# Parallel Pack and Parallel Sort

CSE 332 Spring 2021

**Instructor:** Hannah C. Tang

**Teaching Assistants:**

| | | |
|---|---|---|
| Aayushi Modi | Khushi Chaudhari | Patrick Murphy |
| Aashna Sheth | Kris Wong | Richard Jiang |
| Frederick Huyan | Logan Milandin | Winston Jodjana |
| Hamsa Shankar | Nachiket Karmarkar | |

# gradescope

gradescope.com/courses/256241

- Describe a parallelized reduction to count the values greater than 10
  - Constraint: you must store your intermediate results in an array; we want the result of the “is greater than 10?” boolean

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **input** | 17 | 4 | 6 | 8 | 11 | 5 | 13 | 19 | 0 | 24 |

```
f: "is element > 10"
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **bits** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| **output** | 5 | | | | | | | | | |

# Announcements

- P2 due tomorrow night!

- P3 partner moving to Google Spreadsheets
  - Need to fill out even if you're keeping the same partner; worth one participation point (see Gradescope assignment, coming soon)
  - Can fill it out multiple times; we will take most recent response
  - DO NOT TYPO UWNetIDs!

# Lecture Outline

- **Review: Designing new parallel algorithms**

- Parallel Pack

- Parallel Sort
  - QuickSort
  - MergeSort

**Span** = $T_\infty$ = sum of runtime of all nodes in the DAG's *most-expensive path*
**Work** = $T_1$ = sum of runtime of all nodes in the DAG
**Speed-up** = $T_1 / T_P$
  **Perfect linear speedup** when $T_1 / T_P = P$
**Parallelism** = $T_1 / T_\infty$

# Amdahl's Law

- Let the work ($T_1$) be 1 unit of time and **S** be the unparallelizable portion of execution time:

$$T_1 = 1 = S + (1-S)$$

- Suppose *perfect linear speed-up* on the parallelizable portion. Then:

$$T_P = S + (1-S)/P$$

- Amdahl's Law states the speed-up with P processors is:

$$T_1 / T_P = 1 / (S + \cancel{(1-S)/P}) \quad \text{as } P \to \infty$$

- and the parallelism (maximum possible speed-up) is:

$$T_1 / T_\infty = 1 / S$$

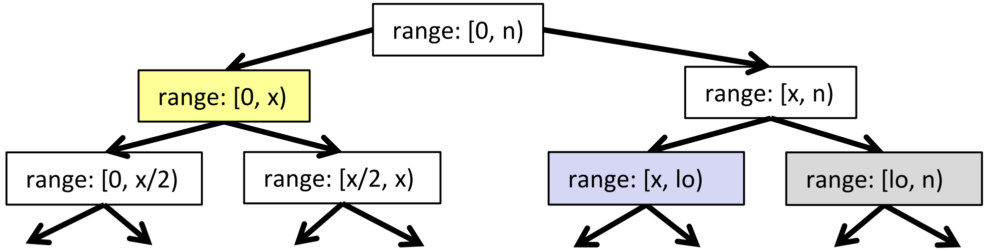
# The Challenge Posed by Amdahl's Law

- Amdahl's Law tells us unparallelized parts become a bottleneck very quickly
  - But it *doesn't* tell us additional processors are worthless

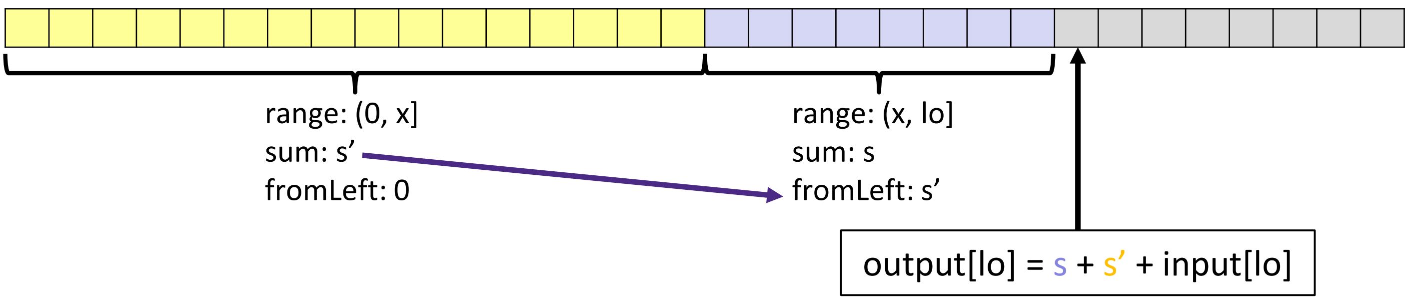
- … because we can find new parallel algorithms
  - Some things that seem sequential turn out to be parallelizable
  - We parallelized a 'running sum' array!

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **input** | 6 | 4 | 16 | 10 | 16 | 15 | 2 | 8 |
| **output** | 6 | 10 | 26 | 36 | 52 | 67 | 69 | 77 |

# Parallel Prefix-Sum is Partial Sums!



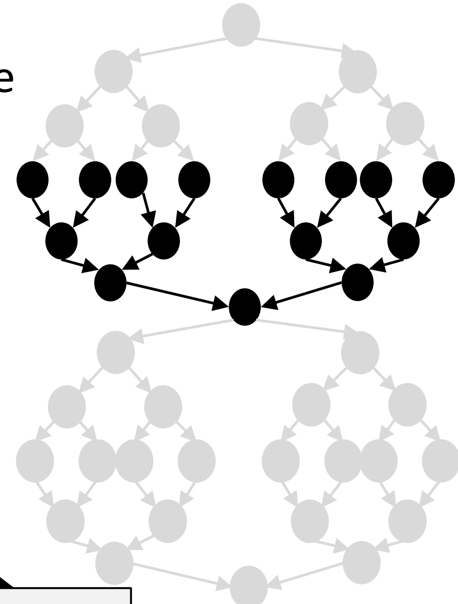

- If we saved the intermediate results from parallel-sum, we could generate the prefix-sum from those results in a second pass



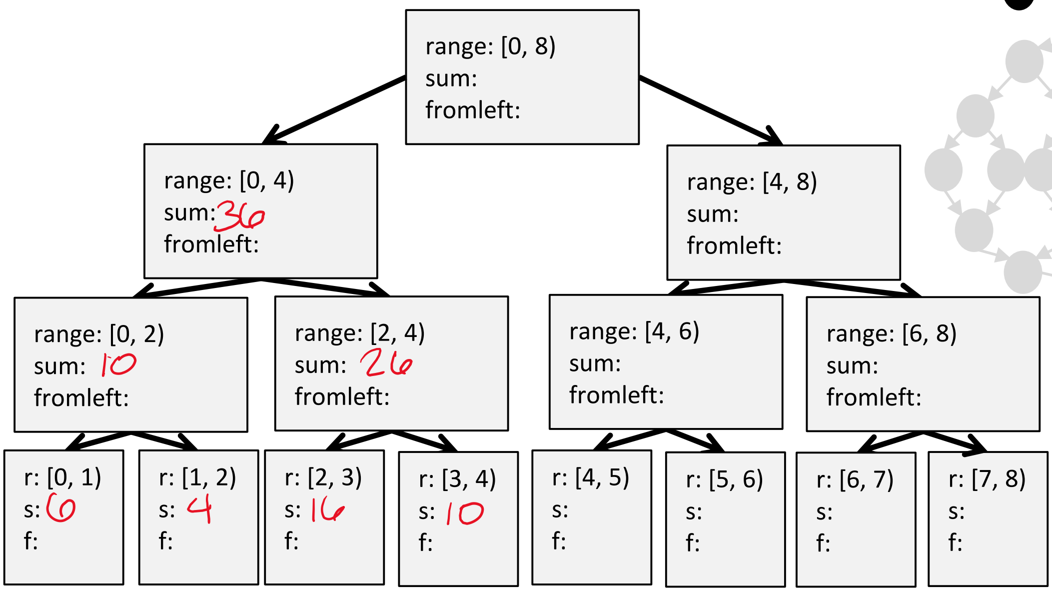

- Internal node takes its fromLeft value and
  - Passes its left child *the same* fromLeft
  - Passes its right child *its fromLeft plus its left child’s sum*

# Parallel Prefix-Sum: The "Up" Pass

- This first pass builds a *binary tree* from the bottom: the "up" pass



| input | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 4 | 16 | 10 | 16 | 15 | 2 | 8 |



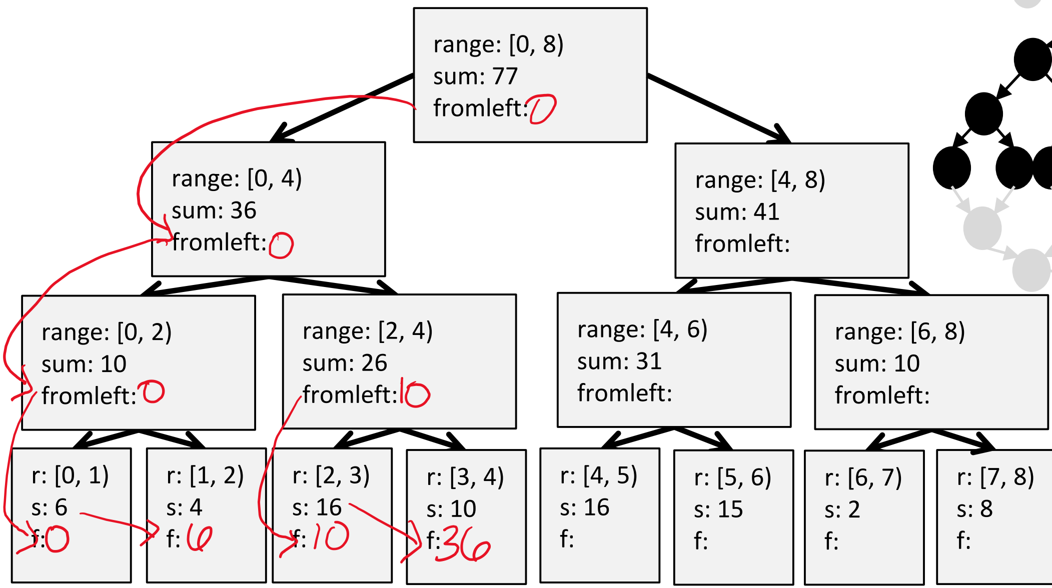
# Parallel Prefix-Sum: The “Down” Pass

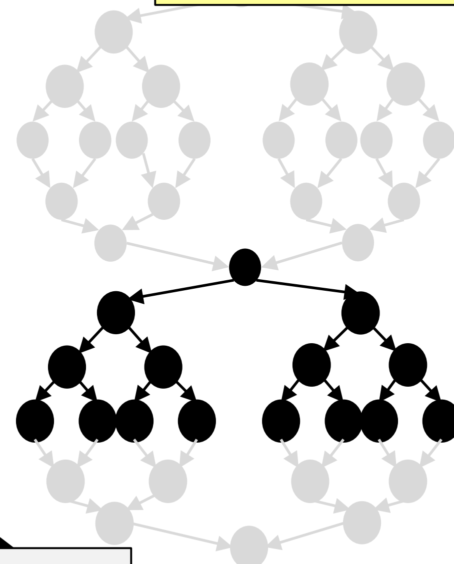
*Internal nodes:*
- *Left: parent’s*
- *Right: parent’s + sibling’s sum*

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

❖ The second pass uses the *binary tree* to populate the fromLeft fields

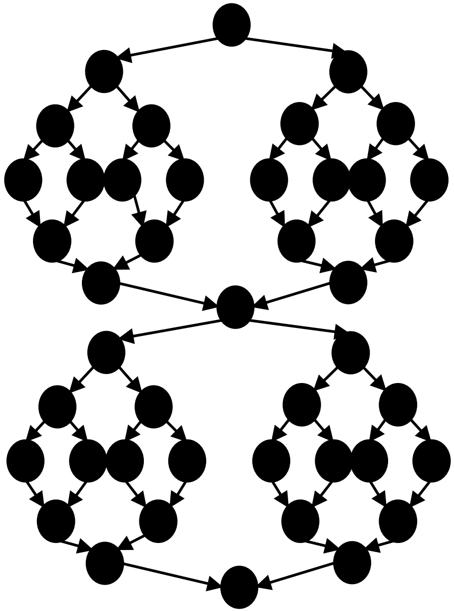
| input | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 6 | 4 | 16 | 10 | 16 | 15 | 2 | 8 |



# Parallel Prefix-Sum Runtime

- Up pass:
  - Work: O(n), Span: O(log n)
- Down pass:
  - Work: O(n), Span: O(log n)



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

# Lecture Outline

- Review: Designing new parallel algorithms
- **Parallel Pack**
- Parallel Sort
  - QuickSort
  - MergeSort

# Pack (aka "Filter")

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

**input**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 17 | 4 | 6 | 8 | 11 | 5 | 13 | 19 | 0 | 24 |

`f: "is element > 10"`

**output**

| | | | | |
|---|---|---|---|---|
| 17 | 11 | 13 | 19 | 24 |

- Parallelizable? Sort of …
  - 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!

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **input** | 17 | 4 | 6 | 8 | 11 | 5 | 13 | 19 | 0 | 24 |

`f: "is element > 10"`

- Parallel-Pack = Parallel-Map + Parallel-Prefix + Parallel-Map!

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **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 |

gradescope

gradescope.com/courses/256241

| input | 17 | 4 | 6 | 8 | 11 | 5 | 13 | 19 | 0 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|

`f: "is element > 10"`

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **bits** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| **bitsum** | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 5 |

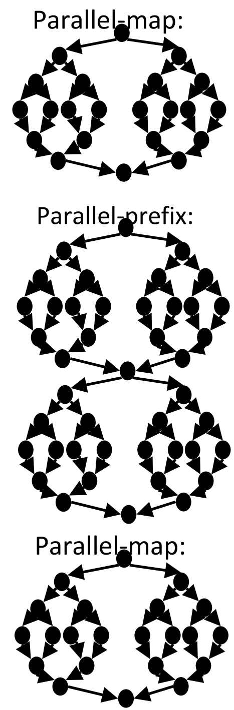
- Write the final parallel-map's pseudocode, generating the output
  - Hint: your code will need to take three inputs: input, bits, and bitsum

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


}
```

# Parallel-Pack Analysis

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

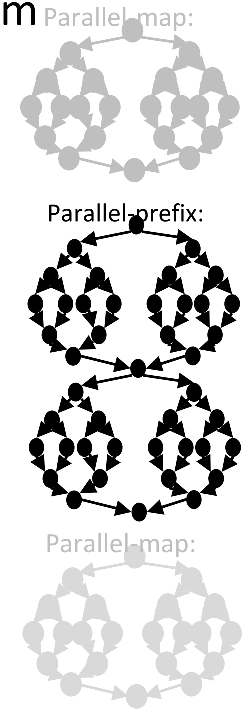



- Each step is O(n) work, O(log n) span
  - So parallel-pack still O(n) work, O(log n) span

# 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
- Still $O(n)$ work, $O(\log n)$ span
  - … but better constants ☺
- Parallelized packs will help us parallelize QuickSort…

# Lecture Outline

- Review: Designing new parallel algorithms
- Parallel Pack
- Parallel Sort
  - **QuickSort**
  - MergeSort

# Sequential QuickSort Review

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

- Recurrence (assuming a good-enough pivot):
  - $T(0) = T(1) = c_1$
  - $T(n) = 2T(\frac{n}{2}) + c_1 + c_2 n$
  - Closed-form $T(n) \in O(n \log n)$

Copied from L5: 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 |

Speedup: $T_1 / T_P$
Max Parallelism: $T_1 / T_\infty$

# Parallelizing QuickSort: Attempt #1

| *Step* | *Runtime Expression* |
|---|---|
| Pick the pivot value(s)<br>• Hopefully these value(s) approximate the median | $c_1$ |
| Partition all the values into:<br>A. The values less than the pivot(s)<br>B. The pivot(s)<br>C. The values greater than the pivot(s) | $c_2 n$ |
| Recursively QuickSort(A) and QuickSort(C) *in parallel* | Somewhere between $T(\frac{n}{2})$ and $2T(\frac{n}{2})$ (depending on number of executors) |

- Recurrence (assuming a good-enough pivot):
  - Work $T_1(n) = 2T(\frac{n}{2}) + c_1 + c_2 n$ ∈ O( $n \log n$ )
  - Span $T_\infty(n) = T(\frac{n}{2}) + c_1 + c_2 n$ ∈ O( $n$ )
  - Parallelism = Work/Span ∈ O( $\log n$ )

# Parallel QuickSort: Doing Better

- $O(\texttt{log}\ n)$ parallelism 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)

- Parallel partition is just two packs!
  1. Pack elements less than pivot into left side of **`aux`** array
  2. Then, pack elements greater than pivot into right side of **`aux`** array
- We know a pack is $O(n)$ work, $O(\log n)$ span

| Step | Work ($T_1$) | Span ($T_\infty$) |
|---|---|---|
| *In parallel*, partition all the values into:<br>A. The values less than the pivot(s)<br>B. The pivot(s)<br>C. The values greater than the pivot(s) | O(n)<br>O(n)<br>O(n) | O(log n)<br>O(log n)<br>O(log n) |

- Parallel Partition (does not include parallel sorting):
  - Work $T_1 \in O($__n__$)$, Span $T_\infty \in O($__log n__$)$

- Can do both packs at once, but no effect on asymptotic complexity

# Parallelizing QuickSort: Attempt #2

| *Step* | *Work ($T_1$)* | *Span ($T_\infty$)* |
|---|---|---|
| Pick the pivot value(s) | $c_1$ | $c_1$ |
| *In parallel*, partition all the values | $c_2 n$ | $c_2 \log n$ |
| *In parallel*, recursively QuickSort(A) and QuickSort(C) | $2T(\frac{n}{2})$ | $T(\frac{n}{2})$ |

- Recurrence (assuming a good-enough pivot):
  - Work $T_1(n) = 2T(\frac{n}{2}) + c_1 + c_2 n \in O(n \log n)$
  - Span $T_\infty(n) = T(\frac{n}{2}) + c_1 + c_2 \log n \in O(\log^2 n)$
  - Parallelism = Work/Span $\in O(n / \log n)$

# 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
   - Packs 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 |

# Lecture Outline

- Review: Designing new parallel algorithms
- Parallel Pack
- Parallel Sort
  - QuickSort
  - **MergeSort**

# Parallelizing MergeSort

| *Step* | *Work ($T_1$)* | *Span ($T_\infty$)* |
|---|---|---|
| *In parallel*, recursively MergeSort(A) and MergeSort(B) | 2T(n/2) | T(n/2) |
| Merge(A, B) | $c_2n$ | $c_2n$ |

- Start like we did with QuickSort: do recursive sorts in parallel
  - Work $T_1(n) = c_2n + 2T(n/2) \in O(n \log n)$
  - Span $T_\infty(n) = c_2n + 1T(n/2) \in O(n)$
  - Parallelism = Work/Span $\in O(\log n)$

- 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

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 6 | 8 | 9 |

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 7 |

↓

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Intuition:
  - Want each parallel executor to merge half of the elements
  - Choose a value that is approximately the median of the final array
    - Suppose the longer subarray has $m$ elements. Then choose $m/2$-th element
  - In parallel:
    - Merge first $m/2$ elements of longer half with "appropriate" elements of shorter half
    - Merge second $m/2$ elements of longer half with rest of the shorter half

# Parallelizing the Merge (2 of 2)

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

| | | | | |
|---|---|---|---|---|
| 1 | 4 | **6** | 8 | 9 |

***longer array***

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 7 |

***shorter array***

- 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)

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 6 | 8 | 9 |

*longer array*

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 7 |

*shorter array*

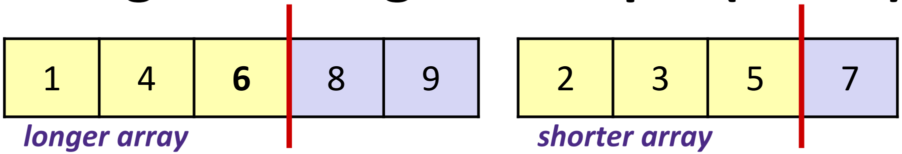
# Parallelizing the Merge: Example (2 of 7)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | **6** | 8 | 9 | 2 | 3 | 5 | 7 |

*longer array* *shorter array*

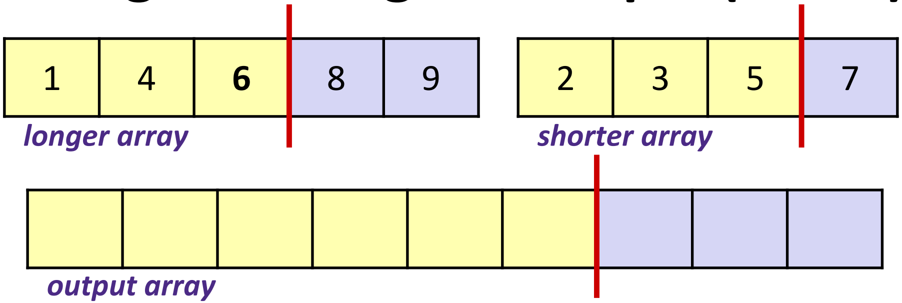
- Pick the median of the *longer array*: O(1) to compute index

# Parallelizing the Merge: Example (3 of 7)



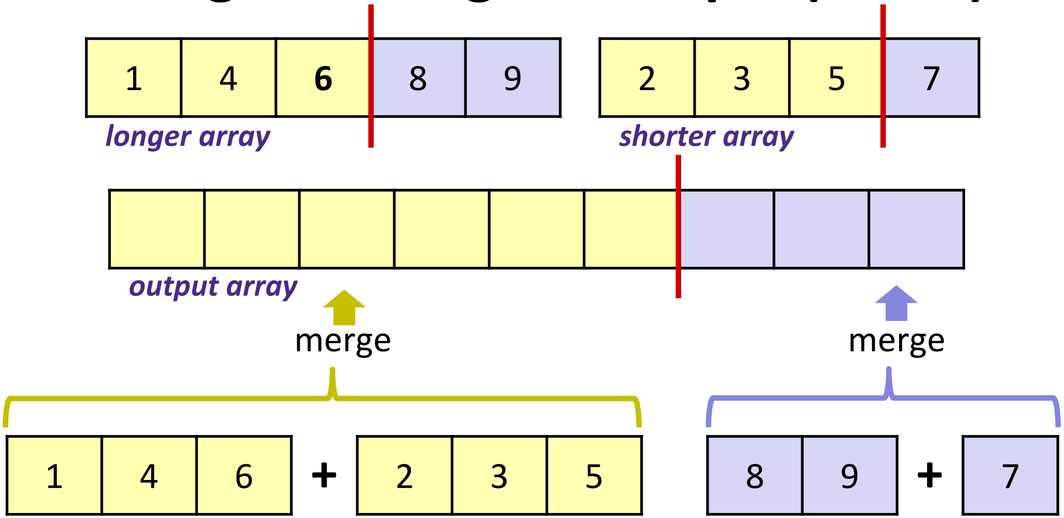

- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search

# 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)

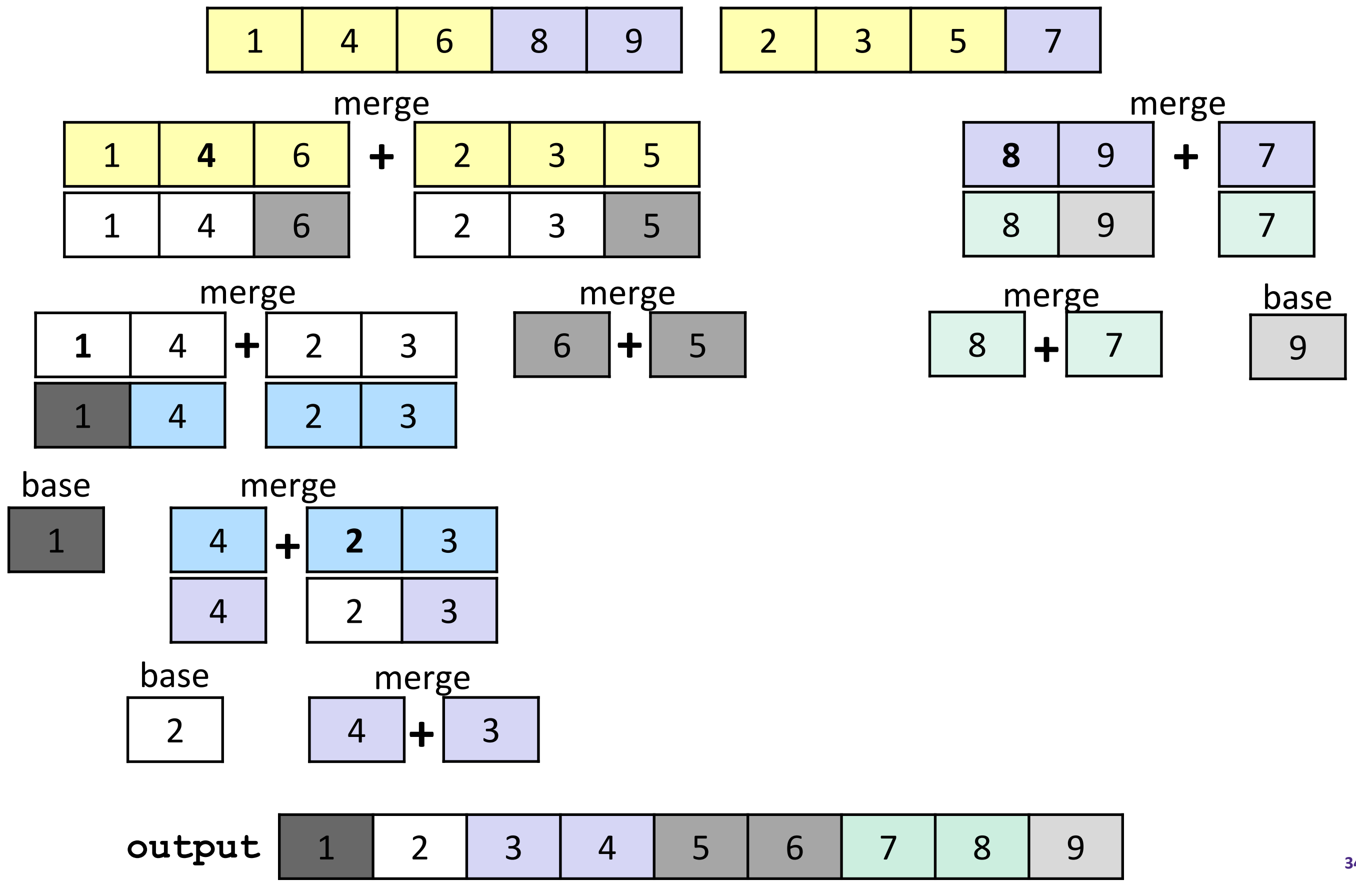
# Parallelizing the Merge: Example (5 of 7)



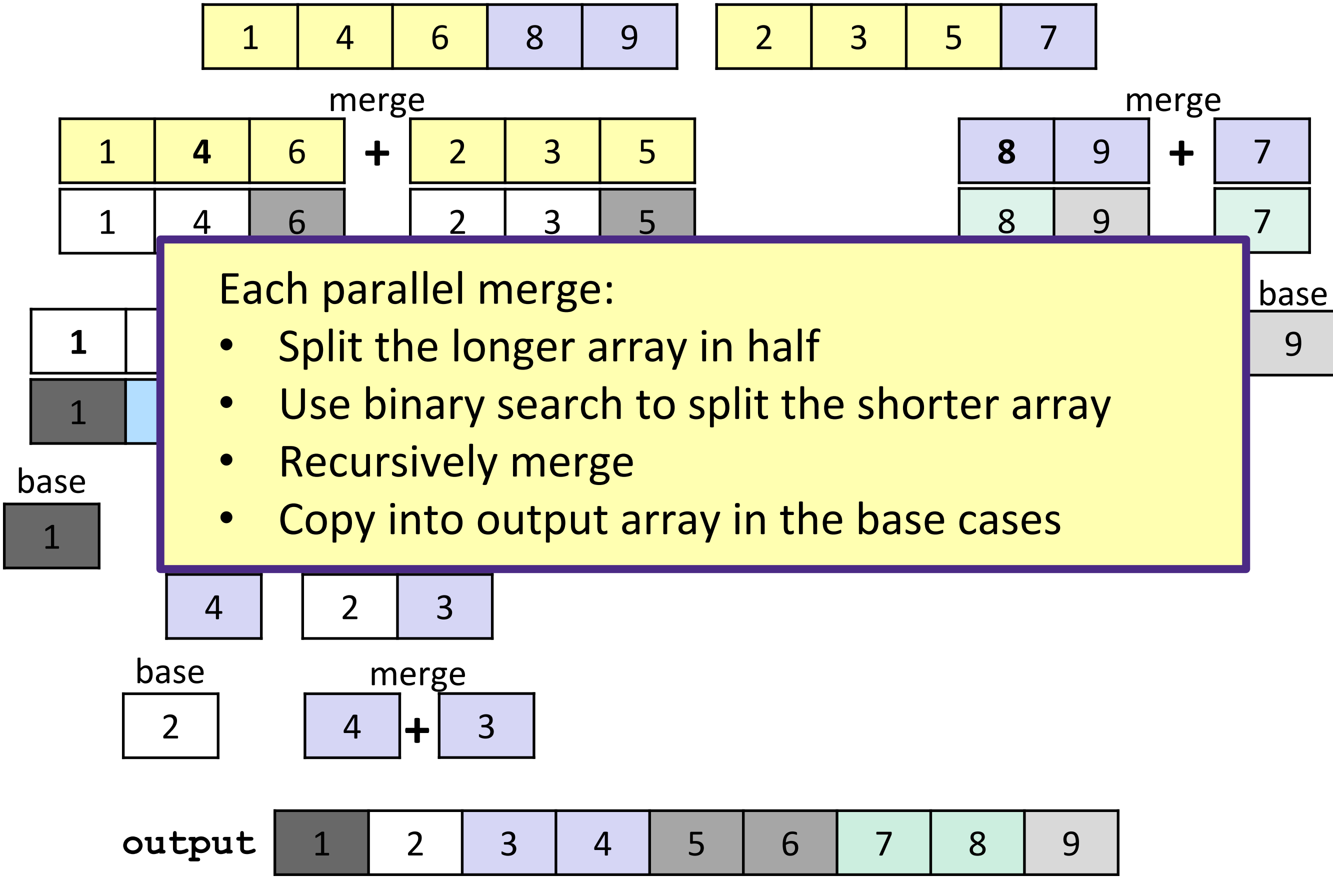

- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search
- Calculate where to split the output array: O(1)
- Do the sub-merges in parallel

*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:

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

- *Parallel* MergeSort with *sequential merge*:
  - **Work**: $O(n \log n)$
  - **Span**: $T(n) = \mathbf{1}T(n/2) + c_2 \quad \in \quad O(n)$

| | | | | |
|---|---|---|---|---|
| 1 | 4 | **5** | 9 | 10 |

***longer array***

| | | | | |
|---|---|---|---|---|
| 2 | 3 | 6 | 7 | 8 |

***shorter array***

# Parallel-MergeSort: Analysis (2 of 3)

- What about *just* the *parallel merge* of two subarrays?
  - Let the total length of the two subarrays be $n$
  - $O(\texttt{log}\ n)$ 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 $T(n) = T(3n/4) + T(n/4) + c_1 \texttt{log}\ n \quad \in \quad O(n)$
- **Span** is $T(n) = T(3n/4) + c_2 \texttt{log}\ n \quad \in \quad O(\texttt{log}^2\ n)$
  - (neither bound is immediately obvious, but "trust me")

| | | | | |
|---|---|---|---|---|
| 1 | 4 | **8** | 9 | 10 |

*longer array*

| | | | | |
|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 7 |

*shorter array*

# Parallel-MergeSort: Analysis (3 of 3)

- *Parallel* MergeSort with a *parallel merge*:
  - **Work** is $T(n) = 2T(n/2) + c_1 n \in O(n \log n)$
  - **Span** is $T(n) = \mathbf{1}T(n/2) + c_2 \log^2 n \in O(\log^3 n)$

- 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