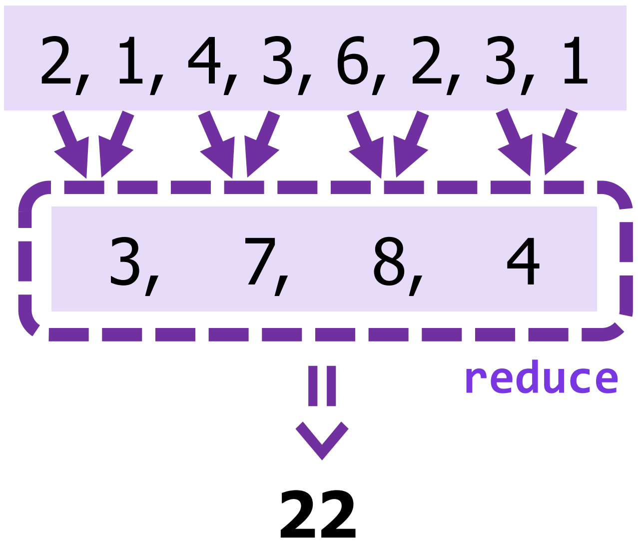
Contraction

Idea: Divide-and-conquer reduces a problem to multiple smaller problems. Contraction reduces a problem to **one** smaller problem.

Reduce (Divide-and-conquer)



Reduce (Contraction)



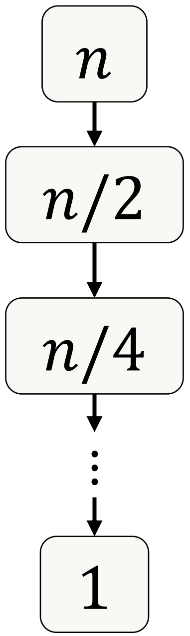
Contraction-based Reduce

```
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)
```

- Divide-and-conquer is parallel by doing the recursive calls in parallel
- Since there's only one recursive call, the parallelism is now instead **in the contraction step**

Contraction-based Reduce (Analysis)

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.



- **Work:**
  - $W(n) = W(n/2) + O(n)$
  - $W(n) = O(n + n/2 + n/4 + \ldots) = O(n)$
- **Span:**
  - $S(n) = S(n/2) + O(1)$
  - $S(n) = O(\log n)$

Scan

Scan

Definition (Scan): Given a sequence S, an associative function and an identity, scan computes the reduction of **every prefix** of S

```
scan : (f : (T, T) -> T, I : T, S : sequence<T>)
          -> (sequence<T>, T)
```

- **Scan returns**:
  - A sequence containing $[I, f(I, S[0]), f(f(I, S[0]), S[1]), \ldots]$
  - The total sum, equivalent to `reduce(f, I, S)`

Sounds quite sequential… but it turns out to be highly parallelizable!

Inefficient Scan

- We could use brute force (but please don't)

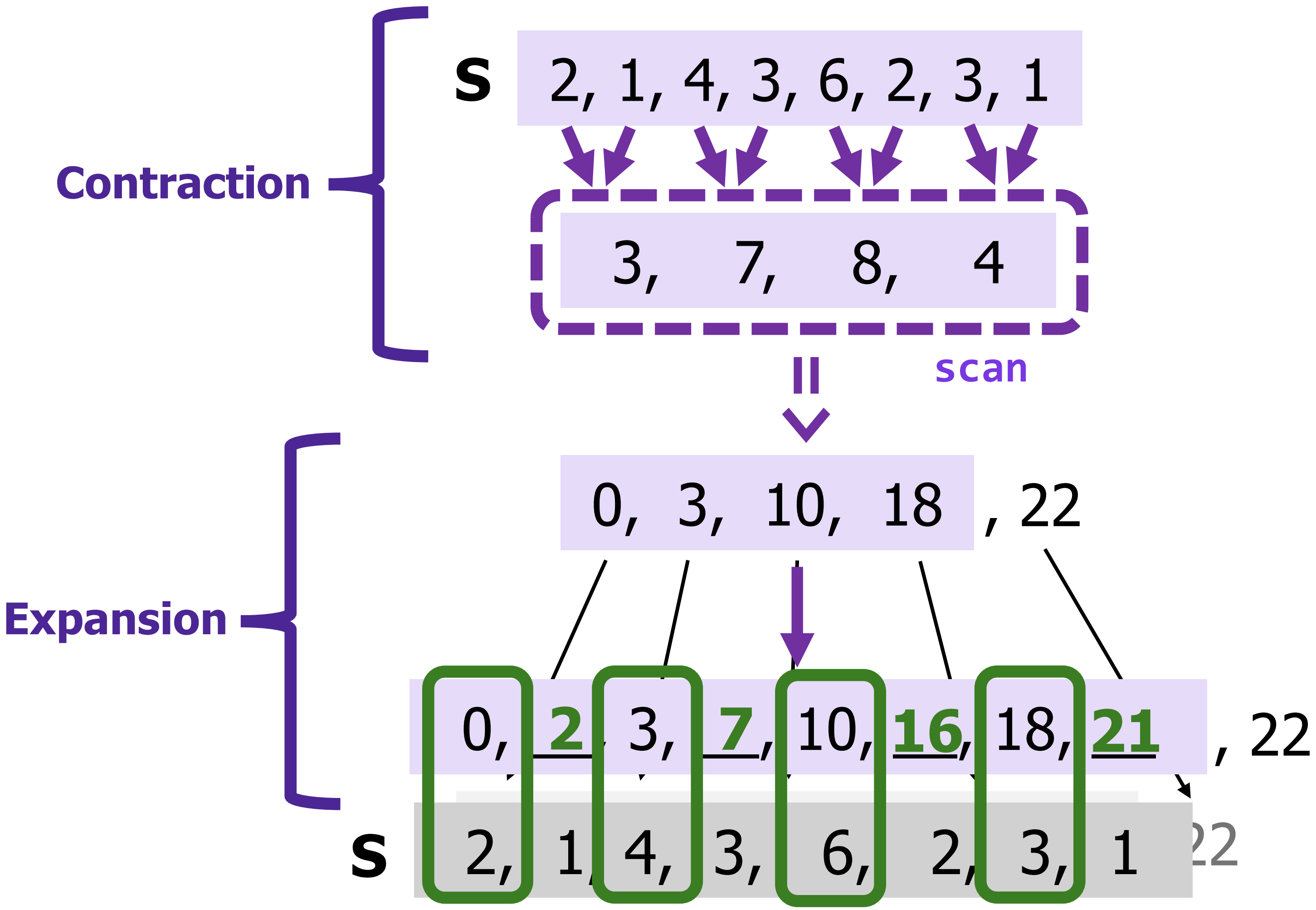
```
fun scan(f : (T, T) -> T, I : T, S : sequence<T>) -> (sequence<T>, T):
  return tabulate(fn i => reduce(f, I, subseq(S, 0, i)), |S|), reduce(f, I, S)
```

- This costs $O(n^2)$ work (but at least its $O(\log n)$ span!)

> **Exercise**: Implement **scan** using divide-and-conquer. This will cost $O(n \log n)$ work and $O(\log n)$ span

- This is much more efficient, but we can still do better!

Contraction-based Scan

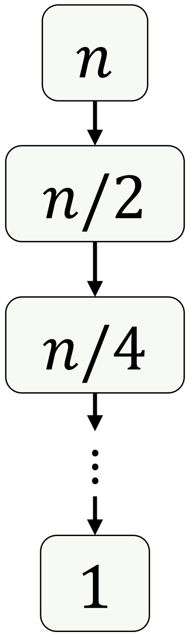


Contraction-based Scan

```
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
```

Analysis of Scan

Theorem (Cost of Scan): Assuming that f can be evaluated in constant time, scan costs $O(|S|)$ work and $O(\log |S|)$ span.



- **Work:**
  - $W(n) = W(n/2) + O(n)$
  - $W(n) = O(n + n/2 + n/4 + \ldots) = O(n)$
- **Span:**
  - $S(n) = S(n/2) + O(1)$
  - $S(n) = O(\log n)$

Applications

MCSS Revisited

- **Recall RefreshLab**: Brute-force MCSS

```
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)
```

- $O(n^2)$ contiguous subsequences; reduce costs $O(n)$ work and $O(\log n)$ span; so, this costs $O(n^3)$ work and $O(\log n)$ span
- What is redundant here?

Improved Brute-Force MCSS

- We are computing `reduce` repeatedly! $O(n^2)$ times!
- Scan can give us the sum of **every prefix**

Fact: `sum(S[i…j)) = sum(S[0…j)) – sum(S[0…i)])`

```
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)
```

Analysis of Improved Brute-Force

- Each sum computation now takes $O(1)$ time!
- Evaluating all $O(n^2)$ contiguous subsequences therefore costs $O(n^2)$ work and $O(\log n)$ span

There is **still** redundancy in this algorithm!

- Suppose the MCSS ends at index `j-1`

- Sum is `prefix_sum[j] - prefix_sum[i]`
- We are still brute-forcing over every starting index i

Optimized Brute-Force: Prefix Min

- Sum is `prefix_sum[j] - prefix_sum[i]`
- To maximize this quantity, we want

$$\texttt{min\_prefix} = \min_{i<j} \texttt{prefix\_sum[i]}$$

- i.e., we want the minimum of `prefix_sum` in every prefix

> `min` is an associative operation. That's just **another scan**!

- The optimal MCSS ending with element `j` is then

`prefix_sum[j] - min_prefix[j]`

Optimized MCSS Example

S	-2	1	-3	4	-1	2	1	-5	4	
prefix_sum	0	-2	-1	-4	0	-1	1	2	-3	1
min_prefix	-∞	0	-2	-2	-4	-4	-4	-4	-4	-4

MCSS = `prefix_sum[j] – min_prefix[j] = 2 – (-4) = 6`

Optimal MCSS With 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)
```

- Two scans and a reduce, each of which costs $O(n)$ work and $O(\log n)$ span, so the total cost is $O(n)$ work and $O(\log n)$ span!
- Same bounds as our previous divide-and-conquer algorithm

Scan With Custom Associative Functions

Review: Associative Functions

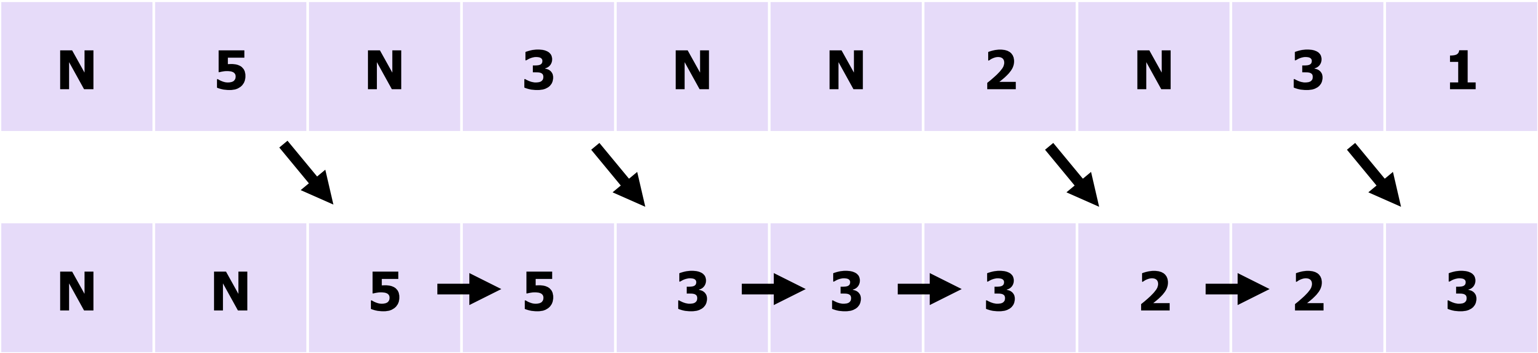
Scan can be used to compute the prefix-"sums" of any associative function

- Recall the two requirements for `reduce` and `scan`:
  - `f` must be an **associative function**. Formally
    - `f(f(x,y),z) = f(x,f(y,z))` for all values `x,y,z` of type `T`
  - `I` must be an **identity**, Formally
    - `f(I,x) = x` and `f(x, I) = x` for all values `x` of type `T`

Propagating Right

- **scan** with clever associative functions can compute things that sound like sequential, but in parallel!

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 one left-to-right).



Left-to-right computation

- We could compute this sequentially by folding from left to right

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

But wait!!

Theorem (Associativity): `take_right_some` is associative.

- Therefore, we can solve Previous SOME in parallel with **`scan`**

Associativity Proof

Theorem (Associativity): `take_right_some` is associative.

- Let `x,y,z` be values of type `optional<T>`.
- WTS that `f(f(x,y),z) = f(x,f(y,z))`

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)

Associativity Proof (continued)

Theorem (Associativity): `take_right_some` is associative.

- Let `x,y,z` be values of type `optional<T>`.
- WTS that `f(f(x,y),z) = f(x,f(y,z))`

x	y	z	f(x,y)	f(y,z)	f(f(x,y),z)	f(x,f(y,z))
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

Parallel Previous SOME

- Since its associative, we can use `take_right_some` with `scan`

```
fun previous_some(S : sequence<optional<T>>) -> sequence<optional<T>>:
  fun take_right_some(a : optional<T>, b : optional<T>) -> optional<T>:
    match b with:
      case SOME(_): return b
      case _: return a
  propagated, _ = scan(take_right_sum, NONE, S)
  return propagated
```

- This gives us, in $O(n)$ work and $O(\log n)$ span, the previous non-NONE option at every position in the sequence

Summary

- **Contraction** differs from divide-and-conquer by reducing a problem to one smaller version of itself, instead of multiple
- Contraction gives us an efficient $O(n)$ work and $O(\log n)$ **implementation of scan**, a highly useful sequence algorithm
- Scan can be used to **compute prefix sums** (or prefix min/max) which can be used to optimize inefficient algorithms
- We can use **custom associative functions** with scan to solve problems that seem inherently sequential (but aren't!)