Name: Harry Lang

JHED: hlang8

Parallel k-means Clustering

Summary:

Consider the following two-phase clustering algorithm: k-means++ selects k centers, and these are used as seeds for Lloyd's algorithm. The goal is to parallelize k-means++ and Lloyd's algorithm for the distributed setting.

We chose to code MPI because it models the distributed setting well. However, most open-source scalable frameworks today (i.e. Spark, H2O) should be coded in Java or Scala. For a real implementation we use Java or Scala, and expect the algorithm to handle large datasets well. The runtime is O(dnkL/P), and the communication complexity is O(dkP) (see the next section for definition of these letters). Our scheme is a data decomposition where each of the P nodes have an approximately equal-sized portion of the entire data.

See "k-means++: The Advantages of Careful Seeding" by David Arthur and Sergei Vassilvitskii.

Definitions:

n = the number of datapoints

x = a datapoint (a weighted vector in R^d)

d = dimension of the data

L = number of Lloyd iterations

P = number of worker nodes

X = the set of datapoints

w(x) = the weight of the point x

k = the number of centroids to find

C = the current set of centroids

D(x) = the distance from x to the nearest centroid in C

k-means clustering seeks to find a set C of size k such that $\sum_x w(x) D(x)^2$ is minimized.

k-means++ (k, X)

k-means++ is an O(log n) approximation in expectation.

```
Let C be an initially empty set.  Add a point to C, adding x with relative probability w(x).
while (|C| < k)
    Add a point to C, adding x with relative probability $w(x)*D(x)^2$
return C
```

Parallel k-means++ (k, X)

For each iteration of the loop, let every node reservoir sample a single vector through their local data, and send a message to the master with the location of the selected data point and the sum of the relative probabilities* of all the datapoints it represents, and then let the master reservoir sample through these results. The selected point becomes the new center.

* *note that the relative probability is different for the first iteration (the weight of the point) verses the other k-1 iterations (the weight times the distance squared).*

On each node, maintain an array to store D(x) for each point. D(x) is calculated in O(k) time, since it is the minimum over O(k) centroids. However, by storing the previous value of D(x), we can update the minimum to consider a new centroid in O(1) time.

The total time complexity is O(dnk/P), which is optimal.

For each loop, each node receives and sends a single weighted vector. Note that this is O(d), which is constant in n and k. The total network communication per iteration is thus O(dP), which is independent of n. This leads to O(dkP) communication for the entire k-means++ procedure.

Lloyd-type step

Define C(i) to be all the data points which are nearer to the centroid i than to any other centroid. Let c(i) be the center of mass of C(i). Let the set of c(i) become the new centroids.

Parallel Lloyd-type step

Send the set of centroids to each node. Let each node calculate the centers of mass of each cluster (by streaming through and maintaining a weighted vector whose position is the center of mass of the points processed so far and whose weight is the total weight of the points processed so far) and send these centers to the master. Let the master, for each i, take the centers of mass of the P different c(i), and make this the new i^th centroid.

The streaming can be achieved as follows:

```
Let c be a weighted vector of weight 0 (and position is irrelevant)
For each point x:
    c.pos = (c.pos*c.weight + x.pos*x.weight)/(c.weight + x.weight)
    c.weight = c.weight + x.weight
```

This operation is iterative (i.e. it satisfies the property required by a combiner in a MapReduce program), which is why this algorithm works. (i.e. the center of mass of X is the same as the center of mass of the center of masses of a partition of X).

The time complexity is O(dnk/P), which is optimal.

The network communication is O(dk) per node, resulting in O(dkP) total network communication.

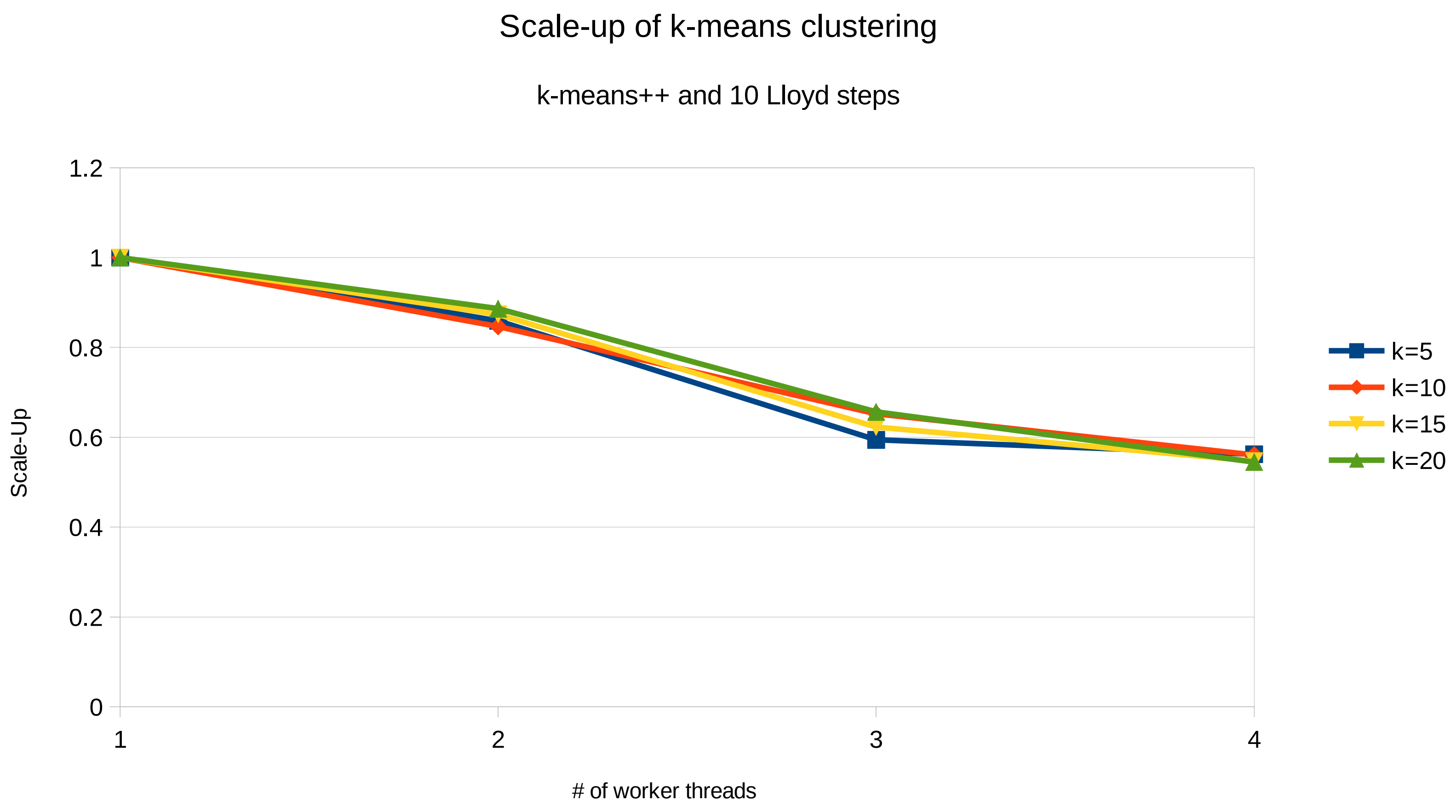
As an example, consider $k = 10^2$, $n = 10^{20}$, $d = 3$, $P = 80$. The total network communication from beginning to end of k-means++ is 125KB. Overall, the communication is trivial! This is fairly remarkable. The network communication for k-means++ is exactly the same as the Lloyd-type steps.

	Time	Network
k-means++	O(dnk/P)	O(dkP)
Lloyd-type step	O(dnk/P)	O(dkP)

Experimental Runtimes:

We show the average of 20 trials at $N = 2P*10^4$, $d = 3$, and 10 Lloyd steps on a single machine with an Intel i7 860 @ 2.8GHz. The interesting notion is scale-up, as the idea is to scale this to the level where N = billions and k = millions. This, for example, would be a useful clustering for research in computational physics.

Runtime (microseconds)		Scale-Up (= P = N / 20,000)			
		1	2	3	4
k	5	65441	76154	110126	116407
	10	97824	115521	149991	174602
	15	128651	147276	206494	235228
	20	156814	176918	238862	287855



Discussion about non-constant scale-up:

I actually expected the scale-up to be constant. I am not sure how to explain it, except that my implementation could be improved. The only physical limitation that comes to mind is that I/O was inherently serialized, whereas it would be parallel on a true distributed system. I explain this in the next paragraph.

The data is loaded from disk among 4 threads, but the hardware only supports sequential reading, so the read-skip-read-skip pattern adds extra latency for each additional process. This would not be a problem on a true distributed system.

About when k is very large:

Very large k, this is still feasible since, representing vectors as floats, 10^7 centroids in 3 dimensions would require P2P communication of about 150MB per Lloyd iteration. On a slow 1 Mbps connection, it would take just 5 minutes for I/O during a Lloyd step.

Note: 150MB = 10^7 centroid * ((3 + 1) centroid) * (4 bytes per float) / (2^{20} B / MB)

Improvement for large k:

The given runtime of O(dnkL/P) only accounts for computation time. When k*P is large, the communication time may become the dominating factor. To greatly reduce this, we could reduce the messages hierarchically instead of having each node communicate directly with the master. For example, if we build a binary tree with the master at the root (and where each node is a worker), we can send the data up the tree, combining at each node. While the total network communication is the same, the maximum communication of any single node is reduced from O(dkP) to O(dk).

This can be achieved by calling MPI_Reduce, where we would write a custom MPI_Op (which would simply do the master's computation) and a custom MPI_Datatype (which would be an array of k weighted vectors). However, it did not make sense to go to the trouble of implementing an MPI_Reduce operation for our small number of workers. When scaling to larger sizes, this would be useful.