CSE332: Data Abstractions

Lecture 20: Parallel Prefix and Parallel Sorting

Dan Grossman
Spring 2010

What next?

Done:

- Simple ways to use parallelism for counting, summing, finding
- Even though in practice getting speed-up may not be simple
- Analysis of running time and implications of Amdahl's Law

Now:

- Clever ways to parallelize more than is intuitively possible
- Parallel prefix:
  - This "key trick" typically underlies surprising parallelization
  - Enables other things like filters
- Parallel sorting: quicksort (not in place) and mergesort
  - Easy to get a little parallelism
  - With cleverness can get a lot

The prefix-sum problem

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

Sequential is easy enough for a CSE142 exam:

```
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;
}
```

This does not appear to be parallelizable

- Work: $O(n)$, Span: $O(n)$
- This *algorithm* is sequential, but we can design a *different algorithm* with parallelism (surprising)

Parallel prefix-sum

The parallel-prefix algorithm has $O(n)$ work but a span of 2**log** n

- So span is $O($**log** $n)$ and parallelism is n/**log** n, an exponential speedup just like array summing

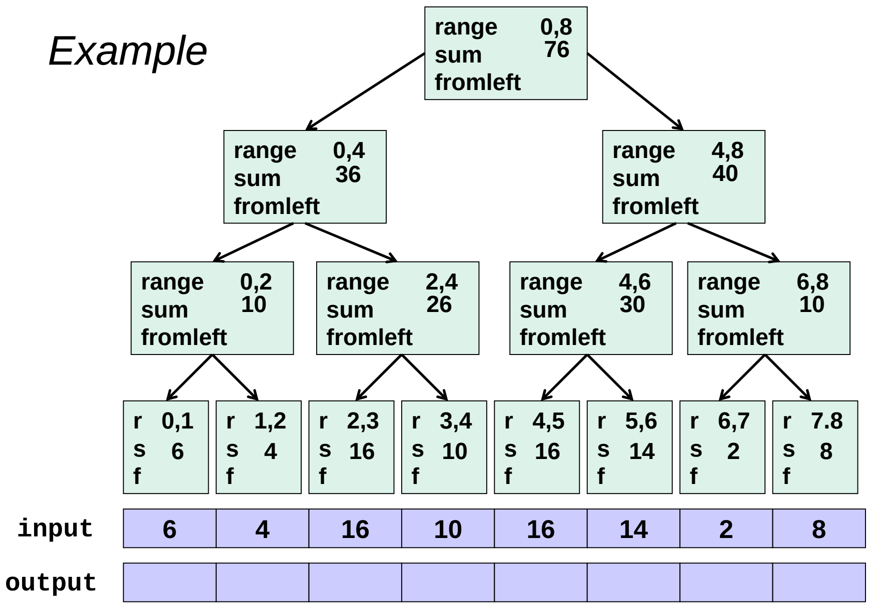
- The 2 is because there will be two "passes" one "up" one "down"

- Historical note / local bragging:
  - Original algorithm due to R. Ladner and M. Fischer in 1977
  - Richard Ladner joined the UW faculty in 1971 and hasn't left

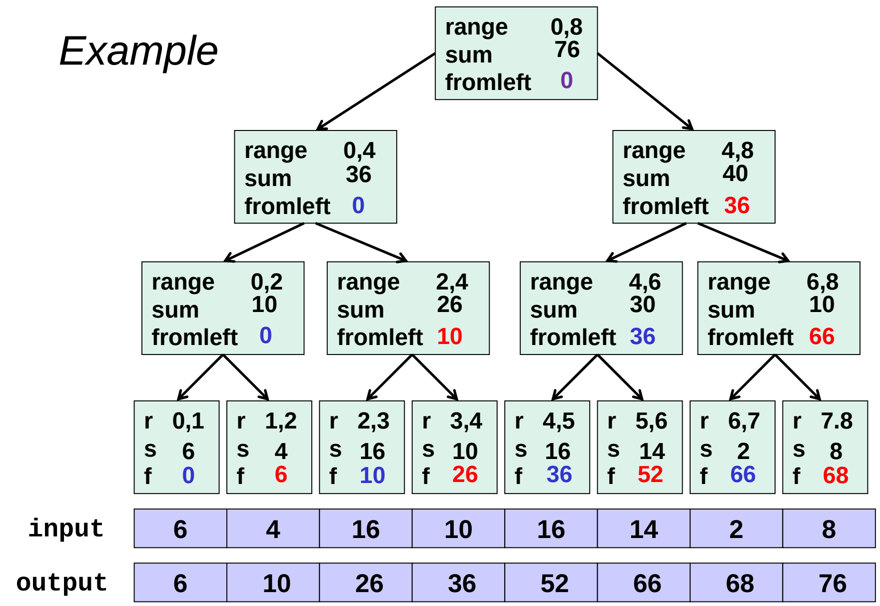

1968? 1973?


recent

Example



Example



The algorithm, part 1

1. Up: Build a binary tree where
   - Root has sum of **`input[0]..input[n-1]`**
   - If a node has sum of **`input[lo]..input[hi]`** and **`hi>lo`**,
     - Left child has sum of **`input[lo]..input[middle]`**
     - Right child has sum of **`input[middle]..input[hi]`**
   - A leaf has sum of **`input[i]..input[i]`**, i.e., **`input[i]`**

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

Analysis: $O(n)$ work, $O(\textbf{log } n)$ 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 (leafs assign to **`output`**)
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\textbf{log } n)$ 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 all elements to the left of **`i`** satisfying some property
  - This last one is perfect for an efficient parallel filter…
  - Perfect for building on top of the “parallel prefix trick”
- We did an *inclusive* sum, but *exclusive* is just as easy

Filter

[Non-standard terminology]

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

Example:
```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
f: is elt > 10
output [17, 11, 13, 19, 24]
```

Looks hard to parallelize
- Finding elements for the output is easy
- But getting them in the right place is hard

Parallel prefix to the rescue

1. Use a 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. Do parallel-prefix sum on the bit-vector
```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```
3. Use a parallel map to produce the output
```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
if(bitsum[0]==1) output[0] = input[0];
FORALL(i=1; i < input.length; i++)
  if(bitsum[i] > bitsum[i-1])
    output[bitsum[i]-1] = input[i];
```

Filter comments

- First two steps can be combined into one pass
  - Just using a different base case for the prefix sum
  - Has no effect on asymptotic complexity
- Parallelized filters will help us parallelize quicksort
- Analysis: $O(n)$ work, $O(\log n)$ span
  - 2 or 3 passes, but 3 is a constant

Quicksort review

Recall quicksort was sequential, in-place, expected time $O(n \log n)$

	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 $O(n) + 1T(n/2) = O(n)$
- So parallelism (i.e., work/span) is $O(\log n)$

Doing better

- An $O(\log n)$ 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
- 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 filters!
  - We know a filter is $O(n)$ work, $O(\log n)$ span
  - Filter elements less than pivot into left side of **`aux`** array
  - Filter elements great than pivot into right size of **`aux`** array
  - Put pivot in-between them and recursively sort
  - With a little more cleverness, can do both filters at once but no effect on asymptotic complexity
- With $O(\log n)$ span for partition, the total span for quicksort is
  $O(\log n) + 1T(n/2) = O(\log^2 n)$

Example

- Step 1: pick pivot as median of three

8	1	4	9	0	3	5	2	7	6

- Steps 2a and 2a (combinable): filter less than, then filter 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)

Now mergesort

Recall mergesort: sequential, not-in-place, worst-case $O(n \log n)$

		Best / expected case *work*
1.	**Sort left half and right half**	**2T(n/2)**
2.	**Merge results**	**O(n)**

Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the span to $O(n) + 1T(n/2) = O(n)$

- Again, parallelism is $O(\log n)$
- To do better we need to parallelize the merge
  - The trick won't use parallel prefix this time

Parallelizing the merge

Need to merge two *sorted* subarrays (may not have the same size)

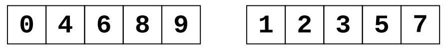
0	1	4	8	9

2	3	5	6	7

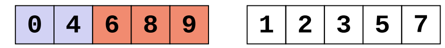
Idea: Suppose the larger subarray has *n* elements. In parallel,

- merge the first $n/2$ elements of the larger half with the "appropriate" elements of the smaller half
- merge the second $n/2$ elements of the larger half with the rest of the smaller half

Parallelizing the merge

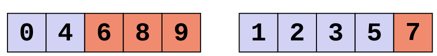


Parallelizing the merge



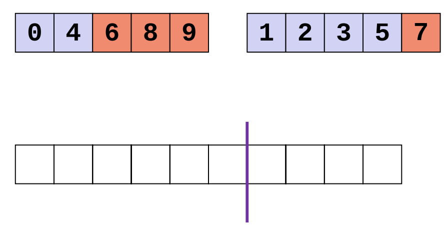
1. Get median of bigger half: *O*(1) to compute middle index

Parallelizing the merge



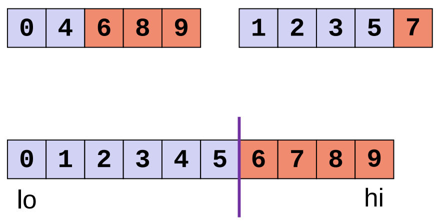
1. Get median of bigger half: *O*(1) to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: *O*(**`log`** *n*) to do binary search on the sorted small half

Parallelizing the merge



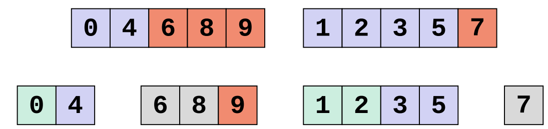
1. Get median of bigger half: *O*(1) to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: *O*(**`log`** *n*) to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: *O*(1)

Parallelizing the merge



1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: $O(\log n)$ to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: $O(1)$
4. Do two submerges in parallel

The Recursion



When we do each merge in parallel, we split the bigger one in half and use binary search to split the smaller one

Analysis

- Sequential recurrence for mergesort:
  $T(n) = 2T(n/2) + O(n)$ which is $O(n \log n)$
- Doing the two recursive calls in parallel but a sequential merge:
  work: same as sequential span: $T(n)=1T(n/2)+O(n)$ which is $O(n)$
- Parallel merge makes work and span harder to compute
  - Each merge step does an extra $O(\log n)$ binary search to find how to split the smaller subarray
  - To merge n elements total , do two smaller merges of possibly different sizes
  - But the worst-case split is $(1/4)n$ and $(3/4)n$
    - When subarrays same size and "smaller" splits "all" / "none"

Analysis continued

For just a parallel merge of n elements:

- Span is $T(n) = T(3n/4) + O(\log n)$, which is $O(\log^2 n)$
- Work is $T(n) = T(3n/4) + T(n/4) + O(\log n)$ which is $O(n)$
- (neither of the bounds are immediately obvious, but "trust me")

So for mergesort with parallel merge overall:

- Span is $T(n) = 1T(n/2) + O(\log^2 n)$, which is $O(\log^3 n)$
- Work is $T(n) = 2T(n/2) + O(n)$, which is $O(n \log n)$

So parallelism (work / span) is $O(n / \log^2 n)$

- Not quite as good as quicksort, but worst-case guarantee
- And as always this is just the asymptotic result