CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 13, 5/9/2016. Scribed by Alfredo Láinez, Luke de Oliveira.

13 Optimization

We will first consider optimization problems in Spark. In particular, we consider two large classes of optimization problems: **convex problems**, where the objective is the minimization of a convex function, and **spectral problems**, such as SVD. In this lecture we will focus on convex problems, since they commonly appear in areas relevant to distributed computations such as machine learning.

13.1 Gradient Descent

We consider the general additive objective function that we saw in previous lectures:

$$\mathcal{L}(w; x, y) = \sum_{i=1}^{n} F(w; x_i, y_i)$$

where we have a set of data points $\{x_i\}_{i=1}^n$, where $x_i \in \mathbb{R}^d$, and we have associated labels $\{y_i\}_{i=1}^n$. In addition, we have a set of weights that we seek to optimize over – the parameters of our model. This function is usually associated with the loss value of the problem in hand that we want to minimize.

In order to minimize this function we perform gradient descent, a first-order method in which we take small steps in the direction of maximum decrease in functional value (the gradient).

Let $g(w; x_i, y_i) = \nabla F(w; x_i, y_i)|_w$, the gradient of our objective. Gradient descent consists of iteratively applying the following update of the parameters (weights) of the problem.

$$w \leftarrow w - \alpha * \sum_{i=1}^{n} g(w; x_i, y_i)$$

13.1.1 Scaling

We briefly consider the ways in which our optimization problem can scale.

- $\uparrow n$, when we increase the number of training points.
- $\uparrow d$, when our parameter space becomes very high dimensional
- When we have a hard problem and need many iterations and/or hyperparameter tuning to find a suitable solution.

13.1.2 Logistic Regression

Now, we turn to solving a minimization problem in a parallel way with Spark. Consider the following code, which uses the example of Logistic Regression.

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
        val gradient = points.map{ p =>
                (1 / (1 + exp(-p.y * w.dot(p.x)))) * p.y * p.x
        }.reduce(_ + _)
        w -= alpha * gradient
}
```

In the first line, we are computing an RDD consisting of points `(p.x, p.y)` where `p.x` is a vector of features and p.y is the label corresponding to that data point. To do that, we call the function `textFile()` from the Spark context. The spark context, represented in this case as *spark*, is the main entry point for Spark functionality. `textFile()` reads a file (usually from HDFS) and distributes its content among the cluster. Then, we apply the `map` function, which as we know is a transformation that creates a new RDD by applying a function to every element in our RDD. In this case, the function applied is a closure defined by `parsePoint`. The closure `parsePoint` is a user-defined operation that in this case constructs the elements of the `point` struct, and fills the new elements in the resultant RDD.

The second line defines a vector of zeroes. In the third line, we open our for loop to govern the number of iterations we perform gradient descent to optimize our model. Note that this number of iterations is not known a priori, and is an additional hyperparameter to tune.

At a macro scale, the inside of this loop performs the gradient descent update that we saw before. Specifically, we are mapping (using the `map()` transform) every point to its respective gradient. Next, we sum all gradients across points using a reduce action with the + operator, and finally, we update our parameter vector w. Note that the resulting dimension post-reduction is a d dimensional vector.

Some points are worth mentioning about both this code and the way Spark works.

- Sparks utilizes `lazy evaluation`. Jobs are not performed until absolutely necessary, usually when an action is performed. In this way, Spark registers the stages of computation that happen in the program and only applies them when necessary. In our code, the reduce function is what triggers the computation. Up to that moment, the data would not be even loaded.

- Note the `cache()` function applied at the end of the first line. This is telling Spark that we plan to use the RDD `points` frequently , and so all machines should try to keep their part of it in RAM as much as possible in order to speed up things.

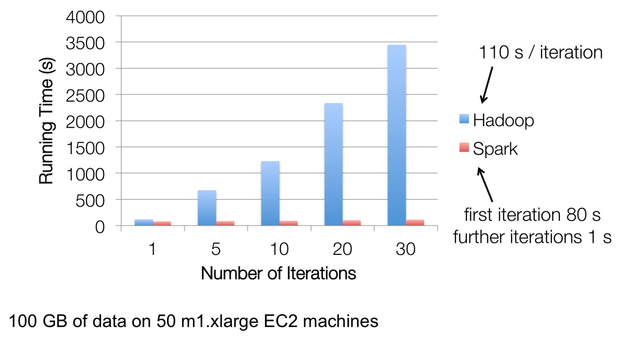
- RDDs are immutable and so every transformation is creating a new RDD. Spark registers every RDD created such that it is able to reconstruct the transformations performed to the data in the case some node fails.
- We are applying the `reduce` function to sum the gradients of every data point. Note that `(_ + _)` is Scala syntactic sugar for the sum of elements. Internally, Spark is making first every machine to locally perform the sum of the gradients that they hold. Secondly, $\log_2(m)$ rounds of network computations are performed to obtain the final result, where m is the number of machines in the cluster.
- The vector w is originally a local vector in the driver machine. Since Spark serializes the code and sends it to every machine in the cluster, w is being sent as well, as it is part of the computation of the gradients. Note that this has to be taken into account in terms of communications costs for the case that w was very big.
- Spark uses `speculative execution`. If a machine is lagging, Spark will automatically set up another machine so that we have the option of finishing earlier, and then will utilize the result of the faster machine.

Also note that as long as the gradients fit in memory this code can be easily generalized. Examples of problems that could be treated in a similar way are

- Unconstrained optimization
- Smooth or non-smooth optimization
- LBFGS, Conjugate Gradient, Accelerated Gradient methods, etc.

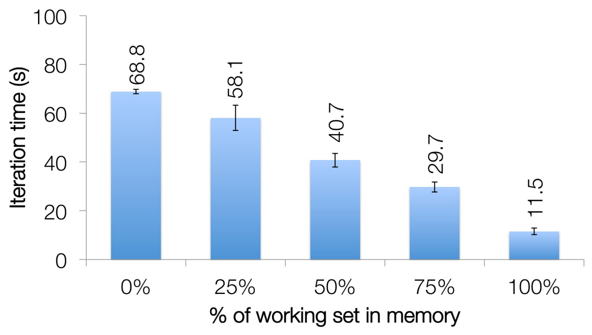
Finally, let's take a look at some performance-wise aspects of the problem. Consider Figure 1.

Figure 1: Logistic Regression, comparison of Map Reduce to Spark



We can see how an increasing number of iterations yields big performance improvements of Spark over simple MapReduce on Hadoop. This is due to the fact that MapReduce saves and loads data from disk after every iteration (which corresponds to a map and reduce process), while Spark takes advantage of maintaining most of the data in memory. Hence, Spark is very fast for iterative

Figure 2: RAM working set percentage vs. performance



algorithms, as opposed to MapReduce. Also, in Figure 2 we can appreciate how keeping more data in memory means performance gains per iteration.

14 Demo in class

In class we saw some sample commands in the Databricks Spark environment. We reproduce some interesting takeaways from that.

We will try to create a vector with 100 random elements in a parallel way. For that, first we create a distributed vector of 100 elements distributed over 10 partitions.

```
scala> val mypoints = sc.parallelize(1 to 100, 10)
```

Note that `sc` is the Spark context mentioned beforehand. Once we have the distributed vector, we map each element to a random number.

```
scala> val randpoints = mypoints.map(x => Math.random)
```

Note that until this moment no computation has been performed, other than Spark registering some transformations over "fake" data. We can then apply some actions on our data:

```
scala> randpoints.reduce(_ + _)  // computes the sum
scala> randpoints.first          // gets first number
scala> randpoints.sum            // computes the sum
```

15 Stages, Scheduler, Cluster Manager and Sorting

In Figure 3 we can see an overview of how RDDs and code are translated into real computations. First, Spark computes a DAG of operations based on the transformations and actions applied to the different RDDs. By doing this, Spark can effectively distinguish `stages` of tasks that can be parallelized and are dependent of other stages. Consider for instance the example stages in Figure 4 where a `groupBy` operation is performed over the RDD A, while a map and a filter are successively applied to C. This can be done in two parallel stages, while there is a final stage to join both results

Figure 3: RDD Data Flow

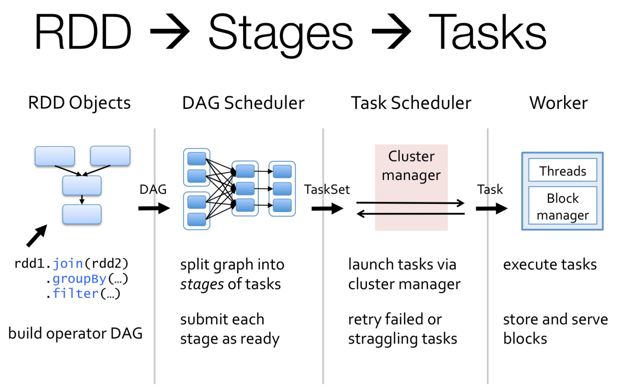
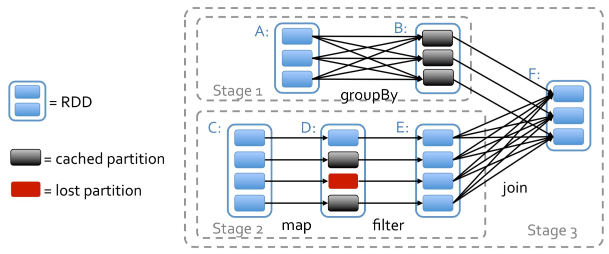


Figure 4: Spark Computation Stages



into F. Note also that Spark has the ability to easily recover from failure by recomputing failing stages.

Continuing with Figure 3, once Spark has knowledge of the different stages, the task scheduler kicks in by sending the different tasks to machines, trying to minimize both the machines makespan and the movement of data around the cluster (hence, the scheduler tries to send tasks ”close” to where the data is). As we already saw in a previous lecture in Figure 5, the complexity of our algorithm will be heavily impacted by the communication between machines. We also note that each worker has a small web server with data indexed in a way that it can be readily sent to other workers as they need it.

Scheduling is related to the concept of the cluster manager, as seen in Figure 6. The cluster manager is an interface between Spark and resources allocation. Spark can use well-established managers such as YARN or Mesos, although it can also be deployed in a simple way via the Spark Standalone Mode.

Finally, we note that shuffle operations, such as `groupByKey`, `sortByKey` and `reduceByKey`, are internally implemented with sorting. Spark utilizes Distributed Timsort for this. Timsort is an algorithm designed to take advantage of already existing partial orderings. In the next lecture we will analyze how Spark effectively parallelizes this algorithm.

Figure 5: Communication Patterns

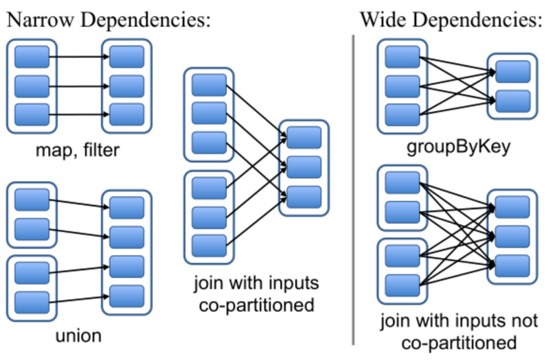
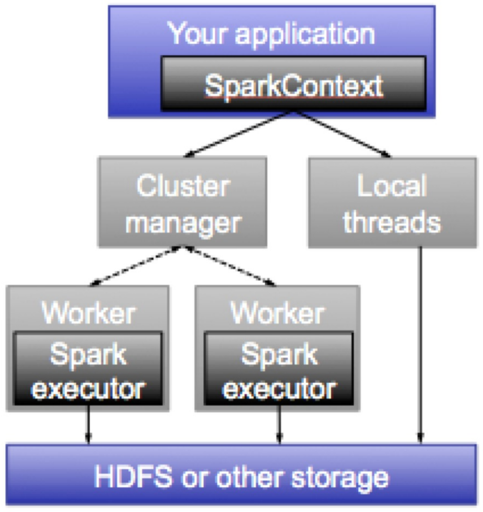

Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned

Figure 6: Cluster Management


Your application
SparkContext
Cluster manager
Local threads
Worker
Spark executor
Worker
Spark executor
HDFS or other storage