CSE 332: Parallel Algorithms

Richard Anderson

Spring 2016

Announcements

Project 2: Due tonight

Project 3: Available soon

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 = $\mathbf{T_\infty}$

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

Amdahl's Fairly Trivial Observation

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

- Let S = proportion that can't be parallelized, and normalize $\mathbf{T_1}$ to 1

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

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

$$\mathbf{T_P = S + (1\text{-}S)/P}$$

- So the overall speed-up on P processors is (Amdahl's Law): $\mathbf{T_1 / T_P = 1 / (S + (1\text{-}S)/P)}$

$$\mathbf{T_1 / T_\infty = 1 / S}$$

Results from Friday

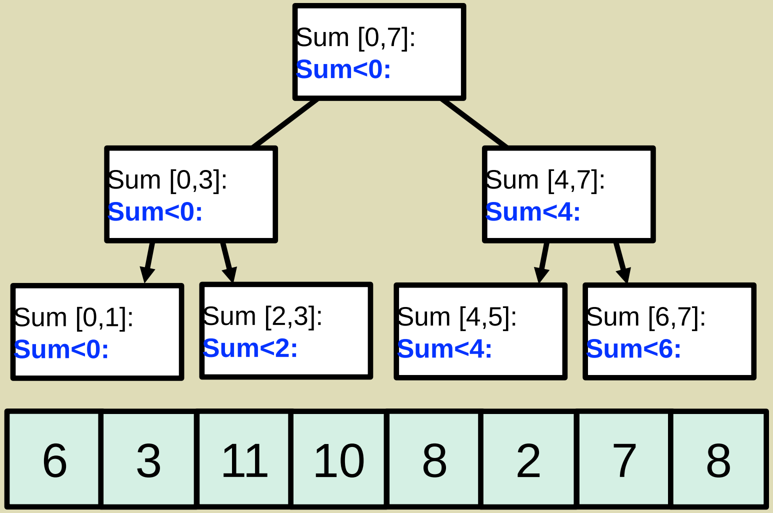
- Parallel Prefix
  - O(N) Work
  - O(log N) Span

- Quicksort
  - Partition can be solved with Parallel Prefix
  - Overall result
    - O(N log N) work, $O(\log^2 N)$ Span

Prefix sum

Prefix-sum:

input	6	3	11	10	8	2	7	8
output	6	9	20	30	38	40	47	55



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 — $O(\log n)$ 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)

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

– Span: $O(\log^2 n)$

– Parallelism (work/span) = $O(n / \log n)$

Sequential Mergesort

Mergesort (review):

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

Complexity (worst case)

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

How to parallelize?

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

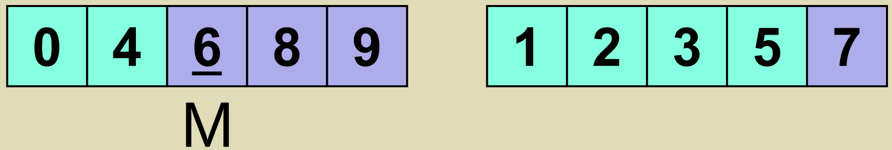
Parallel Merge

0	4	6	8	9

1	2	3	5	7

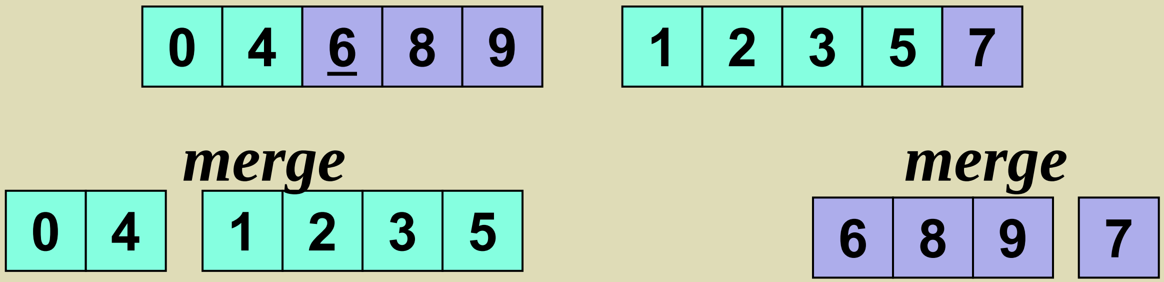
How to merge two sorted lists in parallel?

Parallel Merge

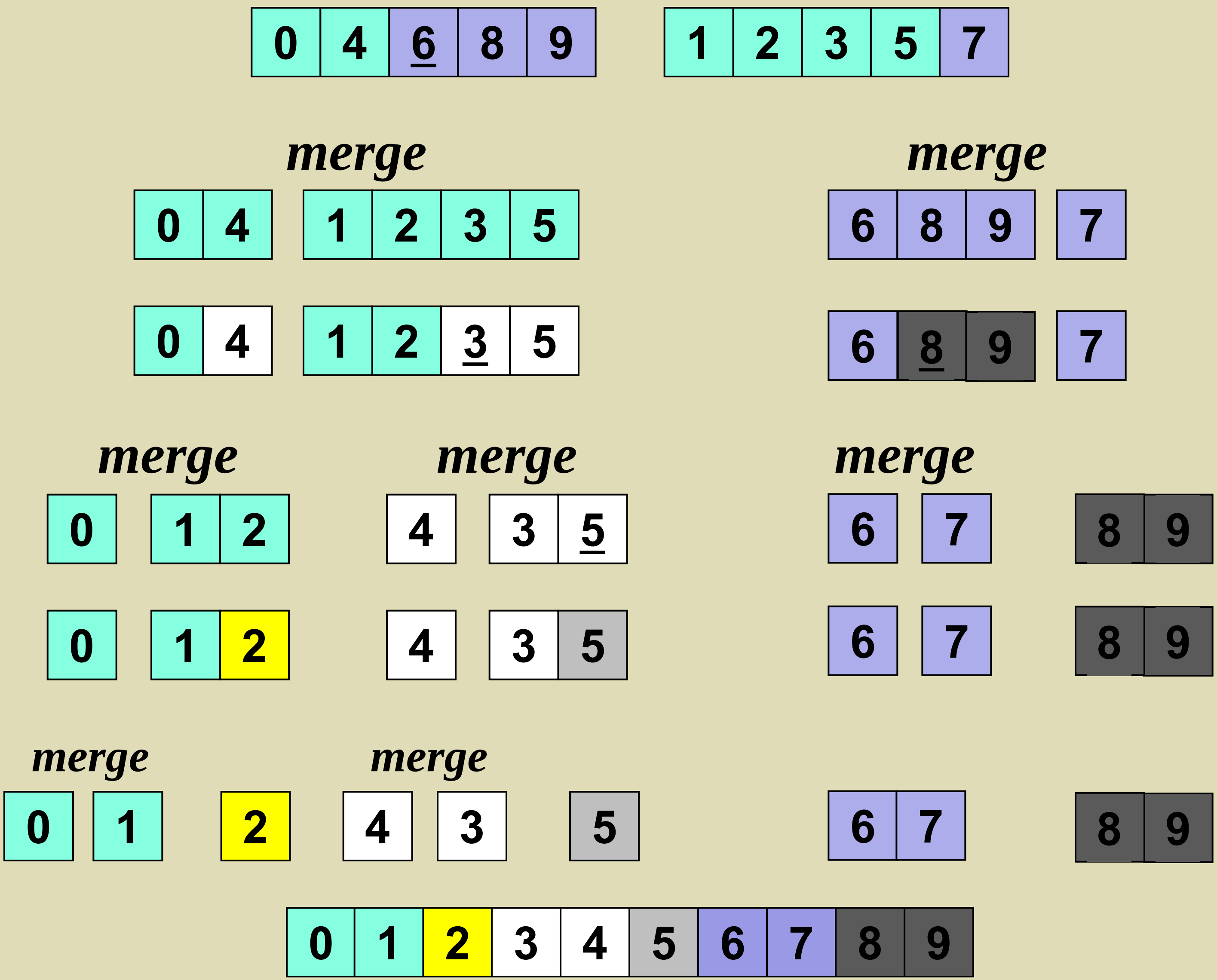


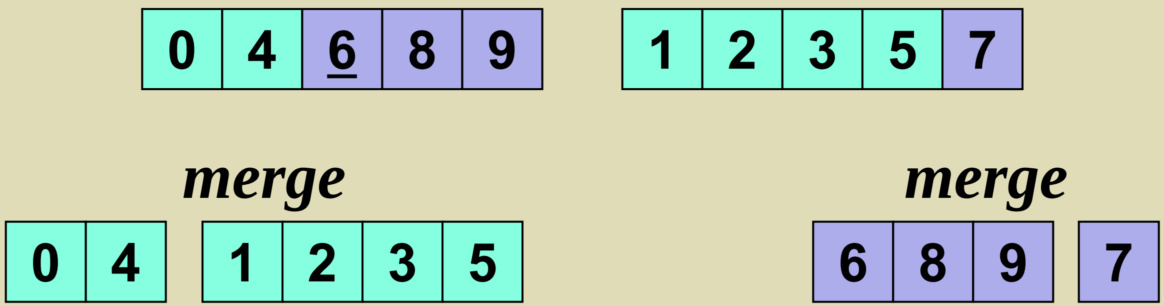
1. Choose median M of left half O( )
2. Split both arrays into < M, >=M O( )
   - how?

Parallel Merge



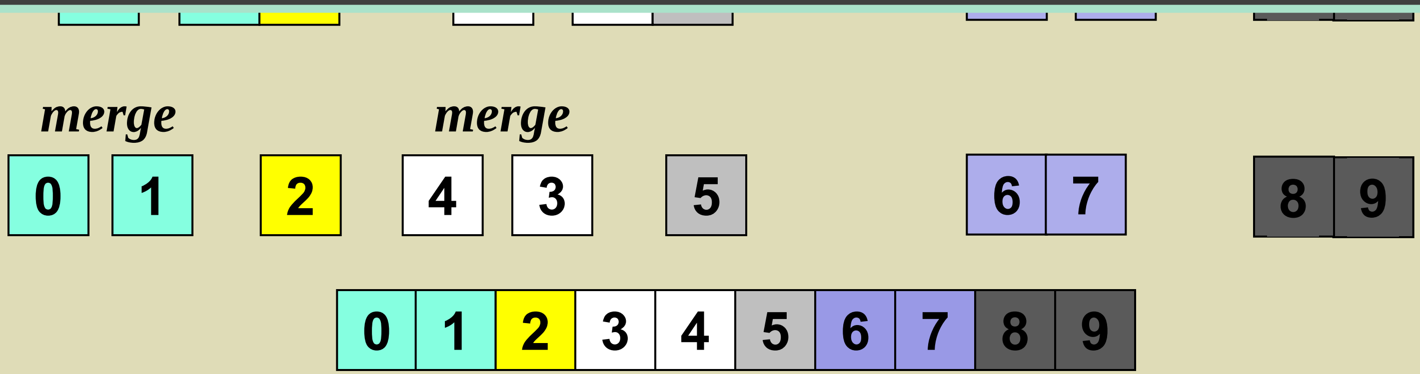
1. Choose median M of left half
2. Split both arrays into < M, >=M
   - how?
3. Do two submerges in parallel


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

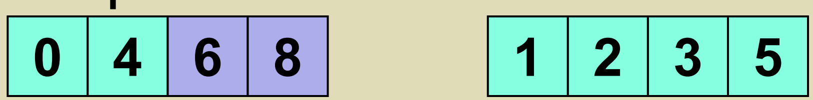
Parallel Merge (worst case)

- 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



- 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