CSE 332
Data Structures and Parallelism

Outline

1 More Parallel Primitives

2 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.

Maps and Parallel Prefix-Sum

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

\[ a: a_0 \ a_1 \ a_2 \ a_3 \]

Then, 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:

\[ \text{Example (Prefix Sum)} \]

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

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

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.
We begin with an array as usual:

```
| 0 | 1 | 2 | 3 | 4 | 5 |
```

Then, transform it into a balanced tree, because log(n) height will allow us to get a span of log(n), eventually:

```
    0
   / \
  1   2
 / \ / \     
3  4 5 6 7 8
```

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).

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

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

**Processing the Input**

**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);
    }
}
```

Notice that this means we must pass the input array to this phase now.
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 \( \text{pack(arr)} = [8, 6, 2, 4] \).

\[\text{The key to doing this in parallel is scan!}\]

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

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 Mergesort  15

\[
1 \text{ int}\[] \text{ mergesort(int}\[] \text{ arr)} \{
2 \hspace{1em} \text{ int}[] \text{ left = getLeftHalf();}
3 \hspace{1em} \text{ int}[] \text{ right = getRightHalf();}
4 \hspace{1em} \text{ return merge(mergesort(left), mergesort(right));} \}
\]

Do The Recursive Calls in Parallel
This will get us the same work and span we got for quicksort when we did this:

- 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.

Another Primitive: Parallel Pack (or “filter”)  11

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

Parallel Pack
\[
\text{input} = [1, 3, 8, 6, 7, 2, 4, 9]
\]

- Use a map to compute a bitset for \( f(x) \) applied to each element
- Do a scan on the bit vector with \( @ = +: \)
- Do a map on the bit sum to produce the output:

\[
\text{output} = [0, 0, 2, 4]
\]

Parallel Quicksort

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

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 \( O(\lg n) \).
Parallelizing Merge

Do The Merge in Parallel

Merge takes as input two arrays:

\[ \text{arr[0]} \text{ arr[1]} \text{ arr[2]} \text{ arr[3]} \text{ arr[4]} \]
\[ \text{arr[5]} \text{ arr[6]} \text{ arr[7]} \text{ arr[8]} \]

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

\[ \text{arr[0]} \text{ arr[1]} \text{ arr[2]} \text{ arr[3]} \text{ arr[4]} \]
\[ \text{arr[5]} \text{ arr[6]} \text{ arr[7]} \text{ arr[8]} \]

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

\[ \text{arr[0]} \text{ arr[1]} \text{ arr[2]} \text{ arr[3]} \text{ arr[4]} \]
\[ \text{arr[5]} \text{ arr[6]} \text{ arr[7]} \text{ arr[8]} \]

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.

\[ \text{arr[0]} \text{ arr[1]} \text{ arr[2]} \text{ arr[3]} \text{ arr[4]} \]
\[ \text{arr[5]} \text{ arr[6]} \text{ arr[7]} \text{ arr[8]} \]

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

Parallel Mergesort Analysis

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

Putting It Together

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

This works out to \( \text{span}(n) = O(\log^2 n) \).

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