CSE 332: Parallel Sorting

Richard Anderson

Spring 2016

Announcements

Recap

Last lectures

- simple parallel programs
- common patterns: map, reduce
- analysis tools (work, span, parallelism)

Now

- Amdahl's Law
- Parallel quicksort, merge sort
- useful building blocks: prefix, pack

Analyzing Parallel Programs

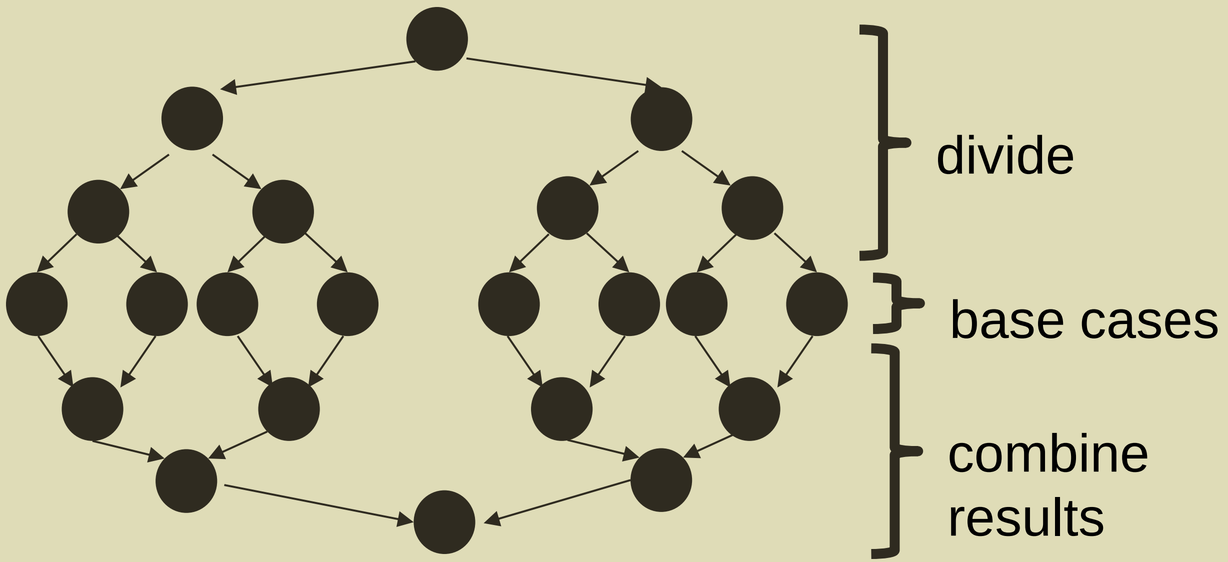
Let $\mathbf{T_P}$ be the running time on **P** processors

Two key measures of run-time:

- Work: How long it would take 1 processor = $\mathbf{T_1}$
- Span: How long it would take infinity processors = $\mathbf{T_\infty}$
  - The hypothetical ideal for parallelization
  - This is the longest "dependence chain" in the computation
  - Example: $O(\log n)$ for summing an array
  - Also called "critical path length" or "computational depth"

Divide and Conquer Algorithms

Our **`fork`** and **`join`** frequently look like this:



In this context, the span (T_∞) is:

- The longest dependence-chain; longest 'branch' in parallel 'tree'
- Example: $O(\log n)$ for summing an array; we halve the data down to our cut-off, then add back together; $O(\log n)$ steps, O(1) time for each
- Also called "critical path length" or "computational depth"

Parallel Speed-up

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$

- If speed-up is **P**, we call it perfect linear speed-up
  - e.g., doubling **P** halves running time
  - hard to achieve in practice

- Parallelism is the maximum possible speed-up: $\mathbf{T_1 / T_\infty}$
  - if you had infinite processors

Estimating T_p

- How to estimate $\mathbf{T_P}$ (e.g., P = 4)?

- Lower bounds on $\mathbf{T_P}$ (ignoring memory, caching...)
  1. $\mathbf{T_\infty}$
  2. $\mathbf{T_1 / P}$
  – which one is the tighter (higher) lower bound?

- The ForkJoin Java Framework achieves the following expected time asymptotic bound:

$$\mathbf{T_P \in O(T_\infty + T_1 / P)}$$

  – this bound is optimal

Amdahl’s Law

- Most programs have
  1. parts that parallelize well
  2. parts that don’t parallelize at all

- The latter become bottlenecks

Amdahl's Law

- Let T_1 = 1 unit of time
- Let S = proportion that can't be parallelized

$$1 = \mathbf{T_1 = S + (1 - S)}$$

- Suppose we get perfect linear speedup on the parallel portion:

$$\mathbf{T_P =}$$

- So the overall speed-up on P processors is (Amdahl's Law):

$$\mathbf{T_1 / T_P =}$$

$$\mathbf{T_1 / T_\infty =}$$

- If 1/3 of your program is parallelizable, max speedup is:

Pretty Bad News

- Suppose 25% of your program is sequential.
  - Then a billion processors won't give you more than a 4x speedup!

- What portion of your program must be parallelizable to get 10x speedup on a 1000 core GPU?
  - $10 <= 1 / (S + (1-S)/1000)$

- Motivates minimizing sequential portions of your programs

Take Aways

- Parallel algorithms can be a big win
- Many fit standard patterns that are easy to implement
- Can’t just rely on more processors to make things faster (Amdahl’s Law)

Parallelizable?

Fibonacci (N)

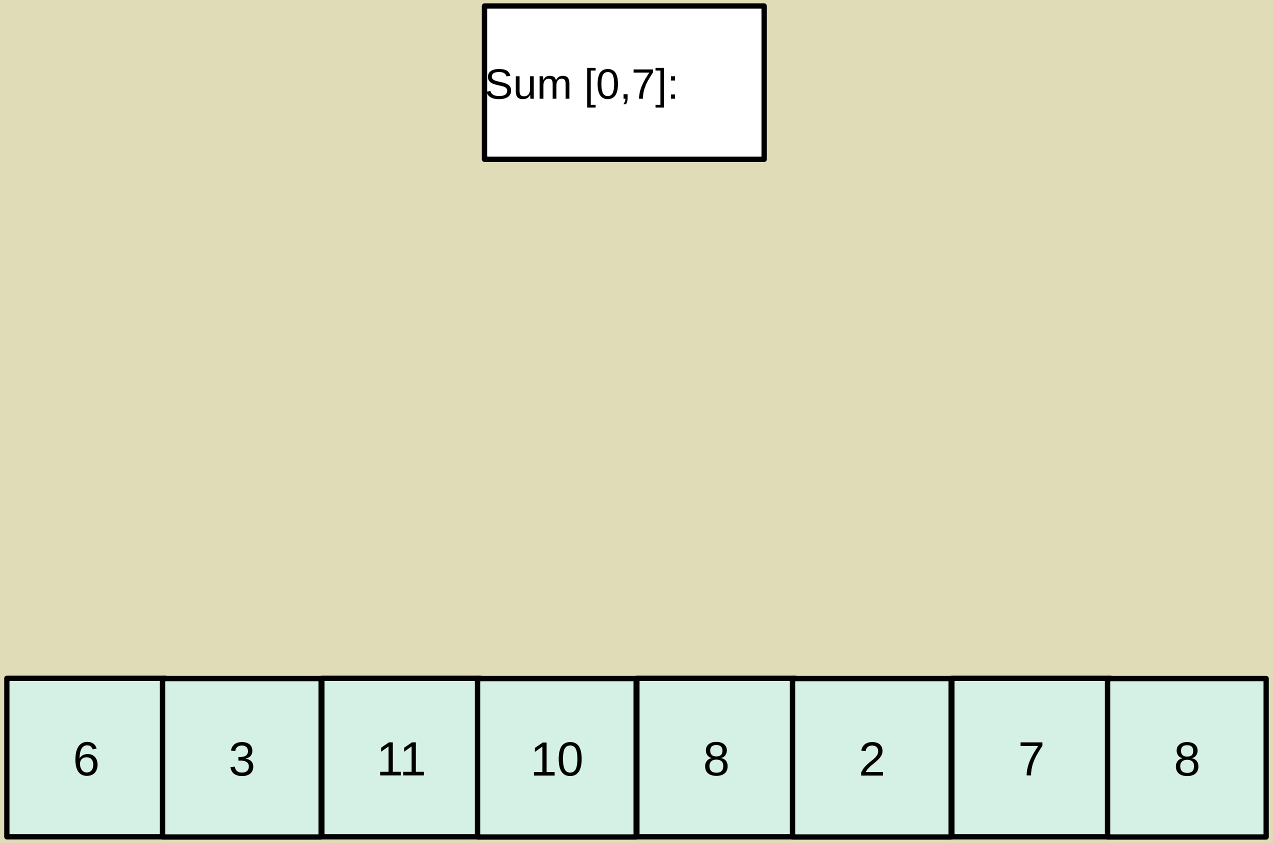
Parallelizable?

Prefix-sum:

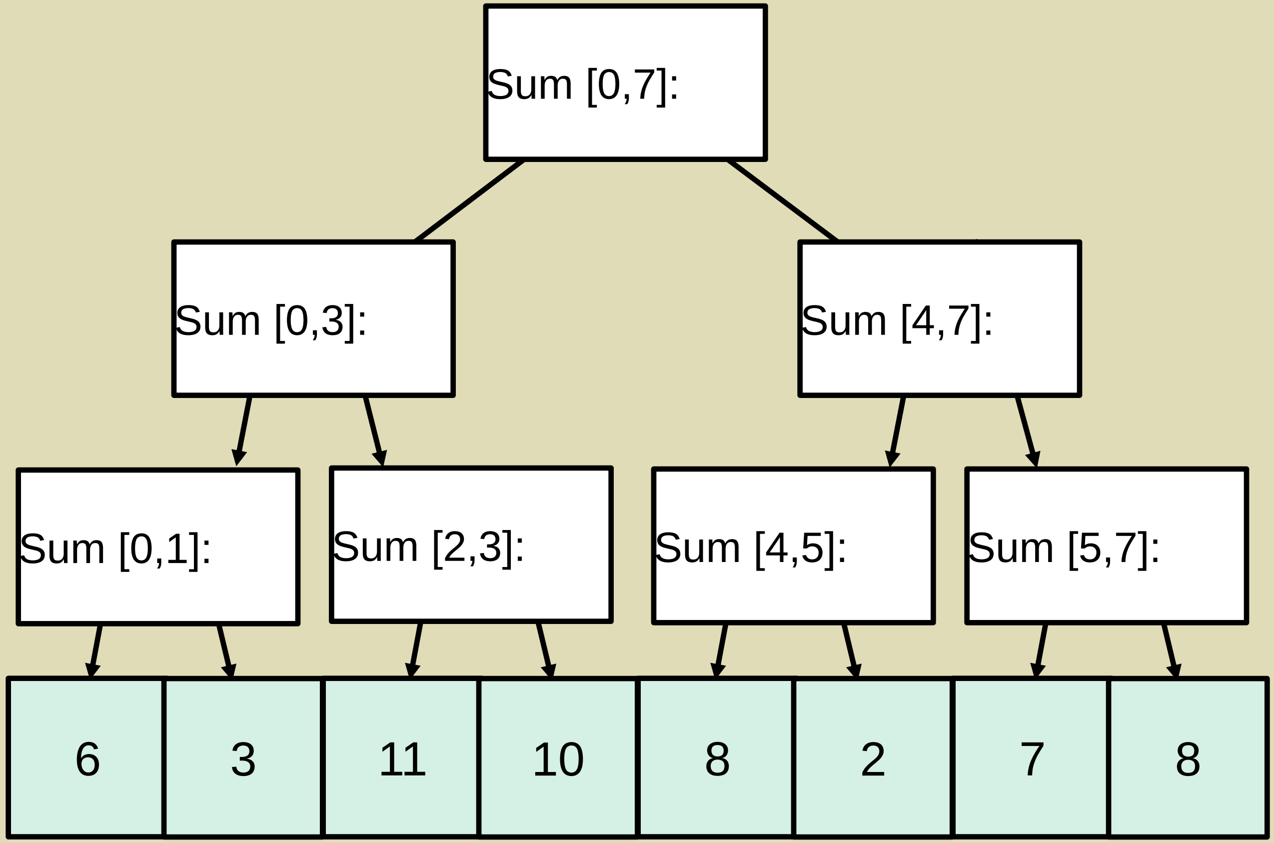
input	6	3	11	10	8	2	7	8
output								

$output[i] = \sum_0^{i-1} input[i]$

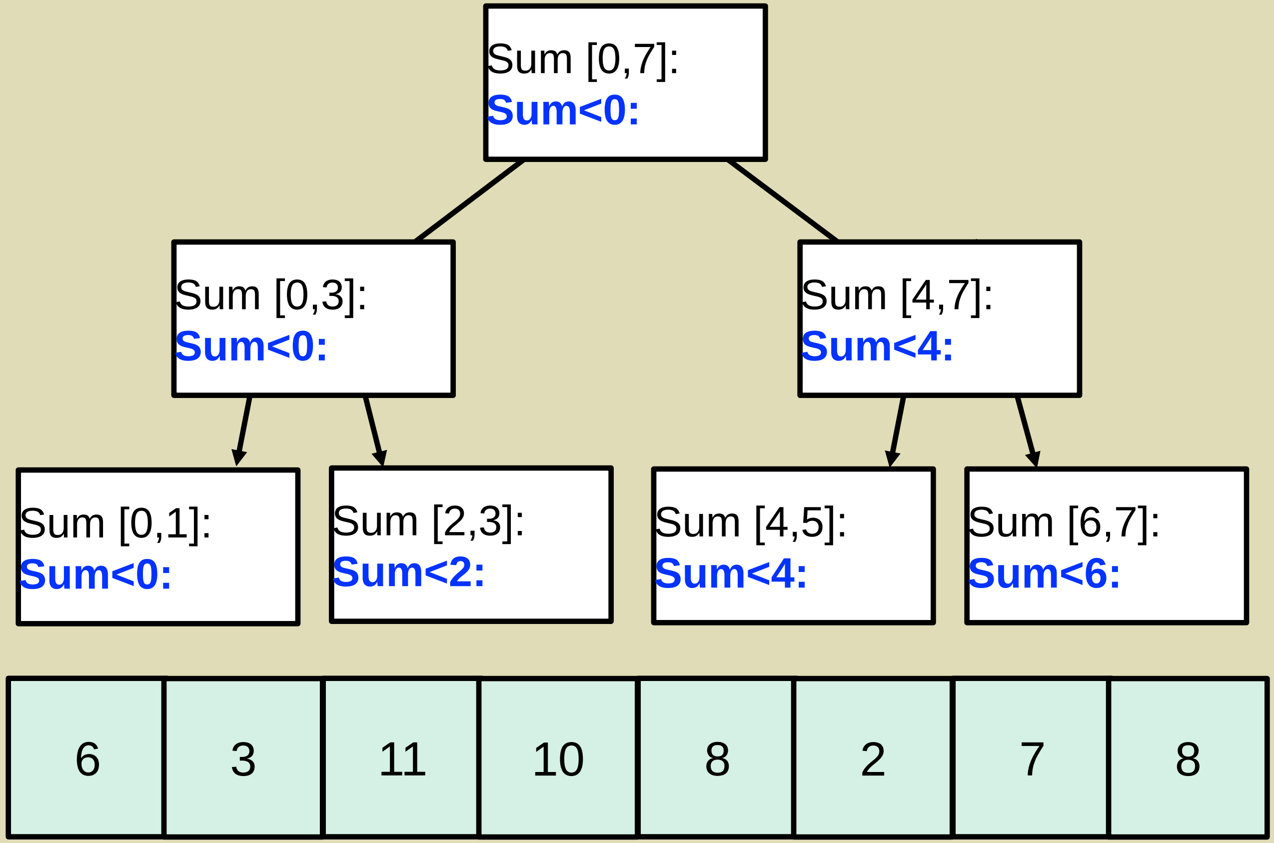
First Pass: Sum



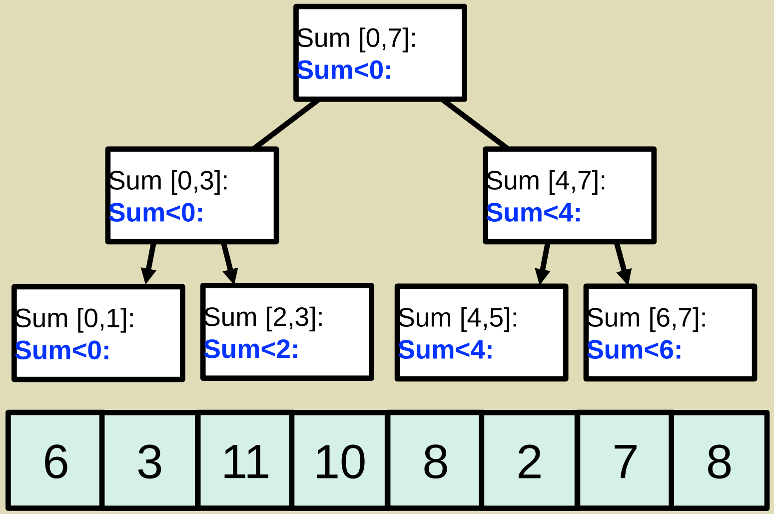
First Pass: Sum



2nd Pass: Use Sum for Prefix-Sum



2nd Pass: Use Sum for Prefix-Sum



Go from root down to leaves

Root

– sum<0 =

Left-child

– sum<K =

Right-child

– sum<K =

Prefix-Sum Analysis

- First Pass (Sum):
  - span =
- Second Pass:
  - single pass from root down to leaves
    - update children's sum<K value based on parent and sibling
  - span =
- Total
  - span =

Parallel Prefix, Generalized

Prefix-sum is another common pattern (prefix problems)

- maximum element to the left of i
- is there an element to the left of i i satisfying some property?
- count of elements to the left of i satisfying some property
- …

We can solve all of these problems in the same way

Pack

Pack:

`input`	6	3	11	10	8	2	7	8
`output`								

`test:  X < 8?`

Output array of elements satisfying **`test`**, in original order

Parallel Pack?

Pack

`input`	6	3	11	10	8	2	7	8
`output`	6	3	2	7				

`test: X < 8?`

- Determining **which** elements to include is **easy**
- Determining **where** each element goes in output is **hard**
  - seems to depend on previous results

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8
test	1	1	0	0	0	1	1	0

test: X < 8?

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8
test	1	1	0	0	0	1	1	0

test: X < 8?

2. transform bit vector into array of indices into result array

pos	1	2				3	4	

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8
test	1	1	0	0	0	1	1	0

test: X < 8?

2. prefix-sum on bit vector

pos	**1**	**2**	2	2	2	**3**	**4**	4

3. map input to corresponding positions in output

output	6	3	2	7				

- `if (test[i] == 1) output[pos[i]] = input[i]`

Parallel Pack Analysis

- Parallel Pack
  1. map: O( ) span
  2. sum-prefix: O( ) span
  3. map: O( ) span

- Total: O( ) span

Sequential Quicksort

Quicksort (review):

1. Pick a pivot — O(1)
2. Partition into two sub-arrays — O(n)
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B — 2T(n/2), avg

Complexity (avg case)

- T(n) = n + 2T(n/2)   T(0) = T(1) = 1
- O(n logn)

How to parallelize?

Parallel Quicksort

Quicksort

1. Pick a pivot O(1)
2. Partition into two sub-arrays O(n)
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B in parallel T(n/2), avg

Complexity (avg case)

- T(n) = n + T(n/2) T(0) = T(1) = 1
- Span: O( )
- Parallelism (work/span) = O( )

Taking it to the next level…

- O(**`log n`**) speed-up with infinite processors is okay, but a bit underwhelming
  - Sort 10^9 elements 30x faster

- Bottleneck:

Parallel Partition

Partition into sub-arrays

A. values less than pivot
B. values greater than pivot

What parallel operation can we use for this?

Parallel Partition

- Pick pivot

8	1	4	9	0	3	5	2	7	6

- Pack (test: **<6**)

1	4	0	3	5	2				

- Right pack (test: **>=6**)

1	4	0	3	5	2	6	8	9	7

Parallel Quicksort

Quicksort

1. Pick a pivot — O(1)
2. Partition into two sub-arrays — O(   ) span
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B in parallel — T(n/2), avg

Complexity (avg case)

- T(n) = O(   ) + T(n/2)   T(0) = T(1) = 1
- Span: O(   )
- Parallelism (work/span) = O(   )

Sequential Mergesort

Mergesort (review):

1. Sort left and right halves — $2T(n/2)$
2. Merge results — $O(n)$

Complexity (worst case)

- $T(n) = n + 2T(n/2)$ $\qquad$ $T(0) = T(1) = 1$
- $O(n \log n)$

How to parallelize?

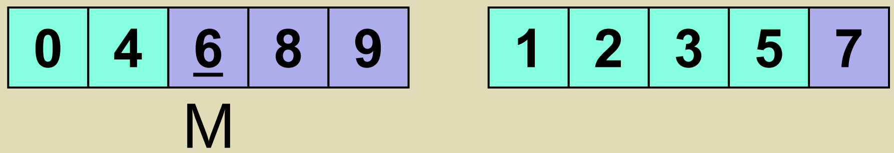
- Do left + right in parallel, improves to $O(n)$
- To do better, we need to…

Parallel Merge

0	4	6	8	9		1	2	3	5	7

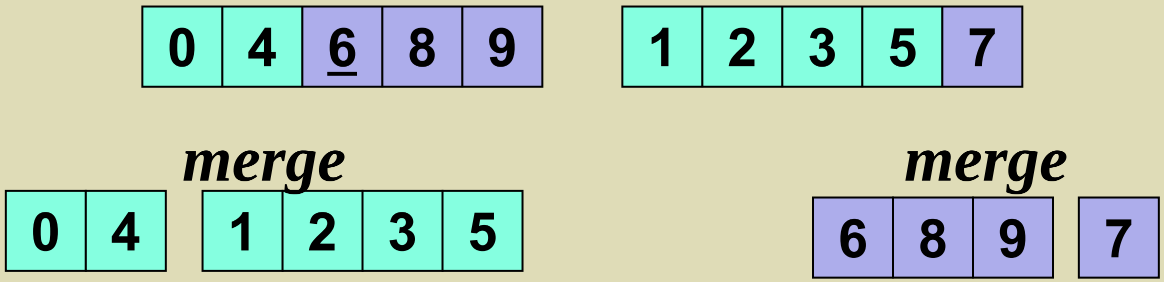
How to merge two sorted lists in parallel?

Parallel Merge

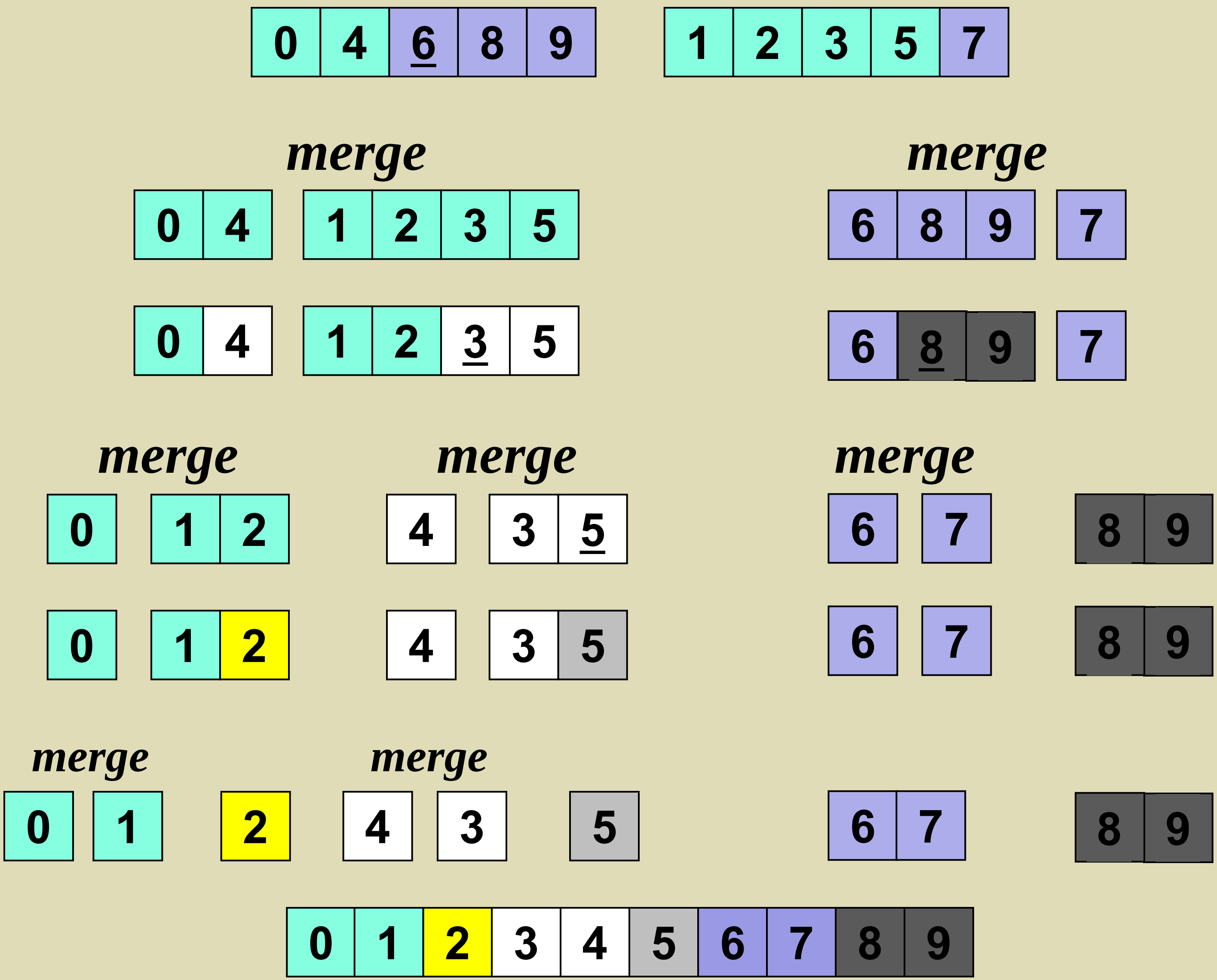


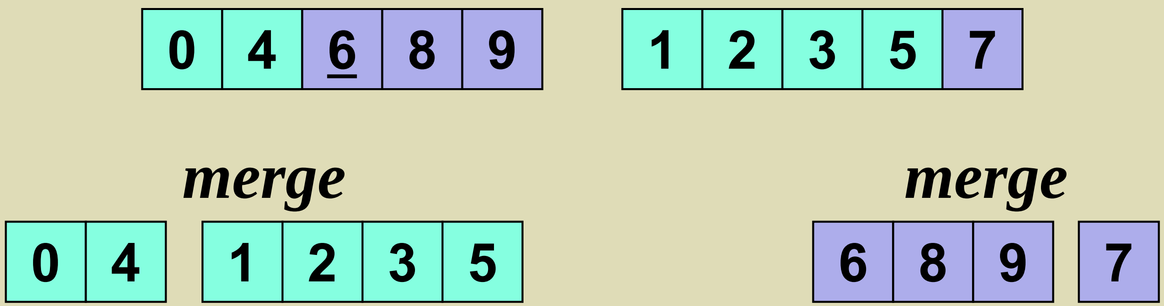
1. Choose median M of left half O( )
2. Split both arrays into < M, >=M O( )
   - how?

Parallel Merge



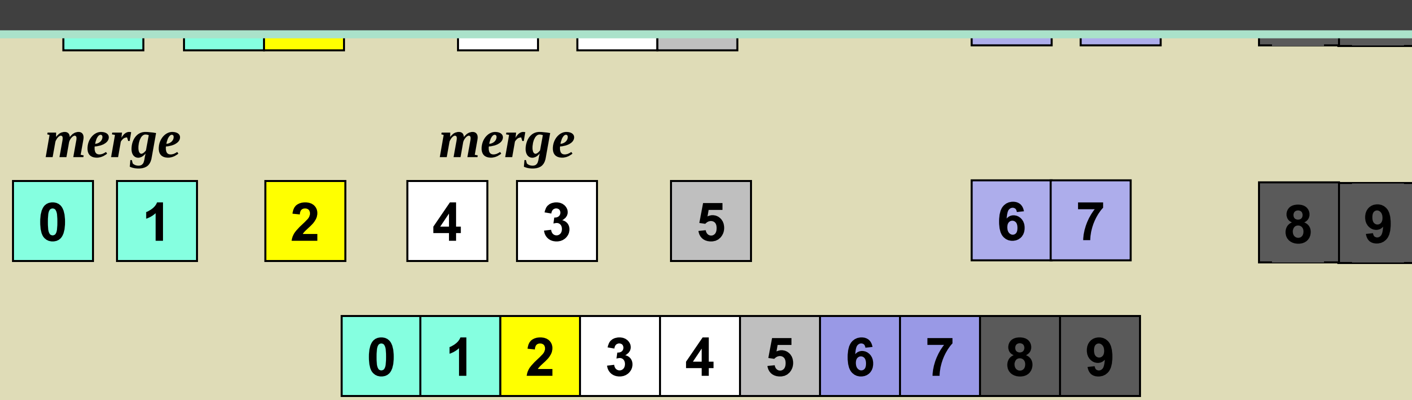
1. Choose median M of left half
2. Split both arrays into < M, >=M
   – how?
3. Do two submerges in parallel


0 4 6 8 9
1 2 3 5 7
merge
merge
0 4
1 2 3 5
6 8 9
7
0 4
1 2 3 5
6 8 9
7
merge
merge
merge
0
1 2
4
3 5
6
7
8 9
0
1 2
4
3 5
6
7
8 9
merge
merge
0
1
2
4
3
5
6 7
8 9
0 1 2 3 4 5 6 7 8 9



When we do each merge in parallel:

- we split the bigger array in half
- use binary search to split the smaller array
- And in base case we copy to the output array



Parallel Mergesort 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

Parallel Merge (worst case)

- Height of partition call tree with n elements: O( )
- Complexity of each thread (ignoring recursive call): O( )
- Span: O( )

Parallel Mergesort (worst case)

- Span: O( )
- Parallelism (work / span): O( )

Subtlety: uneven splits

0	4	6	8

1	2	3	5

- but even in worst case, get a 3/4 to 1/4 split
  - still gives O(log n) height

Parallel Quicksort vs. Mergesort

Parallelism (work / span)

- quicksort: $O(n / \log n)$ avg case
- mergesort: $O(n / \log^2 n)$ worst case