

Parallel Sorts

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-07-31A

Announcements

- ❖ Happy Friday!
- ❖ Ex 10-11 are soft due dates
 - But we highly recommend that you meet these deadlines
 - Also we recommend Minimax by next Tuesday

pollev.com/332summer :: tinyurl.com/332-07-31A

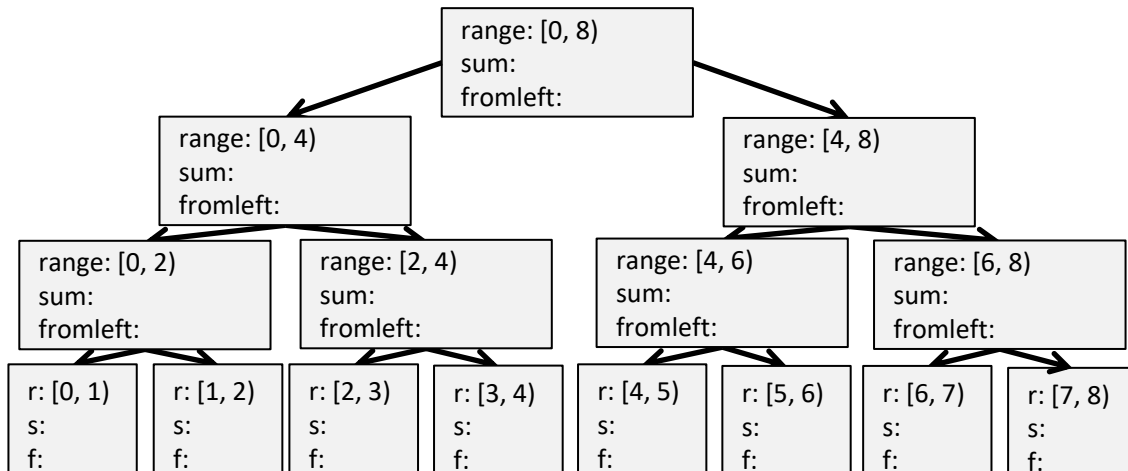
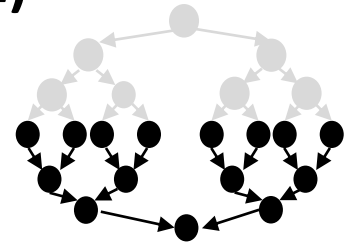
Lecture Outline

- ❖ **Parallel-Pack**
- ❖ Parallel-QuickSort
- ❖ Parallel-MergeSort
- ❖ Sharing Resources
- ❖ Concurrency: Managing Correct Access to Shared Resources

pollev.com/332summer :: tinyurl.com/332-07-31A

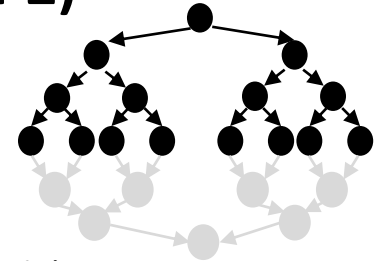
Parallel Prefix-Sum: Summary (1 of 2)

- ❖ 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:

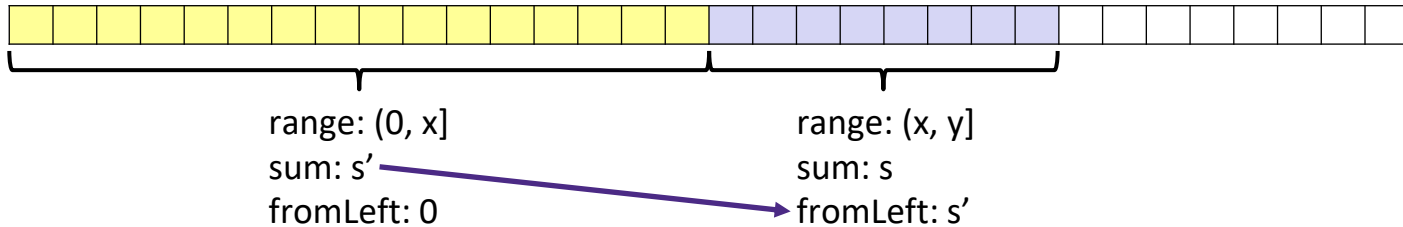


Parallel Prefix-Sum: Summary (2 of 2)

- ❖ 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 $output[i] = fromLeft + input[i]$
- ❖ 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++){

}
```

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];
}
```

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**
- ❖ Parallel-MergeSort
- ❖ Sharing Resources
- ❖ Concurrency: Managing Correct Access to Shared Resources

pollev.com/332summer :: tinyurl.com/332-07-31A

Sequential QuickSort Review

<i>Step</i>	<i>Runtime Expression</i>
Pick the pivot value(s) <ul style="list-style-type: none"> • Hopefully these value(s) approximate the median 	
Partition all the values into: <ol style="list-style-type: none"> The values less than the pivot(s) The pivot(s) The values greater than the pivot(s) 	
Recursively QuickSort(A) and QuickSort(C)	

❖ Recurrence (assuming a good-enough pivot):

- $T(0) = T(1) = c1$
- $T(n) = \underline{\hspace{2cm}}$
- Closed-form $T(n) = \underline{\hspace{2cm}}$

Copied from L7: Algorithm Analysis III

Really Common Recurrences

<i>Recurrence Relation</i>	<i>Closed Form</i>	<i>Name</i>	<i>Example</i>
$T(n) = O(1) + T(n/2)$	$O(\log n)$	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

<i>Step</i>	<i>Runtime Expression</i>
Pick the pivot value(s) <ul style="list-style-type: none">• Hopefully these value(s) approximate the median	c_1
Partition all the values into: <ul style="list-style-type: none">A. The values less than the pivot(s)B. The pivot(s)C. The values greater than the pivot(s)	c_2N
Recursively QuickSort(A) and QuickSort(C)	

- ❖ Let's parallelize the two recursive calls!
 - Work (unchanged): _____
 - $T(n) =$ _____
 - Span: _____

Parallel QuickSort: Doing Better

- ❖ $O(\log n)$ 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
<i>In parallel</i> , 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)	

- ❖ 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 side 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: _____

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

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:

- $T(n) =$ _____

- Span: _____

Lecture Outline

- ❖ Parallel-Pack
- ❖ Parallel-QuickSort
- ❖ **Parallel-MergeSort**
- ❖ Sharing Resources
- ❖ Concurrency: Managing Correct Access to Shared Resources

pollev.com/332summer :: tinyurl.com/332-07-31A

Parallelizing MergeSort

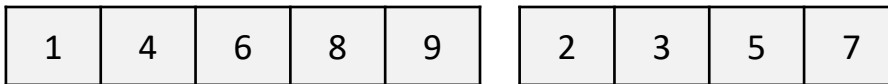
<i>Step</i>	<i>Runtime Expression</i>
Recursively MergeSort(A) and MergeSort(B)	
Merge(A, B)	

- ❖ Just like QuickSort, do the two recursive sorts in parallel:
 - Span is $T(n) = c_1n + \mathbf{1}T(n/2) = O(n)$
 - 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



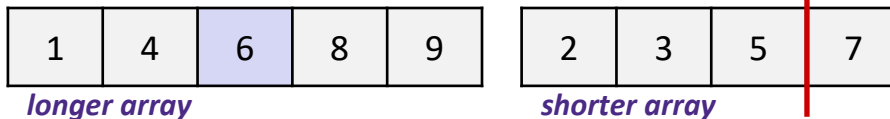
❖ Intuition:

- 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* (\leq 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)

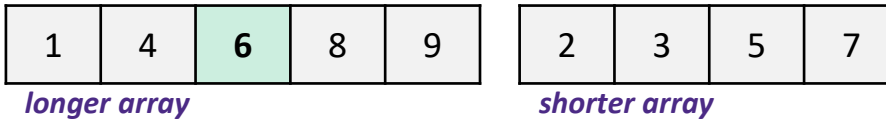
1	4	6	8	9
---	---	---	---	---

longer array

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

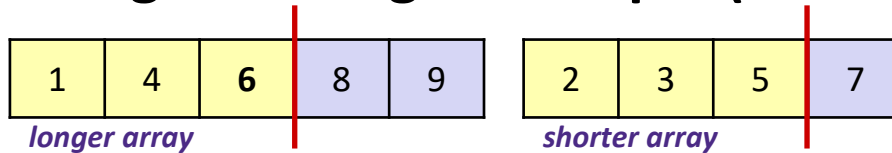
shorter array

Parallelizing the Merge: Example (2 of 7)



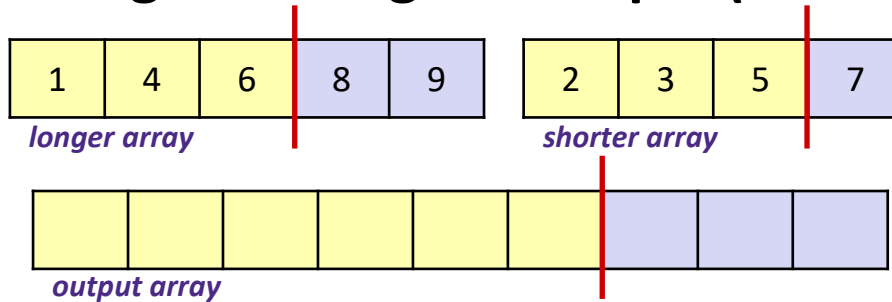
- ❖ Pick the median of the *longer array*: $O(1)$ to compute index

Parallelizing the Merge: Example (3 of 7)



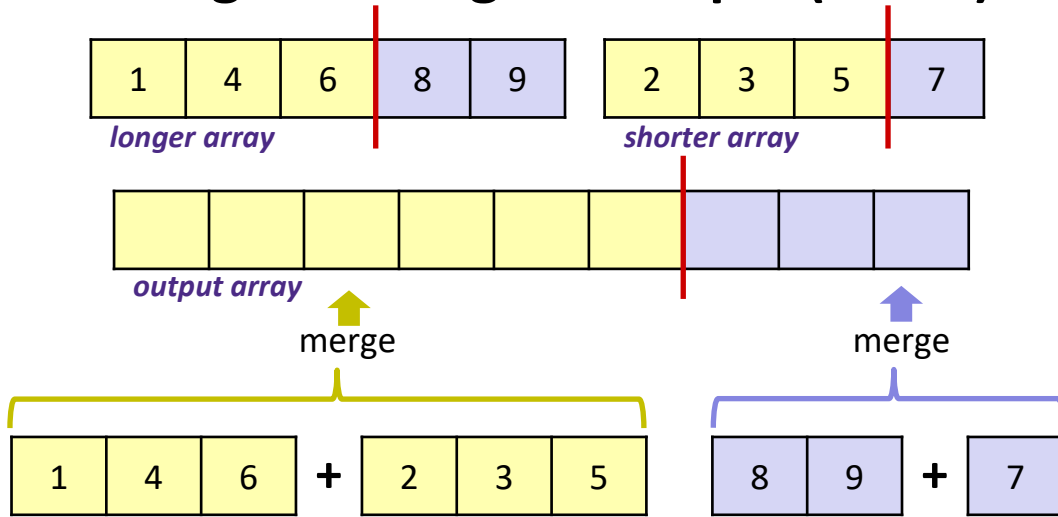
- ❖ Pick the median of the *longer array*: $O(1)$ to compute index
- ❖ Split the *shorter array* at the same value: $O(\log n)$ for binary search

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: $O(1)$

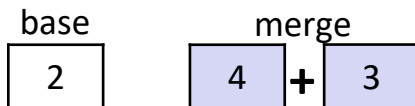
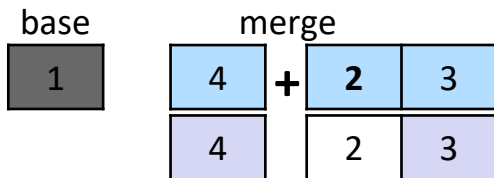
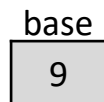
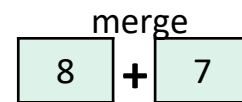
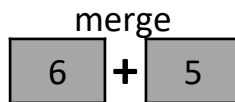
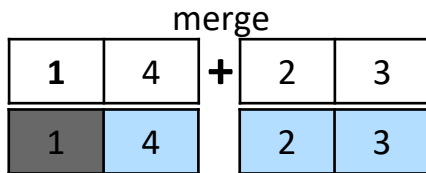
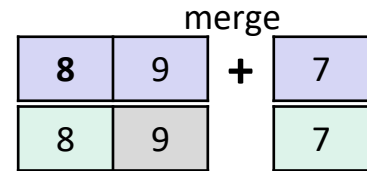
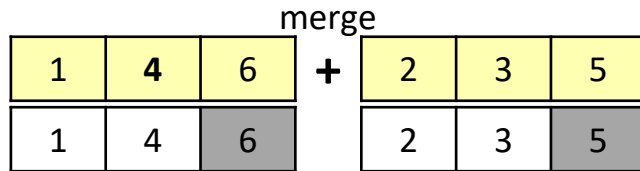
Parallelizing the Merge: Example (5 of 7)



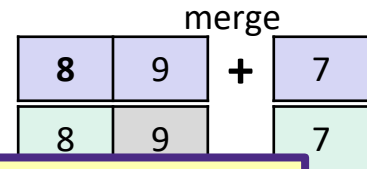
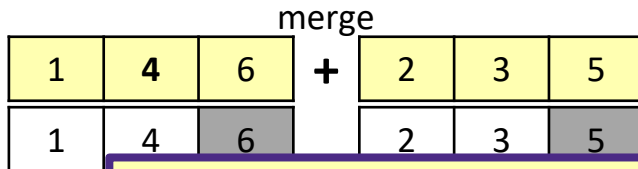
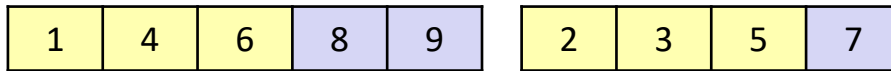
- ❖ Pick the median of the *longer array*: $O(1)$ to compute index
- ❖ Split the *shorter array* at the same value: $O(\log n)$ for binary search
- ❖ Calculate where to split the output array: $O(1)$
- ❖ Do the sub-merges in parallel

🤖 how do we sub-merge? 🤖

Parallelizing the Merge: Example (6 of 7)

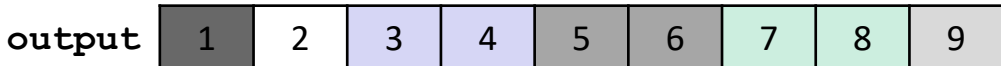
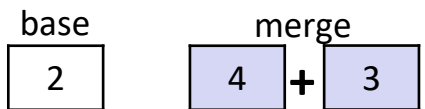
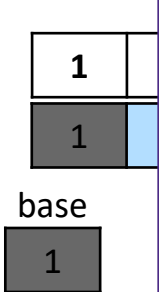


Parallelizing the Merge: Example (7 of 7)



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)

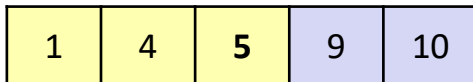
❖ Sequential MergeSort:

$$T(n) = 2T(n/2) + c_2n \quad \in \quad O(n \log n)$$

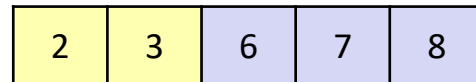
❖ MergeSort with *parallel recursive calls* and a sequential merge:

■ **Work:** $O(n \log n)$

■ **Span:** $T(n) = \mathbf{1}T(n/2) + c_2n \quad \in \quad O(n)$



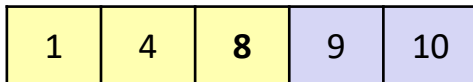
longer array



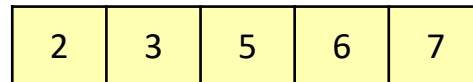
shorter array

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
 - $O(\log n)$ 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”)



longer array



shorter array

Parallel-MergeSort: Analysis (3 of 3)

- ❖ MergeSort with *parallel recursive calls* and a parallel merge:
 - **Work** is $T(n) = 2T(n/2) + c_1n \in O(n \log n)$
 - **Span** is $T(n) = \mathbf{1}T(n/2) + c_2 \log^2 n \in O(\log^2 n)$
 - 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

Lecture Outline

- ❖ Parallel-Pack
- ❖ Parallel-QuickSort
- ❖ Parallel-MergeSort
- ❖ **Sharing Resources**
- ❖ Concurrency: Managing Correct Access to Shared Resources

pollev.com/332summer :: tinyurl.com/332-07-31A

Parallelism and Sharing Resources

- ❖ We've studied **parallel algorithms** using the fork-join model and focused on reducing span via parallel tasks
- ❖ This model has a simple structure to avoid **race conditions**
 - Each thread had a part of memory the “only it accessed”
 - Example: each array sub-range accessed by only one thread
 - Result of forked executor not accessed until after `join()` called
 - Structure (mostly) ensures bad simultaneous access wouldn't occur
- ❖ Model won't work well when:
 - **Memory** accessed by executors is **overlapping** or unpredictable
 - Executors doing **independent tasks** need to **access the same resources**
 - (rather than implementing the same algorithm)

Parallelism's Pitfall

- ❖ Fork-join model doesn't work well when:
 - Executors implementing the same algorithm access **overlapping memory**
 - Executors implementing different algorithms access **the same resources**

Overlapped Sharing (1 of 2)

- ❖ Threads are not just useful for parallelism
 - i.e., not always about implementing algorithms faster

- ❖ Threads are useful for:
 - Responsiveness
 - Respond to events in one thread while another is performing computation
 - Processor utilization (hide I/O latency)
 - If 1 thread “goes to disk,” process still has something else to do
 - Failure isolation
 - Prevent an exception in one task from stopping conceptually-parallel tasks

Overlapped Sharing (2 of 2)

- ❖ What if we have multiple threads:
 - Processing different bank-account operations
 - What if 2 threads modify the same account at the same time?
 - Using a shared cache (e.g., hashtable) of recent files
 - What if 2 threads insert the same file at the same time?
 - Creating a pipeline (think assembly line) with a queue for handing work from one thread to next thread in sequence
 - What if enqueueer and dequeuer adjust a circular array queue at the same time?

Sharing a Queue

- ❖ Imagine 2 threads
 - Running at the same time
 - Accessing a *shared linked-list-based queue*, initially empty

```
enqueue(x) {  
    if (back == null) {  
        back = new Node(x);  
        front = back;  
    }  
    else {  
        back.next = new Node(x);  
        back = back.next;  
    }  
}
```

Overlapped Sharing Needs Concurrency

- ❖ **Concurrency**: *Correctly and efficiently* managing access to shared resources from multiple possibly-simultaneous clients
 - Requires *coordination*, particularly **synchronization**, to avoid incorrect simultaneous access
 - Make thread *block* (wait) until the resource is free
 - `join` is not what we want
 - Want other thread to be “done using *what we need*”, not “completely done executing”
- ❖ Correct concurrent applications are usually highly **non-deterministic**
 - How threads are scheduled affects order of operations
 - Non-repeatability complicates testing and debugging

Attributes of Concurrent Programs

- ❖ In concurrent programs, it is common that:
 - Threads access the same resources in an *unpredictable order*
 - Threads access the same resources at (*approx.*) *the same time*
 - Correctness requires that simultaneous access be prevented
 - Simultaneous access is rare
 - Makes testing and debugging difficult
 - Rare != Impossible; need to be disciplined when designing / implementing

- ❖ In other words: concurrent programs are non-deterministic

Lecture Outline

- ❖ Parallel-Pack
- ❖ Parallel-QuickSort
- ❖ Parallel-MergeSort
- ❖ Sharing Resources
- ❖ **Concurrency: Managing Correct Access to Shared Resources**

pollev.com/332summer :: tinyurl.com/332-07-31A

Concurrency: Canonical Example

- ❖ In a single-threaded world, this code is correct!

```
class BankAccount {
    private int balance = 0;

    protected int getBalance()      { return balance; }
    protected void setBalance(int x) { balance = x; }

    public void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }

    // ... other operations like deposit(), etc.
}
```

Interleaving

- ❖ Suppose:
 - Thread **T1** calls **`x.withdraw(100)`**
 - Thread **T2** calls **`y.withdraw(100)`**
- ❖ If second call starts before first finishes, we say they **interleave**
 - e.g. T1 runs for 50 ms, pauses somewhere, T2 picks up for 50ms
 - Can happen with one processor; if **pre-empted** due to time-slicing
- ❖ If **`x`** and **`y`** refer to different accounts, no problem
 - “You cook in your kitchen while I cook in mine”
 - But if **`x`** and **`y`** alias, possible trouble...

Activity: What is the Balance at the End?

- ❖ Two threads both `withdraw()` from the same account:
 - Assume initial balance == 150

```
class BankAccount {
    private int balance = 0;

    protected int getBalance()      { return balance; }
    protected void setBalance(int x) { balance = x; }

    public void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }

    // ... other operations, etc.
}
```

Thread A

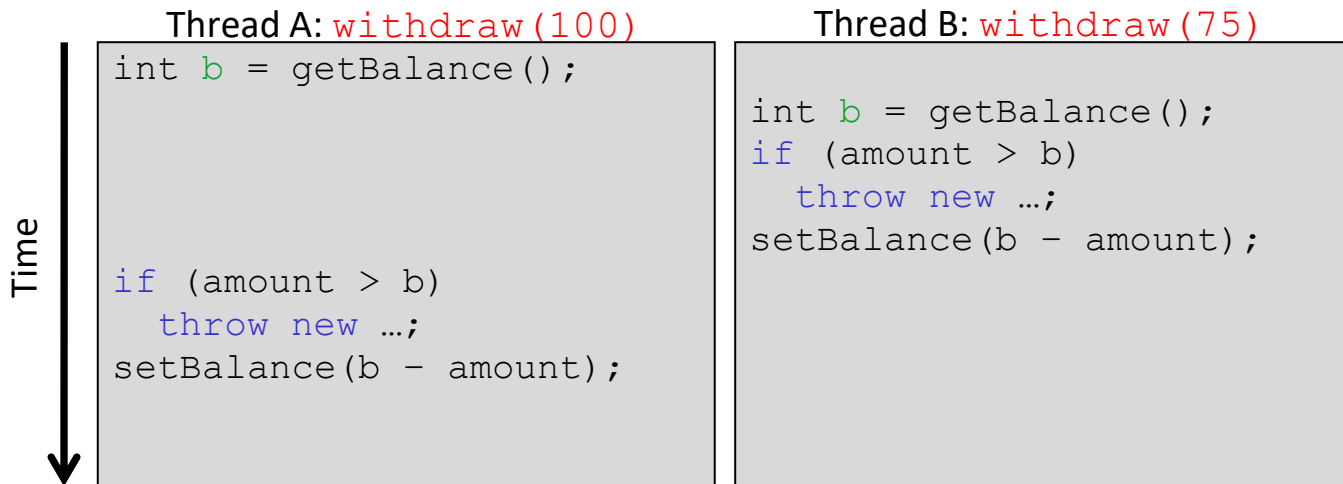
```
x.withdraw(100);
```

Thread B

```
x.withdraw(75);
```

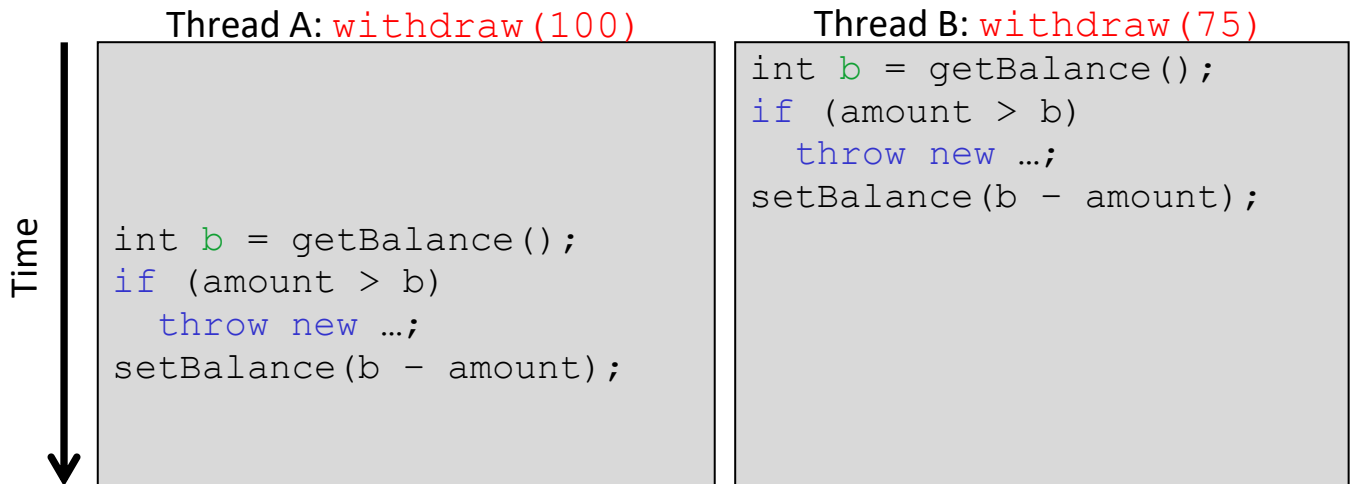
Activity: A Bad Interleaving

- ❖ Interleaved `withdraw()` calls on the same account
 - Assume initial balance == 150
 - This *should* cause a `WithdrawTooLarge` exception (but doesn't)



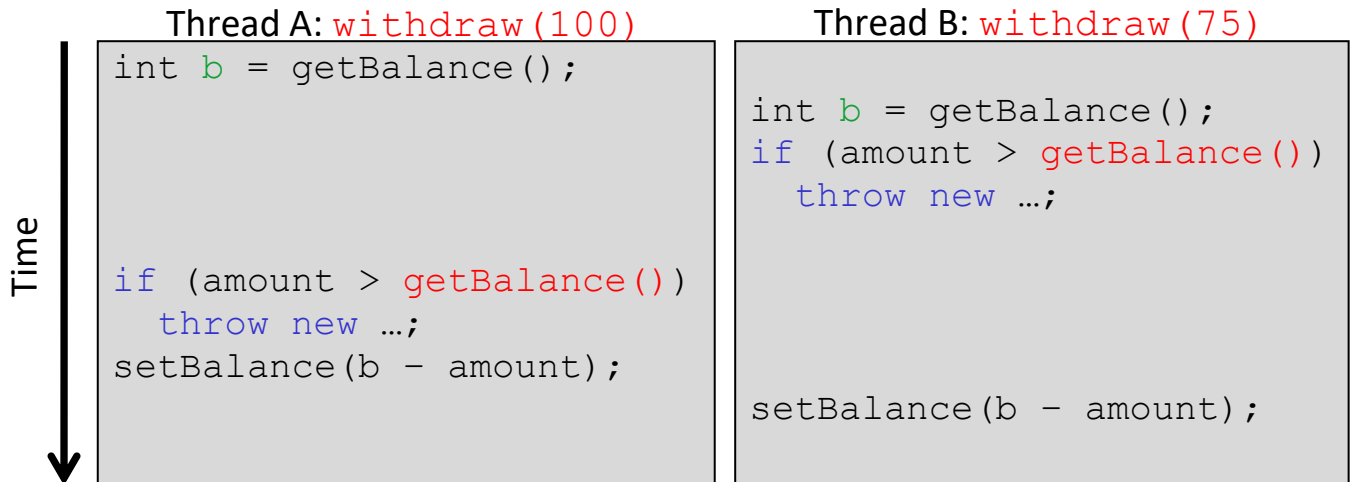
A Good Interleaving is Also Possible

- ❖ Interleaved `withdraw()` calls on the same account
 - Assume initial balance == 150
 - This *does* cause a `WithdrawTooLarge` exception



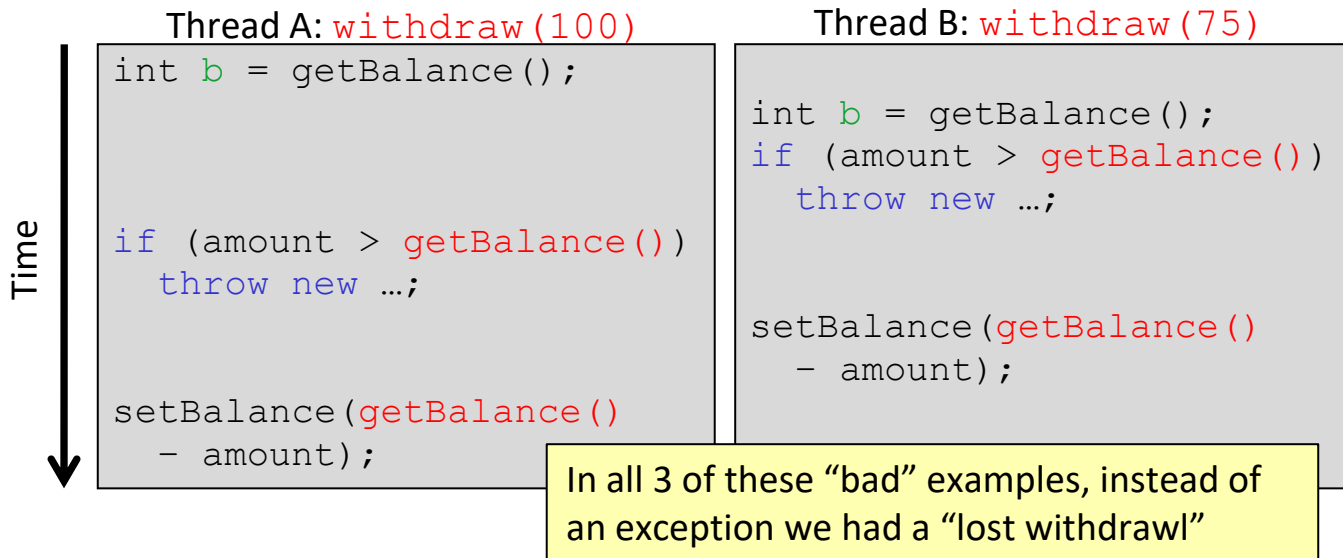
A Bad Fix: Another Bad Interleaving

- ❖ Interleaved `withdraw()` calls on the same account
 - Assume initial balance == 150
 - This *should* cause a `WithdrawTooLarge` exception (but doesn't)



ANOTHER Bad Fix: Another Bad Interleaving

- ❖ Interleaved `withdraw()` calls on the same account
 - Assume initial balance == 150
 - This *should* cause a `WithdrawTooLarge` exception (but doesn't)



Incorrect “Fixes”

- ❖ It is tempting *and almost always wrong* to try fixing a bad interleaving by rearranging or repeating operations, such as:

```
public void withdraw(int amount) {  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
  
    // Maybe the balance was changed  
    setBalance(getBalance() - amount);  
}
```

- ❖ This fixes nothing!
 - Potentially narrows the problem by one statement
 - And that’s not even guaranteed!
 - The compiler could optimize it into the old version, because you didn’t indicate a need to synchronize