CSE332: Data Abstractions

# Lecture 19: Parallel Prefix and Sorting

Ruth Anderson
Winter 2011

## Announcements

- **Homework 6** – due Friday Feb 25th at the BEGINNING of lecture
- **Project 3** – the last programming project!
  - Version 1 & 2 - Tues March 1, 2011 11PM - (10% of overall grade)
  - ALL Code - Tues March 8, 2011 11PM - (65% of overall grade):
  - Writeup - Thursday March 10, 2011, 11PM - (25% of overall grade)



## Outline

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 packs (aka 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]`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **in** | 6 | 4 | 16 | 10 | 16 | 14 | 2 | 8 |
| **out** | 6 | 10 | 26 | 36 | 52 | 66 | 68 | 76 |

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; each cell depends on previous cell
- **Work**: $O(n)$, **Span**: $O(n)$
- This *algorithm* is sequential, but we can design a *different algorithm* with parallelism for the same problem



## Parallel prefix-sum

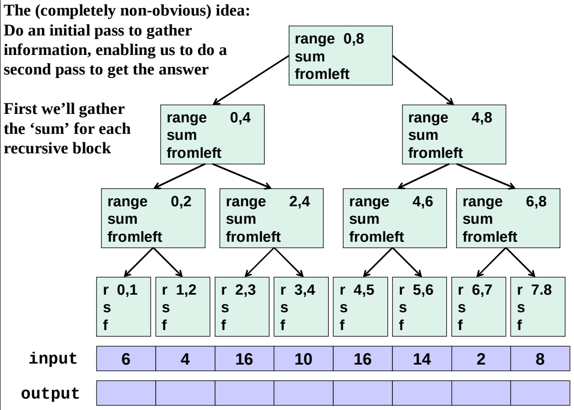
The parallel-prefix algorithm has $O(n)$ **work** but a **span** of $2\log n$
- So **span** is $O(\log n)$ and parallelism is $n/\log n$, an exponential speedup just like array summing

- The 2 is because there will be two "passes" on the tree
  - One "up" one "down"
- Historical note :
  - Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977



**The (completely non-obvious) idea: Do an initial pass to gather information, enabling us to do a second pass to get the answer**

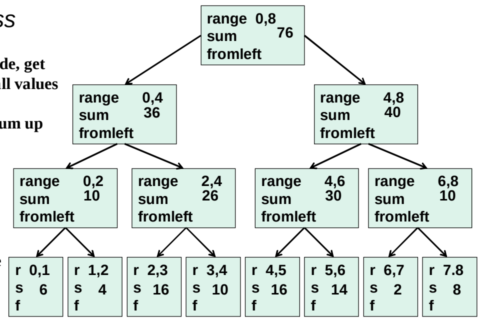
**First we'll gather the 'sum' for each recursive block**



## *First pass*

For each node, get the sum of all values in its range; propagate sum up from leaves

Will work like parallel sum, but recording intermediate information

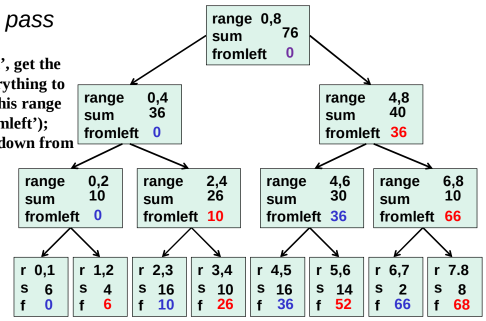



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| input | 6 | 4 | 16 | 10 | 16 | 14 | 2 | 8 |
| output | | | | | | | | |



## *Second pass*

Using 'sum', get the sum of everything to the left of this range (call it 'fromleft'); propagate down from root




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



## *The algorithm, part 1*

1. Propagate 'sum' up: Build a binary tree where
   – Root has sum of **`input[0]..input[n-1]`**
   – Each node has sum of **`input[lo]..input[hi-1]`**
     • Build up from leaves; parent.sum=left.sum+right.sum
   – A leaf's sum is just it's value; **`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; ex. Use an array as tree representation like we did for heaps

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



## *The algorithm, part 2*

2. Propagate 'fromleft' down:
   – 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 of first step: $O(n)$ **work**, $O($**`log`** $n)$ **span**
Analysis of second step: $O(n)$ **work**, $O($**`log`** $n)$ **span**
**Total for algorithm:** $O(n)$ **work**, $O($**`log`** $n)$ **span**



## *Sequential cut-off*

Adding a sequential cut-off isn't too bad:

• Step One: Propagating Up:
  Sequentially compute sum for range
  The tree itself will be shallower

• Step Two: Propagating 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,`** for any **`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

• We did an *inclusive* sum, but *exclusive* is just as easy

## Pack (aka 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!
- Determining *whether* an element belongs in the output is easy
- But *getting them in the right place* in the output is hard; seems to depend on previous results



## Parallel prefix Pack/Filter

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];
```



## Pack/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
- Analysis: $O(n)$ **work**, $O(\log n)$ **span**
  - 2 or 3 passes, but 3 is a constant
- Parallelized packs will help us parallelize quicksort!!



## Sequential 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)** |

Recurrence (assuming a good pivot):
- $T(0)=T(1)=1$
- $T(n)=n + 2T(n/2) = O(n\log n)$

Run-time: O(nlogn)

How should we parallelize this?



## Review: Really common recurrences

Should know how to solve recurrences but also recognize some really common ones:

| | |
|---|---|
| $T(n) = O(1) + T(n-1)$ | linear |
| $T(n) = O(1) + 2T(n/2)$ | linear |
| $T(n) = O(1) + T(n/2)$ | logarithmic |
| $T(n) = O(1) + 2T(n-1)$ | exponential |
| $T(n) = O(n) + T(n-1)$ | quadratic |
| $T(n) = O(n) + T(n/2)$ | linear |
| $T(n) = O(n) + 2T(n/2)$ | $O(n \log n)$ |

Note big-Oh can also use more than one variable
- Example: can sum all elements of an $n$-by-$m$ matrix in $O(nm)$



## Parallel Quicksort (version 1)

| | 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)** |

First: Do the two recursive calls in parallel
- **Work**: unchanged of course, O(n log n)
- Now recurrence takes the form:

  $T(n) = O(n) + \mathbf{1}T(n/2) = O(n)$

  **Span**: 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 an in place sort)
  – 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)*

**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 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 packs at once but no effect on asymptotic complexity
- With $O(\log n)$ **span** for partition, the total **span** for quicksort is $O(\log n) + 1T(n/2) = O(\log^2 n)$



## *Parallel Quicksort Example (version 2)*

- 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

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 $O(n \log n)$

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, **work** is O(nlogn), and
- parallelism is work/span = $O(\log n)$
- 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)
**Idea**: Recursively divide subarrays in half, merge halves in parallel

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 6 | 8 | 9 | | 1 | 2 | 3 | 5 | 7 |

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

- Pick the **median** element of the larger array (here 6) in constant time
- In the other array, use binary search to find the first element greater than or equal to that median (here 7)
- Merge (in parallel) half the larger array (from the median onward) with the upper part of the shorter array
- Merge (in parallel) the lower part of the larger array with the lower part of the shorter array



## *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

0 4 6 8 9    1 2 3 5 7



## Parallelizing the merge

0 4 6 8 9    1 2 3 5 7

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



## Parallelizing the merge

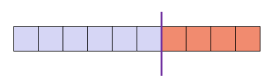
0 4 6 8 9    1 2 3 5 7

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

0 4 6 8 9    1 2 3 5 7



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

0 4 6 8 9    1 2 3 5 7

0 1 2 3 4 5 | 6 7 8 9

lo    hi

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 *sub-merges* in parallel (how?)



## Doing sub-merges in parallel

0 4 6 8 9    1 2 3 5 7

0 4    6 8 9    1 2 3 5    7

When we do each merge in parallel, for each sub piece (e.g blue pieces)

1) we split the bigger of the two pieces (e.g. 1235) in half
2) use binary search to split the smaller piece (e.g. 04)

## *Mergesort Analysis*

- Sequential recurrence for mergesort:

  $T(n) = 2T(n/2) + O(n)$ which is $O(n\,\mathbf{log}\,n)$

- 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 $O(\mathbf{log}\ n)$ 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"



## *Mergesort Analysis (continued)*

For just a parallel merge of *n* elements:

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

So for mergesort with parallel merge overall:

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

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

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