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

Suppose we have an associative operation ⊕ and an array a:

\[
a = [a_0, a_1, a_2, a_3]
\]

Then, \( \text{scan}(a) \) returns an array of "partial sums" (using ⊕):

\[
\text{scan}(a) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, a_0 \oplus a_1 \oplus a_2 \oplus a_3]
\]

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 ⊕ = +)

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

Example (Prefix Sum)

\[
a = [5, 1, 3, 4, 2]
\]

\[
\text{scan}(a) = [5, 6, 9, 13, 15]
\]

Sequential Code

1 int[] prefixSum(int[] input) {
2     int[] output = new int[input.length];
3     int sun = 0;
4     for (int i = 0; i < input.length; i++) {
5         sun = sun + input[i];
6     }
7     return output;
8 }

If you have a really good memory, you'll remember that on the very first day of lecture, we discussed a very similar problem.
Better Prefix-Sum 4

Sequential Code

```java
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 3 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.

Sequential Cut-off 5

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:

```java
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);
    }
}
```

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 $f: E \rightarrow \text{boolean}$ and an array $a$ of type $E$:

$$a = [a_0, a_1, a_2, a_3]$$

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

The key to doing this in parallel is scan!

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: $O(n)$ work and $O(lg n)$ span. (Why?)

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

Parallel Mergesort

This will get us the same work and span we got for quicksort when we did this:
- work($n$) = $O(n lg n)$
- span($n$) = $O(n)$
- Parallelism is $O(lg n)$

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:

\[
\begin{array}{cccccccc}
\text{arr[0]} & \text{arr[1]} & \text{arr[2]} & \text{arr[3]} & \text{arr[4]} & \text{arr[5]} & \text{arr[6]} & \text{arr[7]} & \text{arr[8]}
\end{array}
\]

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

\[
\begin{array}{cccccccc}
\text{X} & & & & & & & & \text{Y}
\end{array}
\]

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

\[
\begin{array}{cccccccc}
\text{X} & & & & & & & & \text{Y}
\end{array}
\]

3. Now, we have four pieces \(\leq X\), \(> X\), \(\leq Y\), and \(> Y\). In the sorted array, the \(\leq\) pieces will be entirely before the \(>\) pieces.

\[
\begin{array}{cccccccc}
\text{X} & & & & & & & & \text{Y}
\end{array}
\]

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 \(O(1) + O(\lg n)\) 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:

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

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

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

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

**Putting It Together**

\[
\text{work}(n) = O(n \lg n)
\]

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

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

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