Parallel And Sequential Data Structures and Algorithms

Algorithms for Sequences

Learning Objectives

- Understanding the sequence ADT and how it can be implemented using arrays
- Implement core Sequence operations like **filter** and **flatten** efficiently

Recall: Sequence ADT

Definition (Sequence): A sequence of length n over elements of type T is an ordered collection of values that can be viewed as a mapping from the indices

$$\{0,1,\ldots,n-1\} \to T$$

Interface (Sequence): A `sequence<T>` (with value type T) supports

- `nth(S : sequence<T>, i : int) -> T`:
  returns the i^{th} element of the sequence S
- `length(S : sequence<T>) -> int`:
  return the length of the sequence S
- `subseq(S : sequence<T>, i : int, k : int) -> sequence<T>`:
  returns a view of the subsequence of S starting at index i with length k

Recall: Creating Array Sequences

- Assume that the sequences we construct are **`ArraySequence<T>`**, a contiguous fixed-size array, which supports $O(1)$ time operations
- This is the type we will assume is returned by the **`tabulate`** primitive, which constructs a sequence from a function

```
tabulate : (f : (int -> T), n : int) -> ArraySequence<T>
```

- `tabulate(f, n)` returns a sequence of length n where $S[i] = f(i)$, i.e.,

$$[f(0), f(1), \dots, f(n-1)].$$

- We may also use Python-like syntax in our pseudocode, e.g., we may write

```
parallel [f(i) for i in 0…n-1]
```

Arrays and Imperative Parallelism

Arrays: Building Block for Array Sequences

- We will assume that arrays support the following operations:
  - **`allocate<T>(n)`**: allocate an array of length n of type T
  - **`A[i]`**: return the i^{th} element of A
  - **`|A|`**: return the length of A
  - **`A[i] ← x`**: write the value x to the i^{th} element of A

Remark: Arrays are **mutable**! We will need mutation to implement low level primitives.

Definitions

Definition (Data Race): A **data race** occurs when there are two unsynchronized parallel operations on the same memory location, and at least one of the operations is a write.

- If two operations are both reads, this is not a data race.
- This can’t happen in a purely functional program.
- In this class, we will say the presence of data race makes the behavior of the program undefined.

Imperative Parallel Loops

Definition (Parallel For Loop): A parallel for loop executes its body for every value of its range in parallel.

```
parallel for i in 0…n-1:
  e(i)
```

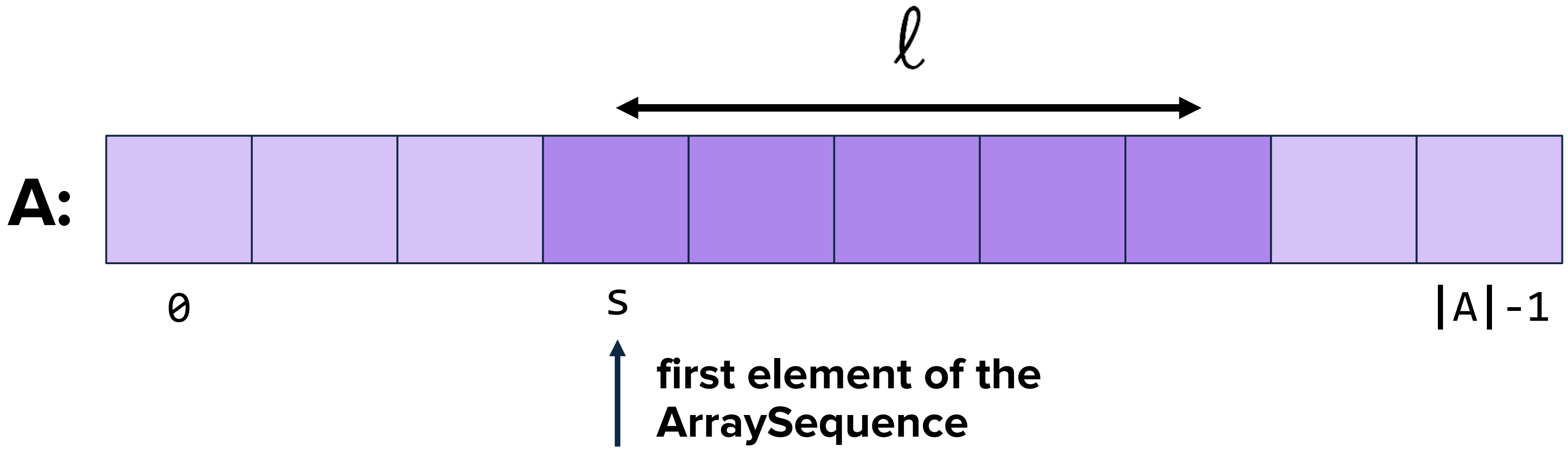
where `e(i)` is some (impure) function of `i`

- This could cause data races.
- The work and span are: $W = \sum_i W(e(i)), \qquad S = \max_i S(e(i))$

Data Type for ArraySequence

ArraySequences

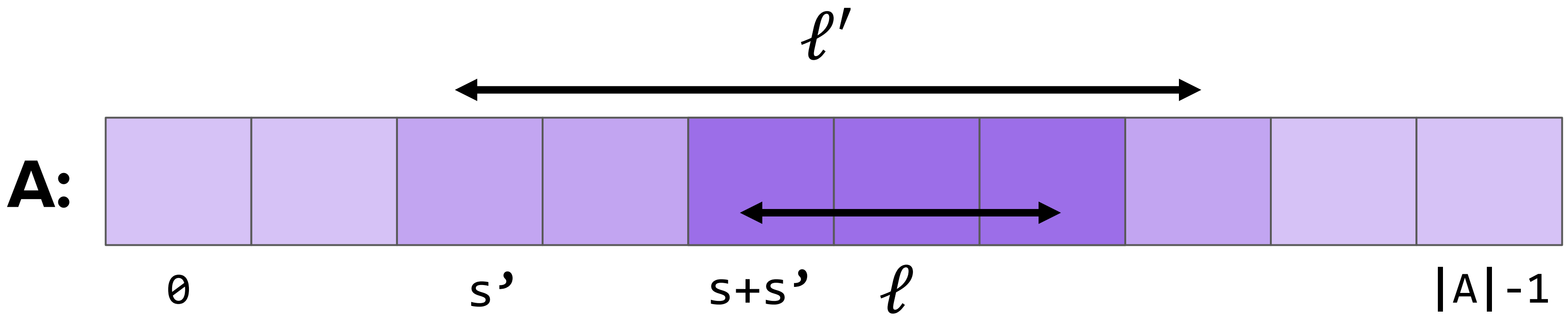
```
type ArraySequence<T> = {
    A : Array<T> // actual Array A
    s : int      // initially 0
    ℓ : int      // initially |A|
}
```



Basic Functions

subseq

```
fun subseq (S : ArraySequence<T>, s : int, ℓ : int) -> ArraySequence<T>:
    (A,s’,ℓ’) = S
    return (A,s+s’,ℓ)
```



tabulate

```
fun tabulate (f : (int -> T), n : int) -> ArraySequence<T>:
  R = allocate<T>(n)
  parallel for i in 0...n-1:
    R[i] ← f(i)
  return (R, 0, n)
```

f(0)	f(1)	f(2)	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)	f(9)

$$W = \sum_{0 \le i < n} W(f(i))$$

$$S = \max_{0 \le i < n} S(f(i))$$

Question: Is there a data race?

Answer: No! Each parallel execution of the loop body operates on a different element of R

Sequence Functions

map

```
fun map (f : (T -> U), S : sequence<T>) -> ArraySequence<U>:
  return tabulate(fn i => f(S[i]), |S|)
```

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]

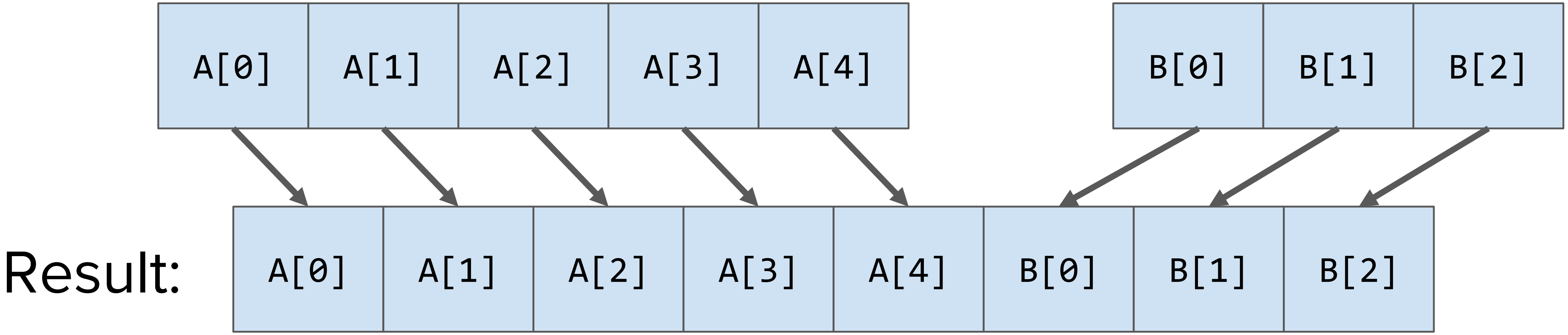
↓

f(S[0])	f(S[1])	f(S[2])	f(S[3])	f(S[4])	f(S[5])	f(S[6])	f(S[7])	f(S[8])	f(S[9])

$$W = \sum_{x \in S} \mathcal{W}(f(x)) \qquad S = \max_{x \in S} \mathcal{S}(f(x))$$

append

```
fun append (A : sequence<T>, B : sequence<T>) -> ArraySequence<T>:
    return tabulate (fn i => (A[i] if i<|A| else B[i-|A|]), |A|+|B|)
```



W = $O(|A| + |B|)$ S = $O(1)$

filter

Definition (Filter):

```
filter : (p : (T -> bool), S : sequence<T>) -> ArraySequence<T>
```

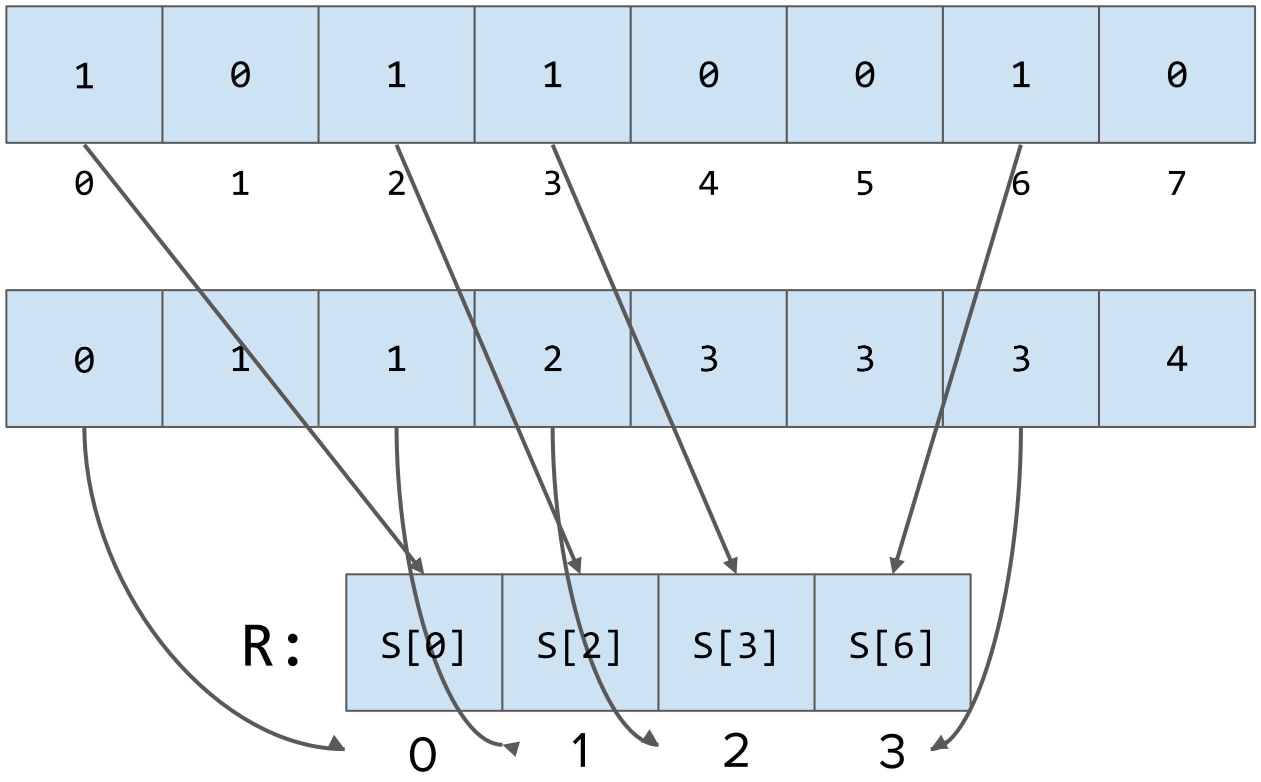
`filter(p, S)` returns a sequence consisting of the elements of S which satisfy the predicate p. The relative order of the elements is preserved.

Example:

`filter (fn x => x < 5, [7,1,3,11,7,2])` returns [1, 3, 2]

filter

Goal: implement filter in linear work and logarithmic span




```
F = map(fn x => 1 if p(x) else 0, S)

x,ℓ = scan(plus, 0, F)

if F[i] == 1: R[x[i]] ← S[i]
```

filter implementation

```
fun filter (p : (T -> bool), S : Sequence<T>) -> ArraySequence<T>:
  F = map (fn x => 1 if p(x) else 0, S)
  X, ℓ  = scan(plus, 0, F)
  R = allocate<T>(ℓ)
  parallel for i in 0...|S|-1:
    if F[i] == 1:
      R[x[i]] ← S[i]
  return (R,0,ℓ)
```

W = $\sum_{x \in S} \mathcal{W}(p(x))$ S = $O(\log |S|) + \max_{x \in S} \mathcal{S}(p(x))$

flatten

Definition (Flatten):

```
flatten: (S : sequence<sequence<T>>) -> ArraySequence<T>
```

Given a nested sequence of sequences, return a sequence consisting of all the elements of the inner sequences in the same relative order.

Example (Flatten):

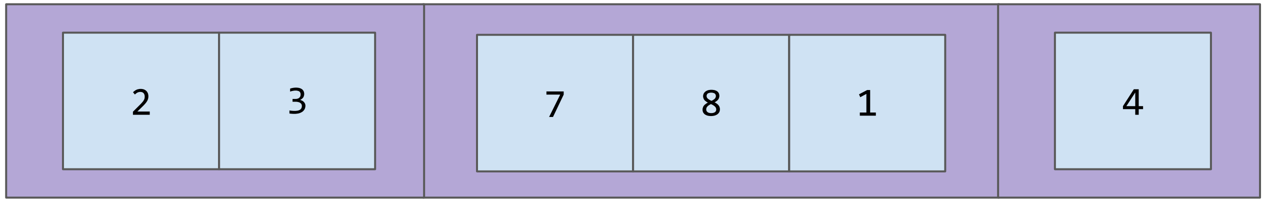
```
flatten([[2,3],[7,8,1],[4]]) returns [2,3,7,8,1,4]
```

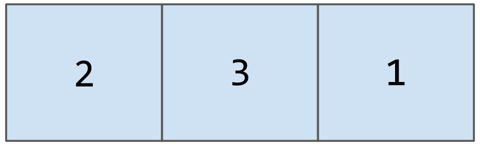
Question: Any ideas how to do this?

Answer: We could use scan again!

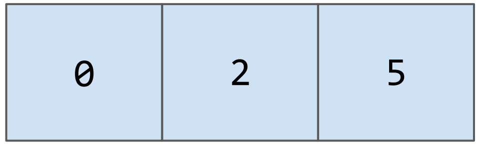
flatten

S:

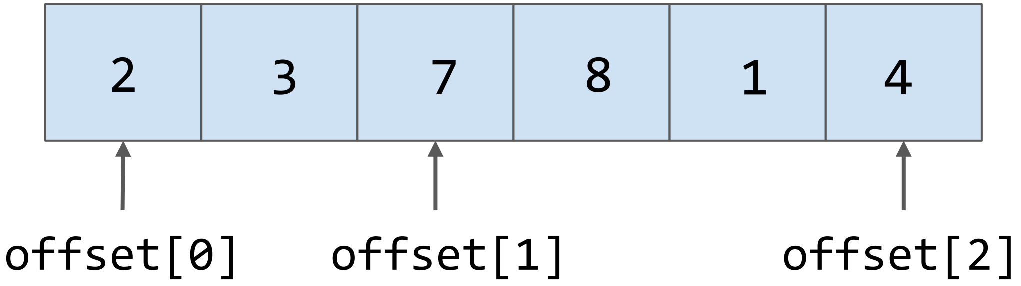





```
L = map (fn x=> |x|, S)  // length of inner sequences
```




```
offset, length = scan(plus, 0, L)
```




```
R[offset[i]+j] ← S[i][j]
```

flatten implementation

```
fun flatten (S : sequence<sequence<T>>) -> ArraySequence<T>:
   L = map (fn x => |x|, S)
   offset, length = scan(plus, 0, L)
   R = allocate<T>(length)
   parallel for i in 0...|S|-1:
     parallel for j in 0...L[i]-1:
       R[offset[i]+j] ← S[i][j]
   return (R, 0, length)
```

$$W = O\left(\sum_{x \in S} (1 + |x|)\right) \qquad S = O(\log |S|)$$

Practice!

collate

Definition (Collate):

```
collate : ((T, T) -> order), sequence<T>, sequence<T>) -> order
```

`collate(f, s1, s2)` returns the lexicographical comparison of sequences `s1` and `s2` using the comparison function `f`

Examples (Collate):

```
[1,5,1,2,2] vs [1,5,2,1,0] → LESS (first difference at index 2)
[1,5,2,1,0] vs [1,5,2,1] → GREATER (longer sequence)
["a","b","c"] vs ["a","b","c"] → EQUAL
```

collate

Question: Any ideas how to implement collate, within the costs:

- Work: $O(\min(m, n))$,
- Span: $O(\log(\min(m, n)))$

where $m = |s_1|$ and $n = |s_2|$

Answer: We could use reduce!

collate

```
type order = LESS | EQUAL | GREATER

fun collate(f: (T,T)->order, a: sequence<T>, b: sequence<T>) -> order:
  pairs = zip(a, b)
  cmps = map(f, pairs)  // compare corresponding pairs in a and b
  fun first_notequal(x, y):
    return (y if x == EQUAL else x)  // propagate leftmost non-EQUAL
  res = reduce(first_notequal, EQUAL, cmps)
  match res with:
    case EQUAL: return compare(|a|, |b|)
    case _:     return res
```

Optimized collate

Note: The obvious sequential algorithm will just compare the two sequences starting from left to right until a difference is found. Let d be the number of comparisons it does. So, the work is $O(\mathrm{d})$. This could be much less than $O(\min(m, n))$.

Can you think of how to achieve a parallel algorithm with

- Work: $O(d)$
- Span: $O(\log^2(d))$

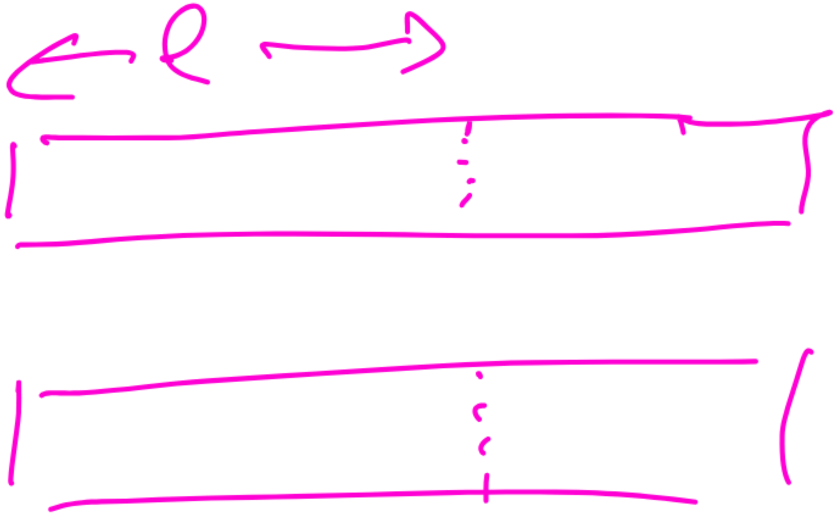
Hint: Remember the **doubling trick** for dynamic arrays

Optimized collate (Solution)

- Recall the trick from dynamic arrays (15-122): when you run out of capacity, double the size. This is efficient because

$$O\left(n + \frac{n}{2} + \frac{n}{4} + \cdots + 1\right) = O(n)$$

- We can use the same trick and do a "doubling search":
  - Check whether `subseq(a,0,1)` and `subseq(b,0,1)` are equal, then `subseq(a,0,2)` and `subseq(b,0,2)`, then `subseq(a,0,4)` and `subseq(b,0,4)`, and so on... until they are not equal
  - This takes $O\left(2d + d + \frac{d}{2} + \frac{d}{4} + \ \ldots + 1\right) = O(d)$ work



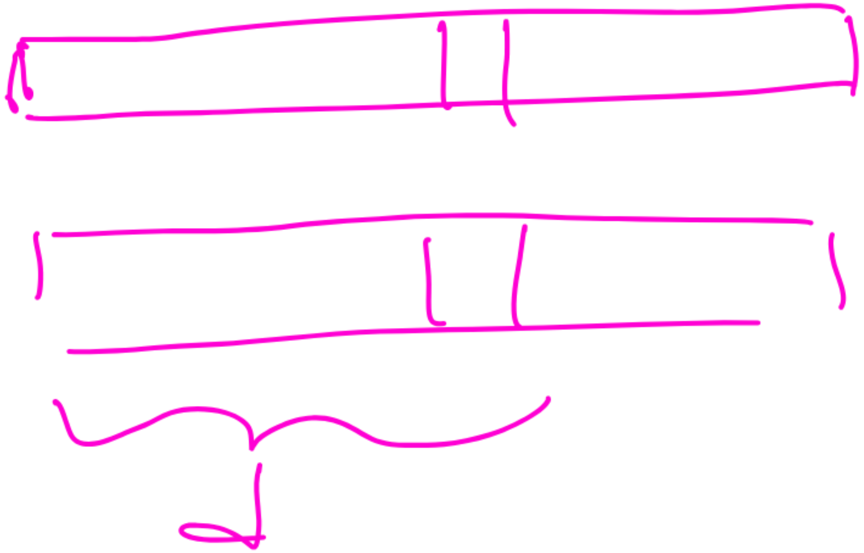
span

$\log 1 + \log 2 + \cdots \log 2d$

$0 + 1 + 2 + \cdots \log 2d$

$O(\log^2 d)$

$l \to 2l$



$l = 1, 2, 4 \ldots 2^{k-1}, 2^k$

$\sum l_i \leq 4d$

$2^{k-1} < d$, $2^k \geq d$

$d \leq 2^k < 2d$

Summary

- Efficiently implementing sequences with arrays requires some imperative (non-functional) parallelism!
- Core sequence operations like **filter** and **flatten** can be implemented efficiently as applications of scan