CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 13, 5/14/2020. Scribed by Andreas Santucci, Edited by Robin Brown.

13 Distributed Summation

Suppose we have data sharded between p machines and we simply want to sum up all the data. If we model each machine with our PRAM model, we can use the parallel summation algorithm discussed in the first half of the course. Note that these parallel summations can happen concurrently at all the machines. Once the individual machines are done, suppose their results are communicated to a master and aggregated there. If we assume that each message has size m with L latency and B bandwidth, then this requires $p(L + \frac{m}{B})$ time.[1]

What's wrong with this approach? If every machine tries to send its partial sums to the master, then its network card will be the bottle-neck. The other machines are capable of aggregating the partial sums as well. Moreover, the network cards on all other machines lay idle. This means we are not efficiently using our computation and communication resources. A more efficient algorithm for distributed summations by using *bit-torrent* broadcasting, and extrapolating our parallel summation algorithm. Specifically, on the first time step machines m_1, m_2 sum up their results, machines m_3, m_4 sum up their results, etc. From each pair, we select exactly one machine to propagate the result to the next level in our tree. This allows us to take full advantage of our compute resources. Moreover, by using as many links in the network as possible, this allows us to avoid bottlenecks at any one network card.

This algorithm results in $\log_2(p)$ rounds of communication before termination. Each round takes $L + m/B$ communication time, for a total of $\log_2(p)\,(L + m/B)$ communication time.

Notice that the choice to compute partial sums between pairs of machines is arbitrary. We could instead sum up between multiple machines to use as much of the bandwidth as possible. In practice, we often will not worry about this $\log p$ factor—this is because machines are costly, so it's very rare to so many machines that the $\log p$ term is the limiting factor.

After all rounds of communication take place, only one machine has the sum. If all the machines need to use this sum, we need to send it to them via a one-to-all broadcast. We can use **bit-torrent broadcasting** to accomplish this.

14 Simple random sampling

On a single machine, if we have n data points and want to sample an observation uniformly at random, it suffices to first generate a random number between 0 and $n-1$, and select the data point

[1] L is on the order of 1-2 milliseconds for a local network, or a couple hundred milliseconds is the data need to be sent across the world. Ultimately, we are limited by the speed of light; we can't transfer physical information or data any faster than that.

with this index. In the distributed setting, we now have to deal with the challenge of incomplete knowledge. We will often face scenarios where we do not have a priori knowledge of the amount of data being fed into our algorithms. Moreover, this data may come from various sources such as the disk, or network.

For example, consider running an algorithm using data from the Twitter firehose[2]With gigabytes of data coming in per second, it is infeasible to store all the data and randomly sample from it. Additionally, with a constant stream of data, there is no notion of when the all of the data has come in.

The following algorithm gives a method to sample from a stream of data uniformly at random.

```
1 Initialize return value r as empty, and stream counter k as 1.
2 while stream is not yet empty do
3 |   Read an item from the stream, and flip a coin with probability 1/k.
4 |   if coin comes up heads then
5 |   |   r ← s
6 |   end
7 |   Increment k
8 end
9 return r
```

Algorithm 1: Sampling from a stream uniformly at random

We will now prove the correctness of the streaming uniform sampling algorithm. Precisely, we need to show that after n items have been read, the value stored in r has equal probability $1/n$ of being any of these n items. We will prove the claim via induction. The base case follows by observing that when $n = 1$, our stream size will have counter value $k = 1$, and so r will be set to the first element in the stream with probability $1/k = 1$, hence r correctly represents a random sample from the singleton set containing only the first item in the stream.

For the inductive step, we assume that our claim holds for a stream up to length $n-1$. Consider the state of the algorithm after the nth item is read from the stream and processed. Our stream size counter has value $k = n$, and so r will be set to the newest element of the stream with probability $1/n$, and we leave r changed with probability $(n-1)/n$. We now need to show that r has equal probability of being any of the stream items seen so far. Specifically, we need to show that that $\Pr(r = s_i) = 1/n$ for all $i = 1, \ldots, n$, where s_i denotes the ith item of the stream.

We already proved that r has value s_n with probability $1/n$, i.e. $\Pr(r = s_n) = 1/n$. Now we need to consider $\Pr(r = s_i)$ for $i < n$.

If r was not replaced on the most recent step, it had the same value as it had after $n-1$ steps. By the inductive hypothesis, this value of r represents a sample uniform at random from the first $n-1$ items of the stream, i.e. $\Pr(r = s_i = \frac{1}{n-1}$ for $i = 1, \ldots, n-1$. Since r is *not replaced* with

[2]Twitter firehose is the complete stream of public messages. For reference, Users send 500 million tweets every day. This corresponds to 6000 tweets per second.

probability $(n-1)/n$, we see that

$$\Pr(r = s_i) = \frac{n-1}{n} \cdot \frac{1}{n-1} = \frac{1}{n}$$

for all $i = 1, \ldots, n-1$. Hence after n items have been read from the stream, we have $\Pr(r = s_n) = 1/n$ for every $i = 1, \ldots, n$. This completes the inductive step.

15 Distributed sort

With distributed summation, simply extending the parallel summation algorithm worked well. What if we tried to implement quicksort on a cluster?

Recall in quicksort, we select a pivot p uniformly at random. Then we construct the sets L and R where L contains all items less than p and R contains all items greater than p. We recursively call quicksort on L and R until all items have been sorted.

Now, suppose we simply try to implement this algorithm on a cluster. Specifically, assume we have B machines, were each machine has some portion of the data. We first need to randomly pick a pivot p. Each machine can sample uniformly at random from its data. From there, we can select one of theses B samples based on a weighted index of the amount of data on each machine.

Now that we have a pivot, we need to construct L and R. This is problematic because we are typically using distributed sorting because the input data is too large to fit on one machine. Consequently, L and R likely will not fit on one machine either. We could try to allocate half of the machines to storing L and half to R, and recursively using this approach. However, in the worst case, this could result in reshuffling all of the data in each recursive call. We might try to get around this by not shuffling data around maintaining $L_1, \ldots, L_M$ and $R_1, \ldots, R_M$ on each machine. The recursive call would involve picking one pivot out of $L_1, \ldots, L_M$ and other from $R_1, \ldots, R_M$, and sorting according to those pivots. Ultimately, this would result in sorted arrays on each machine. However, merging these sorted array would still take linear time in the number of inputs.

Often, recursion requires too much overhead from preparing and returning function inputs and outputs to be used in a distributed computing framework.

It is helpful to think about how our output might differ from the parallel setting. When our algorithm is done, we want the data on each individual machine to be sorted, and for there to be an ordering on the machines. For example, if we select $d_1, d_2, \ldots, d_B$ from machines $1, \ldots, B$ respectively, then we requires that $d_1 < d_2 < \ldots < d_B$. We also want the quantity of data on each of the machines to be balanced.

Suppose we knew the true distribution of our data. Then each machine can sort its own data and send its data from the "first histogram bin" to one machine, second to another and so on. Distributed sort works by estimating the true distribution of the data. Below, we give the

algorithm for distributed sorting.

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 2: 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 is 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 In step 6, because each machine has sorted its data (on disk), we can stream data from disk and send it across the network, where the receiving machine reads the data directly in to disk. This means we don't need to temporarily store the data in RAM, spend timing finding the relevant data on disk. Moreover, the driver machine broadcasts the cutoff points to all machines, so the data exchange in step 6 can be implemented in a peer to peer fashion. This can be done one of two ways. In the first, each machine pushes the appropriate portions of its data to each other machine. In the second, each machines queries every other machine for the appropriate portions of the data. The second is often preferable because fault-tolerance is simple. If a machine dies during this process, it can simply query the other machines for the data again. In contrast, with the first method, other machines cannot inherently detect that one of the machines has died, and thus do not know if they need to resend the data.

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.

16 Introduction to MapReduce

MapReduce is a programming paradigm that is particularly amenable to processing large data sets in a distributed fashion over a cluster of machines. The user defines a *map* function, and *reduce* function. The run-time system then handles the details of parallelization, fault-tolerance, data distribution, and load balancing. This is a powerful framework allows users to automatically parallelize and distributed large-scale computations.

16.1 Programming Model

Users of the MapReduce library write two scripts specifying a *map* function and *reduce* function.

The *map* function takes an input set of key/value pairs and outputs an intermediate set of key/value pairs. The MapReduce library guarantees that all values with the same key are grouped together before passing them to the *reduce* functions.

The *reduce* function takes as input **one** intermediate key and a set of values for that key, and outputs a (potentially smaller) set of values. Typically, each *reduce* invocation produces only one output value, but it is not required.

16.1.1 Examples

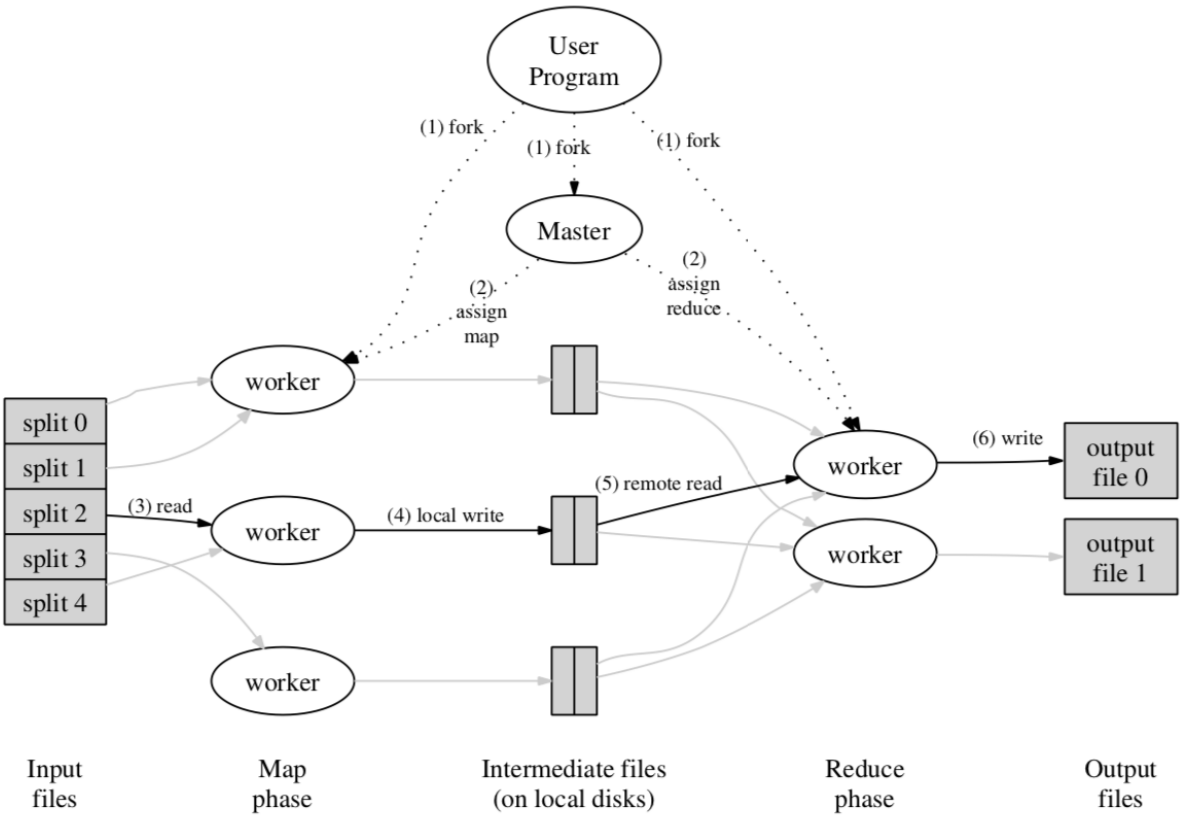
Counting Occurrences Suppose we want to count the number of occurrences of each word in a collection of documents. In this case, the input key/value pairs would be the document name and document contents respectively. The *map* function would output key/value pairs of the form $(word, 1)$ for each *word* in the documents. The reduce function would take an input a single word key and a list of counts for that word. For example, if the word "w" appears k times in all of the documents, then there will be k key/value pairs (w, 1). The *reduce* function would then iterate through the pairs for an individual word and add up the counters.

Inverted Index Suppose we want to locate all occurrences of a word in a set of documents. The map function would take as input document IDs and document contents, parse each document and output a sequence of (word, document ID) pairs. The reduce function takes the (word, document ID) pairs and outputs (word, list(document ID)).

16.2 MapReduce Execution

There are multiple possible implementations of the MapReduce interface, with the appropriate one depending on the computing environment. This section overviews the implementation targeting the computing environment at Google: a large cluster of commodity PCs connected together with switched Ethernet.

The *map* function is executed in parallel by multiple machines by partitioning the input data into M "splits"; the mapping function can be executed independently and in parallel on each of the splits. Similarly, the *reduce* function can be executed in parallel by partitioning the set of keys (output from the *map* function) into R splits. The partitioning function and number of splits, R, are specified by the user.



When a user calls the MapReduce function, the following sequence of steps are executed (depicted in Figure 16.2).

1. The MapReduce library in the user program partitions the input files into M splits (typically of 16 - 64 MBs per split).

2. Copies of the program are also started up on each machine in the cluster. One of the copies is the designated master, while the remaining are workers that are assigned tasks by the master. In total there are $M + R$ tasks that need to be assigned.

3. Workers that are assigned *map* tasks are each assigned a split of the input data, on which it runs the *map* function. The output key/value pairs are buffered in memory.

4. The buffered pairs are written to a one of R partitions of the local disk. The locations of these buffered pairs are passed back to the master program, which passes the locations to the *reduce* workers when they are assigned a *reduce* task.

5. When a *reduce* worked is assigned a *reduce* task, it receives a list of buffered locations from the master, and reads in all the intermediate data from these locations. It sorts the read-in data by the intermediate keys so that all key/value pairs with the same key are grouped

together—there is no guarantee that all of the data passed to a single *reduce* worker have the same key.

6. Each reduce worker iterates over the sorted data, and starts a new reduce task when a new key is encounter. The outputs of each of the reduce invocations is appended to a final output file for the partition.

7. Once all of the map and reduce tasks are complete, the output of the entire MapReduce execution is available as the aggregtion of the R output files. Typically, the outputs for the partitions are not combined.

During the execution of the MapReduce, the master coordinates computation across the worker nodes. Specifically, for each of the map and reduce tasks, the master stores a state ("idle", "in-progress", or "completed") and the ID of the worker assigned to the task if it is "in-progress" or "completed". When a map task is completed, the master stores the location and sizes of the R intermediate file regions produced by the map task. This information is passed onto the reduce worked with "in-progress" reduce tasks.

16.3 Fault Tolerance of MapReduce

One of the key features of the MapReduce paradigm is that it automatically builds in robustness to machine failure. When data is being processed by hundreds or thousands of machines, it is likely that one will fail, so the library must recover from failures efficiently. The master keeps tabs of worker failure by periodically pinging each worker. If it receives no response (within a set time period) it marks that worker as failed, and any map tasks tasks that were completed by that worker are marked as "idle" so they can be assigned to another worker. Similarly, "in-progress" map or reduce tasks are rest to "idle" if their workers have failed. Completed map tasks need to be re-executed because their output are stored on the local disk of the failed machine, and thus cannot be accessed. On the other hand, because the outputs of the reduce functions are stored in a global file system, they are still accessible even if their worker dies. Consequently, completed reduce tasks do not need to be reexecuted if their worker fails. When a map tasks is reassigned, the reduce workers are notified of the reassignment, so they can read data from the appropriate map workers.

MapReduce does not build in safe-guards in case of master failure. This is because the master is only one node, and the chance of one particular node failing are unlikely.

References

[1] R. Zadeh. *Introduction to Distributed Optimization.*

[2] R. Zadeh. *Distributed Computing with Spark and MapReduce.*