Recitation 4 — Reduction, MapCollectReduce

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

September 19, 2012

1 Announcements

- How did Assignment 2 go?
- Assignment 1 will be returned tomorrow. A suspiciously common problem was one character mistakes (e.g. "`lVal`" instead of "`rVal`"). Better testing or better style may have caught these errors!
- Assignment 3 is out—get an early start!
- Questions about homework, lecture, life?

2 `fields` and `tokens`

Two useful string parsing routines are `fields` and `tokens` which breaks a string into a sequence of words. The only difference between the two is that fields will return empty words (which is preferable for parsing data written for or by computers) and tokens will not (which is more useful for human-centric data).

More specifically, we pick some characters to be delimiters (the argument f takes a character and returns whether it is a delimiter) and define a word to be a maximal string without delimiters. Note the input string might consist of multiple consecutive delimiters (in which case `fields` which return an empty string for that word and `tokens` will simply omit it). For example, we'll parse a line from a comma-separated values (CSV) file.

```
val fields :  (char -> bool) -> string -> string seq
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
```

which would return

⟨"a", "", "", "line", "of", "a", "csv", "", "file"⟩.

Using `tokens`, instead of `fields`, the result would have been

⟨"a", "line", "of", "a", "csv", "file"⟩.

Traditionally `fields` and `tokens` have been implemented in a sequential fashion, starting at the front end and processing it one character at a time until reaching the end. During the processing, whenever a whitespace or delimiter is found, the algorithm declares the current token or field finished and starts a new one. However, `fields` and `tokens` can be implemented in parallel.

How do we go about implementing `fields` *in parallel?* We can figure out where each field starts by finding locations of the delimiters. Further, this also tells us where each field ends—right before each delimiter (which starts the next field), and at the end of the string. Therefore, if there is a delimiter at location i and the next delimiter is at $j \geq i$, we have that the field starting after i contains the substring extracted from locations $(i+1)..(j-1)$, which may be empty. This leads to the following code, in which `delims` contains location of each delimiter. We use the notation $\oplus$ to denote sequence concatenation.

```
fun fields f s = let
  val delims = ⟨−1⟩ ⊕ ⟨ i :  0 ≤ i < |s| ∧ f(s[i]) ⟩ ⊕ ⟨|s|⟩
in
  ⟨ s[delims[i]+1, delims[i+1]-delims[i]-1) :  0 ≤ i < |delims| − 1 ⟩
end
```

To illustrate the algorithm, let's run it on our familiar example.

```
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
delims = ⟨−1,1,2,3,8,11,13,17,18,23⟩
result = ⟨ s[0,1), s[2,2), s[3,3), s[4,8), s[9,11), s[12,13), ... ⟩
result = ⟨"a", "", "", "line", "of", "a", "csv" , "", "file" ⟩
```

2.1 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_m and a reduce function f_r supplied by the user. The f_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_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_m and reduce function f_r are the following:

$$\begin{aligned} f_m &: (document \to (key \times \alpha)\ seq) \\ f_r &: (key \times (\alpha\ seq) \to \beta) \end{aligned}$$

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_m and f_r functions are sequential functions. Parallelism comes about since the f_m function is mapped over the documents in parallel, and the f_r function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
1  fun mapCollectReduce f_m f_r docs =
2    let
3      val pairs = flatten ⟨ f_m(s) : s ∈ docs ⟩
4      val groups = collect String.compare pairs
5    in
6      ⟨ f_r(g) : g ∈ groups ⟩
7    end
```

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

$$\begin{array}{ll} & \texttt{flatten}\,\langle\langle a,b,c\rangle,\langle d,e\rangle\rangle \\ \Rightarrow & \langle a,b,c,d,e\rangle \end{array}$$

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 $f_{\texttt{m}}$ and $f_{\texttt{r}}$ functions.

$$\textbf{fun}\ f_{\texttt{m}}(\texttt{doc}) = \langle (w,1) : \texttt{tokens doc}\rangle$$

$$\textbf{fun}\ f_{\texttt{r}}(w,s) = (w, \texttt{reduce}\ +\ 0\ s)$$

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.

$$\texttt{val countWords} = \texttt{mapCollectReduce}\ f_{\texttt{m}}\ f_{\texttt{r}}$$

$$\begin{array}{l} \texttt{countWords}\ \langle \textit{“this is a document”}, \\ \qquad\qquad\quad \textit{“this is is another document”}, \\ \qquad\qquad\quad \textit{“a last document”}\rangle \end{array}$$

$$\Rightarrow \langle (\textit{“a”}, 2), (\textit{“another”}, 1), (\textit{“document”}, 3), (\textit{“is”}, 3), (\textit{“last”}, 1), (\textit{“this”}, 2)\rangle$$

3 Graph Representations

Suppose you're given an undirected graph $G = (V, E)$, where V is a set of vertices (also known as nodes) and $E \subseteq \binom{V}{2}$ is a set of edges. Notice that in this definition, an edge $e = \{x, y\}$ is a set of size two representing the endpoints. Thus, the edge $\{x, y\}$ represents the same edge as $\{y, x\}$. Following this, we can think of representing a graph simply as an edge set. But the edge set representation doesn't provide an efficient means to access the neighbors of a vertex. That's why in class we looked at a different representation which we call an adjacency set. In the following, we'll look at a simple problem that will illustrate the differences in complexity between these two common representations.

Consider writing a simple routine to compute the degree of a vertex v. First, how would we do it in the edge set representation? Let the edge set E be represented as a set.

```
1  fun degree E v =
2    let
3      val nbrs = filter (fn (x,y) => v=x orelse v=y) E
4    in
5      |nbrs|
6    end
```

What's the cost of this function? Linear work and $O(\log n)$ span.

Now suppose we're given a graph represented in the edge set form. Can we convert it into an adjacency set? What is the type of an adjacency set? We use `int set IntTable.table`.

```
1  fun makeAdjSet E =
2    let
3      val biDir = flatten (map (fn (x,y) => %[(x,y), (y,x)]) (toSeq E))
4      val nbrsSeq = collect Int.compare biDir
5    in
6      Table.fromSeq (map (fn (u, nbrs) => (u, Set.fromSeq nbrs)) nbrsSeq)
7    end
```

What's the cost of `makeAdjSet`? $O(n \log n)$ work and $O(\log^2 n)$ span.

Finally, in this adjacency set representation, it's super easy to compute the degree.

```
1  fun degree' G v =
2    let
3      val nbrs = find G v
4    in
5      size nbrs
6    end
```

What's the cost? Ans: $O(\log n)$ cost (both work and span).