Lecture 6 — Sequences II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Margaret Reid-Miller — 31 January 2013

Material in this lecture: Today's lecture is about *reduction*.

- Scan implementation (from the last lecture notes)
- Divide-and-conquer with reduction
- Cost of reduce when f is not constant work

1 Reduce Operation

Recall that `reduce` function has the interface

$$\texttt{reduce}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to \alpha$$

When the combining function f is associative—i.e., $f(f(x,y),z) = f(x,f(y,z))$ for all x, y and z of type α—`reduce` returns the sum with respect to f of the input sequence S. It is the same result returned by `iter` $f\ I\ S$. The reason we include `reduce` is that it is parallel, whereas `iter` is strictly sequential. Note, though, `iter` can use a more general combining function with type: $\beta \times \alpha \to \beta$.

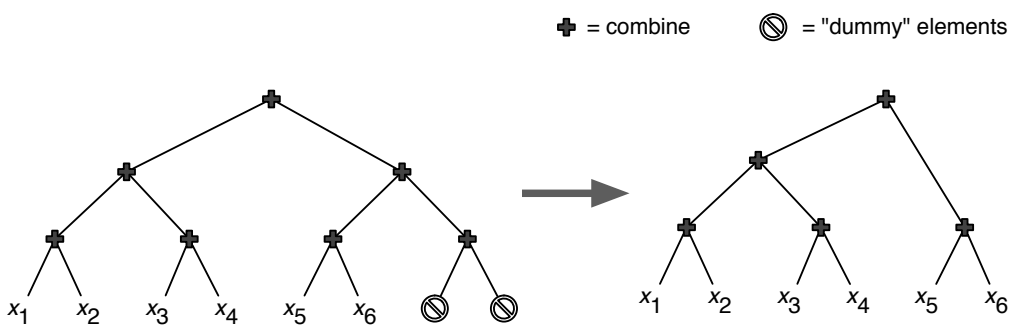
The results of `reduce` and `iter`, however, may differ if the combining function is non-associative. In this case, the order in which the reduction is performed determines the result; because the function is non-associative, different orderings will lead to different answers. While we might try to apply reduce to only associative operations, unfortunately even some functions that seem to be associative are actually not. For instance, floating point addition and multiplication are not associative. In SML/NJ, integer addition is also not associative because of the overflow exception.

To properly deal with combining functions that are non-associative, it is therefore important to specify the order that the combining function is applied to the elements of a sequence. This order is part of the specification of the ADT `Sequence`. In this way, every (correct) implementation returns the same result when applying `reduce`; the results are deterministic regardless of what data structure and algorithm are used.

For this reason, we define a specific combining tree, which is defined quite carefully in the library documentation for `reduce`. This tree is the same as if we rounded up the length of the input sequence to the next power of 2, i.e., $|x| = 2^k$, and then put a perfectly balanced binary tree[1] over the sequence with 2^k leaves. Wherever we are missing children in the tree, we don't apply the combining function. An example is shown in the following figure.

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] A *perfect* binary tree is a tree in which every node other than the leaves have exactly 2 children.



Later on today, we will offer an explanation why we chose this particular combining order.

1.1 Divide and Conquer with Reduce

Now, let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

```
1  function myDandC(S) =
2    case showt(S) of
3      EMPTY ⇒ [emptyVal]
4    | ELT(v) ⇒ [base](v)
5    | NODE(L, R) ⇒ let
6          (L′,R′) = (myDandC(L) || myDandC(R))
7      in
8          [someMessyCombine](L′,R′)
9      end
```

Algorithms that fit this pattern can be implemented in one line using the sequence `reduce` function. Turning a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking `reduce` with the following parameters:

```
reduce [someMessyCombine] [emptyVal] (map [base] S)
```

We will take a look two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm. Both problems should be familiar to you.

Algorithm 4: MCSS Using Reduce.

The first example is the Maximum Contiguous Subsequence Sum problem from last lecture. Given a sequence S of numbers, find the contiguous subsequence that has the largest sum—more formally:

$$\text{mcss}(s) = \max\left\{\sum_{k=i}^{j} s_k \ : \ 1 \le i \le n, i \le j \le n\right\}.$$

Recall that the divide-and-conquer solution involved returning four values from each recursive call on a sequence S: the desired result $\texttt{mcss}(S)$, the maximum prefix sum of S, the maximum suffix sum of S, and the total sum of S. We will denote these as M, P, S, T, respectively. To solve the mcss problem we can then use the following implementations for `combine`, `base`, and `emptyVal`:

function $\texttt{combine}((M_L, P_L, S_L, T_L), (M_R, P_R, S_R, T_L)) =$
$(\max(S_L + P_R, M_L, M_R), \quad \max(P_L, T_L + P_R), \quad \max(S_R, S_L + T_R), \quad T_L + T_R)$

function $\texttt{base}(v) = (v, v, v, v)$

$\texttt{emptyVal} = (-\infty, -\infty, -\infty, 0)$

and then solve the problem with:

function $\texttt{mcss}(S) =$
```
  reduce combine emptyVal (map base S)
```

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail or condense our code into just a few lines that take advantage of the almighty `reduce`. *So which is preferable*, using the divide-and-conquer code or using reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. You should realize, however, that this pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing quick sort as the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot.

2 Analyzing the Costs of Higher Order Functions

Last lecture we looked at using `reduce` to solve divide-and-conquer problems. In the example we gave, the maximum increasing subsequence sum, the combining function f had $O(1)$ cost (i.e., both its work and span are constant). In that case the cost specifications of `reduce` on a sequence of length n is simply $O(n)$ work and $O(\log n)$ span. Does that hold true when the combine function does not have constant cost?

For `map` it is easy to find its costs base on the cost of the function applied:

$$\begin{aligned} W(\texttt{map}\ f\ S) &= 1 + \sum_{s \in S} W(f(s)) \\ S(\texttt{map}\ f\ S) &= 1 + \max_{s \in S} S(f(s)) \end{aligned}$$

Tabulate is similar. But can we do the same for `reduce`?

Merge Sort. As an example, let's consider merge sort. As you have likely seen from previous courses you have taken, merge sort is a popular divide-and-conquer sorting algorithm with optimal work. It is based on a function `merge` that takes two already sorted sequences and returns a sorted sequence containing all elements from both sequences. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`. In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

```
combine = merge_<
base = singleton
emptyVal = empty()
function reduceSort(S) = reduce combine emptyVal (map base S)
```

where $\texttt{merge}_<$ is a merge function that uses an (abstract) comparison operator $<$. Note that merging is an associative function.

Assuming a constant work comparison function, two sequences S_1 and S_2 with lengths n_1 and n_2 can be merged with the following costs:

$$\begin{aligned} W(\texttt{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

What do you think the cost of `reduceSort` *is?*

2.1 Reduce: Cost Specifications

We want to analyze the cost of `reduceSort`. Does the reduction order matter? As mentioned before, if the combining function is associative, which it is in this case, all reduction orders give the same answer so it seems like it should not matter.

To answer this question, let's consider the reduction order we get by sequentially adding the elements in one after the other. On input $x = \langle x_1, x_2, \ldots, x_n \rangle$, the sequence of $\texttt{merge}_<$ calls looks like the following:

$$\texttt{merge}_<(\ldots\ \texttt{merge}_<(\texttt{merge}_<(\texttt{merge}_<(I, \langle\, x_1 \,\rangle), \langle\, x_2 \,\rangle), \langle\, x_3 \,\rangle), \ldots)$$

i.e. we first merge I and $\langle\, x_1 \,\rangle$, then merge in $\langle\, x_2 \,\rangle$, then $\langle\, x_3 \,\rangle$, etc.

With this order $\texttt{merge}_<$ is called when its left argument is a sequence of varying size between 1 and $n-1$, while its right argument is always a singleton sequence. The final merge combines $(n-1)$-element with 1-element sequences, the second to last merge combines $(n-2)$-element with 1-element sequences, so on so forth. Therefore, the total work for an input sequence S of length n is

$$W(\texttt{reduceSort}\ S) \ \leq\ \sum_{i=1}^{n-1} c \cdot (1+i) \ \in\ O(n^2)$$

since merge on sequences of lengths n_1 and n_2 has $O(n_1 + n_2)$ work.

Note that this reduction order is the order that the `iter` function uses, and hence is equivalent to:

```
function reduceSort'(S) =
  iter merge< (empty()) (map singleton S)
```

Furthermore, using this reduction order, the algorithm is effectively working from the front to the rear, "inserting" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. This corresponds to the well-known insertion sort.

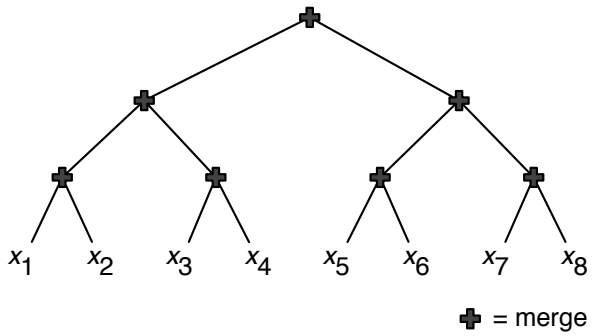
Clearly, this reduction order has no parallelism except within each merge, and therefore the span is

$$S(\texttt{reduceSort}\ x) \ \leq\ \sum_{i=1}^{n-1} c \cdot \log(1+i) \ \in\ O(n \log n)$$

since merge on sequences of lengths n_1 and n_2 has $O(\log(n_1 + n_2))$ span.

Notice that in the reduction order above, the reduction tree was extremely unbalanced. Would the costs change if the merges are balanced? For ease of exposition, let's suppose that the length of our sequence is a power of 2, i.e., $|x| = 2^k$. Now we lay on top the input sequence a "full" binary tree[2] with 2^k leaves and merge according to the tree structure. As an example, the merge sequence for $|x| = 2^3$ is shown below.

[2]This is simply a binary tree in which every node either has exactly 2 children or is a leaf, and all leaves are at the same depth.



Clearly using this balanced combining tree gives a smaller span than the imbalanced tree. But does it also reduce the work cost? At the bottom level where the leaves are, there are $n = |x|$ nodes with constant cost each (these were generated using a `map`). Stepping up one level, there are $n/2$ nodes, each corresponding to a `merge` call, each costing $c(1+1)$. In general, at level i (with $i = 0$ at the root), we have 2^i nodes where each node is a merge with input two sequences of length $n/2^{i+1}$. Therefore, the work of `reduceSort` using this reduction order is the familiar sum

$$\begin{aligned} W(\texttt{reduceSort}\ x) &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$. In fact, this algorithm is essentially the merge sort algorithm. Therefore we see that `mergeSort` and `insertionSort` are just two special cases of `reduceSort` that use different reduction orders.

As discussed earlier, we need to precisely define the reduction order to determine the result of `reduce` when applied to a non-associative combining function. These two examples illustrate, however, that even when the combining function is associative, the particular reduction order we choose can lead to drastically different costs in both work and span. When combining function has $O(1)$ work, using a balanced reduction tree improves the span from $O(n)$ to $O(\log n)$ but does not change the work. But with $\texttt{merge}_<$ as the combining function, we see that the balanced reduction tree also improves the work from $O(n^2)$ to $O(n \log n)$.

In general, how would we go about defining the cost of `reduce` with higher order functions. Given a reduction tree, we'll first define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ as

$$\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S) = \Big\{\text{all function applications } f(a,b) \text{ in the reduction tree}\Big\}.$$

Following this definition, we can state the cost of `reduce` as follows:

$$\begin{aligned} W(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(n + \sum_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} W(f(a,b))\right) \\ S(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(\log n \max_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} S(f(a,b))\right) \end{aligned}$$

The work bound is simply the total work performed, which we obtain by summing across all combine operations. The span bound is more interesting. The $\log n$ term expresses the fact that the tree is at most $O(\log n)$ deep. Since each node in the tree has span at most $\max_{f(a,b)} S(f(a,b))$, any root-to-leaf path, including the "critical path," has at most $O(\log n \max_{f(a,b)} S(f(a,b)))$ span.

This can be used, for example, to prove the following lemma:

Lemma 2.1. *For any combine function $f : \alpha \times \alpha \to \alpha$ and a monotone size measure $s : \alpha \to \mathbb{R}_+$, if for any x, y,*

1. $s(f(x,y)) \leq s(x) + s(y)$ *and*
2. $W(f(x,y)) \leq c_f \left(s(x) + s(y)\right)$ *for some universal constant c_f depending on the function f,*

then

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log |S| \sum_{x \in S} (1 + s(x))\right).$$

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce}\ \texttt{merge}_<\ \langle\,\rangle\ \langle \langle a \rangle : a \in A \rangle) = O(|A| \log |A|)$$