Section 5: More Parallel Algorithms

Michelle Kuttel

mkuttel@cs.uct.ac.za

The prefix-sum problem

Given **`int[] input`**, produce **`int[] output`** where **`output[i]`** is the sum of **`input[0]+input[1]+`**… **`+input[i]`**

Sequential can be a CS1 exam problem:

```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Does not seem parallelizable

- Work: $O(n)$, Span: $O(n)$
- This *algorithm* is sequential, but a *different algorithm* has Work: $O(n)$, Span: $O(\log n)$

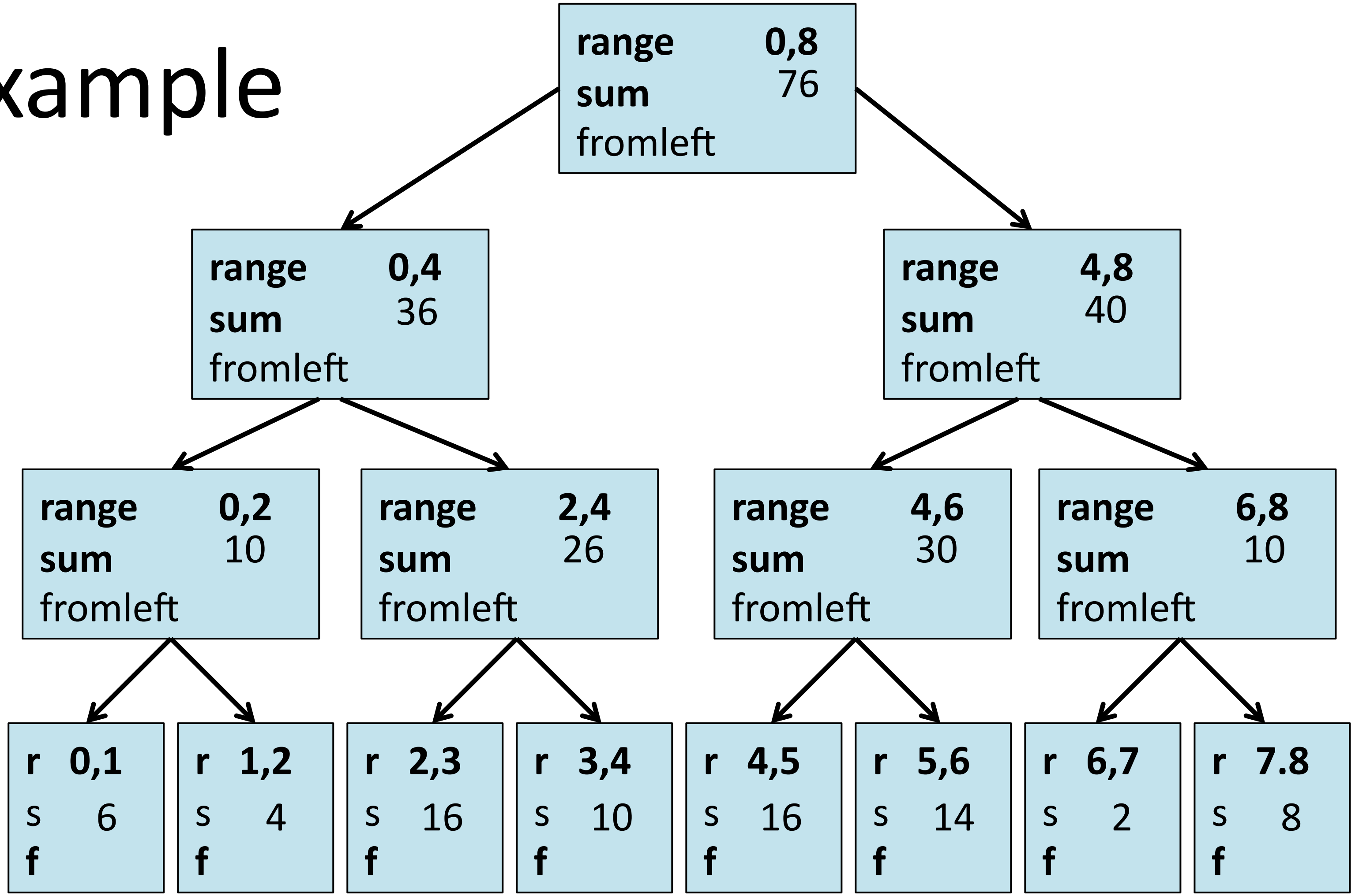
Parallel prefix-sum

- The parallel-prefix algorithm does two passes
  - Each pass has $O(n)$ work and $O(\log n)$ span
  - So in total there is $O(n)$ work and $O(\log n)$ span
  - So just like with array summing, the parallelism is $n/\log n$, an exponential speedup
- The first pass builds a tree bottom-up: the "up" pass
- The second pass traverses the tree top-down: the "down" pass

Historical note:

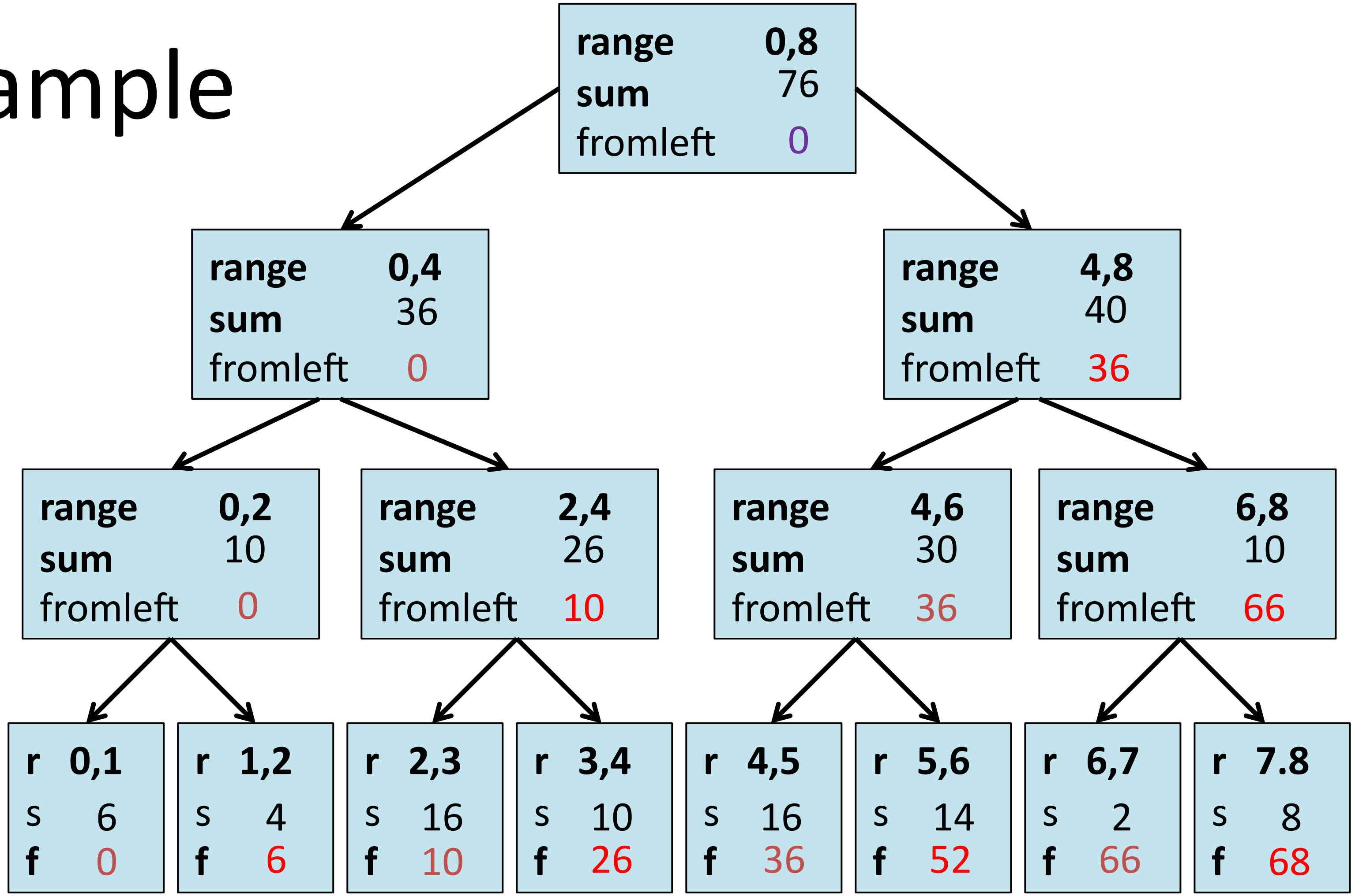
- Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977

Example



input	6	4	16	10	16	14	2	8
output								

Example



input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

The algorithm, part 1

1. Up: Build a binary tree where
   - Root has sum of the range [**`x,y`**)
   - If a node has sum of [**`lo,hi`**) and **`hi>lo`**,
     - Left child has sum of [**`lo,middle`**)
     - Right child has sum of [**`middle,hi`**)
     - A leaf has sum of [**`i,i+1`**), i.e., **`input[i]`**

This is an easy fork-join computation: combine results by actually building a binary tree with all the range-sums
- Tree built bottom-up in parallel
- Could be more clever with an array, as with heaps

Analysis: $O(n)$ work, $O($**`log`** $n)$ span

The algorithm, part 2

2. Down: Pass down a value **`fromLeft`**
   - Root given a **`fromLeft`** of **`0`**
   - 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`** (as stored in part 1)
   - At the leaf for array position **`i`**, **`output[i]=fromLeft+input[i]`**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result

- Leaves assign to **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\log n)$ span

Sequential cut-off

Adding a sequential cut-off is easy as always:

- Up:

  just a sum, have leaf node hold the sum of a range

- Down:

```
    output[lo] = fromLeft + input[lo];
    for(i=lo+1; i < hi; i++)
      output[i] = output[i-1] +
input[i]
```

Parallel prefix, generalized

Just as sum-array was the simplest example of a pattern that matches many, many problems, so is prefix-sum

- 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
  - This last one is perfect for an efficient parallel pack…
  - Perfect for building on top of the "parallel prefix trick"
- We did an *inclusive* sum, but *exclusive* is just as easy

Pack

[Non-standard terminology]

Given an array **`input`**, produce an array **`output`** containing only elements such that **`f(elt)`** is **`true`** in the same order…

Example: **`input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`**

```
f: is elt > 10
output [17, 11, 13, 19, 24]
```

Parallelizable?

- Finding elements for the output is easy
- But getting them in the right place seems hard

Parallel prefix to the rescue

1. Parallel map to compute a bit-vector for true 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 the 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];
}
```

Pack comments

- First two steps can be combined into one pass
  - Just using a different base case for the prefix sum
  - No effect on asymptotic complexity
- Can also combine third step into the down pass of the prefix sum
  - Again no effect on asymptotic complexity
- Analysis: $O(n)$ work, $O(\texttt{log}\ n)$ span
  - 2 or 3 passes, but 3 is a constant
- Parallelized packs will help us parallelize quicksort…

Quicksort review

- Very popular sequential sorting algorithm that performs well with an average sequential time complexity of O(*n*log*n*).
  - First list divided into two sublists.
    - All the numbers in one sublist arranged to be smaller than all the numbers in the other sublist.

- Achieved by first selecting one number, called a *pivot*, against which every other number is compared.
  - If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist.

Quicksort review

sequential, in-place, expected time $O(n \texttt{ log } n)$

		Best / expected case *work*
1.	Pick a pivot element	O(1)
2.	Partition all the data into:	O(n)
	A. The elements less than the pivot	
	B. The pivot	
	C. The elements greater than the pivot	
3.	Recursively sort A and C	2T(n/2)

How should we parallelize this?

Quicksort

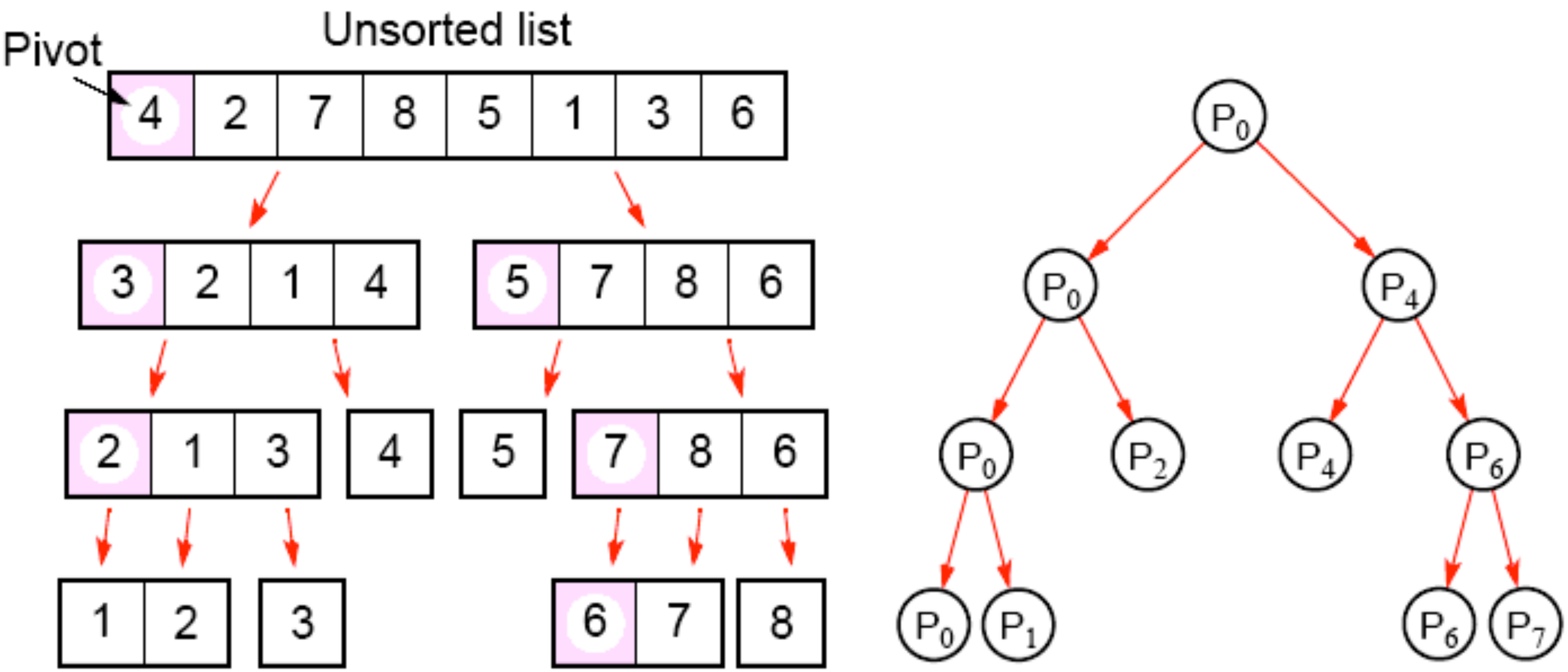
Best / expected case *work*

1. Pick a pivot element — $O(1)$
2. Partition all the data into: — $O(n)$
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C — $2T(n/2)$

Easy: Do the two recursive calls in parallel

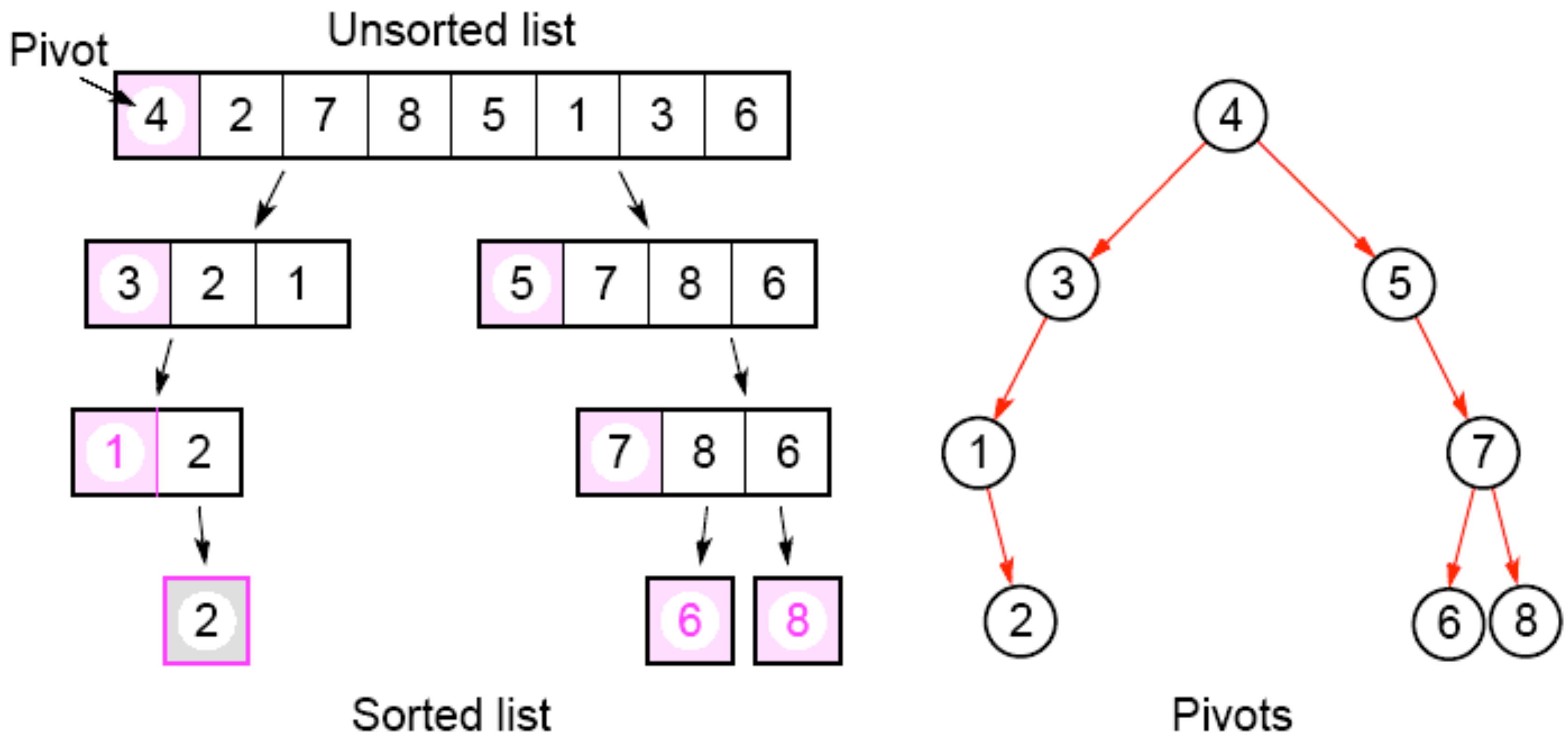
- Work: unchanged, of course, $O(n \log n)$
- Span: Now $T(n) = O(n) + 1T(n/2) = O(n)$
- So parallelism (i.e., work / span) is $O(\log n)$

Naïve Parallelization of Quicksort



Parallelizing Quicksort

With the pivot being withheld in processes:



Analysis

- Fundamental problem with all tree constructions – initial division done by a single thread, which will seriously limit speed.

- Tree in quicksort will not, in general, be perfectly balanced
  - Pivot selection very important to make quicksort operate fast.

Doing better

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

- Already have everything we need to parallelize the partition…

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

- This is just two packs!
  - We know a pack is $O(n)$ work, $O(\log n)$ span
  - Pack elements less than pivot into left side of **`aux`** array
  - Pack elements greater than pivot into right size 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
- With $O(\log n)$ span for partition, the total span for quicksort is $T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$
- Hence the available parallelism is proportional to

$$n \log n/\log^2 n = n/\log n$$

an exponential speed-up.

Example

- Step 1: pick pivot as median of three

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

- Steps 2a and 2c (combinable): pack less than, then pack greater than into a second array
  - Fancy parallel prefix to pull this off not shown

1	4	0	3	**5**	2				

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

- Step 3: Two recursive sorts in parallel
  - Can sort back into original array (like in mergesort)