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

Lecture 9, 5/1/2017. Scribed by A. Santucci.

9 Network Topology

There are all kinds of ways that computers can be connected. Suppose each node a computer. The below is a sufficient network

```
x -- x -- x -- x
|              |
x -- x -- x -- x
```

Whats reasonable to assume about connectedness of network? Adding more connections may avoid bottlenecks, and we can make this a complete graph. This is not realistic, however. Instead, we sometimes use routers, which are dedicated pieces of hardware specifically designed to route packets of information to the relevant computers on the network. In this case, it's as though there is a direct and unfettered line of communication between any two computers which are tethered to the same router. This is often the case in our distributed computing set up.

```
   x
 / | \
 x x x
```

In PRAM, we learned how to apply general associative binary operators in logarithmic depth.

Recall bandwidth is the amount of bits per second that can be sent across a pipeline. In a datacenter, we assume everybody has the same network card, i.e. everybody is connected with 10 gigabit connections; this is the kind of cluster that we would spin up on Amazon AWS, for example.

10 Distributed Summation

Suppose each machine has part of the data in it.

The exact WRONG way to sum up numbers Each machine is a PRAM. We can go to town with our PRAM algorithm and concurrently sum all of the numbers entries stored locally in each machine. If there are p machines, why don't we just communicate them all to one master node and then aggregate the result?

Observe that a message of size m with L latency and B bandwidth requires $L + pm/B)$ time.[1] Here, m is a single integer, so this is no big deal. But realize the summation operator might not be operating on just numbers, but possibly vectors. In this case, m can be very large. If you're serious about distributed computing, p large as well.

What's wrong with this approach? If every machine tries to talk to the master, the node that gets the partial sums, then its network card will be the bottle-neck. We can only transfer data as fast as the bandwidth allows on said machines network card, B bytes per second. Meanwhile, the network cards on all other machines lay idle; realize they could be creating partial sums theirselves.

A better way to do summation in distributed environment - Tree aggregation The proper way is to use *bit-torrent* broadcasting. The idea is similar to what we observed in PRAM: pair off machines and have them communicate with each other.

```
x   x    x    x    x
```

So why not, on the first time step allow m_1, m_2 talk and in parallel m_3, m_4, etc. From each pair, we select exactly one machine to propogate the result to the next level in our tree. So by having all this bandwidth capacity in the network, we can use ideas from PRAM world to communicate within the network. When we have a lot of compute resources, we want to take advantage of all them. We can assume our network topology is a clique within a distributed cluster. So we should be using as much of the network as possible so as to avoid unecessary bottleneck.

Since we paired machines, there are of course $\log_2(p)$ rounds of communication before termination. Each round requires $L + m/B$, for $\log_2(p)\,(L + m/B)$ total shuffle cost.

Notice that we can crank up the base or branching factor to saturate bandwidth. I.e. there's no need to pair off computers; we can instruct one machine to recieve data from many machines. When we do this, in practice (*not* theoretically) this $\log p$ factor is effectively a constant.

This summation algorithm is all-to-one.[2]

Bit Torrent Broadcasting After all rounds of communication take place, only one machine has the sum. Very often, right after you're done with the summation, all the machines need to use this information! What we need now is a one-to-all broadcast. We use a technique known as bit-torrent broadcasting to accomplish this.

Fault-Tolerance: where can our system fail? Much of what we have done so far assumes each machine won't die. But nothing could be further from the truth. There is a listing of things

[1]Note that depending on the network, this could be $p(L + m/B)$; the latencies go away if there is a nice queueing structure that is available in the network that may not typically be avilable. In any case, L is on the order of 1-2 milliseconds for a local network, or a couple hundred milliseconds across the world. We hit fundamental limitations because the speed of light is a physical constant, and we can't transfer physical information or data any faster than that.

[2]Don't confuse network topology with communication patterns. In this class, we always assume all the machines are fully connected i.e. our network topology is a complete graph.

that go wrong in distributed computing systems. We've totally ignored failures. Problems with traditional network programming:

1. Machine failures. If you have 1,000 machines, you can bet than several will die on any given day. This could be something like a hard-disk failure. How do we handle faults? Typically, we might write a temporary snapshot of work in progress to a distributed file-system, in at least several places, such that it can be retrieved if need be.

2. Disk and memory interface has been largely ignored. Last class we saw a streaming algorithm for randomly sampling an observation. We need to deal with the fact that memory access is orders of magnitude faster than hard-disk access.

3. Communication costs. Even analyzing our simple distributed summation was non-trivial: we must distinguish between all-to-one and one-to-all. We will later analyze all-to-all communication patterns when we tackle distributed sorting.

4. Multiple programs must be written for fault tolerance: one program for failures, one for failures, one for scheduling and putting code where it needs to be, and also data locality.

Out of all these problems, the last is the biggest nuissance. This listing describes why distributed computing is difficult. Instead of solving each one of these, we commit to SQL like programming languages and abstract away our computations into a new programming language that then solves these problems for us.[3] The first example was map-reduce. At a basic level, map-reduce just implements sorting, itself an all-to-all communication protocol. Sorting is powerful, and can be used as a building block for other algorithms. There are a cottage industry of people who build algorithms on top of just map-reduce. At some point, we decided just sorting wasn't enough. So, we developed tools like Hive and PIG which take regular-looking SQL and create map-reduce jobs from it. Another more recent example is SPARK, which combines SQL with functional programming and better memory management.

A note on last lecture

Going back to last class's quicksort failure analogy. Last class, we talked about why quicksort fails. Sorting is so fundamental to all-to-all communication patterns. We had something like

Quicksort(A) select a pivot p uniform at random L ¡- all items less than p R ¡- all items greater than p return (QSort(L), p, QSort(R)) END

Machines numbered 0, 1, 2, ... We want the first k items to be on machine 1, and the next k items to be on machine 2, etc. I.e. we need a sorting within *and* across machines.

The assumptions of quicksort in a distributed environment are the following.

- Data are split between machines.

[3] Much like assembly language is a hassle to deal with, where instead we've abstracted away and written for-loops and if-else branching. There are compilers that will take our SQL queries and make them fault-tolerant in a distributed computing world.

- Result needs to be output to all machines.

So these two assumptions must be true for Quicksort algorithm as well. What does this mean in terms of our recursive calls? Well, when ‘Quicksort(L)‘ returns, it may have moved around all the data from one machine to another. In the worst case, it may have to move all items to land on the correct machine. In our recursion, we can have lots of calls to ‘QSort‘. In reality, at each level in the recursion, the entire data may be shuffled. In worst case, with each recursion level, all data gets shuffled. This is not acceptable; we can do much better.