COMPSCI 589
Lecture 14: Data Parallel Programming Abstractions in Spark

Benjamin M. Marlin

College of Information and Computer Sciences
University of Massachusetts Amherst

Slides by Benjamin M. Marlin (marlin@cs.umass.edu).
Created with support from National Science Foundation Award# IIS-1350522.

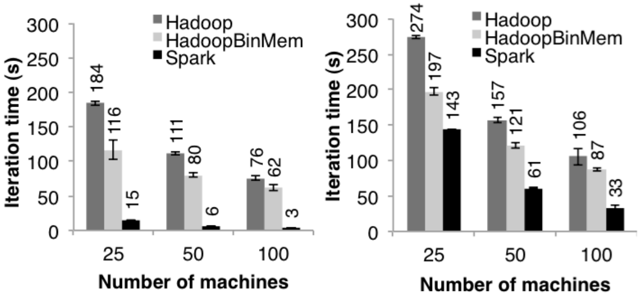
Outline

Apache Spark

- Apache Spark is a parallel and distributed programming framework that adds additional parallel abstractions and allows for distributed in-memory caching as well as distributed on-disk data access. This makes it much faster than MapReduce for ML tasks.

Apache Spark

- Apache Spark is a parallel and distributed programming framework that adds additional parallel abstractions and allows for distributed in-memory caching as well as distributed on-disk data access. This makes it much faster than MapReduce for ML tasks.



(a) Logistic Regression (b) K-Means

Distributed Data Abstraction

- Spark's primary abstraction is a data structure called a *resilient distributed data set* or RDD.

Distributed Data Abstraction

- Spark's primary abstraction is a data structure called a *resilient distributed data set* or RDD.
- An RDD is a read-only partitioned collection of objects that is stored across one or more nodes in a Spark cluster.

Distributed Data Abstraction

- Spark's primary abstraction is a data structure called a *resilient distributed data set* or RDD.
- An RDD is a read-only partitioned collection of objects that is stored across one or more nodes in a Spark cluster.
- RDDs are created by applying a sequence of *transformations* to data. Results are obtained by applying *actions* to RDDs.

Distributed Data Abstraction

- Spark's primary abstraction is a data structure called a *resilient distributed data set* or RDD.
- An RDD is a read-only partitioned collection of objects that is stored across one or more nodes in a Spark cluster.
- RDDs are created by applying a sequence of *transformations* to data. Results are obtained by applying *actions* to RDDs.
- The Spark system keeps track of the transformations used to create each partition of an RDD and can dynamically re-generate lost partitions on other cluster nodes. This makes it very fault-tolerant.

Creating RDDs from Data

Creating RDDs from Data

- An RDD is initially created from a root Spark context object.

Creating RDDs from Data

- An RDD is initially created from a root Spark context object.
- Supported methods include $textFile(path)$ to create an RDD from a text file (or a directory of files), and $parallelize(data)$ to partition an existing collection.

Creating RDDs from Data

- An RDD is initially created from a root Spark context object.
- Supported methods include $textFile(path)$ to create an RDD from a text file (or a directory of files), and $parallelize(data)$ to partition an existing collection.
- Spark can run over a regular file system or the Hadoop file system (HDFS), which provides on-disk distributed storage.

Parallel Programming Abstractions: Transformations

For the reasons covered last class, transformations on RDDs are specified through parallel programming abstractions with functional semantics:

- *map*(f): Return a new distributed dataset formed by passing each element of the source through a function f.

Parallel Programming Abstractions: Transformations

For the reasons covered last class, transformations on RDDs are specified through parallel programming abstractions with functional semantics:

- $map(f)$: Return a new distributed dataset formed by passing each element of the source through a function f.
- $flatMap(f)$: Similar to map, but each input item can be mapped to 0 or more output items (so f should return a list rather than a single item).

Parallel Programming Abstractions: Transformations

For the reasons covered last class, transformations on RDDs are specified through parallel programming abstractions with functional semantics:

- *map*(f): Return a new distributed dataset formed by passing each element of the source through a function f.
- *flatMap*(f): Similar to map, but each input item can be mapped to 0 or more output items (so f should return a list rather than a single item).
- *filter*(f): Return a new dataset formed by selecting those elements of the source on which f returns true.

Parallel Programming Abstractions: Transformations

For the reasons covered last class, transformations on RDDs are specified through parallel programming abstractions with functional semantics:

- *map*(f): Return a new distributed dataset formed by passing each element of the source through a function f.
- *flatMap*(f): Similar to map, but each input item can be mapped to 0 or more output items (so f should return a list rather than a single item).
- *filter*(f): Return a new dataset formed by selecting those elements of the source on which f returns true.
- *reduceByKey*(*func*, [*numTasks*]): When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function f, which must be of type $(V, V) => V$.

Parallel Programming Abstractions: Transformations

- Transformation use lazy evaluation.

Parallel Programming Abstractions: Transformations

- Transformation use lazy evaluation.
- No transformations are applied until you call an action on an RDD.

Parallel Programming Abstractions: Transformations

- Transformation use lazy evaluation.
- No transformations are applied until you call an action on an RDD.
- Why is Spark implemented in this way?

Parallel Programming Abstractions: Actions

Actions on RDDs return actual values to the Spark master process:

- *collect*(): Return all the elements of the dataset as an array to the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

Parallel Programming Abstractions: Actions

Actions on RDDs return actual values to the Spark master process:

- *collect*(): Return all the elements of the dataset as an array to the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- *reduce*(f): Aggregate the elements of the dataset using a function f (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

Parallel Programming Abstractions: Actions

Actions on RDDs return actual values to the Spark master process:

- *collect*(): Return all the elements of the dataset as an array to the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- *reduce*(f): Aggregate the elements of the dataset using a function f (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- *first*(): Return the first element of the dataset.

Parallel Programming Abstractions: Actions

Actions on RDDs return actual values to the Spark master process:

- $collect()$: Return all the elements of the dataset as an array to the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- $reduce(f)$: Aggregate the elements of the dataset using a function f (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- $first()$: Return the first element of the dataset.
- $take(n)$: Return an array with the first n elements of the dataset.

Parallel Programming Abstractions: Actions

Actions on RDDs return actual values to the Spark master process:

- *collect*(): Return all the elements of the dataset as an array to the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- *reduce*(f): Aggregate the elements of the dataset using a function f (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- *first*(): Return the first element of the dataset.
- *take*(n): Return an array with the first n elements of the dataset.
- *saveAsTextFile*(*path*): Write the elements of the dataset as a text file (or set of text files) in a given directory in the filesystem.

Spark API

The Python Spark RDD API is fully documented here:

http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Mixing Imperative and Functional Programming

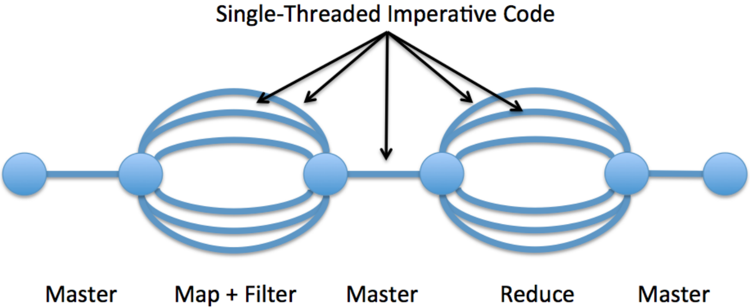
- Importantly, programs written using Spark can mix regular imperative programming with the functional parallel programming abstractions that Spark provides.

Mixing Imperative and Functional Programming

- Importantly, programs written using Spark can mix regular imperative programming with the functional parallel programming abstractions that Spark provides.
- All code in the Spark master process and in individual functions f passed to Spark can be regular single-threaded imperative code.

Mixing Imperative and Functional Programming

- Importantly, programs written using Spark can mix regular imperative programming with the functional parallel programming abstractions that Spark provides.
- All code in the Spark master process and in individual functions f passed to Spark can be regular single-threaded imperative code.



Spark and Numpy

- We can use any Python data types in RDDs and any Python library functions inside the functions we pass to Spark transformations and actions. In particular, we can use Numpy functions.

Spark and Numpy

- We can use any Python data types in RDDs and any Python library functions inside the functions we pass to Spark transformations and actions. In particular, we can use Numpy functions.
- A common operation is to map the rows of an RDD to Numpy arrays so that we can apply Numpy operations to them within Spark map and reduce operations.

In Memory Caching of RDDs

- The main advantage of Spark over MapReduce/Hadoop is that it has the ability to cache RDDs in memory on remote cluster nodes.

In Memory Caching of RDDs

- The main advantage of Spark over MapReduce/Hadoop is that it has the ability to cache RDDs in memory on remote cluster nodes.
- In the absence of caching, an RDD is recomputed from the base data from scratch including all transformations each time an action is called on it.

In Memory Caching of RDDs

- The main advantage of Spark over MapReduce/Hadoop is that it has the ability to cache RDDs in memory on remote cluster nodes.
- In the absence of caching, an RDD is recomputed from the base data from scratch including all transformations each time an action is called on it.
- Caching makes Spark much faster than Hadoop for iterative algorithms or repeated queries since the data doesn't need to be read of disk for each iteration or query.

In Memory Caching of RDDs

- The main advantage of Spark over MapReduce/Hadoop is that it has the ability to cache RDDs in memory on remote cluster nodes.
- In the absence of caching, an RDD is recomputed from the base data from scratch including all transformations each time an action is called on it.
- Caching makes Spark much faster than Hadoop for iterative algorithms or repeated queries since the data doesn't need to be read of disk for each iteration or query.
- An RDD can be marked for caching by calling $cache()$ on it.

Spark and Machine Learning

- Since the computationally intensive part of most machine learning computations on big data involves computations that are embarrassingly parallel with respect to the data, Spark can be a great fit.

Spark and Machine Learning

- Since the computationally intensive part of most machine learning computations on big data involves computations that are embarrassingly parallel with respect to the data, Spark can be a great fit.
- To re-write a Python implementation of a method like logistic regression learning or prediction, we need to replace the loops over the data with map and reduce steps.

Spark and Machine Learning

- Since the computationally intensive part of most machine learning computations on big data involves computations that are embarrassingly parallel with respect to the data, Spark can be a great fit.
- To re-write a Python implementation of a method like logistic regression learning or prediction, we need to replace the loops over the data with map and reduce steps.
- Let's look at the fundamental operation of classifying data cases contained in the rows of an RDD using a linear classifier implemented with Spark:

$$w^T x + b = \sum_{i=1}^{D} w_i x_i + b > 0$$