Recitation 4 — Reduction, MapCollectReduce, Graphs

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

February 8, 2012

Today's Agenda:
- Announcements
- Reductions
- MapCollectReduce
- Fields
- Graph Representations

1 Announcements

- Assignment 2 was due yesterday at 11:59pm. Make sure you reread the late day policy if you plan to hand it in late.
- Assignment 3 went out on yesterday and will be due next Tuesday the 14th.
- Questions about homework, class, life, universe?

2 `kth element`

The kth problem is: given a sequence S and an int k, return the kth smallest element of S.

We can solve this by recalling from $15-150$ how quicksort works. We somehow (more on this later) pick a partition element, split the array into things less than the pivot and things more than the pivot, sort each half, and append them back together.

However, if we are only looking for one element, we can do better: we don't need to recurse on both halves, just the half that has the element we want. We can write `kth` as follows:

```
fun partition S x = (filter (fn y => y < x), x, filter (fn y => y >= x));

fun kth S k =
let
  val (L,m,R) = partition S (choose_pivot S)
in
  case Int.compare(k, length L) of
       LESS => kth S k
     | EQUAL => m
     | GREATER => kth S (k-(length L)-1)
end
```

Then the work recurrence is $W(n) = W(n/2) + O(n)$, so the work is $O(n)$. The span recurrence is $S(n) = S(n/2) + O(\log(n))$, so the span is $O(\log^2(n))$. These recurrences assume we cut the work in half each time: actually, the runtimes remain the same as long as we recurse on at most a constant fraction of the original size each time: that is, we want the size of each recursive step to be bounded above by cn, for some $c < 1$. We'll look at this issue more carefully later.

Waxing philosphical for a moment: this is sort of like reducing kth element into quicksort, but a little different. In a reduction, we use the solution to the other problem as a black box, whereas here we used the details of our solution to produce a slightly different solution. The same distinction is common in mathematics: the difference between using a mathematical result (reduction) in a proof and using the idea that worked in one proof in another (what we just did).

In mathematics (and theoretical computer science), reduction is very common; when writing code, we tend towards reuse. The reason is that when writing code, we care about runtime (reductions can be expensive even if it doesn't increase the asymptotic complexity) and don't have an infinite library (it is useless to reduce a problem to 2-SAT if you don't have code to solve 2-SAT). Mathematics has neither of these problems.

Another detail we glossed over: how to choose a pivot. We'll examine three ideas:

First Element Just always pivot on the first element. This *usually* works, but in the worst case (a sorted sequence, ironically enough) this only reduces the problem size to $n-1$ each time, which isn't good enough.

Randomly There is a 50% chance that we'll choose an element between the first quarter and the third quarter of the array, in which case we will reduce the problem size by at least a quarter. You can use probability (we'll show the analysis later in the course) to show that this implies with high probability that the height of the recursive tree will be $O(\log(n))$.

Median of Medians In 1973, Blum, Floyd, Pratt, Rivest and Tarjan published "Time bounds for selection" which gave a determinstic algorithm for `kth` that runs in linear time. The idea is to split the sequence into blocks of five, take the median of each sequence (in $O(1)$ time) and use the median of this sequence (which can be computed recursively in $W(n/5)$ work and $S(n/5)$ span) as the pivot. You can show that this pivot has at least 30% of the elements of the sequence on each side, so the size of the recursive subproblem is at most $7n/10$. So the overall work is described by $W(n) = W(n/5) + W(7n/10) + O(n)$, which the tree method shows is $O(n)$ pretty easily. The span is described by $S(n) = S(n/5) + S(7n/10) + O(\log(n))$, which you can show is $O(\log^2(n))$.

The asympotic cost is the same, but this algorithm is significantly more complicated to implement and significantly (about 5 times) slower than the random algorithm, so it is rarely used in practice.

3 `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 useful for CSV parsing) and tokens will not (which is more useful for the general case).

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
tokens (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
```

which would return

```
⟨"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. Part of the reason for this sequential nature probably has to do with how strings are loaded from external storage (e.g. tape/disk) many years ago. Here, we'll assume that we have random access to the input string's characters, a common assumption now that a program tends to load a big chuck of data into main memory at a time. This makes it possible to implement `fields` and `tokens` in parallel. We will now discuss a parallel implementation of `fields`.

How do we go about implementing `fields` *in parallel?* Notice that we can figure out where each field starts by looking at the locations of the delimiters. Further, we know where each field ends—this is necessarily right before the delimiter that starts the next field. 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 the starting location of each field. We use the notation $\oplus$ to denote sequence concatenation.

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

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

```
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"

delims = ⟨0,2,3,4,9,12,14,18,19,24⟩

result = ⟨ s[0,1), s[2,2), s[3,3), s[4,8), s[9,11), s[12,13), ... ⟩

result = ⟨"a", "", "", "line", "of","a", ... ⟩.
```

4 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 used 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. But we are stuck with the standard terminology here.

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 ML the types for map and reduce functions are the following:

$$\begin{aligned} mapF : &\quad (document \to (key \times \alpha)\ seq) \\ reduceF : &\quad (key \times (\alpha\ seq) \to \beta) \end{aligned}$$

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 *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.

In ML map reduce can be implemented as follows

```
fun mapCollectReduce mapF reduceF S =
  let
    val pairs = flatten (map mapF S)
    val groups = collect String.compare pairs
  in
    map reduceF groups
  end
```

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

```
  flatten < < a, b, c>, < d, e> >
 = < a, b, c, d, e >
```

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. This can be done with the following *mapF* and *reduceF* functions.

```
fun mapF D = map (fn w => (w,1)) (tokens spaceF D)
fun reduceF(w,s) = (w, reduce op+ 0 s)
```

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

```
  val countWords = mapCollectReduce mapF reduceF

  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) >
```

5 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.

```
fun degree E v =
  let val nbrs = filter (fn (x,y) => v=x orelse v=y) E
  in  size nbrs
  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`.

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

```
fun degree' G v =
  let val nbrs = find G v (** cost?  **)
  in size nbrs
  end
```

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

6 Test Your Code!

There is no substitute for testing your code even when you have carefully reasoned it to be correct. It is equally easy to write a buggy proof as it is to write buggy code. The bottom line is *test your code!* And if you can, use your proof to guide the testing and use the test to double check your proof.

Beware of bugs in the above code; I have only proved it correct, not tried it - Donald Knuth