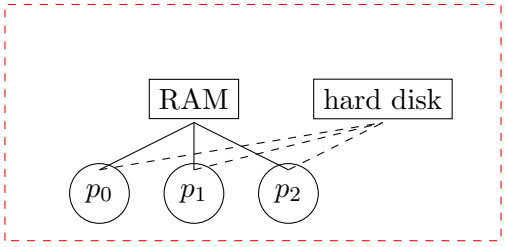
CME 323: Distributed Algorithms and Optimization, Spring 2017
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

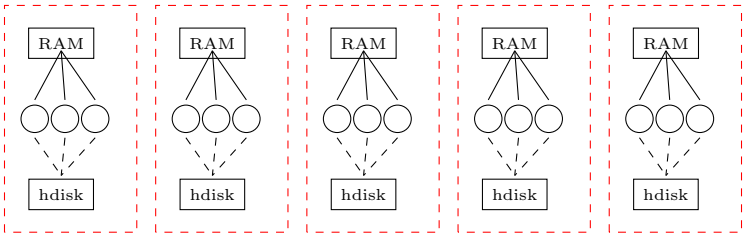
Lecture 8, 4/26/2017. Scribed by A. Santucci.

8 Introduction to Distributed Computing

We're all familiar with the PRAM by now. What does it look like? We have one block of RAM, connected to a bunch of CPUs, all on a single machine.



In this case, one thread has access to their own processor, p_i, and all processors have access to RAM. Even in the PRAM world, we we still must pull the data in from disk; it's possible to stream data directly from disk. The reason we have to use multiple computers now, is because whatever data we have can't fit on a single machine (or single hard-drive).



We call the above a cluster of machines.

Assumptions

- Data has been sharded between machines.[1]
- Network calls are expensive. I.e. network latency and bandwidth are much faster intra-machine than inter-machine.

These are the two main challenges which arise when trying to distribute algorithms.

[1] How we partition the data matters, but for now just understand that the data are split between machines.

Observations It is still the case that $T_p \geq \frac{T_1}{p}$, where now p is the total number of processors available in the cluster. Before, in PRAM, we also had the bound that $T_p \leq \frac{T_1}{p} + T_\infty$; Brent's theorem assumed we had greedy scheduling where the data was locally available.[2] We no longer have this assumption.

Just how large is the difference between RAM and hardisk?

- Random access lookup
  - Main memory on a piece of RAM can take 100 nanoseconds to pick up a piece of data
  - A hard disk[3] can take 10 million nanoseconds to fetch the same data, i.e. 100k$\times$ slower.
- Sequential access lookup
  - Reading one megabyte from RAM: 1/4 million nanoseconds
  - Reading one megabyte from disk: 30 million nanoseconds, i.e. 100x slower

The more data we are reading, the difference between sequential RAM and disk reads shrinks, but the order of magnitude difference is still there. We need to start dealing with this now, even for our algorithms on a single machine. This dramatic difference gives rise to a model of computation called streaming or online algorithms. The basic idea is that data comes at you in a stream, and you are allowed to maintain some amount of state (much smaller than the amount of data you consume) in RAM, and at termination, state outputs results.

9 Simple random sampling

On a single machine, if we want to sample an observation uniformly at random, it suffices to first generate a random number between 0 and $n-1$, the number of observations we have, and then pick an element from our data according to this index. If we know how much data is going to show up, we can just generate a random number and be done with it. Very often, however, we don't know how much data is coming at us; it comes from a mix of disk, network, etc, but we still have to sample one element.

When is this useful One situation this is useful: suppose you're reading from the Twitter firehouse. There are gigabytes of data coming in per second. You can't keep all this data around and *then* randomly sample from it.

Realize that since we only store values of r and k, our algorithm technically only uses a logarithmic amount of memory. Note that technically it's not a constant amount of memory, because k stores the length n of the stream and therefore we require $\log_2(n)$ bits to represent said number.

[2] Recall we proved Brent's theorem by assigning all computations on one level of our DAG to a bunch of processors. Notice that now processors don't have access to all the data, and so we may have to incur a shuffle cost. We don't yet have a nice analagous upper bound in the distributed computing framework.

[3] not a solid state disk

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

Algorithm 1: Sampling from a stream uniformly at random

Correctness We claim 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 prove the claim via induction. The base case is trivial, since 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 wish to show that r has equal probability of being any of the stream items seen so far, i.e. that $\Pr(r = s_i) = 1/n$ for all $i = 1, \ldots, n$, where s_i is the ith item of the stream. We just showed that r has value s_n with probability $1/n$, i.e. $\Pr(r = s_n) = 1/n$. What about $\Pr(r = s_i)$ 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 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.

What would happen if we implement quick-sort on a cluster? Recall how quicksort works. We started off with a pivot, selected uniformly at random. We then constructed L and R for items less than or greater than pivot p respectively, and then we return the concatenation of sorted L, pivot p, and sorted R.

Now, suppose we're in distributed computing world. Say we have $B = 5$ machines, where each machine has $1/5$ of the numbers that we need to sort. We first need to randomly pick a pivot p.

From each machine, we can pick out a sample, to get give samples. Based on a weighted index of how many elements are on each machine, we must choose one of these five elements at random, with weights depending on representative size of sample. OK, so we can pick out a pivot.

Now, constructing L and R. Is there even room to copy our data? We can't afford this. But even if we could, what does it mean to recursively call `quicksort` on each machine's L and R. Suppose even we could do this, then when the result is ready we must return the result in a sorted manner. But where do we even place the result? It can't fit on a single machine. We can't even make a recursive call if our input data on different machines. Recursion in general suffers from this problem in a distributed computing framework: it requires too much overhead incurred from preparing and returning function inputs and outputs.

9.1 Communication Protocols

Several communication patterns exist between machines:

- All to one
- One to all
- All to all

We introduce their communication patterns and the associated communication costs.

9.1.1 All to one communication with *driver* machine

Computation is distributed among multiple machines and the results are sent to a single *driver* machine, as shown in Fig.1. Assume all machines are directly connected to *driver* machine the bottleneck of this communication is the network interface of *driver* machine. Let p be the number of machines (excluding *driver*), L be the latency between each pair of machines and the network interface of *driver* machine has bandwidth B. Assume all machines send a message of size M to the *driver* and *driver*'s network interface is saturated by every single message. i.e. machines queue up to send messages one at a time.

Each single message sent has cost:

$$L + \frac{M}{B}$$

Thus the overal communication cost is:

$$p\left(L + \frac{M}{B}\right)$$

9.1.2 All to one communication as *Bittorent Aggregate*

Another algorithm for all to one communication is known as *Bittorent Aggregate*. As in one to all communication computation is distributed among multiple machines but the results are aggregated through the communication between each pair of machines. The aggregation pattern can be seen as a tree structure or as depicted in Fig.2. Assume we are aggregating results from p machines, the results can be aggregated to a single machine in $\log_2 p$ rounds. Let L be the latency and B be

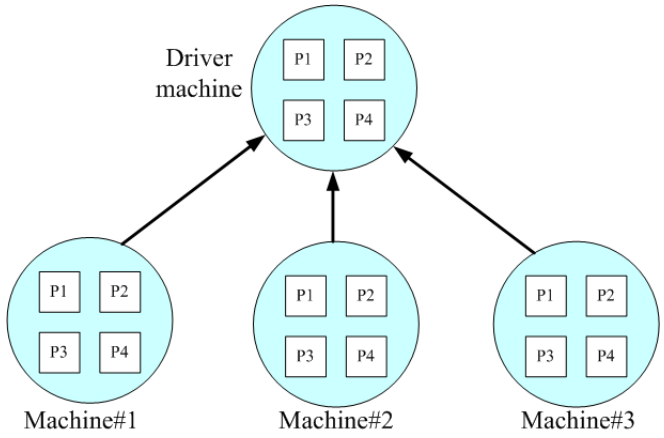


Figure 1: All to one communication with *driver* machine

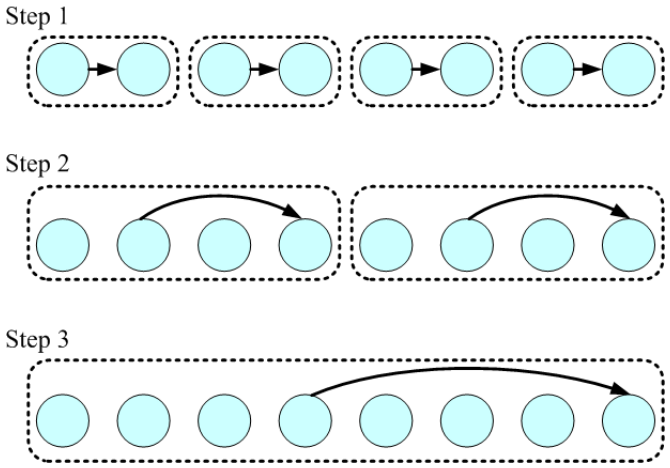


Figure 2: All to one communication with *Bittorent Aggregate*

the bandwidth between each pair of machines and in each aggregation round a message of size M is sent between machine pairs. The total communication cost is:

$$(\log_2 p)\left(L + \frac{M}{B}\right)$$

9.1.3 One to one communication

One to all communication has the same network configuration as all to one communication only with opposite data flow directions. The *driver* machine sends messages of size M to p other machines and *driver* machine's network interface is still the bottleneck of communication. Communication cost is the same as one to all communication with *driver* machine.

Also, we can borrow the concept of *Bittorent Aggregate* such that the message is relayed among machines in tree structure. Then the message can be spread among all machines within $O(\log n)$

rounds.

9.1.4 All to all communication

There are situations where all machines have to communicate with each other. Sorting is one example that requires all to all communication. Assume we have a large number of integers which is exceeds the storage of a single machine. The numbers are shuffled and distributed among multiple machines, each machine cannot determine the correct order of its numbers with communication with all other machines.

Some, but not all problems require all to all communication and it's valuable to find them. Relational database operations JOIN and GROUPBY are two other examples. In fact these two operations are implemented by sorting in many real world applications.