CME 323: Distributed Algorithms and Optimization, Spring 2017
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 10, 5/3/2017. Scribed by A. Santucci.

Topics covered today.

- Distributed sort
- Intro to map reduce
- Applications of map reduce

10 Distributed sort

We had this situation where data is partitioned across p machines, and our old sorting algorithms fail: between each recursive call, we potentially require all data to be shuffled over the network. We wish to avoid such network bottlenecks. We are now in the situation where we don't want to communicate too much, and we don't want to page between disk and RAM too often.

What should our output look like? When we are done, we want the data for each machine to be sorted, and for an ordering between machines; i.e. we wish for everything on machine 1 to be smaller than everything on machine 2, itself containing only data less than machines 3, etc. We also want no single machine to be overloaded with too much data.

Assumptions We restrict ourselves to the case where we sort a billion integers with values ranging from 1 to 1 billion, and we don't know the distribution of our data within this interval.[1] We have one thousand machines available to our disposal. Each machine has a hard-disk, RAM, and lots of CPU's. On each machine, we assume we can't fit all of its local data in RAM, but we can fit its local data on disk of course.

What if we did know the true distribution? Suppose we try sorting data on each machine individually. After this is done, we have no idea how the data in one machine compares to the data in another. Suppose we *did know* the distribution of integers, but that we are only allowed to communicate each data value over the network at most once.[2] Take anything that's in the "first

[1]We only assume that no data value appears more than one thousand times. This is to avoid situations where we end up sending too much data to a single machine.

[2]We can always approximate the distribution by sampling.

histogram bin", and send all this data to one machine. What if the data are stored all over the place? Let's consider a fully correct algorithm, and then analyze it.

```
1 Each machine sorts its own data
2 Each machine samples 100 data points from its local data and sends to a driver node
3 Driver approximates integer distribution by sorting its sampled data
4 From this estimate, driver can determine cutoff thresholds for each machine
5 All thresholds are broadcast to all machines, i.e. each machine knows its relevant data
6 All-to-all communication: each machine shuffles data to correct location
7 Each machine sorts the incoming (sorted) partitions that it gets sent in linear time
```

Algorithm 1: Distributed sort

A note on determining cutoff points In steps 4 and 5, we specifically seek a sequence of cutoff points such that each bin of our histogram contains an equal number of data points, i.e. the total number of data values divided evenly by the number of machines. Notice that these cutoff points may not be uniformly spaced if the distribution of data is not uniform. To get each cutoff point, once the driver has sorted the sampled data, we just need to look at every kth value in the sorted array, where k is a multiple of the length of the array over the number of machines; this gives an estimate of the cutoff point for the kth machine.

A note on all to all communication Crucially, because each machine has sorted its data (on disk), we can simply stream data from disk and send data across network, where the receiving machine reads the data in directly to disk. This means we don't need to bother storing data in RAM temporarily or seeking to find relevant data on disk. Since all machines know all cutoff points, each machine knows exactly which machines to query to get the relevant local data. This all-to-all communication is very expensive, but the network router supports all machines simultaneously talking to all other machines; what happens is that each machine's bandwidth gets saturated, wherein that becomes the bottleneck, not the router switch. [3]

What's left? After the all-to-all communication, we still need to merge the results back to one node. But we've seen how to merge two sorted arrays in linear time on our homework.

What about machine failures? We haven't even touched on this. Assuming that each hard-disk is backed up many times, what would we do in the event of a failure?[4] Let's walk through this. If any machine dies in step 1 or 2, we can just restart the step. If the driver dies in step 3, we need to ask all workers to send results back to a new driver, i.e. we need to repeat steps one through three. Hopefully, our driver lives through step four. By step 5, more machines may have

[3]Asking might be preferable to pushing data, since if a machine dies, a new machine can come up with the responsibilities of the old one to fetch the data corresponding to the dead machine. On the other hand, if data is pushed to a node, it's more difficult to recover from a failure.

[4]We define a distributed filesystem as one in which data is stored redundantly in at least three physically different locations.

died, but at this point we can just restart the step itself with new machines. The go-to response when a machine dies is to retrace our steps (without going back too far).

Takeaway In a distributed setting, sorting is non-trivial. It's worth knowing how it works. Often, sorting is the most trivial sub-routine in an algorithm. Can we abstract away from the difficulties of machine failures? Yes, this leads us into map reduce.

11 Introduction to map reduce

Map reduce as a contract You may have learned about assembly, which is a difficult to read language. So, we choose to write code in a high level language and instead have a compiler write machine code for us. The contract is that we must adhere to `if-else` statements and `for` loops. Similarly, map reduce gives us a contract as well. What is the contract? We're in a distributed computing environment. Map-reduce is a two phase contract.

1. Map phase: data will be fed into a `map` algorithm (user-provided), where pairs of the form `(key, value)` are output.
2. Reduce phase: all pairs with the same key are guaranteed to show up on a single machine.

Between the map and reduce stage, there is a hidden and implicit sort performed. Why is this necessary? Put another way, how can we implement the promise that all pairs with the same key show up on a single machine; how do we implement this shuffle phase? Map each key to an integer, sort the integers, and then the data with the same key will show up on the same machine.

What does map reduce provide? It provides fault-tolerance in the case of hardware failure. They also provide a distributed file system that backs up your data at least three times. It also abstracts away details of communication patterns.