Lecture 7 — Collect and Sets

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

Lectured by Margaret Reid-Miller — 5 February 2013

Material in this lecture:

- Cost of reduce (from last lecture notes)
- The collect operation.
- Set ADT

Challenge: Given sequences of integers A and B of length n, and an initial value X_0, calculate $X_{i+1} = A_i \cdot X_i + B_i$ for all $0 \le i < n$. You need to do it in $O(n)$ work and $O(\log n)$ span.

1 Collect

In many applications it is useful to collect all items that share a common key. For example we might want to collect students by course, documents by word, or sales by date. More specifically let's say we had a sequence of pairs each consisting of a student's name and a course they are taking, such as

$$\begin{aligned}\texttt{Data} = \langle & (\textit{“jack sprat”}, \textit{“15-210”}),\\ & (\textit{“jack sprat”}, \textit{“15-213”}),\\ & (\textit{“mary contrary”}, \textit{“15-210”}),\\ & (\textit{“mary contrary”}, \textit{“15-251”}),\\ & (\textit{“mary contrary”}, \textit{“15-213”}),\\ & (\textit{“peter piper”}, \textit{“15-150”}),\\ & (\textit{“peter piper”}, \textit{“15-251”}),\\ & \ldots\rangle\end{aligned}$$

and we want to collect all entries by course number so we have a list of everyone taking each course. Collecting values together based on a key is very common in processing databases. In relational database languages such as SQL it is referred to as "Group by". More generally it has many applications and furthermore it is naturally parallel.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its interface is:

$$\texttt{collect} : (\alpha \times \alpha \to \textit{order}) \to (\alpha \times \beta)\ \textit{seq} \to (\alpha \times \beta\ \textit{seq})\ \textit{seq}$$

The first argument is a function for comparing keys of type α, and must define a total order over the keys. The second argument is a sequence of key-value pairs. The `collect` function collects all values that share the same key together into a sequence. If we wanted to collect the entries of `Data` given above by course number we could do the following:

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

$$\texttt{collectStrings} = \texttt{collect String.compare}$$
$$\texttt{rosters} = \texttt{collectStrings}(\langle (c,n) : (n,c) \in \texttt{Data} \rangle)$$

This would give something like:

$$\begin{aligned}\texttt{rosters} = \langle & (\text{“15-150”}, \ \langle \text{“peter piper”}, \ldots \rangle) \\ & (\text{“15-210”}, \ \langle \text{“jack sprat”, “mary contrary”}, \ldots \rangle) \\ & (\text{“15-213”}, \ \langle \text{“jack sprat”}, \ldots \rangle) \\ & (\text{“15-251”}, \ \langle \text{“mary contrary”, “peter piper”} \rangle) \\ & \ldots \rangle\end{aligned}$$

We use a map $(\langle (c,n) : (n,c) \in \texttt{Data} \rangle)$ to put the course number in the first position in the tuple since that is the position used to collect on.

Collect can be implemented by sorting the keys based on the given comparison function, and then partitioning the resulting sequence. In particular, the sort will move the pairs so that all equal keys are adjacent. A partition function can then identify the positions where the keys change values, and pull out all pairs between each change. Doing this partitioning can be done relatively easily by filtering out the indices where the value changes. The dominant cost of `collect` is therefore the cost of the sort. Assuming the comparison has complexity bounded above by W_c work and S_c span, then the costs of `collect` are $O(W_c n \log n)$ work and $O(S_c \log^2 n)$ span. It is also possible to implement a version of collect that runs in linear work using hashing. But hashing would require that a hash function is also supplied and would not return the keys in sorted order. Later we discuss tables which also have a `collect` function. However tables are specialized to the key type and therefore neither a comparison nor a hash function need to be passed as arguments.

1.1 Using Collect in Map Reduce

Some of you have probably heard of the map-reduce paradigm first developed by Google for programming certain data intensive parallel tasks. It is now widely used within Google as well as by many others to process large data sets on large clusters of machines—sometimes up to tens of thousands of machines in large data centers. The map-reduce paradigm is often used to analyze various statistical data over very large collections of documents, or over large log files that track the activity at web sites. Outside Google the most widely used implementation is the Apache Hadoop implementation, which has a free license (you can install it at home). The paradigm is different from the `mapReduce` function you might have seen in 15-150 which just involved a map then a reduce. The map-reduce paradigm actually involves a map followed by a collect followed by a bunch of reduces, and therefore might be better called the map-collect-reduces.

The map-reduce paradigm processes a collection of documents based on a map function $f_{\texttt{m}}$ and a reduce function $f_{\texttt{r}}$ supplied by the user. The $f_{\texttt{m}}$ function must take a document as input and generate a sequence of key-value pairs as output. This function is mapped over all the documents. All key-value pairs across all documents are then collected based on the key. Finally the $f_{\texttt{r}}$ function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In ML the types for map function $f_{\mathtt{m}}$ and reduce function $f_{\mathtt{r}}$ are the following:

$$\begin{aligned} f_{\mathtt{m}} &: (\mathit{document} \rightarrow (\mathit{key} \times \alpha)\ \mathit{seq}) \\ f_{\mathtt{r}} &: (\mathit{key} \times (\alpha\ \mathit{seq}) \rightarrow \beta) \end{aligned}$$

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Typically the α and β types are limited to certain types. Also, in most implementations both the $f_{\mathtt{m}}$ and $f_{\mathtt{r}}$ functions are sequential functions. Parallelism comes about since the $f_{\mathtt{m}}$ function is mapped over the documents in parallel, and the $f_{\mathtt{r}}$ function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
function mapCollectReduce $f_m$ $f_r$ docs =
  let
     pairs = flatten $\langle f_m(s) : s \in$ docs $\rangle$
     groups = collect String.compare pairs
  in
     $\langle f_r(g) : g \in$ groups $\rangle$
  end
```

The function `flatten` simply flattens a nested sequence into a flat sequence, e.g.:

$$\begin{aligned} & \mathtt{flatten}\,\langle \langle a,b,c \rangle, \langle d,e \rangle \rangle \\ \Rightarrow\ & \langle a,b,c,d,e \rangle \end{aligned}$$

As an example application of the paradigm, suppose we have a collection of documents, and we want to know how often every word appears across all documents. This can be done with the following $f_{\mathtt{m}}$ and $f_{\mathtt{r}}$ functions.

```
function $f_m$(doc) = $\langle (w, 1)$ : tokens doc $\rangle$
function $f_r(w, s) = (w,$ reduce + 0 $s)$
```

Here `tokens` is a function that takes a string and breaks it into tokens by removing whitespace and returning a sequence of strings between whitespace.

Now we can apply `mapCollectReduce` to generate a `countWords` function, and apply this to an example case.

```
val countWords = mapCollectReduce $f_m$ $f_r$

countWords ⟨"this is a document",
            "this is is another document",
            "a last document"⟩
⇒ ⟨("a", 2), ("another", 1), ("document", 3), ("is", 3), ("last", 1), ("this", 2)⟩
```

2 An Abstract Data Type for Sets

Sets undoubtedly play an important role in mathematics and are often needed in the implementation of various algorithms. Whereas a sequence is an ordered collection, a set is its *unordered* counterpart. In addition, a set has no duplicate element.

Most programming languages either support sets directly (e.g., Python and Ruby) or have libraries that support them (e.g., in the C++ STL library and Java collections framework). They sometimes have more than one implementation of sets. For example, Java has sets based on hash tables and balanced trees. Unsurprisingly, the set interface in different libraries and languages differ in subtle ways. So, when using one of these interfaces you should always read the documentation carefully.

We now define an abstract data type for sets. The definition follows the mathematical definition of sets from set theory and is purely functional. In particular, when updating a set (e.g. with insert or delete) it returns a new set rather than modifying the old set.

Definition 2.1. For a universe of elements $\mathbb{U}$ (e.g. the integers or strings), the SET abstract data type is a type $\mathbb{S}$ representing the power set of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

$$\begin{array}{llll}
\texttt{empty} & : \mathbb{S} & = & \emptyset \\
\texttt{size}(S) & : \mathbb{S} \to \mathbb{N} & = & |S| \\
\texttt{singleton}(e) & : \mathbb{U} \to \mathbb{S} & = & \{e\} \\
\texttt{filter}(f,S) & : ((\mathbb{U} \to \{\mathrm{T},\mathrm{F}\}) \times \mathbb{S}) \to \mathbb{S} & = & \{s \in S \mid f(s)\} \\
\\
\texttt{find}(S,e) & : \mathbb{S} \times \mathbb{U} \to \{\mathrm{T},\mathrm{F}\} & = & |\{s \in S \mid s = e\}| = 1 \\
\texttt{insert}(S,e) & : \mathbb{S} \times \mathbb{U} \to \mathbb{S} & = & S \cup \{e\} \\
\texttt{delete}(S,e) & : \mathbb{S} \times \mathbb{U} \to \mathbb{S} & = & S \setminus \{e\} \\
\\
\texttt{intersection}(S_1,S_2) & : \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = & S_1 \cap S_2 \\
\texttt{union}(S_1,S_2) & : \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = & S_1 \cup S_2 \\
\texttt{difference}(S_1,S_2) & : \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = & S_1 \setminus S_2
\end{array}$$

where $\mathbb{N}$ are the natural numbers (non-negative integers).

We write this definition to be generic and not specific to Standard ML. In our library, the type $\mathbb{S}$ is called `set` and the type $\mathbb{U}$ is called `key`, the arguments are not necessarily in the same order, and some of the functions are curried. For example, the interface for find is `find` : `set` → `key` → `bool`. Please refer to the documents for details. In the pseudocode, we will give in class and in the notes we will use standard set notation as in the right hand column of the table above.

A Note about `map`. You may notice that the interface does not contain a `map` function. If we try to generalize the notion of `map` from sequences, a `map` function does not make sense in the context of a set: if we interpret map to take in a collection, apply some function to each element and return a collection of the same structure. Consider a function that always returns 0. Mapping this over a set would return all zeros, which would then be collapsed into a singleton set, containing exactly 0. Therefore, such a `map` would allow reducing the set of arbitrary size to a singleton, which doesn't match the `map` paradigm (which traditionally preserves the structure and size).

2.1 Cost Model

So far, we have laid out a semantic interface, but before we can put it to use, we need to worry about the cost specification. The most common efficient ways to implement sets are either using hashing or balanced trees. They have various tradeoffs in cost. For simplicity, we'll consider a cost model based on a balanced-tree implementation. We will cover how to implement these set operations when we talk about balanced trees later in the course. For now, a good intuition to have is that we use a comparison function to keep the elements in sorted order in a balanced tree. Since this requires comparisons inside the various set operations, the cost of these comparisons affects the work and span. For this, we'll assume that compare has C_w work and C_s span.

We have the following cost specification:

	Work	*Span*
`size`(S) `singleton`(e)	$O(1)$	$O(1)$
`filter`(f, S)	$O\left(\sum_{e \in S} W(f(e))\right)$	$O\left(\log\|S\| + \max_{e \in S} S(f(e))\right)$
`find`(S, e) `insert`(S, e) `delete`(S, e)	$O(C_w \cdot \log\|S\|)$	$O(C_s \cdot \log\|S\|)$
`intersection`(S_1, S_2) `union`(S_1, S_2) `difference`(S_1, S_2)	$O\big(C_w \cdot m \cdot \log(1 + \frac{n}{m})\big)$	$O\big(C_s \cdot \log(n + m)\big)$

where $n = \max(|S_1|, |S_2|)$ and $m = \min(|S_1|, |S_2|)$.

The work bounds for `intersection`, `union`, and `difference` deserve further discussion—at a glance they might seem a bit funky. These turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later.

Notice that when the two sets have the same length ($n = m$), the work is simply

$$O(C_w \cdot m \cdot \log(1 + 1)) = O(C_w \cdot n).$$

This should not be surprising, because it corresponds to the cost of merging two approximately equal length sequences (effectively what these operations have to do).

Moreover, you should also observe that in the case that one of the sets is a singleton, then the work is $O(\log n)$.

Buy in Bulk and Save. On inspection, the functions `intersection`, `union`, and `difference` are simply the "parallel" counterparts of the functions `find`, `insert`, and `delete`, to wit:

- `intersection` — search for multiple elements instead of one.
- `union` — insert multiple elements.
- `difference` — delete multiple elements.

In fact, it is easy to implement `find`, `insert`, and `delete` in terms of the others.

$$\begin{aligned}
\texttt{find}(S,e) &= \texttt{size}(\texttt{intersection}(S,\texttt{singleton}(e))) = 1\\
\texttt{insert}(S,e) &= \texttt{union}(S,\texttt{singleton}(e))\\
\texttt{delete}(S,e) &= \texttt{difference}(S,\texttt{singleton}(e))
\end{aligned}$$

Since `intersection`, `union`, and `difference` can operate on multiple elements they are well suited for parallelism, while `find`, `insert`, and `delete` have no parallelism. Consequently, in designing parallel algorithms it is good to think about how to use `intersection`, `union`, and `difference` instead of `find`, `insert`, and `delete` if possible. For example, one way to convert a sequence to a set would be to insert the elements one by one, which can be coded as

```
fun fromSeq(S) = Seq.iter (fn (S',e) => Set.insert e S') Set.empty S
```

However, the above is sequential. To do it in parallel we could instead do

```
fun fromSeq(S) = Seq.reduce Set.union Set.empty (Seq.map Set.singleton S)
```

Exercise 1. *What is the work and span of the first version of* `fromSeq`.

Exercise 2. *Show that on a sequence of length n the second version of* `fromSeq` *does $O(C_w n \log n)$ work and $O(\log^2 n)$ span.*