CSE 332
Data Structures & Parallelism

Parallel Prefix Sum

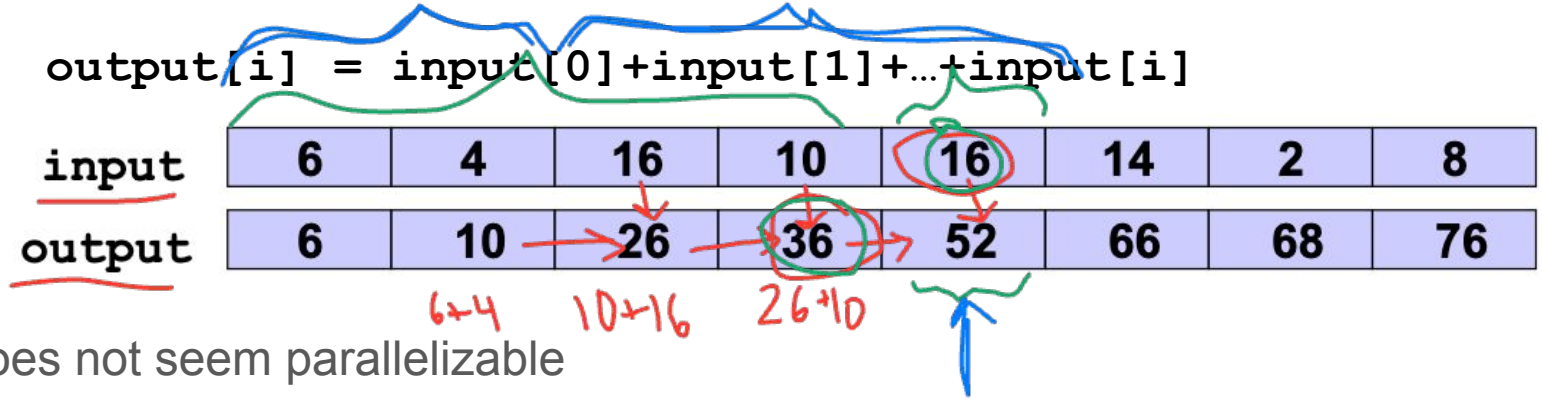
Melissa Winstanley

Spring 2024

The prefix-sum problem

Given `int[] input`, produce `int[] output` where:

`output[i] = input[0]+input[1]+…+input[i]`



Does not seem parallelizable

- Work: O(n), Span: O(n)
- This algorithm is sequential, but a different algorithm has Work: O(n), Span: O(log n

The prefix-sum problem

Sequential can be an intro to CS exam problem

input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

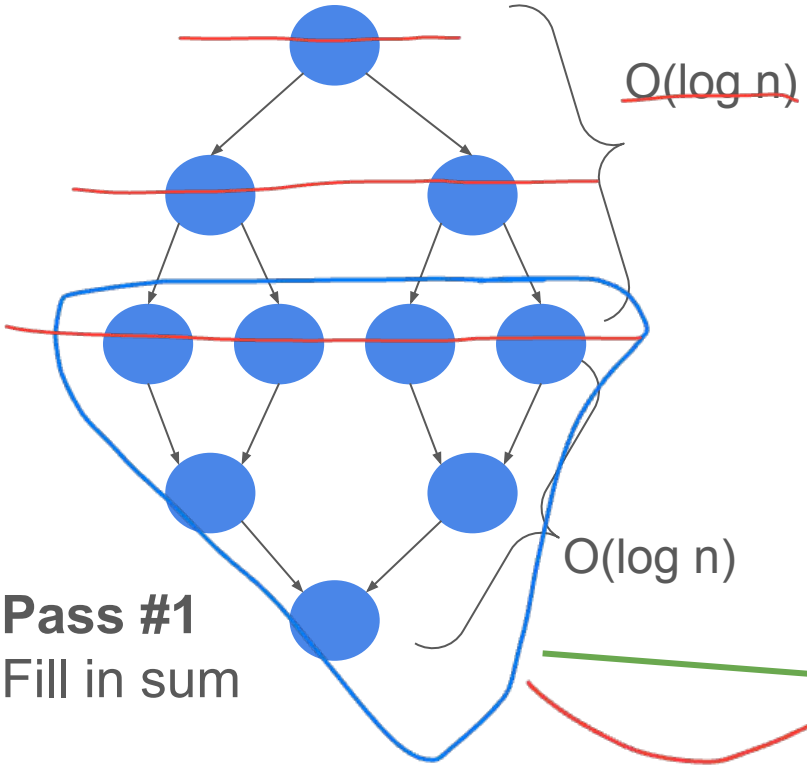
```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Parallel prefix-sum

- The parallel-prefix algorithm does two passes
  - Each pass has O(n) work and O(log n) span
  - So in total there is O(n) work and O(log n) span
  - So like with array summing, parallelism is n/log n
    - An exponential speedup

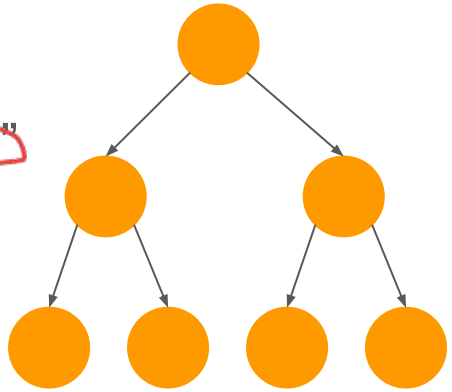
- First pass builds a tree bottom-up: the “up” pass
- Second pass traverses the tree top-down: the “down” pass

The Passes

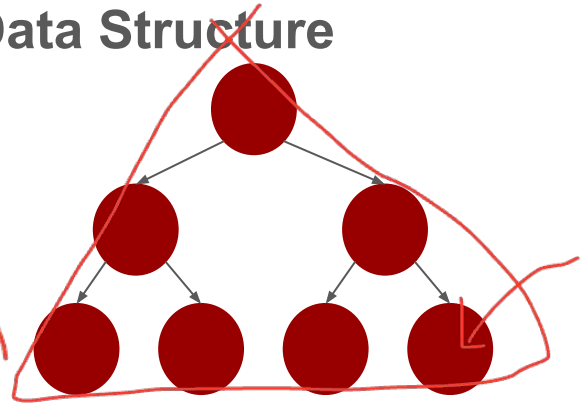


Pass #1
Fill in sum

Pass #2
Fill in
"From Left"



Data Structure



output

Parallel Prefix: The Up Pass

We build want to build a binary tree where

- Root has sum of the range [x,y)
- If a node has sum of [lo,hi) and hi>lo,
  - Left child has sum of [lo,middle)
  - Right child has sum of [middle,hi)
  - A leaf has sum of [i,i+1), which is simply input[i]

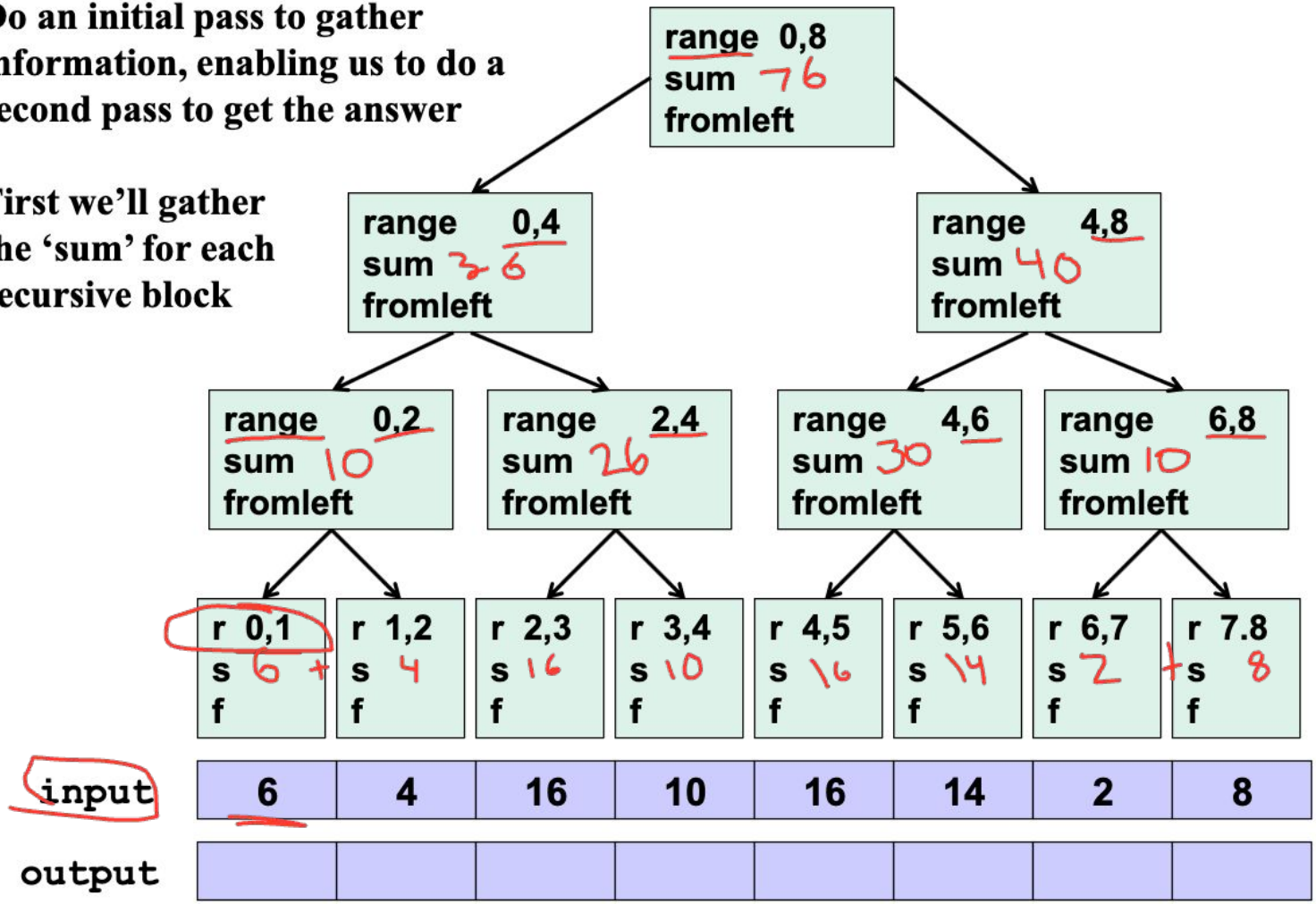
It is critical that we actually create the tree as we will need it for the down pass

- We do not need an actual linked structure
- We could use an array as we did with heaps

Analysis of first step: Work = Span =

The (completely non-obvious) idea:
Do an initial pass to gather information, enabling us to do a second pass to get the answer

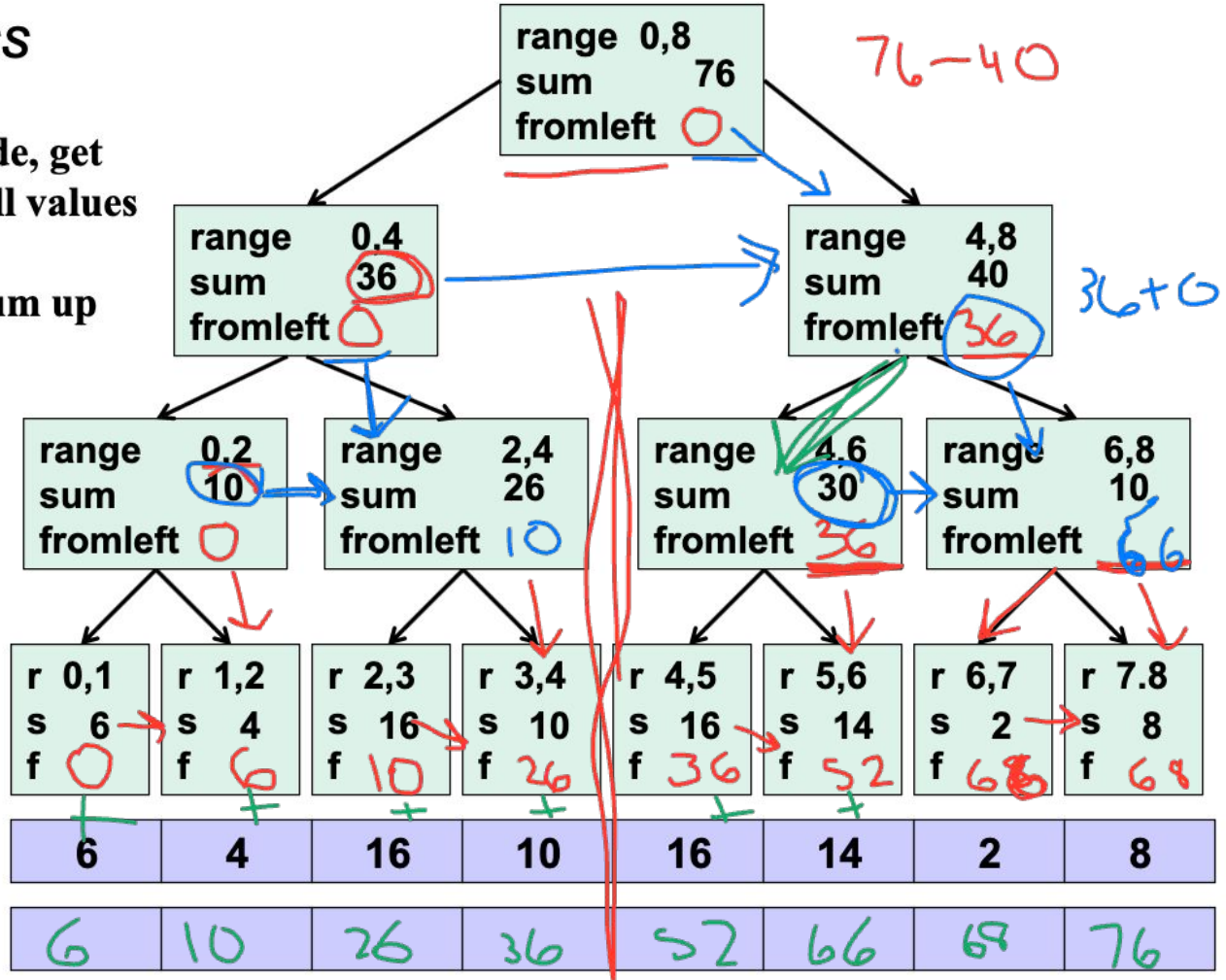
First we'll gather the 'sum' for each recursive block



First pass

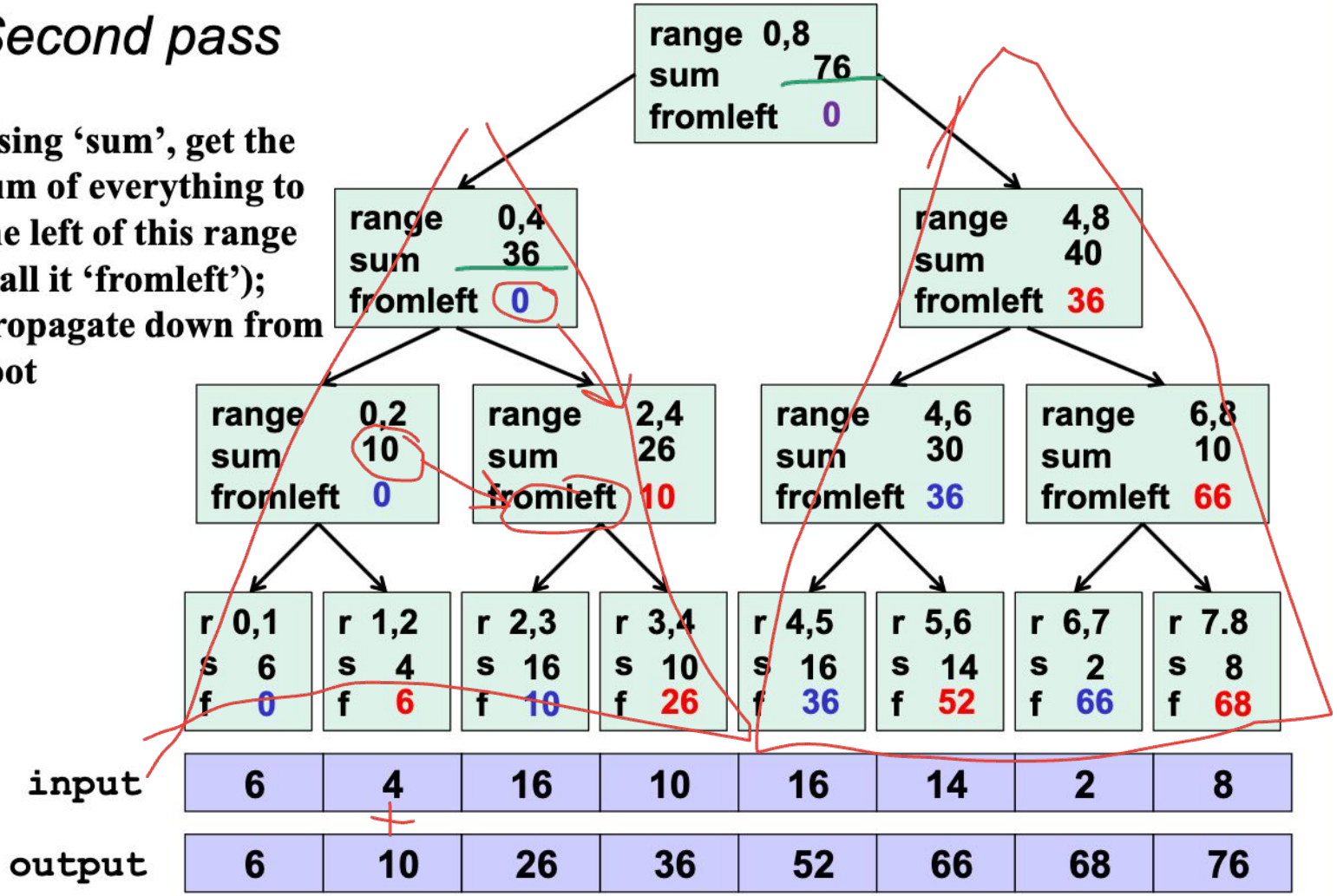
For each node, get the sum of all values in its range; propagate sum up from leaves

Will work like parallel sum, but recording intermediate information



Second pass

Using 'sum', get the sum of everything to the left of this range (call it 'fromleft'); propagate down from root



Parallel prefix, generalized

Just as sum-array was the simplest example of a common pattern, prefix-sum illustrates a pattern that arises in many, many problems

- Minimum, maximum of all elements to the left of `i`
- Is there an element to the left of `i` satisfying some property?
- Count of elements to the left of `i` satisfying some property
  - This last one is perfect for an efficient parallel pack…
  - Perfect for building on top of the “parallel prefix trick”

Pack (think “Filter”)

[Non-standard terminology]

Given an array `input`, produce an array `output` containing only elements such that `f(element)` is `true`

Example:

```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]

f: “is element > 10”

output [17, 11, 13, 19, 24]
```

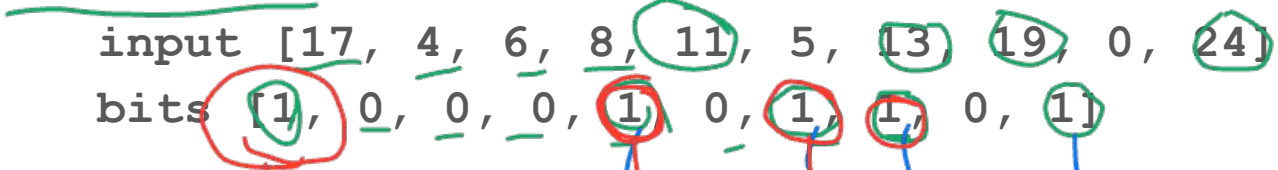
Parallelizable?

- Determining *whether* an element belongs in the output is easy
- But determining *where* an element belongs in the output is hard; seems to depend on previous results….

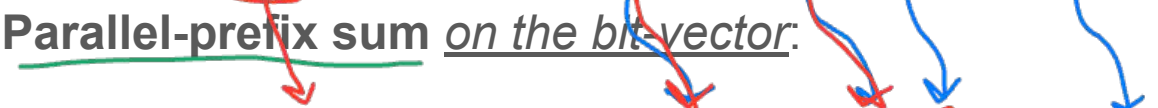
Parallel Pack = (Soln)
parallel map + parallel prefix + parallel map

In this example, Filter = element > 10

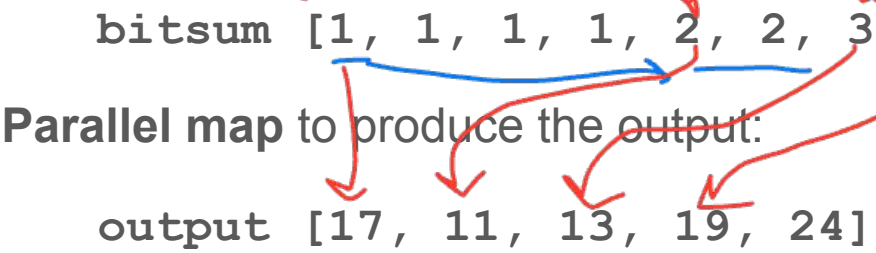
1. **Parallel map** to compute a bit-vector for true elements:



2. **Parallel-prefix sum** *on the bit-vector*:



3. **Parallel map** to produce the output:



Parallel Pack = (Soln)
parallel map + parallel prefix + parallel map

In this example, Filter = element > 10

1. **Parallel map** to compute a bit-vector for true elements:

   `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`

   `bits [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]`

2. **Parallel-prefix sum** *on the bit-vector*:

   `bitsum [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]`

3. **Parallel map** to produce the output:

   `output [17, 11, 13, 19, 24]`

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1]=input[i]
}
```

Parallel Pack = (Soln)
parallel map + parallel prefix + parallel map

In this example, Filter = element > 10

1. **Parallel map** to compute a bit-vector for true elements:

```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits  [1,  0, 0, 0, 1,  0, 1,  1,  0, 1]
```

2. **Parallel-prefix sum** *on the bit-vector*:

```
bitsum [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]
```

1-1=0 2-1=1

3. **Parallel map** to produce the output:

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if (bits[i] == 1)
    Output[bitsum[i]-1] = input[i]
}
```

Parallel Pack = (Soln)
parallel map + parallel prefix + parallel map

In this example, Filter = element > 10

1. **Parallel map** to compute a bit-vector for true elements:

```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]
```

Span = O(log n)

2. **Parallel-prefix sum** *on the bit-vector*:

```
bitsum [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]
```

Span = O(log n)

3. **Parallel map** to produce the output:

```
output [17, 11, 13, 19, 24]
```

Span = O(log n)

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if (bits[i] == 1)
    Output[bitsum[i]-1] = input[i]
}
```

OVERALL = O(log n)

Sequential Quicksort review

Recall quicksort was sequential, in-place, expected time O(n log n)

Best / expected case work

1. **Pick a pivot element** — **O(1)**
2. **Partition all the data into:** — **O(n)**
   A. **The elements less than the pivot**
   B. **The pivot**
   C. **The elements greater than the pivot**
3. **Recursively sort A and C** — **2T(n/2)**

Recurrence (assuming a good pivot): T(0)=T(1)=1

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

Run-time: O(nlogn)

How should we parallelize this?

Parallel Quicksort (version 1)

Best / expected case work

1. **Pick a pivot element** — **O(1)**
2. **Partition all the data into:** — **O(n)**
   A. **The elements less than the pivot**
   B. **The pivot**
   C. **The elements greater than the pivot**
3. **Recursively sort A and C** — ~~**2**~~**T(n/2)**

First: Do the two recursive calls in parallel

- **Work**: O(nlogn)
- **Span**: T(n) = O(n) + T(n/2) → O(n)

Review: Really common recurrences

Should know how to solve recurrences but also recognize some really common ones:

T(n) = O(1) + T(n-1)	linear
T(n) = O(1) + 2T(n/2)	linear
T(n) = O(1) + T(n/2)	logarithmic
T(n) = O(1) + 2T(n-1)	exponential
T(n) = O(n) + T(n-1)	quadratic
T(n) = O(n) + T(n/2)	linear
T(n) = O(n) + 2T(n/2)	O(n log n)

Note big-Oh can also use more than one variable

- Example: can sum all elements of an n-by-m matrix in O(nm)

Pack (think “Filter”)

[Non-standard terminology]

Given an array **`input`**, produce an array **`output`** containing only elements such that **`f(element)`** is **`true`**

Example:

```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]

f: “is element > 10”

output [17, 11, 13, 19, 24]
```

Parallelizable?

- Determining *whether* an element belongs in the output is easy
- But determining *where* an element belongs in the output is hard; seems to depend on previous results….

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two packs!

- We know a pack is O(n) work, O(log n) span
- Pack elements less than pivot into left side of **`aux`** array
- Pack elements greater than pivot into right size of **`aux`** array
- Put pivot between them and recursively sort
- With a little cleverness, can do both packs at once (!) but no big-O change

With O(log n) span for partition, the total span for quicksort is

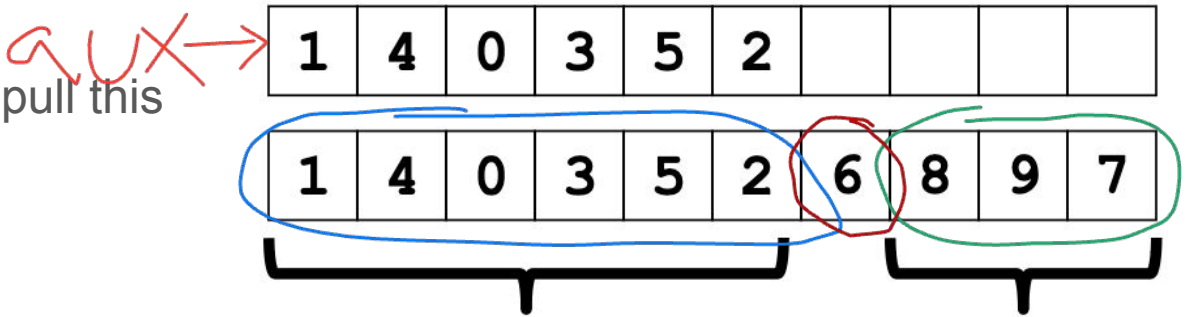
T(n) = O(log n) + T(n/2)

Parallel Quicksort Example (version 2)

Step 1: pick pivot as median of three

Steps 2a and 2c (combinable): pack less than pivot, then pack greater than pivot into a second array

- Fancy parallel prefix to pull this off (not shown)



Step 3: Two recursive sorts in parallel

- Can sort back into original array (like in mergesort)

Parallelize Mergesort?

Recall mergesort: sequential, **not**-in-place, worst-case O(n log n)

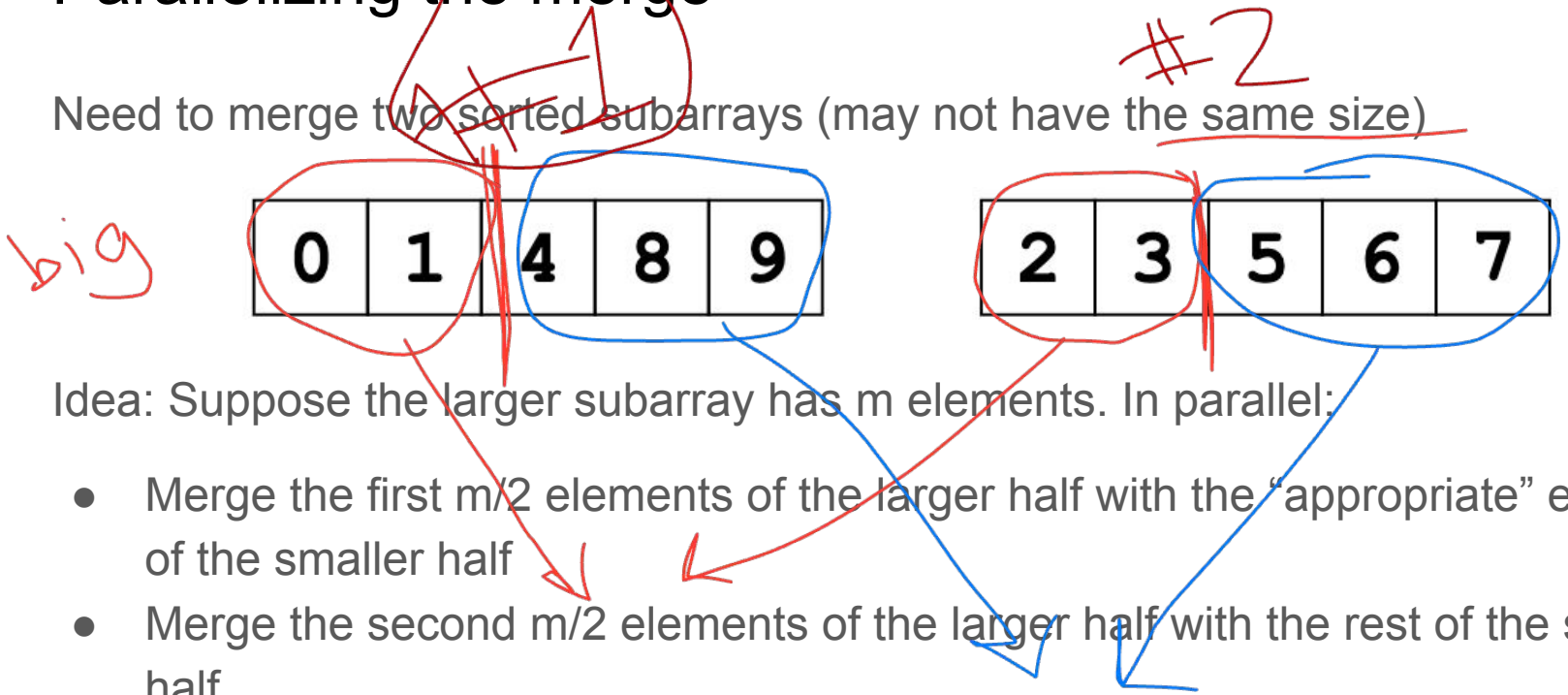
1. **Sort left half and right half** **2T(n/2)**
2. **Merge results** **O(n)**

Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the **Span** to *T(n) = O(n) + 1T(n/2) = O(n)*

- Again, **Work** is O(nlogn), and
- parallelism is **work/span** = O(log n)
- To do better, *need to parallelize the merge*
  - The trick won't use parallel prefix this time…

Parallelizing the merge

Need to merge two sorted subarrays (may not have the same size)



Idea: Suppose the larger subarray has m elements. In parallel:

- Merge the first m/2 elements of the larger half with the “appropriate” elements of the smaller half
- Merge the second m/2 elements of the larger half with the rest of the smaller half

Parallelizing the merge (in more detail)

Need to merge two ***sorted*** subarrays (may not have the same size)
Idea: Recursively divide subarrays in half, merge halves in parallel

Suppose the larger subarray has *m* elements. In parallel:

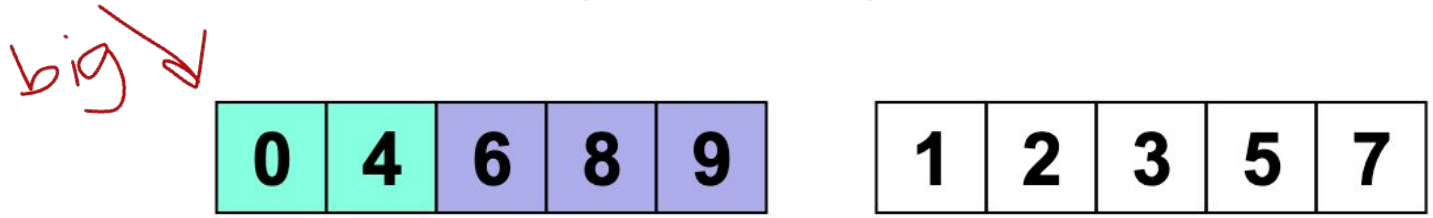
- Pick the **median** element of the larger array (here 6) in constant time
- In the other array, use binary search to find the first element greater than or equal to that median (here 7)

Then, in parallel:

- Merge half the larger array (from the median onward) with the upper part of the shorter array
- Merge the lower part of the larger array with the lower part of the shorter array

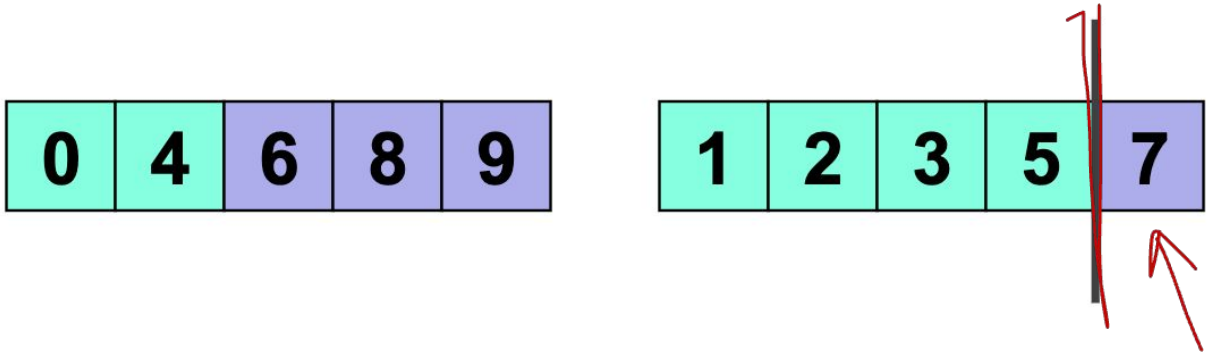
Example: Parallelizing the merge

Example: Parallelizing the merge



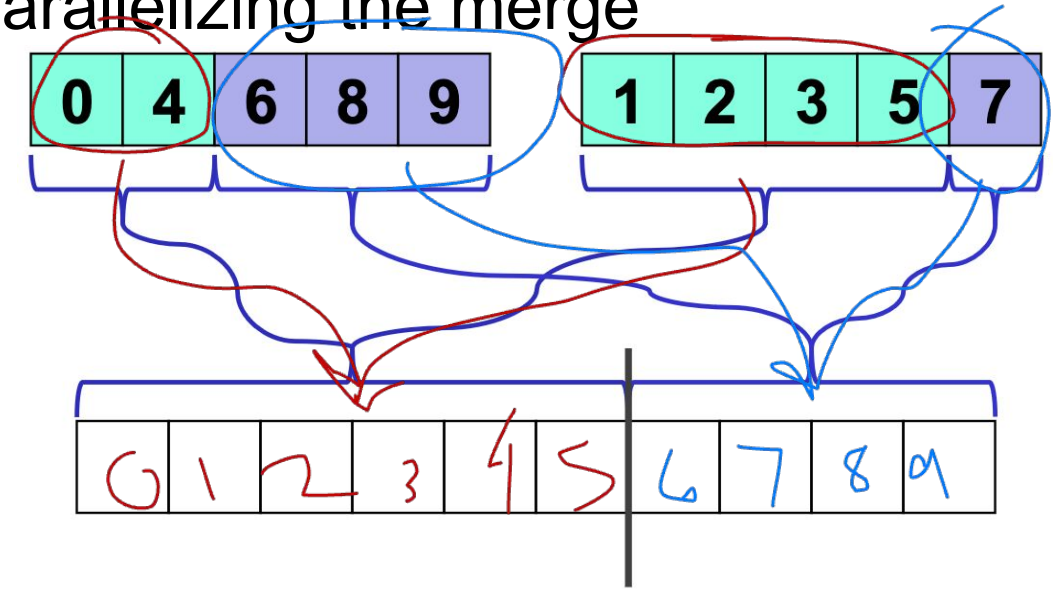
1. Get median of bigger half: O(1) to compute middle index

Example: Parallelizing the merge



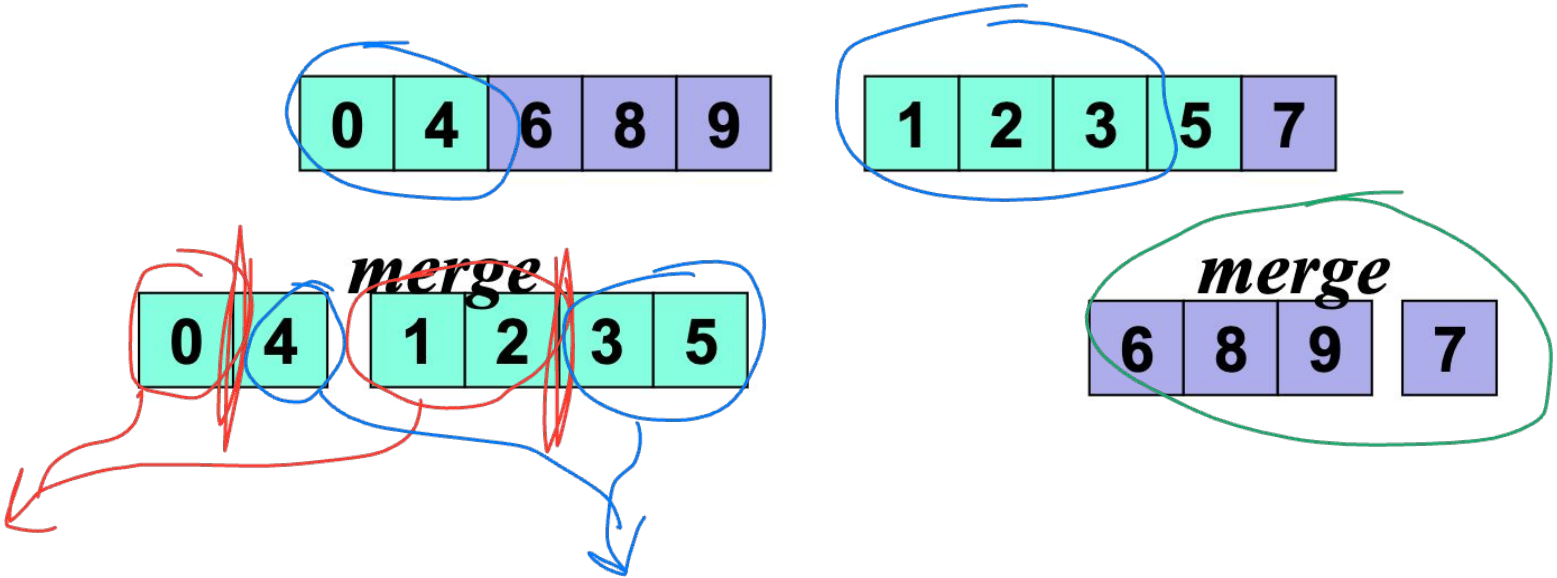
1. Get median of bigger half: O(1) to compute middle index
2. **Find how to split the smaller half at the same value: O(log n) to do binary search on the sorted small half**

Example: Parallelizing the merge



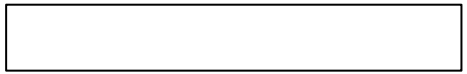
1. Get median of bigger half: O(1) to compute middle index
2. Find how to split the smaller half at the same value:
   O(log n) to do binary search on the sorted small half
3. **Size of two sub-merges conceptually splits output array: O(1)**

Example: Parallelizing the merge



1. Get median of bigger half: O(1) to compute middle index
2. Find how to split the smaller half at the same value:
   O(log n) to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: O(1)
4. **Do two submerges in parallel**

Parallel Merge Pseudocode




```
Merge(arr[], left1, left2, right1, right2, out[], out1, out2)
    int leftSize = left2 – left1
    int rightSize = right2 – right1
    // Assert: out2 – out1 = leftSize + rightSize
    // We will assume leftSize > rightSize without loss of generality
    if (leftSize + rightSize < CUTOFF)
        sequential merge and copy into out[out1..out2]
    int mid = (left2 – left1)/2
    binarySearch arr[right1..right2] to find j such that
        arr[j] ≤ arr[mid] ≤ arr[j+1]
    Merge(arr[], left1,     mid, right1, j,        out[], out1,          out1+mid+j)
    Merge(arr[], mid+1, left2, j+1,    right2, out[], out1+mid+j+1, out2)
```

Analysis

- Doing the *two recursive calls* in parallel but a *sequential merge*:

  Work: same as sequential O(n log n)
  Span: T(n)=**1**T(n/2)+O(n) which is O(n)

- *Parallel merge* makes **work** and **span** harder to compute…
  - Each merge step does an extra O(log n) binary search to find how to split the smaller subarray
  - To merge *n* elements total, do two smaller merges of possibly different sizes
  - But worst-case split is (3/4)*n* and (1/4)*n*
    - Happens when the two subarrays are of the same size (n/2) and the "smaller" subarray splits into two pieces of the most uneven sizes possible: one of size n/2, one of size 0

Analysis continued

For **just** a parallel merge of n elements:

- **Work** is $T(n) = T(3n/4) + T(n/4) + O(\log n)$ which is $O(n)$
- **Span** is $T(n) = T(3n/4) + O(\log n)$, which is $O(\log^2 n)$
- (neither bound is immediately obvious, but "trust me")

So for **mergesort** with parallel merge overall:

- **Work** is $T(n) = 2T(n/2) + O(n)$, which is $O(n \log n)$
- **Span** is $T(n) = 1T(n/2) + O(\log^2 n)$, which is $O(\log^3 n)$

So parallelism (work / span) is $O(n / \log^2 n)$

- Not quite as good as quicksort's $O(n / \log n)$
  - But (unlike Quicksort) this is a worst-case guarantee
- And as always this is just the asymptotic result

$\log^2 n$

Summary

Quicksort (best case)

O(n log n) - sequential n^2 worst case

--> **O(n)** span - parallel calls to quicksort

--> **O(log^2 n)** span - parallel partition

Mergesort (worst case)

O(n log n) - sequential

--> **O(n)** span - parallel calls to mergesort

--> **O(log^3 n)** span - parallel merge

Toward sharing resources (memory)

So far, we have been studying parallel algorithms using the fork-join model

- Reduce span via parallel tasks

Fork-Join algorithms all had a very simple *structure* to avoid race conditions

- Each thread had memory “only it accessed”
  - Example: each array sub-range accessed by only one thread
- Result of forked process not accessed until after join() called
- So the structure (mostly) ensured that bad simultaneous access wouldn’t occur

Strategy won’t work well when:

- **Memory** accessed by threads **is overlapping** or unpredictable
- Threads are doing **independent tasks** needing **access to same resources** (rather than implementing the same algorithm)

Heap for all objects and static fields, *shared* by all threads

2 Threads, each with own *unshared* call stack and "program counter"