CSE341T/CSE549T **10/01/2014**

Collect, Map Reduce, and Abstract Data Types

Lecture 11

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

```
D = <("jack smith", "CSE 341"),
     ("jack smith", "CSE 332"),
     ("mary poppins", "CSE 360"),
     ("mary poppins", "ESE 443"),
     ("mary poppins", "CSE 441"),
     ("bruce wayne", "CSE 341"),
     ("bruce wayne", "CSE 441"),
     ... >
```

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. More generally it has many applications and furthermore it is naturally parallel.

We will use the function COLLECT$(comp, A)$ for this purpose. The first argument is a function for comparing keys, 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 `D` given above by course number we should first swap the key and value pairs.

This would give something like:

```
rosters = < ("CSE 332", < "jack smith", ... >)
            ("CSE 341", < "jack smith",peter cope", ... >),
            ("CSE 360", < "mary poppins", ... >)
           ... >
```

It is often the case that the key needs to be extracted before applying collect.

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 all the equal keys so they

are adjacent. A partition function can then identify the positions where the keys change values (using scan), and pull out all pairs between each change. Thus, 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 $O(S_c \log^3 n)$ 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.

2 Map Reduce Using Collect

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 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. But we are stuck with the standard terminology here.

Map: Map is just applying a function to all the elements of a data structure and returning a data structure of the same format. Therefore, if you have an array, and you apply MAP(f, A), you would get an array where function f has been applied to all its elements.

The map-reduce paradigm processes a collection of documents based on MAPF and REDUCEF functions supplied by the user. The MAPF 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 REDUCEF function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Also, in most implementations both the MAPF and REDUCEF functions are sequential functions. Parallelism comes about since the MAPF function is mapped over the documents in parallel, and the REDUCEF function is mapped over the keys with associated values in parallel.

Example 1: Computing Word Counts

We now consider 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. Here's the algorithm:

1. Tokenize the documents using the algorithm you developed in Homework 2. Each token is now a word in the document.

2. For each token, set the value as 1.

3. Perform a collect using the tokens as keys.

4. Now perform a sum reduction on all individual keys.

The first two steps are performed in MAPF.

Example 2: Breadth-First Search to Find Shortest-Path Length

One can also use the Map Reduce paradigm to do breadth-first search (BFS), and one of the main application of BFS is computing the shortest path.

We will assume the graph is stored as follows — each node is an object, with member fields such as node id, list of neighbors, and distance to root. The list of neighbors contains pointers to other nodes that are adjacent to this node. The distance to root is initialized to infinity.

We can compute BFS by traversing the graph one "frontier" at a time using one round of map-reduce, and repeat the map-reduce process until no more nodes are discovered:

1. MAPF takes `<node_id, (my_distance, node*)>` as its input, where the `node*` is just the pointer to the actual node with the corresponding `node_id`. It maps this node to its neighboring nodes and generate the next frontier to traverse, i.e., neighboring nodes that have distance greater than `my_distance+1`. So, it outputs a list of `<node_id, (distance, node*)>` pair, where the keys are the node id's of its neighboring nodes that have distance greater than `my_distance+1`, and the distance value is `my_distance+1`. The node* is the pointer to the actual node with the corresponding `node_id`. This additional parameter is necessary, because the REDUCEF needs to be able to update the actual node.

2. We can get duplicate nodes, because some nodes have multiple incoming edges. Collect will get rid of the duplicated nodes.

3. REDUCEF then do a min-reduce based on the distance in the value (but with unit-weight edge, the value of a given node should be the same) and update the distance in the node to the result of min-reduce.

4. Repeat these steps until no more nodes are discovered.

Exercise 1 *Extend this map-reduce algorithm to handle positive-weighted graph, where the edges can have positive weights.*

3 Sets

Sets play an important role in mathematics and often needed in the implementation of various algorithms. It is therefore useful to have an abstract data type that supports operations on sets. Indeed most programming languages either support sets directly (e.g., python) or have libraries that support them (e.g., in the C STL library and Java collections framework). Such languages sometimes have more than one implementation of sets. Java, for example, has sets bases on hash tables and balanced trees.

The most common efficient ways to implement sets are either using hashing or balanced trees. Here we will specify a cost model based on a balanced-tree implementation. For now, we won't describe the implementation in detail, but will later in the course. Roughly speaking, however, the idea is to 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. In the table below we assume that the work and span of a comparison on the elements is bounded by C_w and C_s respectively.

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

where $n = \max(|S_1|, |S_2|)$ and $m = \min(|S_1|, |S_2|)$. We didn't talk about the function `filter` in class. `filter`(f, s) basically returns all the elements e of S for which $f(e)$ returns true.

The work for `intersection`, `union`, and `difference` might seem a bit funky, but the bounds turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later, but for now you should observe that in the special case that the two input lengths are within a constant, the work is simply $O(n)$. This bound corresponds to the cost of merging two approximately equal length sequences, which is effectively what these operations have to do. You should also observe that in the case that one of the sets is a singleton, then the work is $O(\log n)$.

On inspection, the three functions `intersection`, `union`, and `difference` have a certain

symmetry with the functions `find`, `insert`, and `delete`, respectively. In particular `intersection` can be viewed as a version of find where we are searching for multiple elements instead of one. Similarly `union` can be viewed as a version of `insert` that inserts multiple elements, and `difference` as a version of `delete` that deletes multiple elements. In fact it is easy to implement `find`, `insert`, and `delete` in terms of the others. 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. This has the span of $O(n \lg n)$ if your are inserting n elements. Instead, you should try to do this in parallel using the union operation.

Exercise 2 *Show that on a sequence of length n can be converted to a set with $O(C_w n \log n)$ work and $O(C_s \log^2 n)$ span.*

4 Tables or Dictionaries

A table is an abstract data type for storing data associated with keys. They are similar to sets, but along with each element (key) we store some data. The table supplies operations for finding the value associated with a key, for inserting new key-value pairs, and for deleting keys and their associated value. Tables are also called dictionaries, associative arrays, maps, mappings, or, in set theory, functions. For the purpose of parallelism the interface we will discuss also supplies "parallel" operations that allow the user to insert multiple key-value pairs, to delete multiple keys, and to find the values associated with multiple keys.

As with sets, tables are very useful in many applications. Most languages have tables either built in (e.g. dictionaries in python), or have libraries to support them (e.g. map in the C STL library and the Java collections framework). We note that the interfaces for these languages and libraries have common features but typically differ in some important ways, so be warned. Most do not support the "parallel" operations we discuss. Here we will define tables mathematically in terms of set theory before committing to a particular language.

Formally, a table is set of key-value pairs where each key appears only once in the set.

Technically a table is a set of pairs and can therefore be written as

$$\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\},$$

as long as the keys are distinct. Distinct from sets, the `find` function does not return a Boolean, but instead it returns the value associated with the key k. As it may not find the key in the table, its result may be bottom ($\perp$). Also note that, in general, a table should have some default behavior for

insert if the key you are trying to insert is already in the table. A similar default behavior should be specified for the merge operation.

The costs of the functions of the table are very similar to the functions in the set.

	Work	*Span*
`size`(T) `create`	$O(1)$	$O(1)$
`filter`(f, T)	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\log\|T\| + \max_{(k,v)\in T} S(f(v))\Big)$
`map`(f, T)	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\log\|T\| + \max_{(k,v)\in T} S(f(v))\Big)$
`find`(T, k) `insert`$(T, (k, v))$ `delete`(T, k)	$O(C_w \log\|T\|)$	$O(C_s \log\|T\|)$
`extract`(T_1, T_2) `merge`(T_1, T_2) `erase`(T_1, T_2)	$O\big(C_w m \log(\frac{n+m}{m})\big)$	$O\big(C_s \log(n+m)\big)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$.

As with sets there is a symmetry between the three operations `extract`, `merge`, and `erase`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three.