

Parallel And Sequential Data Structures and Algorithms

Divide-and-Conquer and Reduce

Learning Objectives

- Understand the general **divide-and-conquer** design pattern
- Understand **reduce** as a parallel generalization of sums
- Learn a work-efficient parallel divide-and-conquer algorithm for **merging** sorted sequences
- Analyze divide-and-conquer algorithms for their **work** and **span** by using **recurrence relations**

Reminder: Recitation

- This semester we are experimenting with new recitation policies
- The recitation handout (on Ed) each week will include a few simple problems that you must complete **before** recitation
- You must **handwrite** your answers (except for approved ODR accommodations) with pen/pencil on paper and give them to your TA within the first ten minutes of recitation starting
- **Laptops should not be used** during recitation (except for approved ODR accommodations). Please bring a **pen/pencil**

What is Divide and Conquer

Definition (Divide and Conquer): A divide-and-conquer algorithm solves a problem by **dividing** it into smaller instances, solving those **recursively**, then **combining** those solutions to solve the original.

Base Case: When the problem size is sufficiently small, solve it directly (without further recursion)

Inductive Step:

⤵ *(Does not have to be two!)*

1. **Divide** the instance into some number of smaller instances
2. **Recursively** solve the smaller instances
3. **Combine** the solutions to solve the original instance

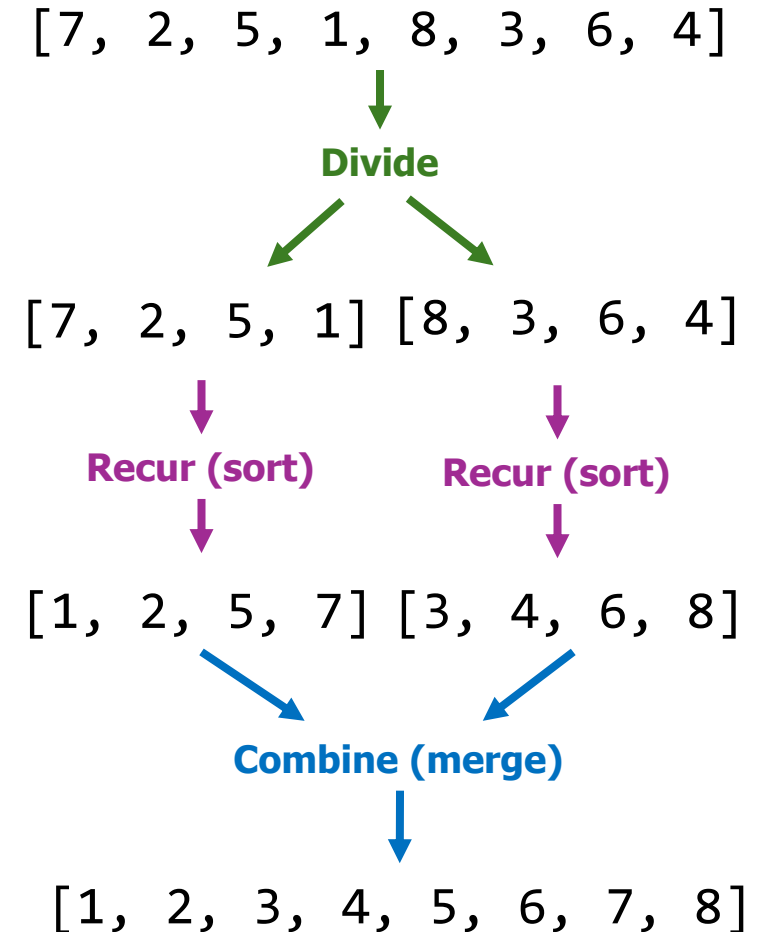
Advantages of Divide and Conquer

- Correctness usually uses **proof by induction**: Prove that the base cases are correct, then assume that the recursive solutions are correct to show that the combined solution is correct.
- Cost analysis can typically be done with **recurrence relations**, which we now have a strong toolbox for solving.
- Divide and conquer is **naturally parallel**; the smaller subproblems can be solved recursively in parallel

A Familiar Example: MergeSort

```
fun mergesort(S : sequence<T>):  
  if |S| <= 1: return S  
  mid = |S|/2  
  L, R = split_mid(S) ← Divide  
  Lsort = mergesort(L) ← Recur  
  Rsort = mergesort(R) ← Recur  
  return merge(Lsort, Rsort) ← Combine
```

```
fun mergesort(S : sequence<T>):  
  if |S| <= 1: return S  
  mid = |S|/2  
  L, R = split_mid(S)  
  Lsort, Rsort = parallel (  
    mergesort(L),  
    mergesort(R))  
  return merge(Lsort, Rsort)
```



Divide-and-Conquer Parallelism

If the sub-instances are **independent**, we can solve them in **parallel**

If our algorithm is written in **pure functional style** (no side effects), all instances are necessarily independent and hence **safe for parallelism**!

- Work is the **sum** of the divide, recursive, and combine steps

$$W(n) = W_{\text{divide}}(n) + \sum W(n_i) + W_{\text{combine}}(n)$$

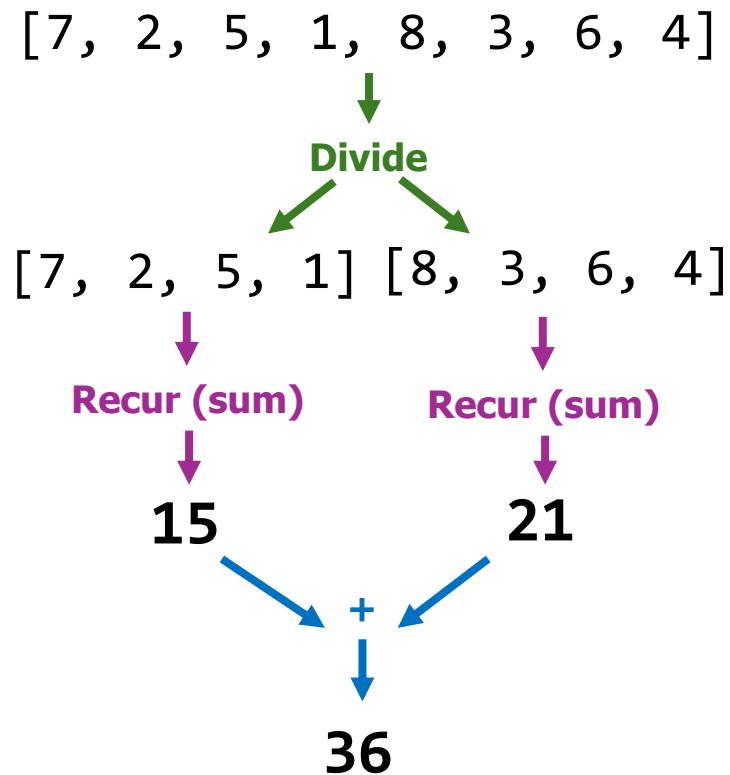
- Span is the sum of the D&C plus **max** of the recursive steps

$$S(n) = S_{\text{divide}}(n) + \max S(n_i) + S_{\text{combine}}(n)$$

Reduce

Example: Parallel Sum

Problem (Sum): Given a sequence of integers, compute their total sum



```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
  case 0: return 0          // Empty sequence
  case 1: return S[0]       // Singleton
  case _:
    L, R = split_mid(S)    // Helper function
    Lsum, Rsum = parallel (sum(L), sum(R))
    return Lsum + Rsum
```

Analysis of Parallel Sum

- The **divide step** is `split_mid`: costs $O(1)$ work and span
- The **combine step** is `+`: costs $O(1)$ work and span
- There are two recursive calls on subproblems of size $n/2$

Work: $W_{\text{sum}}(n) = 2W_{\text{sum}}(n/2) + O(1)$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Number of leaves: n

Work per leaf: $O(1)$

Solution: $O(n)$

Analysis of Parallel Sum

- The **divide step** is `split_mid`: costs $O(1)$ work and span
- The **combine step** is `+`: costs $O(1)$ work and span
- There are two recursive calls on subproblems of size $n/2$

Span: $S_{\text{sum}}(n) = S_{\text{sum}}(n/2) + O(1)$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

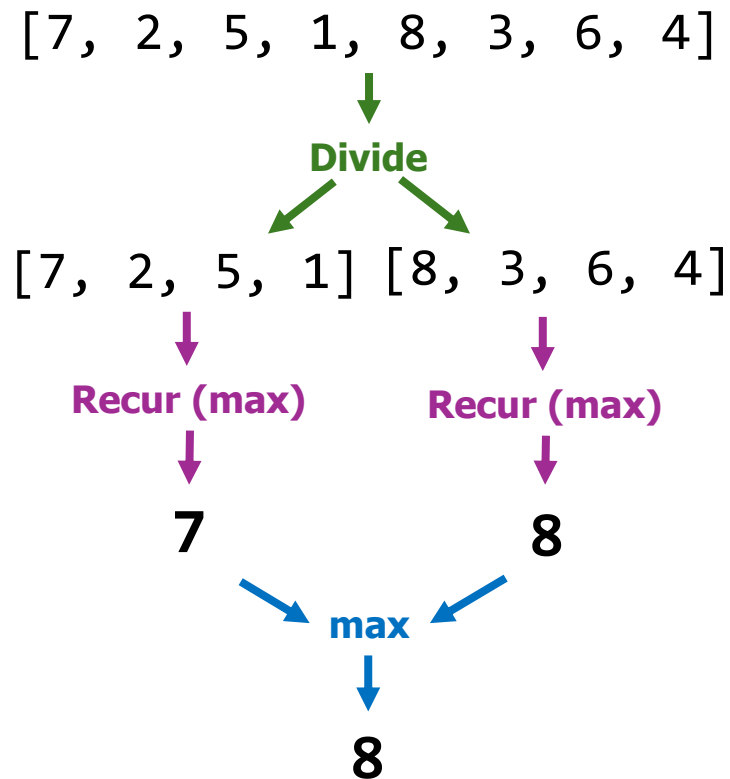
Number of levels: $\lceil \log_2(n + 1) \rceil$

Span per level: $O(1)$

Solution: $O(\log n)$

Example: Parallel Max

Problem (Max): Given a sequence of integers, compute their maximum



```
fun maxv(S : sequence<int>) -> int:  
  match length(S) with:  
  case 0: return -∞      // Empty sequence  
  case 1: return S[0]    // Singleton  
  case _:  
    L, R = split_mid(S)  // Helper function  
    Lmax, Rmax = parallel (maxv(L), maxv(R))  
    return max(Lmax, Rmax)
```

Example: Parallel Min

Problem (Min): Given a sequence of integers, compute their minimum

[7, 2, 5, 1, 8, 3, 6, 4]

↓
Divide

Hang on...

This seems like it is just one algorithm with a different base case and combine!

```
fun minv(S : sequence<int>) -> int:
  match length(S) with:
  case 0: return ∞           // Empty sequence
  case 1: return S[0]        // Singleton
  case _:
    L, R = split_mid(S)      // Helper function
    Lmin, Rmin = parallel (minv(L), minv(R))
    return min(Lmin, Rmin)
```

A General Algorithm

- This same pattern could be used for computing many different aggregates over a sequence:
 - sum, product,
 - min, max,
 - Boolean or, Boolean and
- The only change is the operation used to **combine** answers and the values used for the **base case**
- This primitive is called **reduce**

Reduce

- Given a type T with a **combining function** f and a **base-case value** for an empty sequence I
- **Reduce** computes the aggregate value of a sequence of Ts

```
fun reduce(f : (T, T) -> T, I : T, S : sequence<T>) -> T:  
  match length(S) with:  
  case 0: return I  
  case 1: return S[0]  
  case _:  
    L, R = split_mid(S)  
    Lres, Rres = parallel (reduce(f, I, L), reduce(f, I, R))  
    return f(Lres, Rres)
```

Work and Span of Reduce

Cost (Reduce): The reduce function costs $O(n)$ work and $O(\log n)$ span, assuming that f can be evaluated in $O(1)$ cost.

Proof: Same recurrences we did for parallel sum!

Work: $W_{\text{sum}}(n) = 2W_{\text{sum}}(n/2) + O(1)$

Span: $S_{\text{sum}}(n) = S_{\text{sum}}(n/2) + O(1)$

(c) Leaf Dominated

Number of leaves: n

Work per leaf: $O(1)$

Solution: $O(n)$

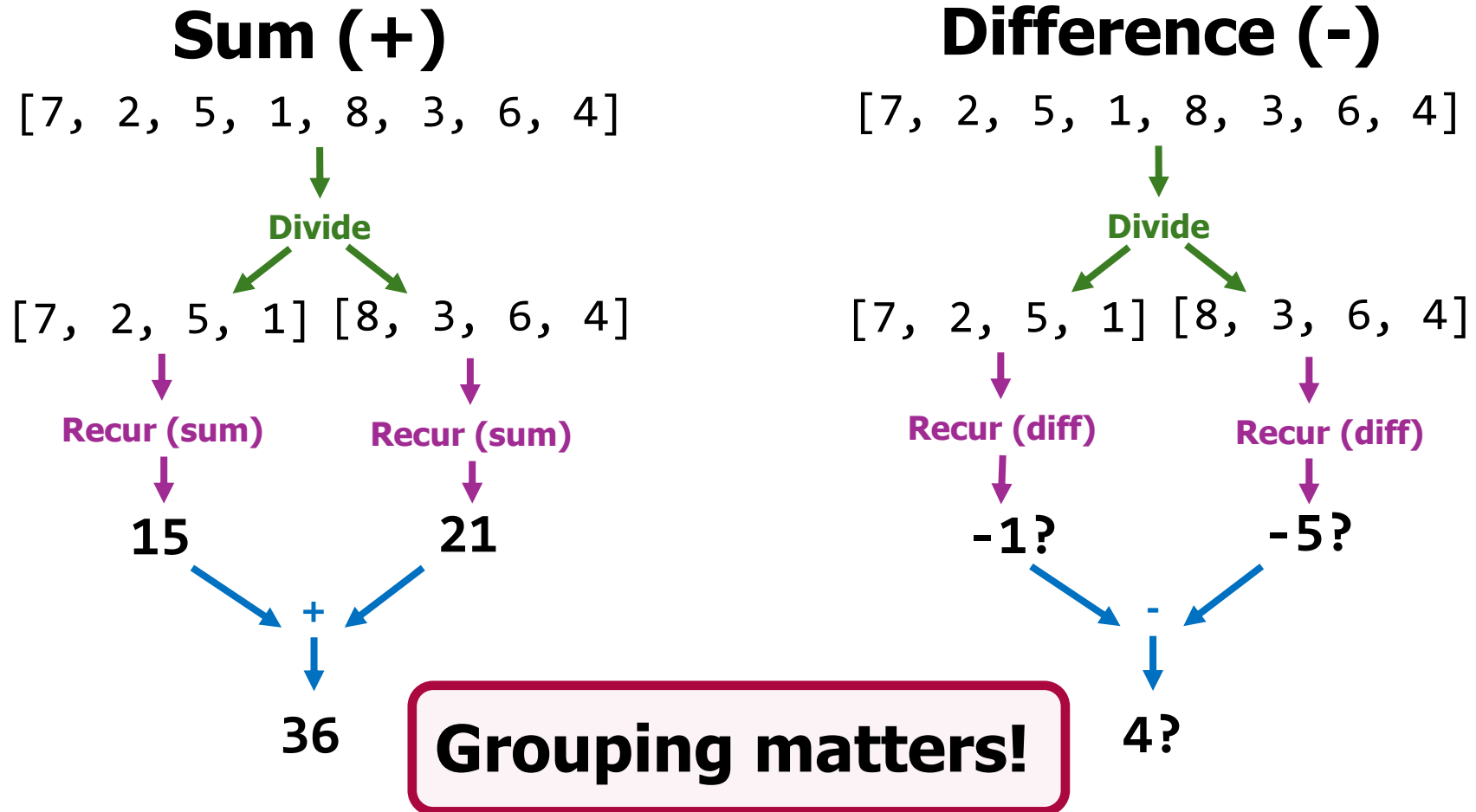
(b) Balanced

Number of levels: $\lceil \log_2(n + 1) \rceil$

Span per level: $O(1)$

Solution: $O(\log n)$

Can We Use Any Operation?



Preconditions for Reduce

- Reduce only produces a well-defined answer if the combine function gives the same result regardless of how its applications are **parenthesized/grouped**.

Definition (Associative): A binary operator f is **associative** if for all values x, y, z , we have $f(f(x, y), z) = f(x, f(y, z))$

- Furthermore, the base case value I must be an **identity value**

Definition (Identity): A value I is an **identity** for an associative binary operator f if for any value x , we have $f(I, x) = x$ and $f(x, I) = x$

Merge

Merge

- Recall the merge operation from 15-122 and 15-150:

Definition (Merge): Given two sorted sequences, return a sorted sequence containing the elements of both

fun merge(A : sequence<T>, B : sequence<T>) -> sequence<T>

- A simple sequential implementation runs in $O(n)$ time
- How can we make this parallel?

Answer: Let's try divide-and-conquer!

Divide-and-conquer merge

Question: Right off the bat, what makes merge more complicated than most divide-and-conquer algorithms?

- Input is *two sequences*, not one! May not be the same length!
- Which one(s) do we divide? What if we divide both in half?

1, 2, 4, 6 5, 8 8, 9, 12, 16
1, 2, 3, 4, 5, 6, 7, 10 8, 9, 11, 12, 13, 14, 15, 16
3, 5, 7, 10, 11, 13, 14, 15
Not sorted!

Finding the Split Point

- Let's still split the first sequence (A) in half

Question: Where should B be split?

Answer: Left side should have numbers less than $A[\text{mid}]$ (i.e., 8)

1, 2, 4, 6	5, 8	8, 9, 12, 16
1, 2, 3, 4, 5, 6, 7		8, 9, 10, 11, 12, 13, 14, 15, 16
3, 5, /	10, 1	10, 11, 13, 14, 15

Making Merge Efficient

- Divide-and-conquer is efficient when splitting the input in halves
- We always split A in half, but B might be split badly

Question: How can we fix this?

Answer: Always split the *larger* of the two sequences

Question: How do we find the split point?

Answer: Use binary search!

Implementing Merge

```
fun merge(A: sequence<T>, B : sequence<T>) -> sequence<T>:  
    if |B| > |A|: swap(A, B) // WLOG assume A is larger than B  
  
    if |B| == 0: return A  
    if |A| == |B| == 1: return [min(A[0], B[0]), max(A[0], B[0])]  
  
    LA, RA = split_mid(A)  
    k = binary_search(B, RA[0]) // k = smallest index such that B[k] >= RA[0]  
    LB, RB = subseq(B,0,k), subseq(B,k,|B|-k)  
    ML, MB = parallel (merge(LA, LB), merge(RA, RB))  
    return append(ML, MB)
```

Oops: This implementation still has an efficiency problem!

Cost Analysis of Merge

- For our analysis, we will define the size of the problem as $n = |A| + |B|$
- Our algorithm splits the larger sequence in half, so WLOG $|A| \geq |B|$



- Define the size of the left subproblem $n_L = |LA| + |LB|$
- Define the size of the right subproblem $n_R = |RA| + |RB|$

Lemma (Even Split): Both subproblem sizes decrease geometrically:

$$\frac{n}{4} \leq n_L, n_R \leq \frac{3n}{4}$$

Cost Analysis of Merge

Lemma (Even Split): Both subproblem sizes decrease geometrically:

$$\frac{n}{4} \leq n_L, n_R \leq \frac{3n}{4}$$

Proof:

- Both subproblems get half of A , which is at least half of n
- The smallest a subproblem can be is if it gets none of B :

$$\frac{|A|}{2} \geq \frac{1}{2} \cdot \frac{n}{2} = \frac{n}{4}$$

Cost Analysis of Merge

Lemma (Even Split): Both subproblem sizes decrease geometrically:

$$\frac{n}{4} \leq n_L, n_R \leq \frac{3n}{4}$$

Proof:

- Both subproblems get half of A , which is at least half of n
- The largest a subproblem can be is if it gets all of B :

$$\frac{|A|}{2} + |B| = \frac{n - |B|}{2} + |B| = \frac{n}{2} + \frac{|B|}{2} \leq \frac{3n}{4}$$

Cost Analysis of Merge

Claim (Cost of Merge): This merge function costs $\Theta(n \log n)$ work

- The **combine step** is $\text{append}(ML, MB)$, which costs $O(n)$
- This makes the work recurrence:

$$W(n) = W(n_L) + W(n_R) + \Theta(n)$$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Cost of Parent: n

Cost of Children: $n_L + n_R = n$

Number of levels: $\Theta(\log n)$

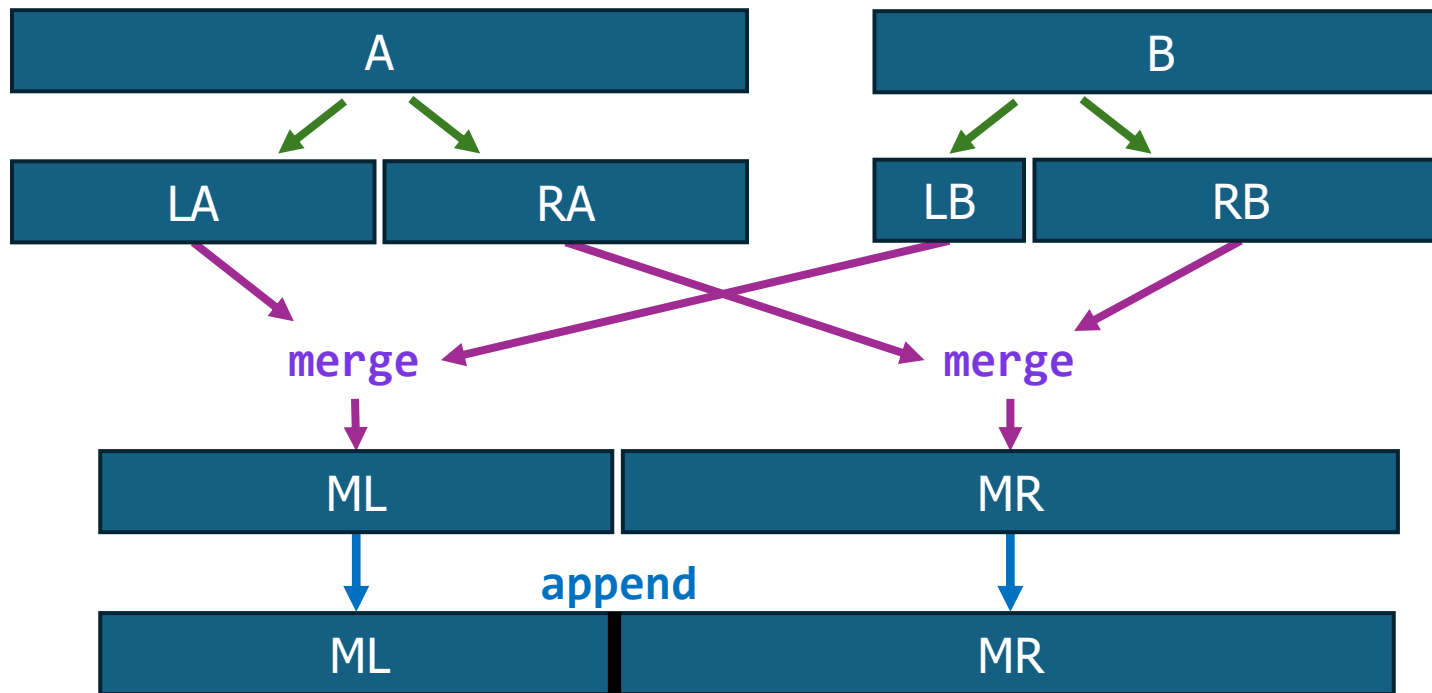
Work per level: $\Theta(n)$

Solution: $\Theta(n \log n)$

Making it Faster

The issue with append-based merge is that the combine step costs $O(n)$

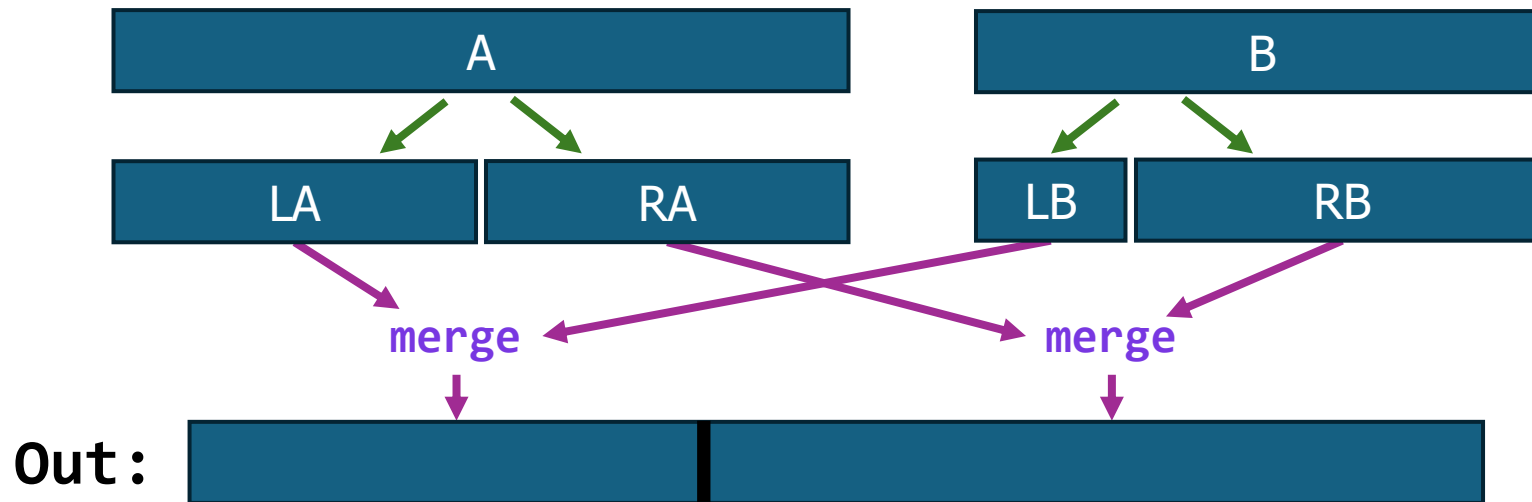
- Claim: the append is **entirely redundant work**



Observation: We know in advance exactly how large each recursive result will be and exactly where it belongs in the final output.

We Need Imperative Code!

Key Idea: Allocate the output just once, and have each recursive call write each element of its answer directly into the correct final location.



Observation: The combine step is now **empty**. When the recursive calls finish, **the answer is already there!**

Efficient (Imperative) Merge

```
fun merge(A: sequence<T>, B : sequence<T>, Out : mutable array_sequence<T>):  
    if |B| > |A|: swap(A, B)  
  
    if |B| == 0: Out[0...|A|] ← A; return  
    if |A| == |B| == 1: Out[0...2] ← [min(A[0], B[0]), max(A[0], B[0])]; return  
  
    LA, RA = split_mid(A)  
    k = binary_search(B, RA[0])  
    LB, RB = subseq(B,0,k), subseq(B,k,|B|-k)  
    Lout, Rout = subseq(Out,0,|LA|+|LB|), subseq(Out,|LA|+|LB|,|RA|+|RB|)  
    _, _ = parallel (merge(LA, LB, Lout), merge(RA, RB, Rout))
```

Analysis of Work-Efficient Merge

Claim (Cost of Merge): The efficient (imperative) merge function costs $\Theta(n)$ work and $\Theta(\log^2 n)$ span.

- The **combine step** is **nothing**, which costs $O(1)$
- The dominant cost of the divide is the binary search, which costs $O(\log n)$

$$W(n) = W(n_L) + W(n_R) + \Theta(\log n)$$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Cost of Parent: $\log n$

Cost of Children: $\log n_L + \log n_R$
 $\geq \log(n/4) + \log(n/4)$
 $= 2\log(n) - 4$

Analysis of Work-Efficient Merge

Claim (Cost of Merge): The efficient (imperative) merge function costs $\Theta(n)$ work and $\Theta(\log^2 n)$ span.

$$W(n) = W(n_L) + W(n_R) + \Theta(\log n)$$

(a) Root dominated, (b) Balanced (c) Leaf Dominated

Number of leaves: ??

Work per leaf: ??

Solution: $O(n)$

How do we use the brick method when the number of leaves and work per leaf varies?

Answer: *It doesn't matter; we just need to know the **total work** of all leaves!*

Analysis of Work-Efficient Merge

Claim (Cost of Merge): The efficient (imperative) merge function costs $\Theta(n)$ work and $\Theta(\log^2 n)$ span.

$$\begin{aligned} S(n) &= \max(S(n_L), S(n_R)) + \Theta(\log n) \\ &\leq S(3n/4) + \Theta(\log n) \end{aligned}$$

(a) Root dominated, **(b) Balanced** (c) Leaf Dominated

Cost of Parent: $\log n$

Cost of Children: $\log(3n/4)$
 $= \log(n) - \log(4/3)$

Number of levels: $\Theta(\log n)$

Span per level: $\Theta(\log n)$

Solution: $\Theta(\log^2 n)$

A Better Merge?

- This merge costs $\Theta(n)$ work and $\Theta(\log^2 n)$ span
- Clearly, we can't beat $\Theta(n)$ work, but can we have better span?

Claim: merge can be implemented with cost $\Theta(n)$ work and $\Theta(\log n)$ span.

- No time to prove it; the idea is simple, but the details are tricky
- **Key idea:** Instead of dividing the big sequence into two pieces, divide it into **lots** of pieces. Specifically, $\sqrt{n}$ pieces
- This makes the recursive problem size decrease much faster, improving the span without increasing the work

Summary

- **Divide-and-conquer** divides a problem into smaller instances, recursively solves them, and combines their solutions
- Independent recursive calls can be performed **in parallel**
- **reduce** generalizes sums to arbitrary associative operations
- Reduce costs $O(n)$ work and $O(\log n)$ span (for $O(1)$ -cost f)
- **Merge** can be implemented with $O(n)$ work and $O(\log^2 n)$ span using divide-and-conquer