Spark - Cluster Computing with Working Sets

Prakhar Srivastav (ps2894)

Summary

Spark is a new cluster computing framework developed at the AMPlab, UC Berkeley. There's one class of specific applications that Spark focuses on - those that reuse a working set of data across multiple parallel operations. To address these kinds of applications in a scalable and fault-tolerant way, Spark introduces the concept of resilient distributed datasets (RDDs) which acts as a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

Need

Most cluster computing systems such as MapReduce work by providing the user a different programming model. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. While this data flow model is useful for a large class of applications, there are a certain kinds of problems for which this model provides an extremely inefficient solution. One such class of applications are those that **reuse a working set of data** across multiple parallel operations. Examples of such jobs include machine learning applications or interactive analytics.

The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.

Programming Model

Spark provides two main abstractions for parallel programming: resilient distributed datasets and parallel operations on these datasets.

RDDs A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. There are four primary ways in which Spark lets programmers create RDDs

1. From a file in a shared file system
2. By *parallelizing* a Scala collection in the driver program
3. By *transforming* an existing RDD using one of the parallel operations or any other `map` functions.
4. By changing the persistence of an existing RDD. In other words, instead of keeping the RDD lazy, either *caching* it for later re-use or *saving* it by writing it to the file system.

Parallel Operations Parallel operations are operations are applied on the RDDs and are invoked by passing a function. Spark supports the following parallel operations -

- `reduce`: Combines dataset elements using an associative function to produce a result at the driver program.
- `collect`: Sends all elements of the dataset to the driver program.
- `foreach`: Passes each element through a user provided function.

Shared Variables When Spark runs a function on a worker node, the variables that are visible to the function (from the outer scope) are copied to the worker. This can result in useless copying of variables which can be slow and consume network bandwidth. To address this, Spark lets programmers create two restricted types of shared variables to support two simple but common usage patterns:

- *Broadcast variables*: An object that wraps the value and ensures that it is only copied to each worker once. This is useful when a large read-only piece of data (e.g., a lookup table) is used in multiple parallel operations. Hence it is preferable to distribute it to the workers only once instead of packaging it with every closure.

- *Accumulators*: These are the variables that workers can only "add" to using an associative operation, and that only the driver can read.

Implementation

Spark is designed to run alongside existing cluster computing frameworks, such as Mesos ports of Hadoop and MPI, and share data with them. This is possible due to the fact that Spark is built on top Mesos, which is a cluster computing system that lets multiple parallel applications share a cluster in a fine-grained manner.

As mentioned before, the core of the spark implementation is RDDs which are stored as a chain of objects capturing the lineage of sequence of operations. When a parallel operation is invoked on a dataset, Spark creates a task to process each partition of the dataset and sends these tasks to worker nodes. Spark uses the technique of *delay scheduling* to send a task to one of its preferred location.

Shipping tasks to workers requires shipping closures to them—both the closures used to define a distributed dataset, and closures passed to operations such as reduce. This is achieved by serializing the Scala closure (which in turn is just a Java object). Lastly, the two types of shared variables in Spark, broadcast variables and accumulators, are implemented using classes with custom serialization formats.

Conclusion

Spark provides three simple data abstractions for programming clusters: resilient distributed datasets (RDDs), and two restricted types of shared variables: broadcast variables and accumulators. While these abstractions are limited, the paper demonstrates that these are are powerful enough to express several applications that pose challenges for existing cluster computing frameworks, including iterative and interactive computations.