More Parallel Primitives
and Parallel Sorting

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

$$
a: \begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
\end{array}
$$

Then, $\text{scan}(a)$ returns an array of “partial sums” (using $\oplus$):

$$
\text{scan}(a): \begin{array}{cccc}
  b_0 & b_1 & b_2 & b_3 \\
\end{array}
$$

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:

$$
a: \begin{array}{cccc}
  5 & 1 & 3 & 4 & 2 \\
\end{array}
$$

$$
\text{scan}(a): \begin{array}{cccc}
  5 & 6 & 9 & 13 & 15 \\
\end{array}
$$

**Sequential Code**

```java
1 int[] prefixSum(int[] input) {
2     int[] output = new int[input.length];
3     int sum = 0;
4     for (int i = 0; i < input.length; i++) {
5         sum += input[i];
6         output[i] = sum;
7     }
8     return output;
9 }
```

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**

We begin with an array as usual:

$$
a: \begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\
\end{array}
$$

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

**Sequential Prefix-Sum**

**Sequential Code**

```java
1 int[] prefixSum(int[] input) {
2     int[] output = new int[input.length];
3     int sum = 0;
4     for (int i = 0; i < input.length; i++) {
5         sum += input[i];
6         output[i] = sum;
7     }
8     return output;
9 }
```

**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: Processing the Input 6

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

```java
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 7

Now, we have the entire tree filled out:

```
To fill in all the pre-scans, 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 8

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 9

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:

```
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);
    }
}
```
Another Primitive: Parallel Pack (or “filter”) 10

Here the idea is that we’d like to filter the array given some predicate (e.g., \( \leq 7 \)). 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 \( f(x) = x \% 2 == 0 \), then \( \text{pack}(\text{arr}) = [8, 6, 2, 4] \).

The key to doing this in parallel is scan!

---

More on Pack 12

- 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 Quicksort 13

1 int[] quicksort(int[] arr) {
2   int pivot = choosePivot();
3   int[] left = filterLessThan(arr, pivot);
4   int[] right = filterGreaterThan(arr, pivot);
5   return quicksort(left) + quicksort(right);
6 }

Do The Recursive Calls in Parallel

Assuming a good pivot, we have:

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

and

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

These solve to \( O(n \lg n) \) and \( O(n) \). So, the parallelism is \( O(\lg n) \).
Parallel Quicksort 14

```java
1 int[] quicksort(int[] arr) {
2   int pivot = choosePivot();
3   int[] left = filterLessThan(arr, pivot);
4   int[] right = filterGreaterThan(arr, pivot);
5   return quicksort(left) + quicksort(right);
6 }
```

Do The Partition in Parallel

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

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

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

Parallel Mergesort 15

```java
1 int[] mergesort(int[] arr) {
2   int[] left = getLeftHalf();
3   int[] right = getRightHalf();
4   return merge(mergesort(left), mergesort(right));
5 }
```

Do The Recursive Calls in Parallel

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 16

Do The Merge in Parallel

Merge takes as input two arrays:

```
```

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

```
```

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

```
arr[0] arr[1] arr[2] \phantom{arr[3]} \phantom{arr[4]} \phantom{arr[5]} \phantom{arr[6]} \phantom{arr[7]}
```

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.

```
arr[0] arr[1] \phantom{arr[2]} \phantom{arr[3]} \phantom{arr[4]} \phantom{arr[5]} \phantom{arr[6]} \phantom{arr[7]}
```

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

Parallel Mergesort Analysis 17

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:

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

and

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

These solve to $work(n) = O(n)$ and $span(n) = O(lg^2 n)$. 
Now, we calculate the work and span of the entire parallel mergesort.

Putting It Together

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

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!