CLIENT-SERVER PARADIGM, MAPREDUCE 22

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 29, 2010

1 Threads, Callbacks

```
(define (im-server-start)
  ;;;Start the server.
  ;
  ;Set! server-socket variable
  ;Set thunk for handling handshake with new client
  ;
  (format logging "~%Server starting...~%")
  (set! server-socket (make-server-socket))
  (format #t "Server IP address: ~A, server port: ~A~%"
  (get-ip-address-as-string)
  (socket-port-number server-socket))
  (when-socket-ready server-socket
    (lambda ()
      (begin
        (format logging "New client connecting.~%")
          (handshake (socket-dup server-socket)))))
  (format logging "(im-server-start) done.~%~%")
  'okay)
```

2 The 3-Way Handshake

```
(define (handshake sock)
  ;;;Handle the three-way handshake with a client.
  ;
  ;Handshaking should go as follows:
```

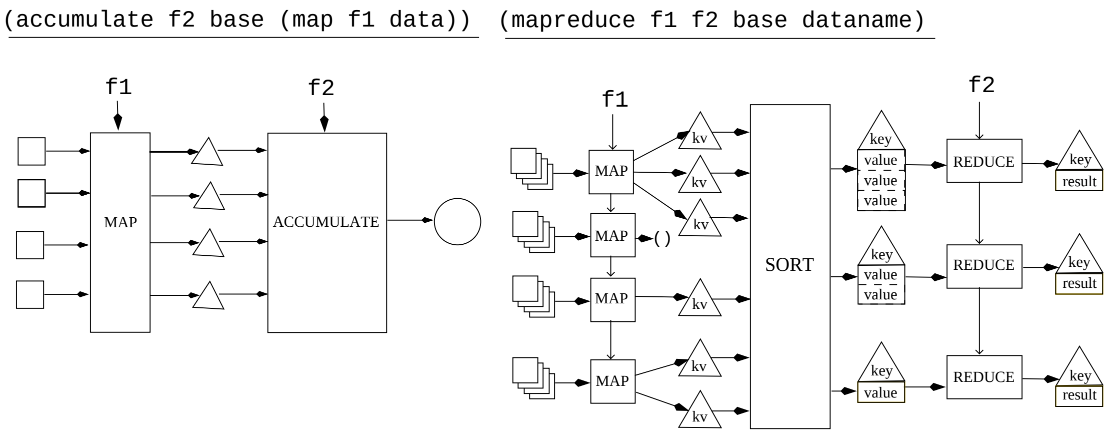
```
  ;client->server:
  ;   request from CLIENT to server with request "hello" and data nil
  ;server->client:
  ;   request from server to CLIENT with request "welcome" and data nil
  ;client->server:
  ;   request from CLIENT to server with request "thanks" and data nil
  ;
  ;;Accept the socket connection
  (socket-accept-connection sock)
  (format logging "Connection accepted for ~A...~%" sock)
  (let* ((port-from-client (socket-input sock))
 (port-to-client (socket-output sock))
 (req (get-request port-from-client)))
    (if (not req)
(socket-shutdown sock #f)
(begin
  (format logging "Request received: ~S~%" req)

  ;; Check message is "hello".
  (cond ((not (equal? 'hello (request-action req)))
 (format #t "Bad request from client: ~S"
 req)
 (socket-shutdown sock #f))
((member (request-src req) (get-clients-list))
 ;; name already exists, send "sorry" to client
 (format logging "Sending 'sorry' to client~%")
 (send-request (make-request 'server
     (request-src req)
     'sorry
     nil)
       port-to-client)
 (format #t "Name ~A already exists."
 (request-src req))
 (socket-shutdown sock #f))
(else
 ;;Send "welcome" message back.
 (format logging "Sending welcome message.~%")
 (if (not
      (send-request (make-request 'server
  (request-src req)
  'welcome
  nil)
    port-to-client))
     (socket-shutdown sock #f)
     (begin

       ;; Check response is "thanks"
       (set! req (get-request port-from-client))
       (if (not req)
   (socket-shutdown sock #f)
   (begin
     (format logging "Response received: ~S~%" req)
     (if (not (equal? 'thanks (request-action req)))
 (begin (format #t "Bad response from client: ~S" req)
(socket-shutdown sock #f))
 (begin
   ;; Finally, we can register the client
   (format logging "~A has logged on.~%"
   (request-src req))
   (register-client (request-src req)
     sock)
   (format logging "Finished handshake~%") )) ) ))))) )))
  'okay)
```

3 MapReduce

Here's the diagram of mapreduce again:



(accumulate f2 base (map f1 data))
(mapreduce f1 f2 base dataname)
f1
f2
MAP
ACCUMULATE
f1
MAP
kv
()
SORT
key
value
REDUCE
result
f2

The seemingly unpoetic names `f1` and `f2` serve to remind you of two things: `f1` (the mapper) is used before `f2` (the reducer), and `f1` takes one argument while `f2` takes two arguments (just like the functions used with ordinary `map` and `accumulate` respectively).

mapper: `kv-pair→list-of-kv-pairs`

reducer: `value,partial-result → result`

All data are in the form of key-value pairs. Ordinary `map` doesn't care what the elements of the data list argument are, but `mapreduce` works only with data each of which is a key-value pair. In the Scheme interface to the distributed filesystem, a file is a stream. Every line of the file is a key-value pair whose key is the filename and whose value is a list of words, representing the text of the line.

(Actually, "a file is a stream" is a slight handwave. The STk running on the parallel cluster has special versions of `stream-car`, etc., that accept both ordinary streams and special file streams that are really pointers to data in the distributed file system, but that simulate the behavior of streams. So if you print one of these special streams directly, rather than using `show-stream`, you'll see something that doesn't look familiar.)

Each processor runs a separate `stream-map`**.** The overlapping squares at the left of the mapreduce picture represent an entire stream. (How is a large distributed file divided among map processes? It doesn't really matter, as far as the `mapreduce` user is concerned; `mapreduce` tries to do it as efficiently as possible given the number of processes and the location of the data in the filesystem.) The entire stream is the input to a `map` process; each element of the stream (a kv-pair) is the input to your mapper function `f1`.

For each key-value pair in the input stream, the mapper returns a list of key-value pairs. In the simplest case, each of these lists will have one element; the code will look something like

```
(define (my-mapper input-kv-pair)
  (list (make-kv-pair ... ...)))
```

The interface requires that you return a list to allow for the non-simplest cases:

(1) Each input key-value pair may give rise to more than one output key-value pair. For example, you may want an output key-value pair for each *word* of the input file, whereas the input key-value pair represents an entire line:

```
(define (my-mapper input-kv-pair)
  (map (lambda (wd) (make-kv-pair ... ...))
    (kv-value input-kv-pair)))
```

(2) There are *three* commonly used higher order functions for sequential data, `map`, `accumulate/reduce`, and `filter`. The way `mapreduce` handles the sort of problem for which `filter` would ordinarily be used is to allow a mapper to return an empty list if this particular key-value pair shouldn't contribute to the result:

```
(define (my-mapper input-kv-pair)
  (if ...
      (list input-kv-pair)
      '()))
```

Of course it's possible to write mapper functions that combine these three patterns for more complicated tasks.

The keys in the kv-pairs returned by the mapper need not be the same as the key in the input kv-pair.

Instead of one big accumulation, there's a separate accumulation of values for each key. The non-parallel computation in the left half of the picture has two steps, a `map` and an `accumulate`. But the `mapreduce` computation has *three* steps; the middle step sorts all the key-value pairs produced by all the mapper processes by their keys, and combines all the kv-pairs with the same key into a single aggregate structure, which is then used as the input to a `reduce` process.

This is why the use of key-value pairs is important! If the data had no such structure imposed on them, there would be no way for us to tell `mapreduce` which data should be combined in each reduction.

Although it's shown as one big box, the sort is also done in parallel; it's a "bucket sort," in which each `map` process is responsible for sending each of its output kv-pairs to the proper `reduce` process. (Don't be confused; your mapper function doesn't have to do that. The `mapreduce` program takes care of it.)

Since all the data seen by a single `reduce` process have the same key, the reducer doesn't deal with keys at all. This is important because it allows us to use simple reducer functions such as `+`, `*`, `max`, etc. The Scheme interface to `mapreduce` recognizes the special cases of `cons` and `cons-stream` as reducers and does what you intend, even though it wouldn't actually work without this special handling, both because `cons-stream` is a special form and because the iterative implementation of `mapreduce` would do the combining in the wrong order.

In the underlying `mapreduce` software, each `reduce` process leaves its results in a separate file, stored on the particular processor that ran the process. But the Scheme interface to `mapreduce` returns a single value, a stream that effectively `stream-append`s the results from all the `reduce` processes.

Running mapreduce: The `mapreduce` function is not available on the standard lab machines. You must connect to the machine that controls the parallel cluster. To do this, from the Unix shell you say this:

```
ssh icluster1.eecs.berkeley.edu
```

If you're at home, rather than in the lab, you'll have to provide your class login to the `ssh` command:

```
ssh cs61a-XY@icluster.eecs.berkeley.edu
```

replacing `XY` above with your login account. `Ssh` will ask for your password, which is the same on the parallel cluster as for your regular class account. Once you are logged into `icluster1`, you can run `stk` as usual, but `mapreduce` will be available:

```
(mapreduce mapper reducer reducer-base-case filename-or-special-stream)
```

The first three arguments are the mapper function for the `map` phase, and the reducer function and starting value for the `reduce` phase. The last argument is the data input to the `map`, but it is restricted to be either a distributed filesystem folder, which must be one of these:

"/beatles-songs" This one is small and has all Beatles song names

"/gutenberg/shakespeare" The collected works of William Shakespeare

"/gutenberg/dickens" The collected works of Charles Dickens

"/sample-emails" Some sample email data for the homework

"/large-emails" A much larger sample email dataset. Use this only if you're willing to wait a while.

(the quotation marks above are required), or the stream returned by an earlier call to `mapreduce`. (Streams you make yourself with `cons-stream`, etc., can't be used.) Some problems are solved with two `mapreduce` passes, like this:

```
(define intermediate-result (mapreduce ...))
(mapreduce ... intermediate-result)
```

(Yes, you could just use one `mapreduce` call directly as the argument to the second `mapreduce` call, but in practice you'll want to use `show-stream` to examine the intermediate result first, to make sure the first call did what you expect.)

Here's a sample. We provide a file of key-value pairs in which the key is the name of a Beatles album and the value is the name of a song on that album. Suppose we want to know how many times each word appears in the name of a song:

```
(define (wordcount-mapper document-line-kv-pair)
  (map (lambda (wd-in-line)
              (make-kv-pair wd-in-line 1))
       (kv-value document-line-kv-pair)))
(define wordcounts (mapreduce wordcount-mapper + 0 "/beatles-songs"))
(ss wordcounts)
```

The argument to `wordcount-mapper` will be a key-value pair whose key is an album name, and whose value is a song name. (In other examples, the key will be a filename, such as the name of a play by Shakespeare, and the value will be a line from the play.) We're interested only in the song names, so there's no call to `kv-key` in the procedure. For each song name, we generate a list of key-value pairs in which the key is a word in the name and the value is 1. This may seem silly, having the same value in every pair, but it means that in the `reduce` stage we can just use + as the reducer, and it'll add up all the occurrences of each word.

You'll find the running time disappointing in this example; since the number of Beatles songs is pretty small, the same computation could be done faster on a single machine. This is because there is a significant setup time both for `mapreduce` itself and for the `stk` interface. Since your mapper and reducer functions have to work when run on parallel machines, your Scheme environment must be shipped over to each of those machines before the computation begins, so that bindings are available for any free references in your procedures. It's only for large amounts of data (or long computations that aren't data-driven, such as calculating a trillion digits of π, but `mapreduce` isn't really appropriate for those examples) that parallelism pays off.

By the way, if you want to examine the input file, you can't just say

```
(ss "/beatles-songs") ; NO
```

because a distributed filename isn't a stream, even though the file itself is (when viewed by the `stk` interface to `mapreduce` a stream. These filenames only work as arguments to `mapreduce` itself. But we can use `mapreduce` to examine the file by applying null transformations in the map and reduce stages:

```
(ss (mapreduce list cons-stream the-empty-stream "/beatles-songs"))
```

The mapper function is `list` because the mapper must always return a list of key-value pairs; in this case, `map` will call `list` with one argument and so it'll return a list of length one.

Now we'd like to find the most commonly used word in Beatle song titles. There are few enough words so that we could really do this on one processor, but as an exercise in parallelism we'll do it partly in parallel. The trick is to have each reduce process find the most common word starting with a particular letter. Then we'll have 26 candidates from which to choose the absolutely most common word on one processor.

```
(define (find-max-mapper kv-pair)
  (list (make-kv-pair (first (kv-key kv-pair))
                      kv-pair)))
(define (find-max-reducer current so-far)
  (if (> (kv-value current) (kv-value so-far))
      current
      so-far))
(define frequent (mapreduce find-max-mapper find-max-reducer
                  (make-kv-pair 'foo 0) wordcounts))
(ss frequent)
(stream-accumulate find-max-reducer (make-kv-pair 'foo 0) frequent)
```

This is a little tricky. In the `wordcounts` stream, each key-value pair has a word as the key, and the count for that word as the value: `(back . 3)`. The mapper transforms this into a key-value pair in which the key is the first letter of the word, and the value is *the entire input key-value pair*: `(b . (back . 3))`. Each `reduce` process gets all the pairs with a particular key, i.e., all the ones with the same first letter of the word. The reducer sees only the values from those pairs, but each value is itself a key-value pair! That's why the reducer has to compare the `kv-value` of its two arguments.

As another example, here's a way to count the total number of lines in all of Shakespeare's plays:

```
(define will
  (mapreduce (lambda (kv-pair)
               (list (make-kv-pair 'line 1)))
             + 0 "/gutenberg/shakespeare"))
```

For each line in Shakespeare, we make exactly the same pair `(line . 1)`. Then, in the `reduce` stage, all the ones in all those pairs are added. But this is actually a bad example! Since all the keys are the same (the word `line`), only one `reduce` process is run, so the counting isn't done in parallel. A better way would be to count each play separately, then add those results if desired. You'll do that in lab.