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

Lecture 12, 5/7/2020. Scribed by Andreas Santucci, Edited by Robin Brown.

12 Communication Networks

We will model the communication structure of a distributed system as a network of processors of connected by communication links. Specifically, in a network of n processors can be modeled as a graph $G = (V, E)$ where $V = [n]$ represents the processors and there is an edge $(i, j) \in E$ if processor i can send a message to processor j. Each processor has its own local memory and computational capabilities, and shares intermediate results with other processors through groups of bits (called *packets* sent over the communication links. The communication graph G may be directed or undirected depending on whether the links are uni-directional or bi-directional. Moreover, each link is characterized by its **latency** and **bandwidth**. Latency is defined as the time it takes to for data to travel from one point to another. Bandwidth is defines as the rate at which data can be transferred over a link during a fixed amount of time. Ultimately, these limits on communication will fundamentally limit the efficiency of our distributed algorithms.

There are a number of causes of delay in communications:

1. *Communication Processing Time* is defined as the time needed to prepare a message for transmission. Before transmitting, the information needs to be assembled into packets with addressing and control information appended to each packet, the link on which to transmit each packet needs to be selected, and the packets need moved to the appropriate buffers.

2. *Queueing Time* is defined as the time a packet waits in the queue before it is transmitted. This can happen for a number of reasons: for example, the link is being used to send other packets, or the allocation of a link to several contending packages is being decided, or the transmission is delayed to ensure the appropriate resources i.e., buffer space is available at the destination.

3. *Transmission Time* is defined as the time required to transmit all of the bits in the packet.

4. *Propagation Time* is defined as the time between the transmission of the last bit of a packet and depends on the physical distance that data must travel and the medium it travels through. For example, in a fiber optic cable, data travels at the speed of light.

We won't deal with the specifics of networking, but instead will lump together our communication costs in terms of latency and bandwidth.

13 Cluster Computing, Broadcast Networks, and Communication Patterns

Commodity cluster computing refers to using a large number of low-cost, low-performance commodity computers, connected via ethernet, working in parallel instead of using fewer high-performance and high-cost computers. The ethernet link between processors means that every processor can communicate with every other processor in the cluster. In other words, the communication network is a complete graph. This doesn't mean that all processors should always be passing messages to each other, though, as this could easily overload the bandwidth of the communication links. Consequently, communication will need to be limited to ensure that the bandwidth constraints are respected. One could theoretically design communication patterns that are tailored to the problem at hand, however, the overhead of fault tolerance makes this unreasonable. Multiple programs must be written for fault tolerance: one program for failures, one for scheduling and putting code where it needs to be, and another for data locality. MapReduce and Spark will automatically handle this overhead, however, this comes at the cost of full control of the communication patterns, which are typically restricted to one of three patterns: (1) All to One, (2) One to All, and (3) All to All. We will now analyze the cost of communication for each of these patterns.

13.0.1 All to one communication with a *driver* machine

Consider a scenario where computation is distributed among multiple machines and the results are sent to a single *driver* machine, as shown in Fig.1. If all machines are directly connected to the *driver* machine, the bottleneck of this communication is the network interface of the *driver* machine. Let p be the number of machines (excluding the *driver*), L be the latency between each pair of machines (specifically, L is the time it takes for the first message from a machine to arrive at the driver machine), and B be the bandwidth of the network interface of the *driver* machine. If all machines send a message of size M to the *driver* and *driver*'s network interface is saturated by every single message (meaning that the driver can only take in messages from other machines one at a time), then each single message sent takes time proportional to

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

Thus the overall communication time scales with:

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

13.0.2 All to one communication as *Bittorent Aggregate*

Another algorithm for all to one communication is known as *Bittorent Aggregate*. Just as before, we assume that each machine carries out a portion of the computation, and results need to be aggregated at one machine. Instead of performing the aggregation at once, it is done incrementally aggregating the results held by pairs of machines. The aggregation pattern can be seen as a tree

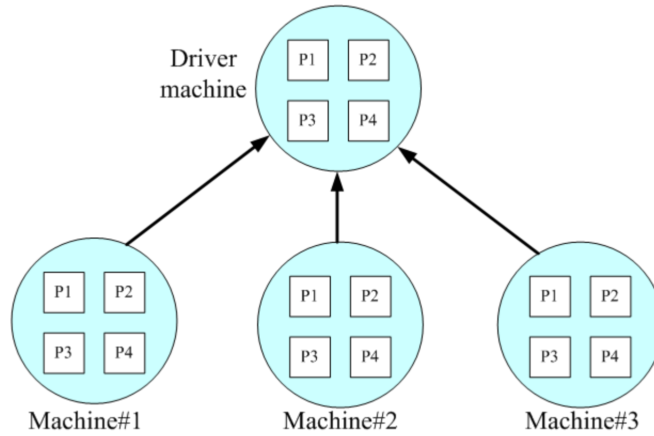

Figure 1: All to one communication with *driver* machine

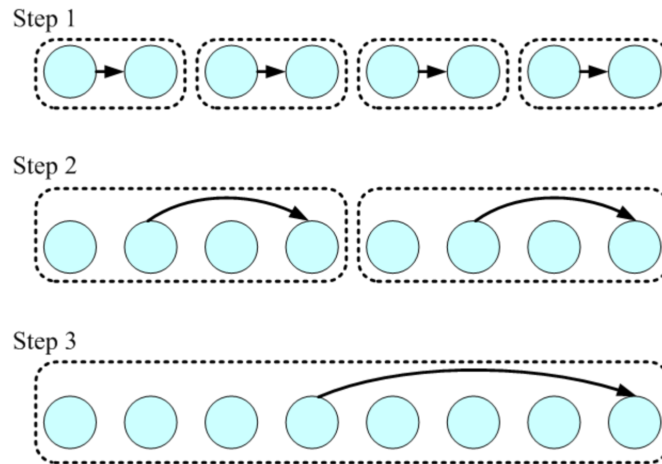

Figure 2: All to one communication with *Bittorent Aggregate*

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 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 time is:

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

13.0.3 One to All communication

We can think of one to all communication as all to one communication with data flow in the opposite direction. Suppose the *driver* machine needs to sends messages of size M to p other machines. The *driver* machine's network interface is still the bottleneck of communication. Following the an

analogous computation, communication cost is the same as one to all communication with *driver* machine.

Similarly, we can use the concept of *Bittorent Aggregate* for one to all communication as well. In this case, the message is relayed among machines in a tree structure. Then the message can be spread among all machines within $O(\log p)$ rounds.

13.0.4 All to all communication

While we have previously stated that all to all communication can easily saturate bandwidth constraints, there are some operations where all machines have to communicate with each other.

For example, sorting requires all to all communication. Suppose we are trying to sort a large set of integers which exceeds the storage of a single machine. If the numbers are shuffled and distributed among multiple machines, without communication with the other machines, no individual machine can know which number are stored on the other machines. Therefore, we require all to all communication for each machine to determine how the numbers relate to the numbers stored in other machines.

Other examples of problems that require all to all communication are JOIN and GROUPBY. In fact, sorting is often a subroutine in implementing these two operations.

14 Optimization, scaling, and gradient descent in Spark

We will now consider two different optimization algorithms: gradient descent, and stochastic gradient descent. In this class, we'll implement gradient descent in Spark, and then see why SGD not suitable for Spark. Then we'll learn how to implementSGD regardless of the data-flow paradigm of the language we're computing in. For the remainder of this section, we will assume we have a separable objective functions, of the form

$$\min_x \sum_{i=1}^{n} F_i(x)$$

where $w \in \mathbb{R}^d$ and F_i is the loss-function applied to a training point i. We won't make any assumption about the specific form of each F_i.

The main question we are interested in is how our algorithm scales both in n, the number of data points we have, and d the dimension of our model. In the first part of the lecture, we'll assume n is too large to fit on a single machine but that d numbers can fit on a single machine's RAM. Recall that gradient descent starts with a random initial vector, x_0, say initialized to all zeros. Then at each iteration $x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$, where $\nabla F_i(\cdot)$ is a vector itself.

14.0.1 Implementing Gradient Descent in Spark

The following block of code demonstrates an implementation of gradient descent in Spark:

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
For (i <- 1 to numIterations):
  val gradient = points.map(p => ∇F_p(w)) .reduce(_+_)
  w -= alpha * gradient
```

The first line reads in a text-file of training points and their associated labels. The operation, `parsePoint` is a closure which takes as argument a single like of the text-file of training points and their associated labels, and outputs a clean d-dimensional x_i vector and a single label y_i. After parsing the files, we cache the RDD with the training points. The action `cache` tells Spark that we're going to use a particular RDD frequently, so each machine should try to hold that its part of the RDD in memory as much as possible rather than flushing data to disk. Since we need to access this RDD every iteration of gradient descent, we will want to keep it in memory. Recall that `cache` is an action (as opposed to a transformation), so Spark actually kicks off computation and reads in the text file, applying the `parsePoint` closure to each part of the distributed file.[1] The next line initializes the model parameters to a d-dimensional vector of all zeros. Finally, each iteration of the for loop implements an iteration of gradient descent. The closure

`points.map(p => ∇F_p(w))`

maps each of the training points to their associated gradients of the loss function. Specifically, $\nabla F_p(w)$ is the gradient of F_p evaluated at point w. For example, if $F_i(w) = (w^T x_i - y_i)^2$, then ∇F_i may be calculated symbolically, and we may plug this formula into $\nabla F_p(\cdot)$. Then we calculate the full gradient as the sum of the individual gradients—we use reduce for this. Recall that Spark uses lazy evaluations, the the computation is not kicked off until this reduce step. Under the hood, Spark takes into account communication cost between machines, scheduling, and executing operations locally in a PRAM model.

14.0.2 Broadcasting in Spark

Recall that by default, when a `map` happens, anything that is needed for the `map` to happen is shipped out to the workers. So in the line

`points.map(p => ∇F_p(w)).reduce(_+_)`

since the function that maps each training point to it gradient depends on w, a copy of w will be sent to **every** CPU of **every** machine. This happens, because by default, we assume w is being modified. Consequently, we must store w separately to avoid any concurrent write issues. However, when we calculate each gradient step, w is not being modified so there is no chance of a

[1] Suppose we have a 1 terabyte dataset, and we have a small amount of memory, say 10 gigabytes. Suppose we wish to process the large dataset. We can read directly off the large terabyte slowly, or we can read directly off the ram for faster access. The question becomes which part of the data set to store in ram since we can't fit the entire object in memory. We commonly employ Least Recently Used caching, which evicts items from ram according to which piece of data was least recently used. This is a simple heuristic, but works in practice.

concurrent write. We are wasting storage by not storing w in shared memory. To get around this, we can broadcast `w` so that it only gets sent to each machine a single time. To do this, we define `w` as a broadcast variable. The following block of code illustrates gradient descent with the model parameters as a broadcast variable.

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
var w_br = sc.broadcast(w)
For (i <- 1 to numIterations):
  val gradient = points.map(p => ∇F_p(w_br.value)) .reduce(_+_)
  w -= alpha * gradient
  w_br = sc.broadcast(w)
End
```

When we initialize w, we first declare it as a broadcast variable. In each of the iterations of gradient descent, we need to refer to the *value* of w. Finally, we re-broadcast w after it gets updated. In addition to being more storage efficient on each machine, i.e. storing a single copy of the data object instead of multiple, we also take advantage of bit-torrent broadcasting when we use Spark's broadcasting.

14.1 Analysis

Recall, gradient descent requires $\mathcal{O}(\log \frac{1}{\epsilon})$ iterations to achieve an ϵ-optimal solution if f is L-smooth, and μ-strongly convex. While the computation of the gradients can be distributed across the cluster of machines, the iterations of gradient descent are inherently serial. At each iteration, Spark implements a synchronization barrier to ensure that w is fully updated before the workers begin computing the next iteration. Just as in the parallel case, if there are stragglers, the other workers are idle waiting for the next iteration. The depth of the computation of typically is the bottleneck rather than the gradient updates.

Communication time The map is embarrassingly parallel, and requires no communication. The broadcast requires one to all communication, and the reduce requires all-to-one communication. So, per iteration, our total communication time scales with

$$2 \log_2(p)(L + \frac{m}{B})$$

[2]

[2] Technically, when we call `points.map()`, we serialize code and send it to each machine. One reason we chose Scala to implement Spark is that it's easy to serialize code. We remark that there is a function called `allReduce` which merges the concept of a reduce and broadcast. After a reduce, the result of the reduce ends up on the driver machine. After an `allReduce` the result of the `reduce` gets sent to all machines via a broadcast; the broadcast happens simultaneously with the regular reduce.

References

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

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