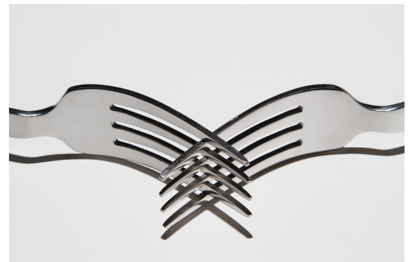
Adam Blank | Lecture 16 | Winter 2016

CSE 332

Data Abstractions

More Parallel Primitives and Parallel Sorting



Outline

Maps and Reductions

Reductions

INPUT: An array
OUTPUT: A combination of the array by an associative operation
The general name for this type of problem is a **reduction**. Examples include: `max`, `min`, `has-a`, `first`, `count`, `sorted`

Maps

INPUT: An array
OUTPUT: Apply a function to every element of that array
The general name for this type of problem is a **map**. You can do this with any function, because the array elements are independent.

Today, we'll add in two more:

- Scan
- Pack (or filter)

As we'll see, both of these are quite a bit less intuitive **in parallel** than map and reduce.

Scan and Parallel Prefix-Sum

Scan

Suppose we have an associative operation $\oplus$ and an array a:

a:	a_0	a_1	a_2	a_3
	a[0]	a[1]	a[2]	a[3]

Then, `scan(a)` returns an array of "partial sums" (using $\oplus$):

scan(a):	a_0	$a_0 \oplus a_1$	$a_0 \oplus a_1 \oplus a_2$	$a_0 \oplus a_1 \oplus a_2 \oplus a_3$
	b[0]	b[1]	b[2]	b[3]

It's hard to see at first, but this is actually a really powerful tool. It gives us a "partial trace" of the operation as we apply it to the array (for free).

No Seriously

splitting, load balancing, quicksort, line drawing, radix sort, designing binary adders, polynomial interpolation, decoding gray codes

Sequential Scan (with $\oplus = +$)

For the sake of being clear, we'll discuss scan with $\oplus = +$.
That is, "prefix sums" of an array":

Example (Prefix Sum)

a:	5	1	3	4	2
	a[0]	a[1]	a[2]	a[3]	a[4]

scan(a):	5	6	9	13	15
	b[0]	b[1]	b[2]	b[3]	b[4]

Sequential Code

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

If you have a really good memory, you'll remember that on the **very first day of lecture**, we discussed a very similar problem.

Sequential Prefix-Sum

Sequential Code

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

Bad News

This **algorithm** does not parallelize well. Step i needs the outputs from all the previous steps. This might as well be an algorithm on a linked list.

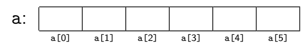
So, what do we do?

Come Up With A Better Algorithm!

The solution here will be to add a "pre-processing step". This is essentially what we did in the first lecture.

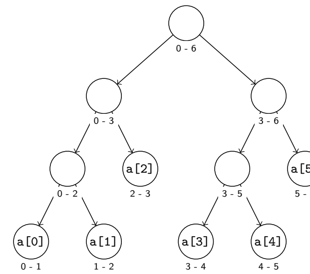
Better Prefix-Sum

We begin with an array as usual:



Then, transform it into a **balanced tree**, because $\lg n$ height will allow us to get a span of $\lg n$, eventually:

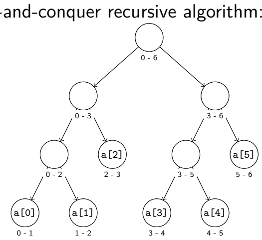
```
PSTNode {
   int lo, hi;
   int sum;
   PSTNode left, right;
}
```



Better Prefix-Sum: Processing the Input

Creating the tree is a standard divide-and-conquer recursive algorithm:

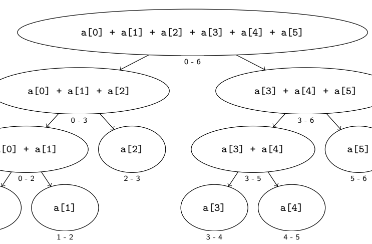
```
PSTNode {
   int lo, hi;
   int sum;
   PSTNode left, right;
}
```




```
PSTNode processInput(int[] input, int lo, int hi) {
   if (hi − lo == 1) {
      return new PSTNode(lo, hi, input[lo]);
   }
   else {
      mid = lo + (hi − lo)/2;
      PSTNode left = processInput(lo, mid);
      PSTNode right = processInput(mid, hi);
      return new PSTNode(lo, hi, left.sum + right.sum, left, right);
   }
}
```

Better Prefix-Sum: Constructing the Output

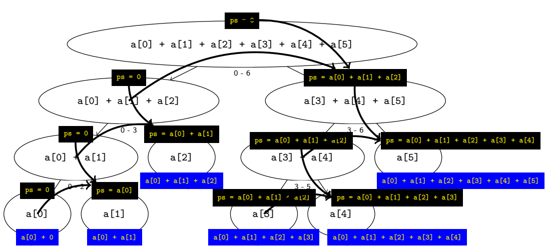
Now, we have the entire tree filled out:



To fill in all the prefix sums, we recursively fill them in down the tree. Since the non-leaf nodes don't have access to the elements of the array, we fill in a **pre-scan** (everything up to, but not including the range).

Better Prefix-Sum: Constructing the Output

To fill in all the **pre-scans**, we recursively fill them in down the tree:




```
void makeOutput(int[] output, PSTNode current, int prescan) {
   if (current is a leaf) {
      output[current.lo] = prescan + current.sum;
   }
   else {
      makeOutput(output, current.left, prescan);
      makeOutput(output, current.right, prescan + current.left.sum);
   }
}
```

Sequential Cut-off

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

Processing the Input

This is just a normal sequential cut-off. The leaves end up being cutoff size ranges instead of ranges of one.

Constructing the Output

We must sequentially compute the prefix sum at our leaves as well:

```
output[lo] = prescan + input[lo];
for (i = lo + 1; i < hi; i++) {
   output[i] = output[i−1] + input[i]
}
```

Notice that this means we must pass the `input` array to this phase now.

Another Primitive: Parallel Pack (or "filter")

Here the idea is that we'd like to filter the array given some predicate (e.g., ≤ 7). More specifically:

Pack/Filter

Suppose we have a function $f : \texttt{E} \to \texttt{boolean}$ and an array a of type E:

a:

a_0	a_1	a_2	a_3
a[0]	a[1]	a[2]	a[3]

Then, `pack(a)` returns an array of elements x for which $f(x) = \texttt{true}$. For example, if `arr = [1, 3, 8, 6, 7, 2, 4, 9]` and `f(x) = x % 2 == 0`, then `pack(arr) = [8, 6, 2, 4]`.

The key to doing this in parallel is scan!

Another Primitive: Parallel Pack (or "filter")

Let `f(x) = x % 2 == 0`.

Parallel Pack

input:

1	3	8	6	7	2	4	9
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

1. Use a **map** to compute a bitset for $f(x)$ applied to each element

bitset:

0	0	1	1	0	1	1	0
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]

2. Do a **scan on the bit vector** with $\oplus = +$:

bitsum:

0	0	1	2	2	3	4	4
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]

3. Do a **map on the bit sum** to produce the output:

output:

8	6	2	4
d[0]	d[1]	d[2]	d[3]

```
output = new E[bitsum[n-1]];
for (i=0; i < input.length; i++) {
   if (bitset[i] == 1) {
      output[bitsum[i] - 1] = input[i];
   }
}
```

More on Pack

- We can combine the first two passes into one (just use a different base case for prefix sum)
- We can also combine the third step into the second part of prefix sum
- Overall: $\mathcal{O}(n)$ work and $\mathcal{O}(\lg n)$ span. (Why?)

We can use scan and pack in all kinds of situations!

Parallel Quicksort


```
int[] quicksort(int[] arr) {
   int pivot = choosePivot();
   int[] left = filterLessThan(arr, pivot);
   int[] right = filterGreaterThan(arr, pivot);
   return quicksort(left) + quicksort(right);
}
```

Do The Recursive Calls in Parallel

Assuming a good pivot, we have:

$$\text{work}(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ 2\text{work}(n/2) + \mathcal{O}(n) & \text{otherwise} \end{cases}$$

and

$$\text{span}(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ \max(\text{span}(n/2), \text{span}(n/2)) + \mathcal{O}(n) & \text{otherwise} \end{cases}$$

These solve to $\mathcal{O}(n \lg n)$ and $\mathcal{O}(n)$. So, the parallelism is $\mathcal{O}(\lg n)$.

Parallel Quicksort


```
int[] quicksort(int[] arr) {
   int pivot = choosePivot();
   int[] left = filterLessThan(arr, pivot);
   int[] right = filterGreaterThan(arr, pivot);
   return quicksort(left) + quicksort(right);
}
```

Do The Partition in Parallel

The partition step is just two filters or packs. Each pack is $\mathcal{O}(n)$ work, but $\mathcal{O}(\lg n)$ span! So, our new span recurrence is:

$$\text{span}(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ \max(\text{span}(n/2), \text{span}(n/2)) + \mathcal{O}(\lg n) & \text{otherwise} \end{cases}$$

Master Theorem says this is $\mathcal{O}(\lg^2 n)$ which is neat!

Parallel Mergesort


```
int[] mergesort(int[] arr) {
   int[] left = getLeftHalf();
   int[] right = getRightHalf();
   return merge(mergesort(left), mergesort(right));
}
```

Do The Recursive Calls in Parallel

This will get us the same work and span we got for quicksort when we did this:

- $\text{work}(n) = \mathcal{O}(n \lg n)$
- $\text{span}(n) = \mathcal{O}(n)$
- Parallelism is $\mathcal{O}(\lg n)$

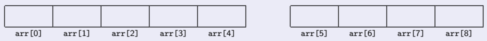
Now, let's try to parallelize the `merge` part.

As always, when we want to parallelize something, we can turn it into a divide-and-conquer algorithm.

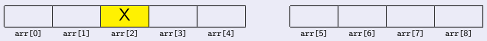
Parallelizing Merge

Do The Merge in Parallel

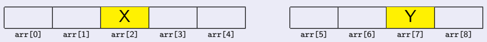
Merge takes as input two arrays:



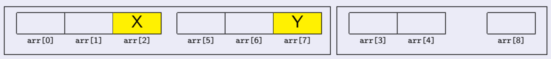
1. Find the median of the **larger** array (just the middle index):



2. Partition the **smaller** array using X as a pivot. To do this, **binary search** the smaller array:



3. Now, we have four pieces $\boxed{\leq X}$, $\boxed{> X}$, $\boxed{\leq Y}$, and $\boxed{> Y}$. In the sorted array, the $\leq$ pieces will be entirely before the $>$ pieces.



4. Recursively apply the merge algorithm (until some cut-off)!

Parallel Mergesort Analysis

First, we analyze **just the parallel merge**:

Parallel Merge Analysis

The non-recursive work is $\mathcal{O}(1) + \mathcal{O}(\lg n)$ to find the splits.

The **worst case** is when we split the bigger array in half and the smaller array is all on the left (or all on the right). In other words:

$$\text{work}(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ \text{work}(3n/4) + \text{work}(n/4) + \mathcal{O}(\lg n) & \text{otherwise} \end{cases}$$

and

$$\text{span}(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ \max(\text{span}(3n/4) + \text{span}(n/4)) + \mathcal{O}(\lg n) & \text{otherwise} \end{cases}$$

These solve to $\text{work}(n) = \mathcal{O}(n)$ and $\text{span}(n) = \mathcal{O}(\lg^2 n)$.

Parallel Mergesort Analysis

Now, we calculate the work and span of **the entire parallel mergesort**.

Putting It Together

$$\text{work}(n) = \mathcal{O}(n \lg n)$$

$$\text{span}(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ \text{span}(n/2) + \mathcal{O}(\lg^2 n) & \text{otherwise} \end{cases}$$

This works out to $\text{span}(n) = \mathcal{O}(\lg^3 n)$.

This isn't quite as much parallelism as quicksort, but **this one is a worst case guarantee**!