

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
- Understand **associativity** and its applications in scan and reduce

Algorithmic Techniques

- Brute Force
- Divide-and-Conquer
- **Contraction**
- Dynamic programming
- Greedy

Contraction

What is Contraction

Definition (Contraction): A contraction algorithm solves a problem by **contracting** to generate **one** smaller instance, solving it **recursively**, then **expanding** to solve the original instance.

Base Case: When the problem size is sufficiently small, solve it directly (without further recursion)

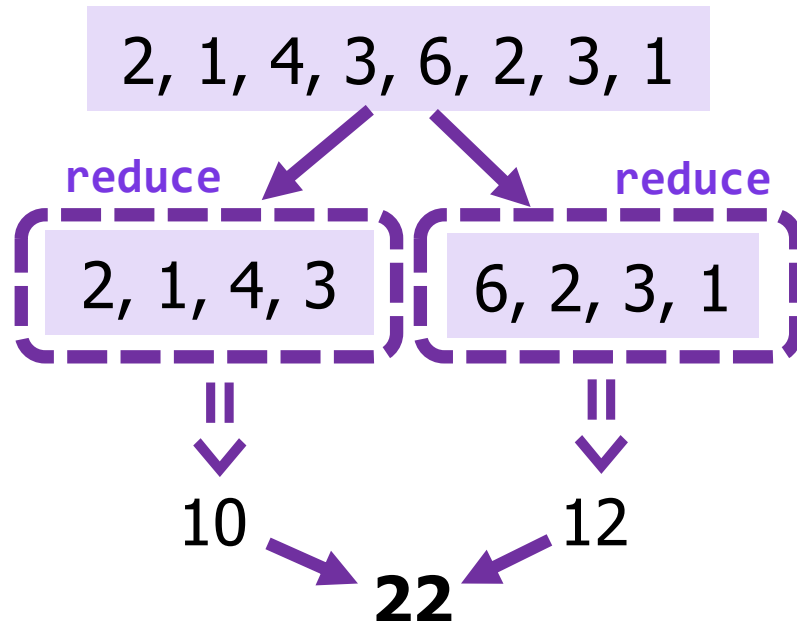
Inductive Step:

1. **Divide Contract** the instance into **one** smaller instance
2. **Recursively** solve the smaller instances
3. **Combine Expand** the solutions to solve the original instance

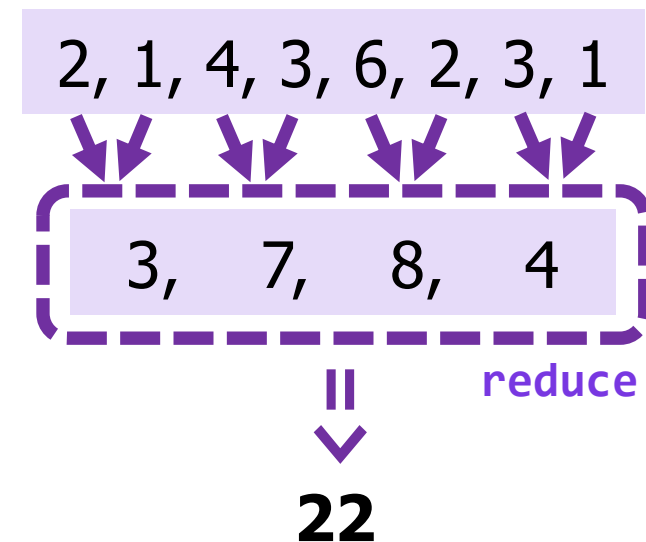
What is Contraction

Definition (Contraction): A contraction algorithm solves a problem by **contracting** to generating **one** smaller instance, solving it **recursively**, then **expanding** to solve the original.

Reduce (Divide-and-conquer)



Reduce (Contraction)



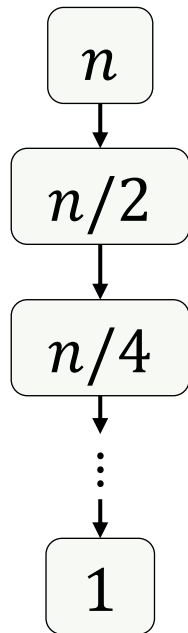
Contraction-based Reduce

```
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]
      + [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**
- **Expansion step** is trivial (identity)

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

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Cost of root: $O(n)$

Span: $S(n) = S(n/2) + O(1)$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Cost across levels: $\lg n \text{ levels} \times O(1)/\text{level} = O(\lg n)$

Scan

Scan

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

$\text{scan} : (f : (T, T) \rightarrow T, I : T, S : \text{sequence}\langle T \rangle) \rightarrow (\text{sequence}\langle T \rangle, T)$

- **Scan returns:**

- A sequence containing $[I, f(I, S[0]), f(f(I, S[0]), S[1]), \dots]$
- The total sum, equivalent to $\text{reduce}(f, I, S)$

Scan

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

$\text{scan} : (f : (T, T) \rightarrow T, I : T, S : \text{sequence}\langle T \rangle) \rightarrow (\text{sequence}\langle T \rangle, T)$

- **E.g.:**

$\text{scan} + 0 \quad [2, 1, 4, 3, 6, 2, 3, 1]$
 $\rightarrow [0, 2, 3, 7, 10, 16, 18, 21], 22$

Seems 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)
```

e.g.

```
scan + 0 [2, 1, 4, 3, 6, 2, 3, 1] ->  
  ([reduce + 0 [],  
    reduce + 0 [2],  
    reduce + 0 [2, 1],  
    reduce + 0 [2, 1, 4],  
    reduce + 0 [2, 1, 4, 3],  
    ...  
  ],  
  reduce(+, 0, [2, 1, 4, 3, 6, 3, 3, 1]))
```

Inefficient Scan

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

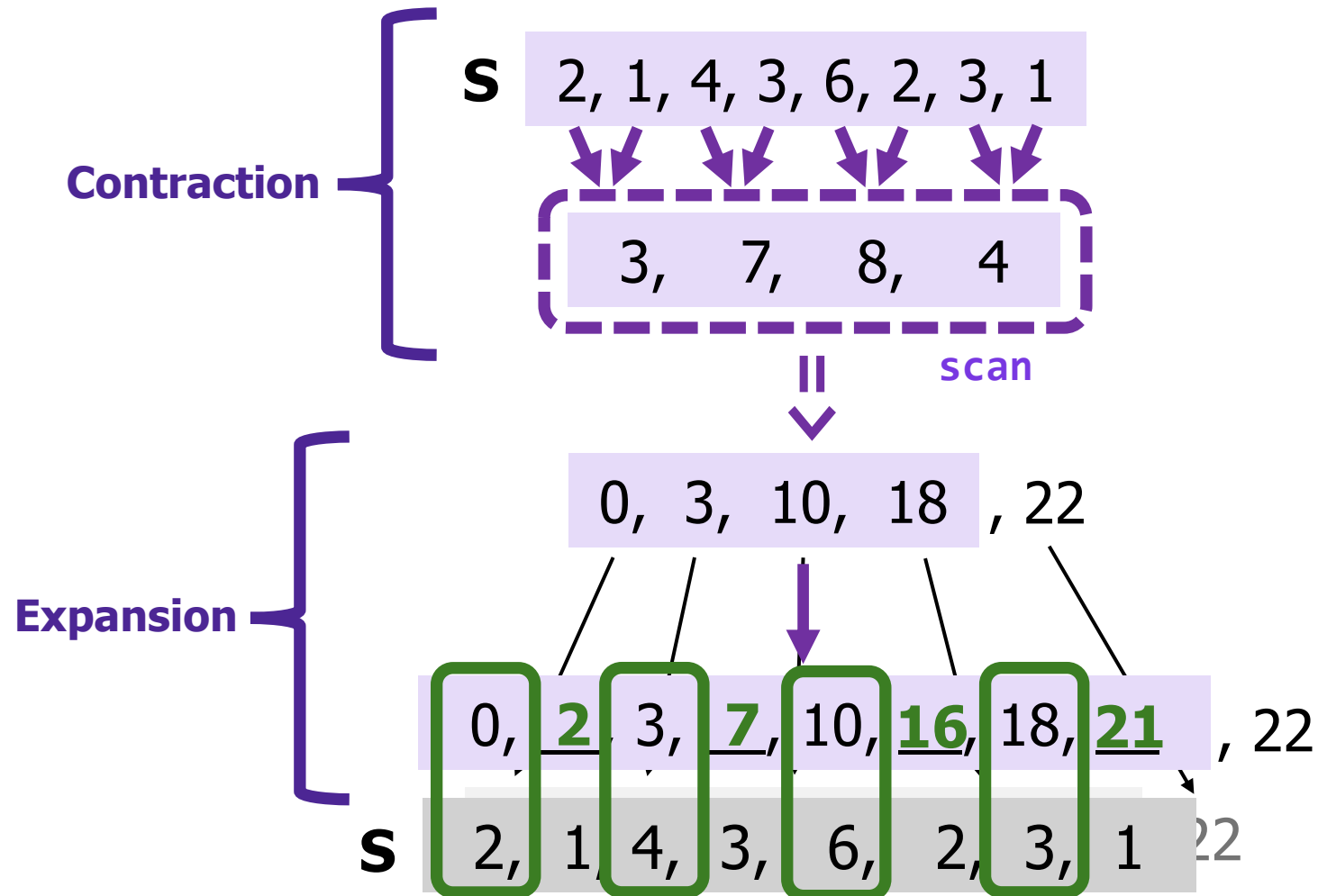
```
fun scan(f : (T, T) -> T, I : T, S : sequence<T>) -> (sequence<T>, T):  
    return tabulate(fn i => reduce(f, I, subseq(S, 0, i)), |S|), reduce(f, I, S)
```

- 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

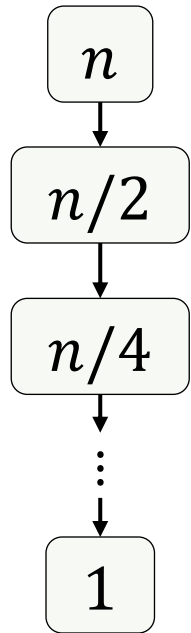


Contraction-based Scan

```
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]  
        + ([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: $W(n) = W\left(\frac{n}{2}\right) + O(n)$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Cost of root: $O(n)$

Span: $S(n) = S(n/2) + O(1)$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Cost across levels: $\lg n \text{ levels} \times O(1)/\text{level} = O(\lg n)$

Associative Functions

Associativity + Identity

- Needed for reduce and scan

Definition (Associative): A binary operator $f : (T, T) \rightarrow T$ is called **associative** if for all x, y, z , we have $f(f(x, y), z) = f(x, f(y, z))$

Definition (Identity): A value I is an **identity** for an associative binary operator f if for any value x , we have $f(I, x) = x$ and $f(x, I) = x$

- Technically (T, f, I) is called a “Monoid”

Examples:

- max, min, +, x, or, and, xor (mentioned in last class)
- append, merge
- union, intersection, symmetric set difference of sets
- matrix multiply, composing permutations
- function composition: $(f(x) \circ g(y)) = f(g(y))$

Note that **not all associative functions are commutative**.
Can you list some of the above that are not commutative?

copy_right

- Consider the following binary function: ©

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

- E.g. $f(\text{SOME}(2), \text{SOME}(5)) \rightarrow \text{SOME}(5)$ $f(\text{NONE}, \text{SOME}(5)) \rightarrow \text{SOME}(5)$
 $f(\text{SOME}(2), \text{NONE}) \rightarrow \text{SOME}(2)$ $f(\text{NONE}, \text{NONE}) \rightarrow \text{NONE}$

Theorem (Associativity): copy_right is associative.

Associativity Proof

Theorem (Associativity): `copy_right` 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): `copy_right` 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

Scan / Reduce With Custom Associative Functions

Recall from MCSSLab

```
fun smcss(S : sequence<int>) -> (int,int,int,int):  
  match length(S) with:  
  case 0: return (0,0,0,0)  
  case 1:  
    m = max(0, A[0])  
    return (m, m, m, A[0])  
  case _:  
    L, R = split_mid(S)  
    (m1,p1,s1,t1), (m2,p2,s2,t2) = parallel (smcss(L), smcss(R))  
    return (max(s1 + p2, m1, m2),  
            max(p1, t1 + p2),  
            max(s2, t2 + s1),  
            t1+t2)
```

Hang on, That's Just Reduce!

```
type sums = (int,int,int,int)

fun combine_smcss((m1,p1,s1,t1) : sums, (m2,p2,s2,t2) : sums) -> sums:
  return (max(s1 + p2, m1, m2),
          max(p1, t1 + p2),
          max(s2, t2 + s1),
          t1+t2)
```

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  fun base(x : int): return (max(0,x),max(0,x),max(0,x),x)
  return reduce(combine_smcss, (0,0,0,0), map(base, S))
```

Recall from Paren Match (Reci)

```
fun excessParens(p : sequence<Paren>) -> (int,int):  
  match length(p) with:  
  case 0: return (0,0)  
  case 1:  
    if p[0] == L: return (0,1)  
    else:         return (1,0)  
  case _:  
    L, R = split_mid(S)  
    (i,j), (k,l) = parallel (excessParens(L), excessParens(R))  
    if j <= k: return (i + k - j, l)  
    else: return (i, l + j - k)
```

It's Reduce Again!

```
fun combine_paren((i : int, j : int), (k : int, l : int)) -> (int, int):  
    return (i + k - j, l) if j <= k else (i, l + j - k)
```

```
fun excessParens(p : sequence<Paren>) -> (int,int):  
    fun base(x : Paren): return (0,1) if x == L else (1,0)  
    return reduce(combine_paren, (0,0), map(base, p))
```

Map+Reduce as "Generic D&C"

```
fun algo(S : sequence<T>):  
  match length(S) with:  
  case 0: return empty  
  case 1: return base(S[0])  
  case _:  
    L, R = split_mid(S)  
    Lres, Rres = parallel (algo(L), algo(R))  
    return combine(Lres, Rres)
```



```
fun algo(s : sequence<T>):  
  return reduce(combine, empty, map(base, s))
```

Map+Reduce as "Generic D&C"

Works under these conditions:

- **The divide step is trivial:** just split input in half and recurse; don't do anything with the halves before recursion
- **The conquer step is an associative function** combine (along with an appropriate identity) which combines the left and right result
- The base case is a pure function of the single element $S[0]$

```
fun algo(S : sequence<T>):  
  match length(S) with:  
  case 0: return empty  
  case 1: return base(S[0])  
  case _:  
    L, R = split_mid(S)  
    Lres, Rres = parallel (algo(L), algo(R))  
    return combine(Lres, Rres)
```

Reduce Merge

Recall: Merge is associative

- Therefore, we can reduce it!

Note: singleton(x) returns [x]

```
reduce(merge, [], map(singleton, s))
```

```
reduce(merge, [[8], [5], [7], [4], [1], [3], [9], [2]], [9], [2])
```

Question: What algorithm is this?

Answer. Its **MergeSort**! In just one line of code!

Merge but in a different order?

- `fold_left(f : (A, T) -> A, I : A, S : sequence<T>) -> A` (called `Seq.iterate` in the 210 SML library) returns the same answer as `reduce` if $T = A$, f is associative, and I an identity.
- So, what happens if we replace `reduce` with `fold_left`?

```
fold_left(merge, [], map(singleton, s))
```

Question: What algorithm is this?

Answer: Its Insertion Sort!

Scan Merge

Question: What happens if we replace reduce with scan?

- What does the following code produce:

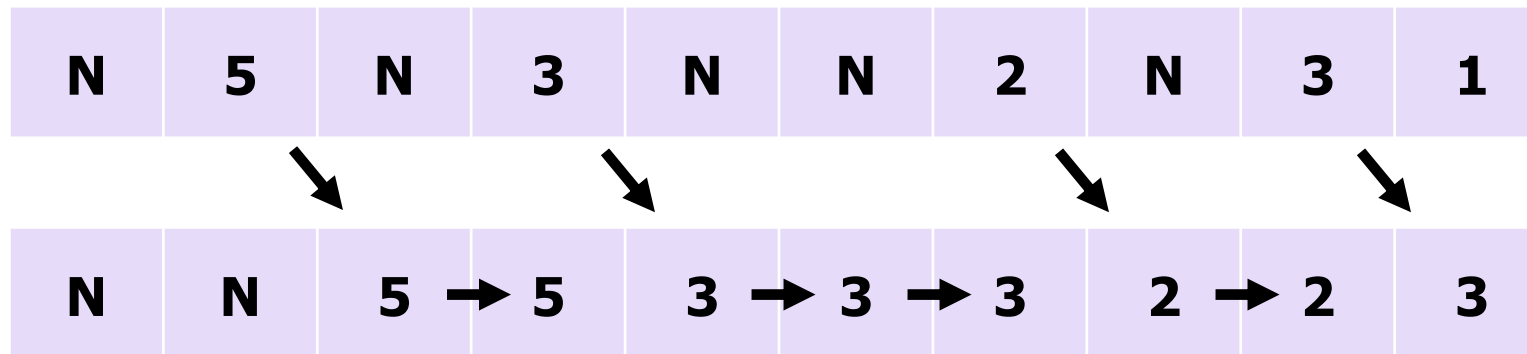
```
scan(merge, [], map(singleton, s))
```

Answer: Sorted copy of **every prefix** of *s* and the fully sorted *s*

Propagating Right

- **scan** with clever associative functions can compute things that appear to be sequential, but in parallel!

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



Hint:

- Recall the binary function from earlier: ©

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

- E.g. $f(\text{SOME}(2), \text{SOME}(5)) \rightarrow \text{SOME}(5)$ $f(\text{NONE}, \text{SOME}(5)) \rightarrow \text{SOME}(5)$
 $f(\text{SOME}(2), \text{NONE}) \rightarrow \text{SOME}(2)$ $f(\text{NONE}, \text{NONE}) \rightarrow \text{NONE}$
- Remember, `copy_right` is associative!

Parallel Previous SOME

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

```
fun previous_some(S : sequence<optional<T>>) -> sequence<optional<T>>:  
  fun copy_right(a : optional<T>, b : optional<T>) -> optional<T>:  
    match b with:  
      case SOME(_): return b  
      case _: return a  
  propagated, _ = scan(copy_right, 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
- We can use **custom associative functions** with scan to solve problems that seem inherently sequential (but aren't!)