15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 6

Sequences - II

Synopsis

- The `reduce` operation
- Implementing divide and conquer with `reduce`
  - Implementing MCSS with `reduce`
- Analyzing cost of higher order functions
  - Cost analysis for `reduce`

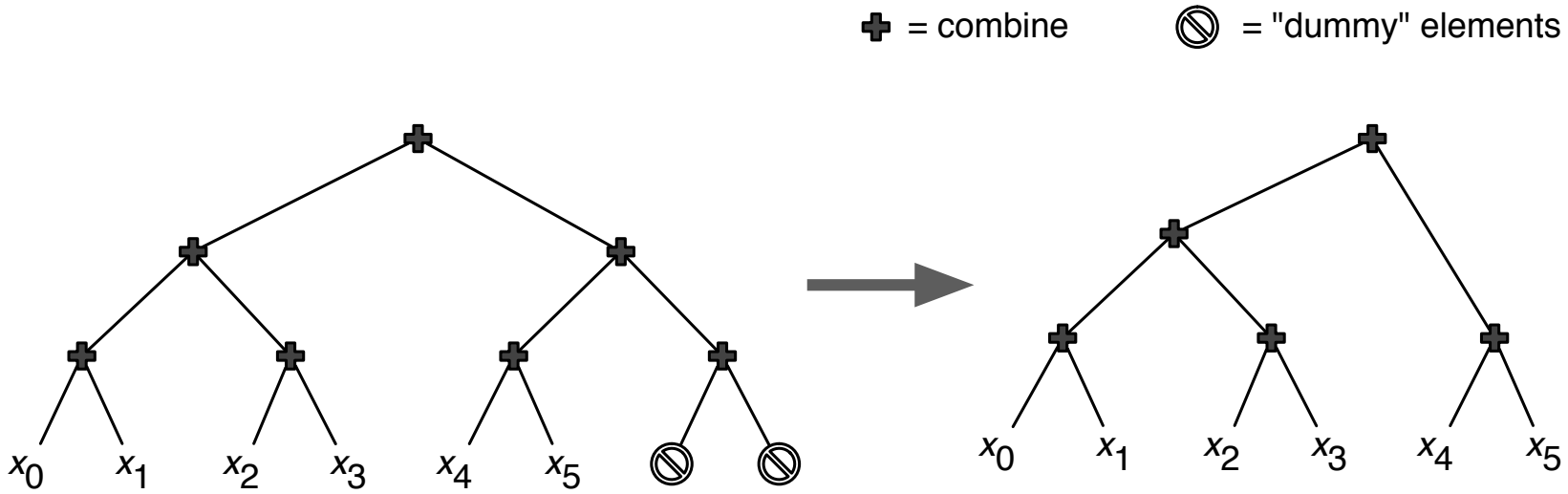
The Reduce Operation

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

- When *f* is associative, `reduce` returns sum with respect to *f*.
- Same result as `iter` *f I S*
  - `iter` is sequential and allows more general *f* (e.g., $\beta \times \alpha \to \beta$
- If *f* is not associative, `reduce` and `iter` results may differ.

The Reduce Operation

- Specific combination based on a perfect binary tree.



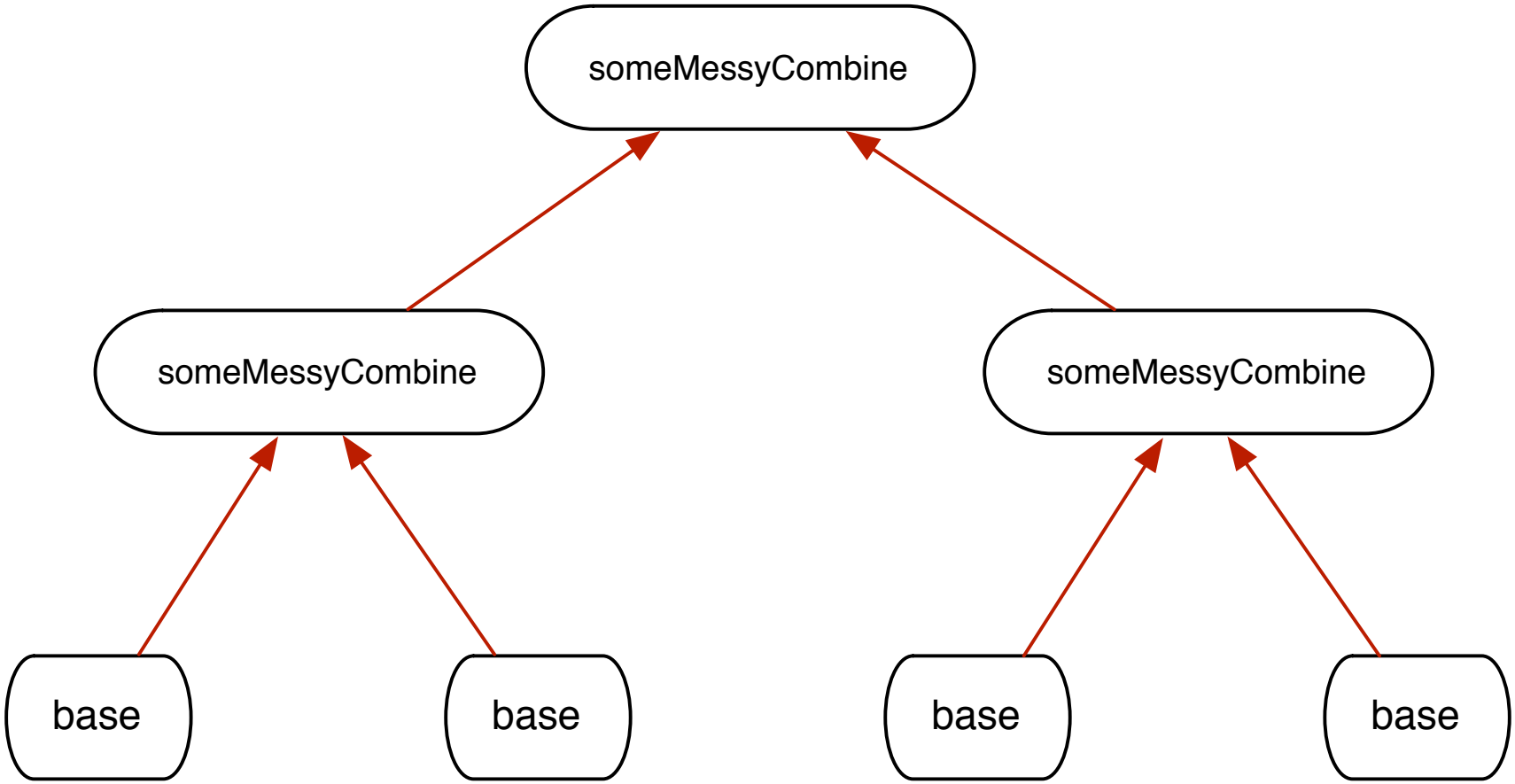
Divide and Conquer with Reduce

- Many divide and conquer have the following structure

```
1  fun 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
```

- This corresponds to a binary tree combination scheme.

Divide and Conquer with Reduce



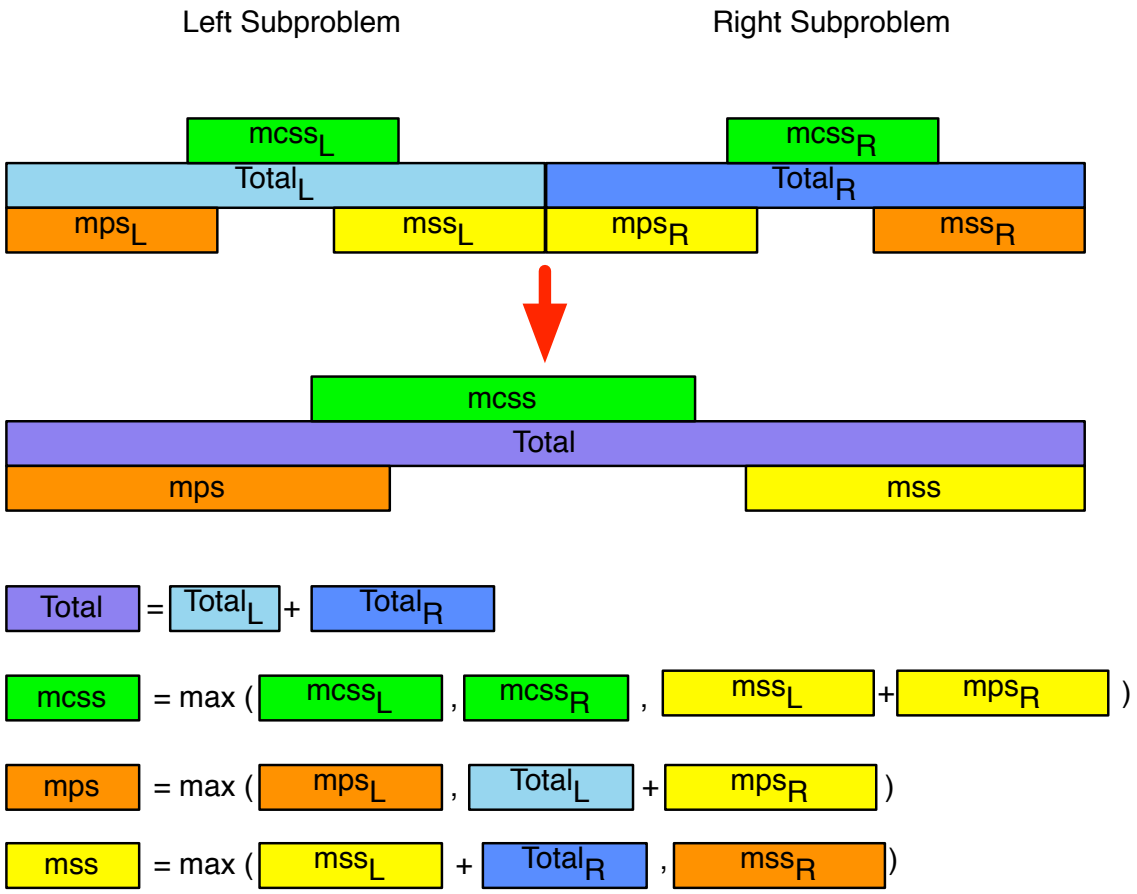
Divide and Conquer with Reduce

```
1  fun 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
```

reduce someMessyCombine emptyVal (*map* base S)

MCSS using Reduce

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



MCSS USING REDUCE

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

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

fun $\mathtt{base}(v) = (v, v, v, v)$

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

fun $\mathtt{mcss}(S) =$
$\mathtt{reduce\ combine\ emptyVal}\ (\mathtt{map\ base}\ S)$

Some Observations

- Which code to use is a matter of taste
  - `reduce` version is shorter
  - Divide and Conquer version exposes the inductive structure.
- `reduce` version not applicable when split is complicated
  - e.g., in Quicksort

Scan via Reduce

- How do we implement the divide and conquer *scan* with this template?
  - $\mathit{scan}\ f\ I\ S \equiv$
    $\mathit{reduce}\ \mathit{combine}\ \mathit{emptyVal}\ (\mathit{map}\ \mathit{base}\ S)$
- `emptyVal`$=?$ $(\langle\ \rangle, I)$
- **fun** $\mathit{base}(v) =?$ $(\langle I \rangle, f(I, v))$
- **fun** $\mathit{combine} =?$

  fun $\mathit{combine}((S_1, T_1), (S_2, T_2)) =$
  $\mathit{append}(S_1, (\mathit{map}\ (\mathit{fn}\ x\ \Rightarrow\ f(x, T_1))\ S_2),\ f(T_1, T_2))$

  - Is this right?

  fun $\mathit{combine}((S_1, T_1), (S_2, T_2)) =$
  $(\mathit{append}(S_1, (\mathit{map}\ (\mathit{fn}\ x\ \Rightarrow\ f(T_1, x))\ S_2),\ f(T_1, T_2))$

Cost of Higher Order Functions

- We assume that f runs in $O(1)$ work and span.
  - $\Rightarrow$ *reduce* has $O(n)$ work and $O(\log n)$ span
- Easy for `map` and `tabulate`

$$W(\texttt{map}\ f\ S) = 1 + \sum_{s \in S} W(f(s))$$

$$S(\texttt{map}\ f\ S) = 1 + \max_{s \in S} S(f(s))$$

- How about `reduce`?

MERGESORT VIA REDUCE

- Remember the reduce template for divide and conquer?

$$\texttt{reduce}\ \boxed{\texttt{combine}}\ \boxed{\texttt{emptyVal}}\ (\texttt{map}\ \boxed{\texttt{base}}\ S)$$

$$\texttt{combine} = \texttt{merge}_{<}$$

$$\texttt{base} = \texttt{singleton}$$

$$\texttt{emptyVal} = \texttt{empty}()$$

fun $\texttt{reduceSort}(S) =$

$$\texttt{reduce combine emptyVal}\ (\texttt{map base}\ S)$$

Cost of ReduceSort

- $\texttt{merge}_<$ is an associative function with costs:

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

- f has no longer $O(1)$ work and span.
- What is the cost of `reduceSort`.

Cost of Reducesort

- For costs, reduction sequence matters!
- Sequential order
- On input $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$, we get

$$\mathtt{merge}_<(\ldots\ \mathtt{merge}_<(\mathtt{merge}_<(\mathtt{merge}_<(I, \langle x_0 \rangle), \langle x_1 \rangle), \langle x_2 \rangle), \ldots)$$

- Left arg. has length 0 through $n-1$
- Right arg. always has length 1.
- Work:

$$W(\texttt{reduceSort}\ S) \ \leq\ \sum_{i=0}^{n-1} c \cdot (1+i) \ \in\ O(n^2) \text{ Why?}$$

MERGESORT WITH REDUCE

- Equivalent to `iter` version

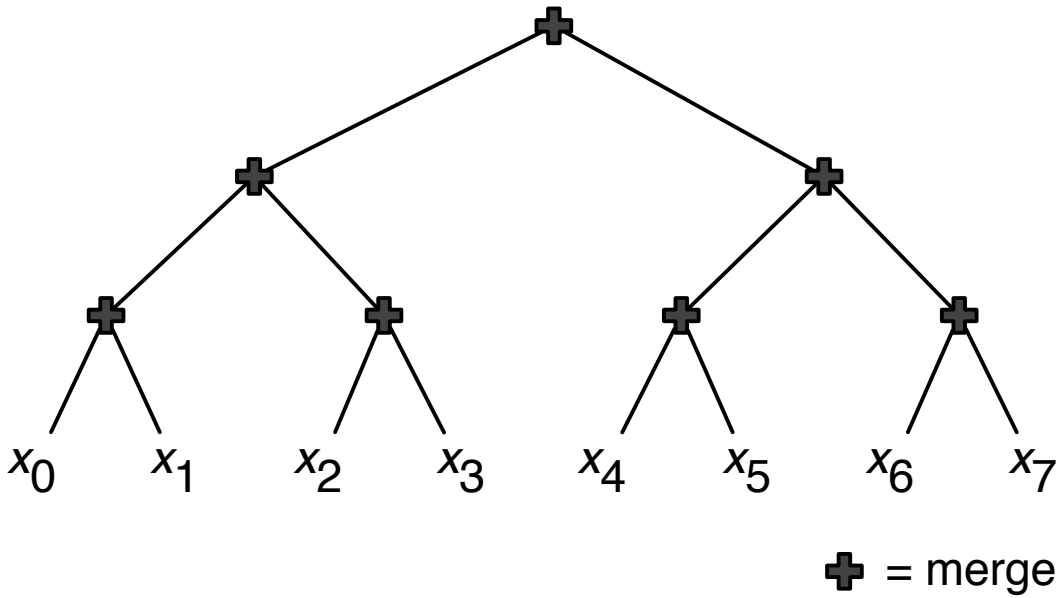
$$\textbf{fun}\ \mathit{reduceSort'}(S) = \mathit{iter}\ \mathit{merge}_{<}\ (\mathit{emptyVal}\ (\mathit{map}\ \mathit{base}\ S))$$

- Works really as an Insertion Sort.
  - Inserts each element into a sorted prefix!
- No parallelism except in `merge`

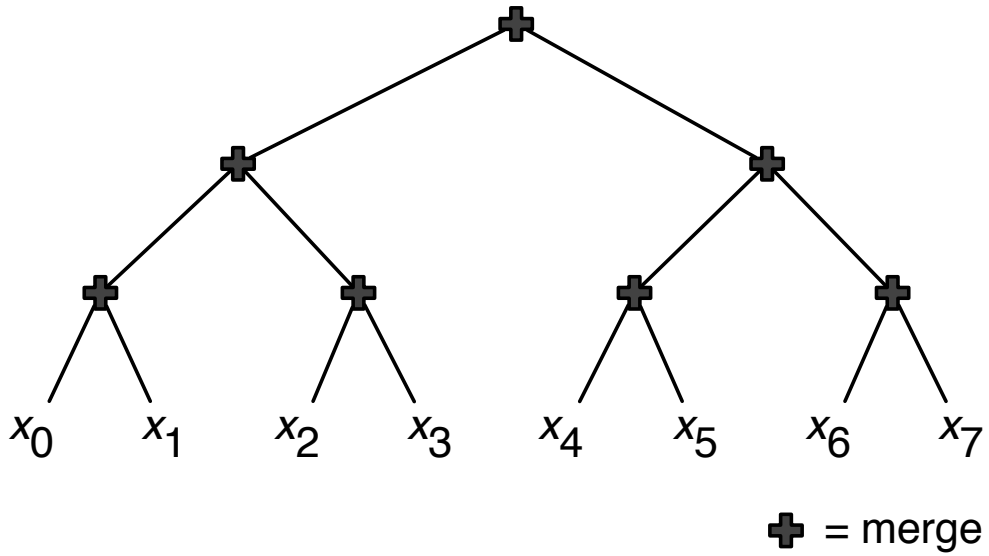
$$S(\texttt{reduceSort}\ S) \leq \sum_{i=0}^{n-1} c \cdot \log(1+i) \in O(n \log n)$$

Mergesort with Reduce

- The reduction tree is very unbalanced!
- Suppose $n = 2^k$ and merge tree is as follows

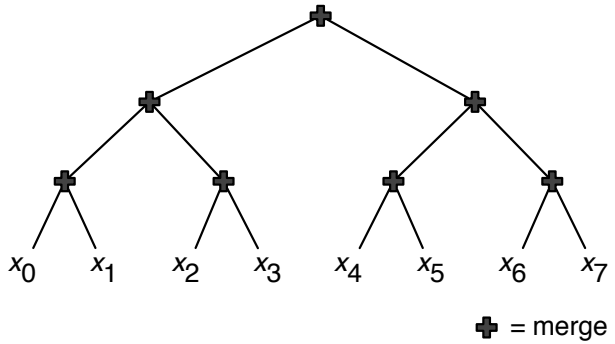


Mergesort with Reduce



- n nodes at constant cost at each leaf ($i = \log_2 n$)
- $n/2$ nodes one level up, each costing $c(1 + 1)$ ($i = \log_2 \frac{n}{2}$) (Why?)
- In general, for level i, we have 2^i nodes merging two sequences each length $\frac{n}{2^{i+1}}$

MERGESORT WITH REDUCE

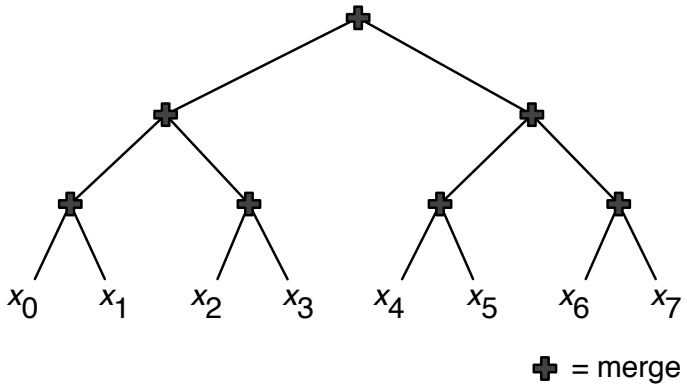


- For level i, we have 2^i nodes merging two sequences each length $\frac{n}{2^{i+1}}$

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

$$= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \in O(n \log n)$$

MERGESORT WITH REDUCE



- $W(\texttt{reduceSort}\,S) \in O(n \log n) \Rightarrow$ `mergeSort`.
- `mergeSort` and `insertionSort` are special cases of `reduceSort` using different reduction orders.

Reduce Order

- Result of `reduce` depends on the order when *f* is not associative
- When *f* is associative, different orders result in different costs.