Parallel Sorts

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Warm-Up

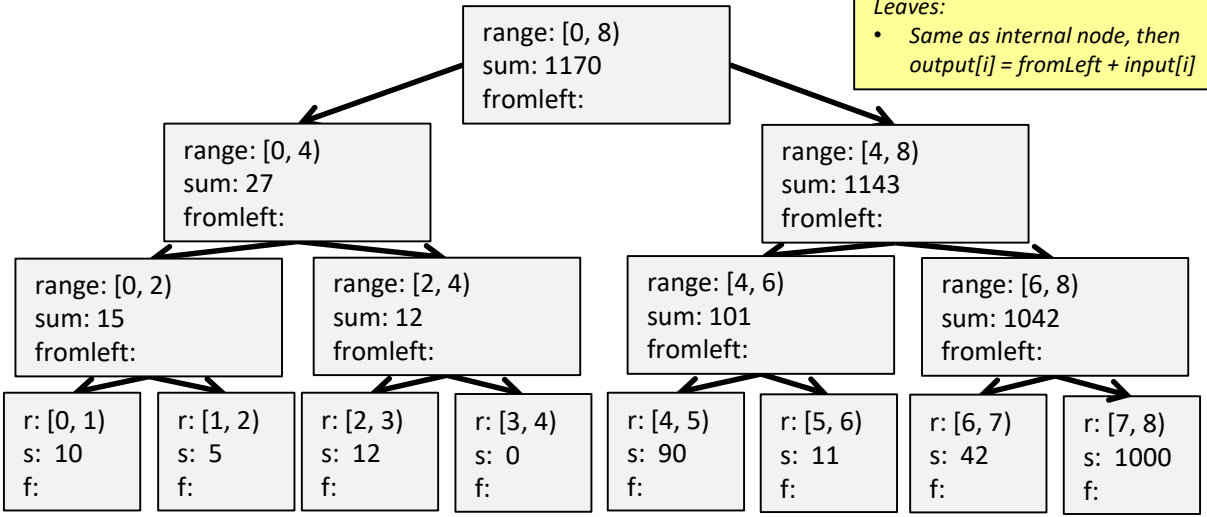
- Write your answers on a piece of scratch paper:
  - Given the following array, write pseudocode that creates a new array consisting *only* of values >10:

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

  - Finish populating fromLeft in this tree:

Internal nodes:
- *Left child: parent's*
- *Right child: parent's + sibling's sum*

Leaves:
- *Same as internal node, then output[i] = fromLeft + input[i]*



Learning Objectives

- See a specific application of parallel-prefix and how it combines with other parallelism techniques we've seen (ie, parallel-sum, map, reduce)

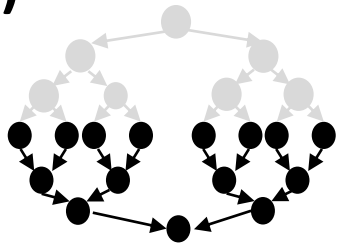
- Understand parallel-pack and parallel-quicksort

Lecture Outline

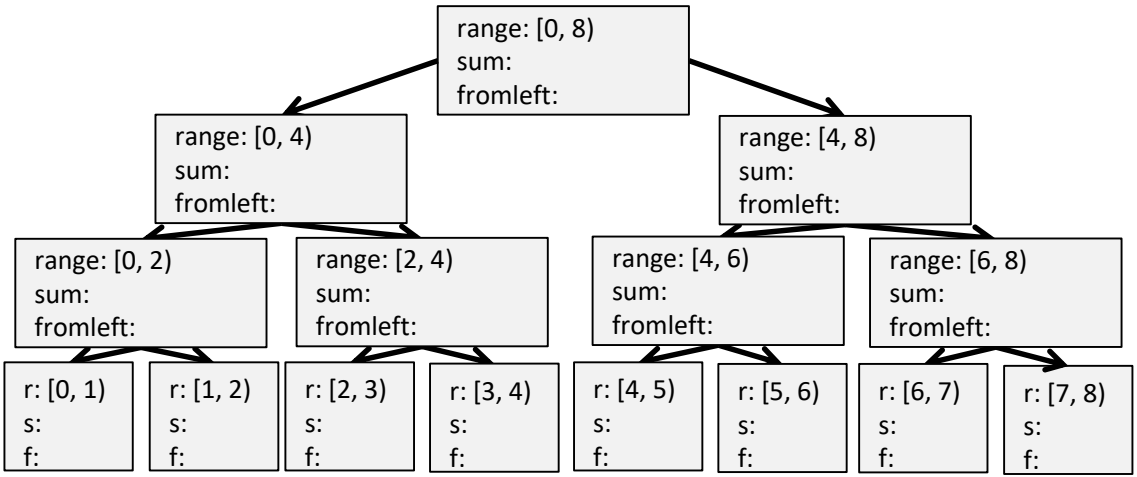
- **Parallel-Pack**
- Parallel-QuickSort
- Bonus Material (not covered in lecture): Parallel-MergeSort

Lecture questions: pollev.com/cse332

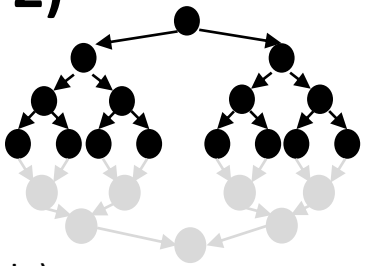
Parallel Prefix-Sum: Summary (1 of 2)



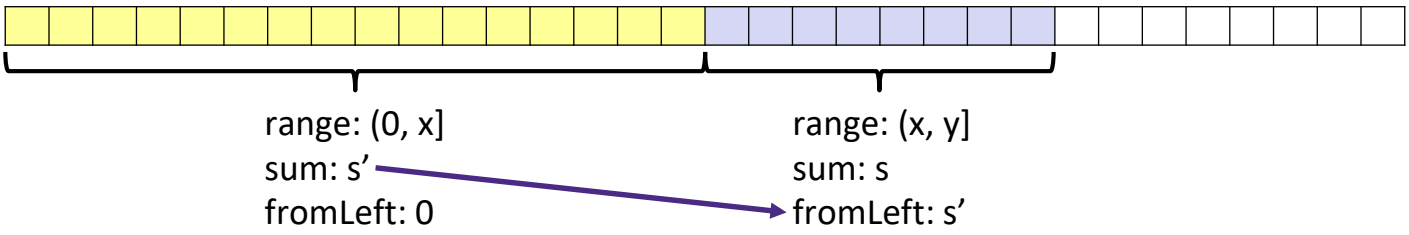
- Parent has range and sum of [lo, hi)
  - left has [lo, middle), and right has [middle, hi)
- "Up" Pass: build sum from the bottom of the tree:
  - A leaf's sum is just its value: input[i]
- Output of the "up" pass is this binary tree:



Parallel Prefix-Sum: Summary (2 of 2)



- The "down" pass: process the binary tree to populate the fromLeft fields
  - fromLeft is sum of elements left of the node's range: [0, lo)



  - Internal node takes its fromLeft value and
    - Passes its left child the same fromLeft
    - Passes its right child its fromLeft plus its left child's sum
  - At the leaf, must also output[i] = fromLeft + input[i]

Parallel work is $\underbrace{2(n-1)+n}_{\text{up}} + \underbrace{2(n-1)+n}_{\text{down}} \in O(n)$

- Total for algorithm: Work: O(n), Span: O(log n)

Parallel-Prefix

- Prefix-sum is also a pattern that arises in 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

You now know the "one weird trick": parallel-prefix!

Pack (aka "Filter")

- Given an array `input`, produce an array `output` containing only elements such that `f(element)` is true
  - E.g.: `input: [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`
    - `f: "is element > 10"`
    - `output: [17, 11, 13, 19, 24]`

- Parallelizable?
  - Yes: determining *whether* an element belongs in the output is easy
  - No: determining *where* an element belongs in the output is hard; seems to depend on previous results….

We Already Know Parallel-Pack!

In this example, filter = element > 10?

- Parallel-Pack = Parallel-Map + Parallel-Prefix + Parallel-Map!
  1. **Parallel map** to compute a bit-vector for filtered 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 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]
}
```

summing the bits gave us array indices!

Parallel-Pack Comments

Parallel-Pack:
1. Parallel-map: compute bit-vector
2. Parallel-prefix: compute bit-sum
3. Parallel-map: produce output

- First two steps can be combined into a prefix-sum
  - Different base case for the prefix sum
  - No effect on asymptotic complexity
- Combine third step into the down pass of the prefix-sum
  - Again, no effect on asymptotic complexity
- Analysis: $O(n)$ work, $O(\log n)$ span
  - 2 or 3 passes, but both are constants ☺
- Parallelized packs will help us parallelize quicksort…

Lecture Outline

- Parallel-Pack
- **Parallel-QuickSort**
- Bonus Material (not covered in lecture): Parallel-MergeSort

Lecture questions: pollev.com/cse332

Sequential QuickSort Review

Step	*Runtime Expression*
Pick the pivot value(s) • Hopefully these value(s) approximate the median	c_1
Partition all the values into: A. The values less than the pivot(s) B. The pivot(s) C. The values greater than the pivot(s)	$c_2 n$
Recursively QuickSort(A) and QuickSort(C)	$2\,T(\frac{n}{2})$

- Recurrence (assuming a good-enough pivot):
  - T(0) = T(1) = c1
  - T(n) = $2\,T(\frac{n}{2}) + c_2 n$
  - Closed-form T(n) = $O(n \log n)$

Copied from L7: Algorithm Analysis III

Really Common Recurrences

Recurrence Relation	*Closed Form*	*Name*	*Example*
T(n) = O(1) + T(n/2)	O(log n)	Logarithmic	Binary Search
T(n) = O(1) + T(n-1)	O(n)	Linear	Sum (v1: "Recursive Sum")
T(n) = O(1) + 2T(n/2)	O(n)	Linear	Sum (v2: "Recursive Binary Sum")
T(n) = O(n) + T(n/2)	O(n)	Linear	
T(n) = O(n) + 2T(n/2)	O(n log n)	Loglinear	MergeSort
T(n) = O(n) + T(n-1)	$O(n^2)$	Quadratic	
T(n) = O(1) + 2T(n-1)	$O(2^n)$	Exponential	Fibonacci

Parallelizing QuickSort: Attempt #1

Step	*Runtime Expression*
Pick the pivot value(s) • Hopefully these value(s) approximate the median	c_1
Partition all the values into: A. The values less than the pivot(s) B. The pivot(s) C. The values greater than the pivot(s)	$c_2 N$
Recursively QuickSort(A) and QuickSort(C)	$T(\frac{n}{2})$ (twice, but in parallel)

- Let's parallelize the two recursive calls!
  - Work (unchanged): $O(n \log n)$
  - T(n) = $1\,T(\frac{n}{2}) + c_2 n$
  - Span: $O(n)$

Parallel QuickSort: Doing Better

- $O(\texttt{log}\ n)$ speed-up with an infinite number of processors is okay, but a bit underwhelming
  - Sort 10^9 elements 30 times faster

- Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized
  - The Internet has been known to be wrong ☺
  - But we need auxiliary storage (no longer in place)
  - In practice, constant factors may make it not worth it, but remember Amdahl's Law…(exposing parallelism is important!)

- Already have everything we need to parallelize the partition…

Parallel Partition (not in place)

Step	*Span*
In parallel, partition all the values into: A. The values less than the pivot(s) B. The pivot(s) C. The values greater than the pivot(s)	O(log n)

- Parallel partition is just two packs!
  - We know a pack is $O(n)$ work, $O(\log n)$ span
    1. Pack elements less than pivot into left side of **`aux`** array
    2. Pack elements greater than pivot into right size of **`aux`** array
  - Put pivot between them and recursively sort
  - With a little more cleverness, can do both packs at once but no effect on asymptotic complexity
- Parallel Partition Span: O(log n)

Parallel QuickSort, Attempt #2: Example

1. Pick pivot (we'll use median-of-3)

8	1	4	9	0	3	5	2	7	6

2. Pack less-than, then pack greater-than
   - Must be sequential, since second pack needs a starting index

1	4	0	3	5	2

1	4	0	3	5	2	6	8	9	7

3. Recursively sort, in parallel
   - Can sort back into original array (like in mergesort)

0	1	2	3	4	5	6	7	8	9

- Parallel QuickSort:
  - T(n) = $1\,T(\frac{n}{2}) + c_1 \log n$
  - Span: $O(\log^2 n)$

Lecture Outline

- Parallel-Pack
- Parallel-QuickSort
- **Bonus Material (not covered in lecture): Parallel-MergeSort**

Lecture questions: pollev.com/cse332

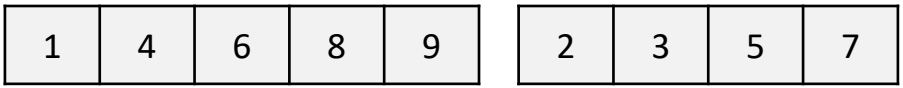
Parallelizing MergeSort

Step	*Runtime Expression*
Recursively MergeSort(A) and MergeSort(B)	
Merge(A, B)	

- Just like QuickSort, do the two recursive sorts in parallel:
  - Span is $T(n) = c_1 n + 1T(n/2) = O(n)$
  - Work is $O(n \log n)$
  - Parallelism = 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 (1 of 2)

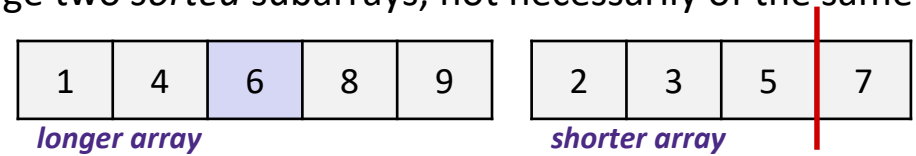
- Problem statement:
  - Merge two *sorted* subarrays, not necessarily of the same size



- Intuition:
  - Suppose the longer subarray has *m* elements. In parallel:
    - Merge the first *m*/2 elements of the longer half with the "appropriate" elements of the shorter half
    - Merge the second *m*/2 elements of the longer half with the rest of the shorter half

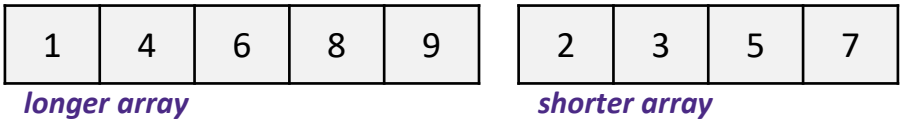
Parallelizing the Merge (2 of 2)

- Problem statement:
  - Merge two *sorted* subarrays, not necessarily of the same size

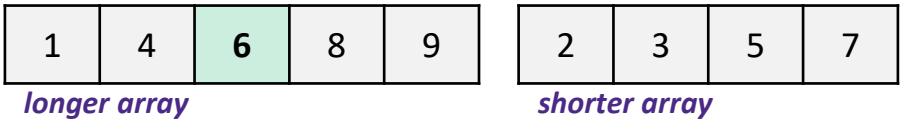


- Step #1:
  - Pick the median of the *longer array* in constant time
  - Binary search the *shorter array* to find the first element >median
- Step #2 (in parallel):
  - Merge the lower part of the *longer array* (<=median) with the lower part of the *shorter array*
  - Merge upper part of the *longer array* (>median onward) with the upper part of the *shorter array*

Parallelizing the Merge: Example (1 of 7)

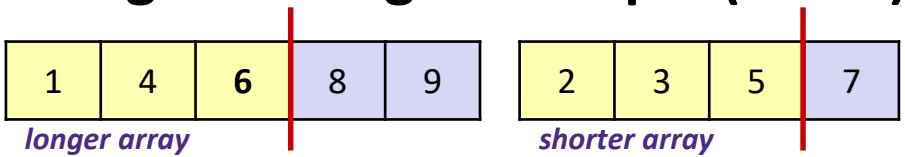


Parallelizing the Merge: Example (2 of 7)



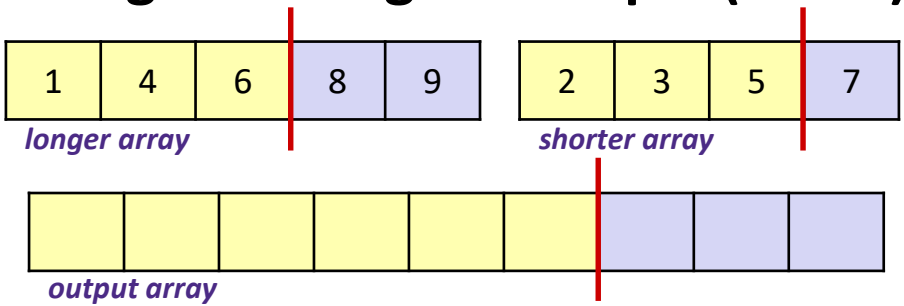
- Pick the median of the *longer array*: O(1) to compute index

Parallelizing the Merge: Example (3 of 7)



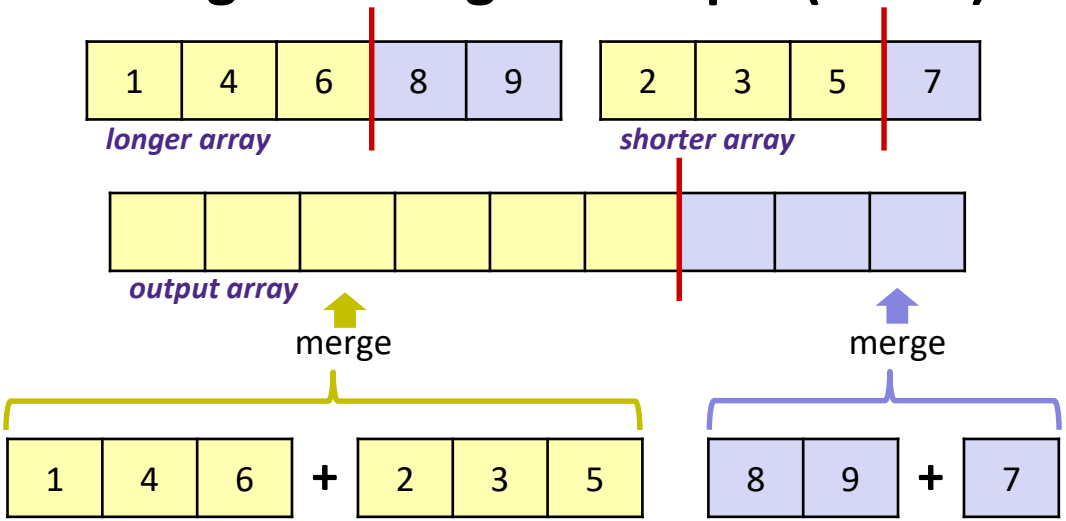
- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search

Parallelizing the Merge: Example (4 of 7)



- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search
- Calculate where to split the output array: O(1)

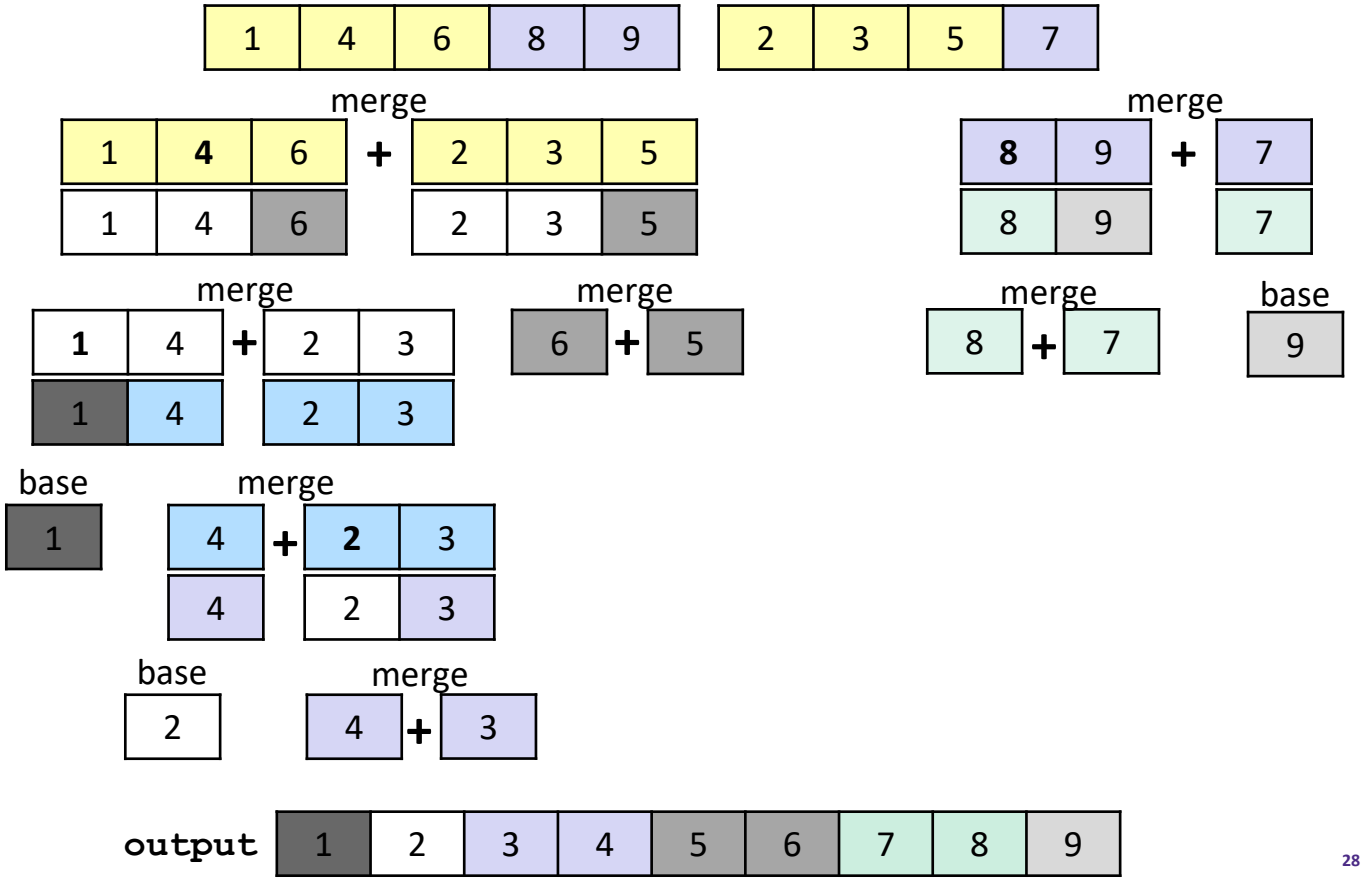
Parallelizing the Merge: Example (5 of 7)



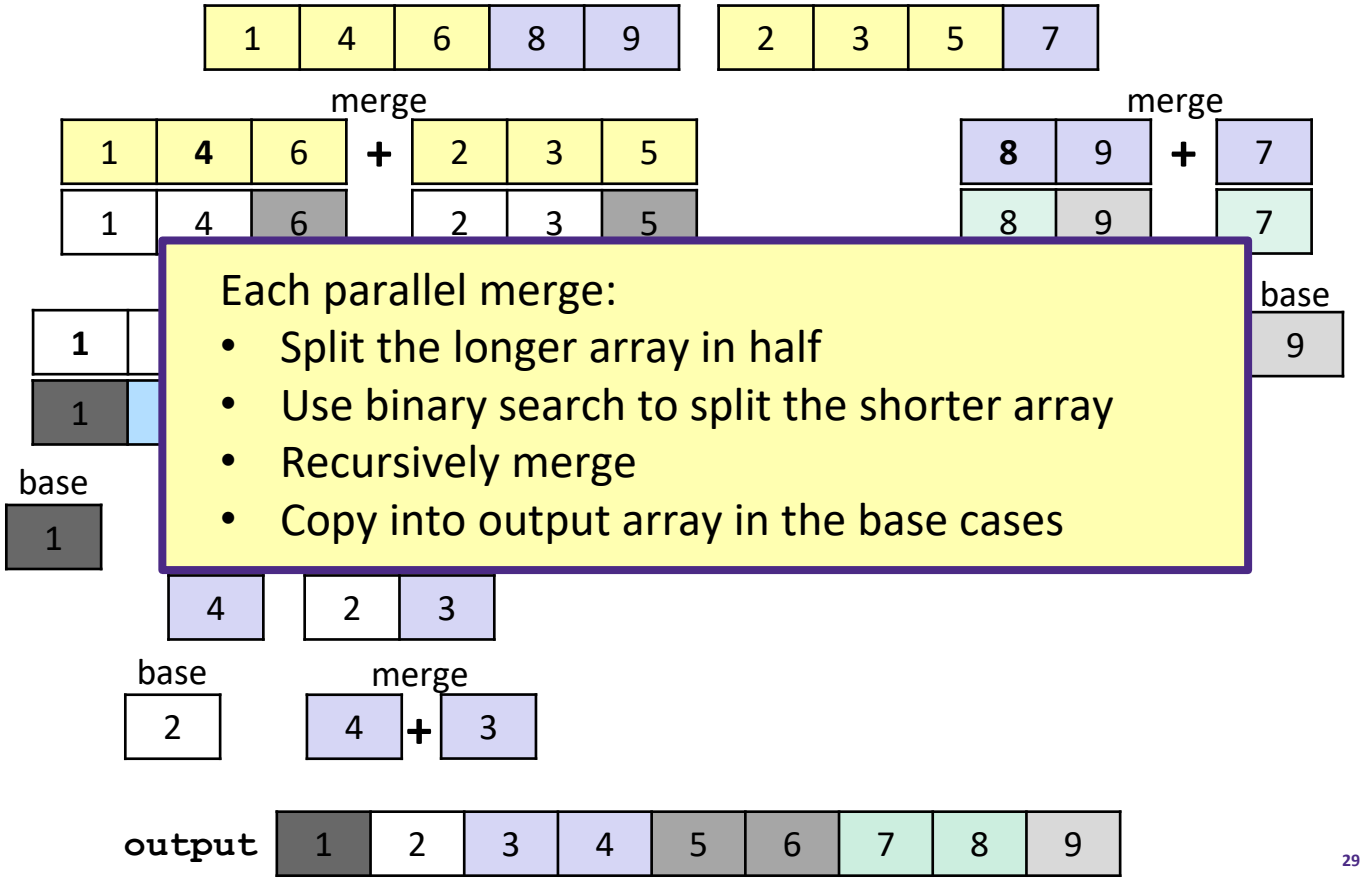
- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search
- Calculate where to split the output array: O(1)
- Do the sub-merges in parallel

🤔 *how do we sub-merge?* 🤔

Parallelizing the Merge: Example (6 of 7)



Parallelizing the Merge: Example (7 of 7)



Each parallel merge:

- Split the longer array in half
- Use binary search to split the shorter array
- Recursively merge
- Copy into output array in the base cases

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)
```

Parallel-MergeSort: Analysis (1 of 3)

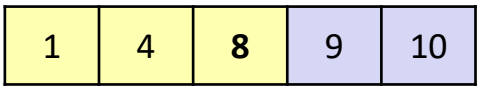
- *Sequential* MergeSort:

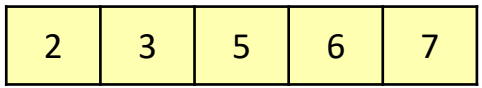
  $T(n) = 2T(n/2) + c_2 \quad \in \quad O(n \log n)$

- MergeSort with *parallel recursive calls* and a *sequential merge*:
  - **Work**: $O(n \log n)$
  - **Span**: $T(n) = 1T(n/2) + c_2 \quad \in \quad O(n)$

Parallel-MergeSort: Analysis (2 of 3)

- What about *just* the *parallel merge* of two subarrays?
  - Let the total length of the two subarrays be n
  - $O(\log n)$ binary search to split the shorter subarray
  - Worst-case split is $(3/4)n$ and $(1/4)n$
    - Happens when the two subarrays are of the same length (n/2) and the shorter subarray splits into two pieces of the most uneven sizes possible: one of size n/2, one of size 0
  - **Work** is $T(n) = T(3n/4) + T(n/4) + c_1 \log n \in O(n)$
  - **Span** is $T(n) = T(3n/4) + c_2 \log n \in O(\log^2 n)$
    - (neither bound is immediately obvious, but "trust me")





Parallel-MergeSort: Analysis (2 of 3)

- MergeSort with *parallel recursive calls* and a *parallel merge*:
  - **Work** is $T(n) = 2T(n/2) + c_1 n \in O(n \log n)$
  - **Span** is $T(n) = 1T(n/2) + c_2 \log^2 n \in O(\log^3 n)$
  - So, **parallelism** (work / span) is $O(n / \log^2 n)$
    - Not quite as good as QuickSort's $O(n / \log n)$ parallelism
    - But, unlike Quicksort, this is a worst-case guarantee
    - And, as always, this is just the asymptotic result

Summary

- Parallel-prefix sum can be used on a bit-vector to generate indices for a packed array
  - Which can then be used for parallel-pack

- Parallel-QuickSort starts with parallelizing its recursive calls
  - But its runtime is lower-bounded by the O(n) pivot operation
  - Parallel pivot is two