MapReduce, Hadoop and Spark

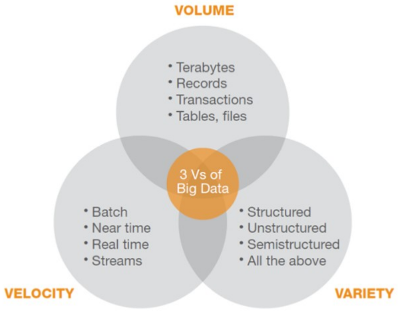
Salvatore Filippone, PhD

School of Aerospace, Transport and Manufacturing
salvatore.filippone@cranfield.ac.uk

Big Data

Dealing with Big Data:

- RDBMS (thanks, but no, thanks)
- MapReduce: store and process data-sets at massive scale (Volume+Variety)
- Data stream processing: process fast data without storing them



The 3 Vs (plus: Variability and Value)

HDFS: Hadoop File System

Large, Distributed file system

Designed in the context of Apache Hadoop. Main goals/requirements:

Hardware Failure Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines: hence some component of HDFS is always non-functional. Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Streaming Data Access Applications that run on HDFS need streaming access to their data sets, not general purpose applications. HDFS is designed for batch processing: emphasis is on high throughput rather than low latency.

Large Data Sets A typical file in HDFS is gigabytes to terabytes in size; thus, tuning to large files, high aggregate data bandwidth, scale to hundreds of nodes and tens of millions of files.

Large, Distributed file system

Simple Coherency Model Write-once-Read-many access model: files tyipically need not be changed except for appends and truncates. This assumption simplifies data coherency issues.

Moving Computation is Cheaper than Moving Data A computation is much more efficient if executed near the data, especially when the data set is huge.

Portability Across Heterogeneous Hardware and Software Platforms Facilitates widespread adoption

Parallel programming:
$\Rightarrow$ Break processing into parts that can be executed concurrently on multiple processors

Challenges

- Identify tasks that can run concurrently and/or groups of data that can be processed concurrently
- Not all problems can be parallelized!

Simplest environment for parallel programming is:

- No dependency among data
  $\Rightarrow$ Data can be split into equal-size chunks;
- Each process can work on a chunk;
- Master/worker approach:

  Master:
  - Initializes array and splits it according to the number of workers
  - Sends each worker the sub-array
  - Receives the results from each worker

  Worker:
  - Receives a sub-array from master
  - Performs processing
  - Sends results to master

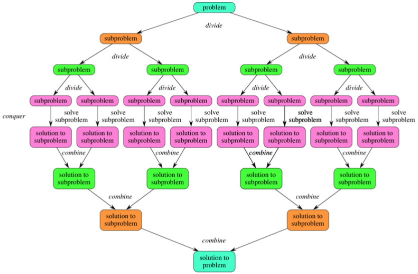
Single Program, Multiple Data (SMPD): technique to achieve parallelism
$\Rightarrow$ The most common style of parallel programming

Divide and Conquer

Divide and Conquer

Basic technique:

- Split in subproblems, assign them to workes, then combine
- Workers can be threads, processes, cores, nodes,etc.
- How to decompose the work?
- How to allocate tasks?



Big Data Features

Data intensive workloads prefer large numbers of commodity servers:

- Cost is not linear in capacity;
- Efficiency is an issue

Processing is fast, I/O is slow

Sharing is a problem

Best of all, share NOTHING:

- No deadlocks;
- No overhead in bandwidth;
- No synchronizations and restarts;

MapReduce

Programming model for large data sets

- Originally from Google
- Shared-Nothing approach

Multiple implementations.
Applications

- Web Indexing
- Web-link Graphs
- Sort
- Statistics

Typical Big Data program:

MAP

- Iterate over many records;
- Extract something from each record;
- Shuffle/sort intermediates;

REDUCE

- Aggregate intermediates;
- Produce final output

Main idea: give a functional abstraction of the MAP and REDUCE operations

Input and output: sets of key-value pairs.
Programmers specify two functions:

- Map
$$map(k1, v1) \to [(k2, v2)]$$
- Reduce
$$reduce(k2, [v2]) \to [(k3, v3)]$$

where

- (k, v) denotes a (key, value) pair
- $[\dots]$ denotes a list
- Keys do not have to be unique: different pairs can have the same key
- Normally the keys of input elements are not relevant

Map

Execute a function on a set of key-value pairs (input shard) to create a new list of values:

$$map(in_{key}, in_{value}) \rightarrow list(out_{key}, intermediate_{value})$$

Example: $square\, x = x * x$; then

$$map\ square[1, 2, 3, 4, 5]$$

returns

$$[1, 4, 9, 16, 25]$$

- Map calls are distributed across machines by automatically partitioning the input data into M "shards"
- MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function

Reduce

Combine values in sets to create a new value

$$reduce(out_{key}, list(intermediate_{value})) \rightarrow list(out_{value})$$

Example:

$$sum = (each\, elem\, in\, arr, total+ =)$$

then

$$reduce[1, 4, 9, 16, 25]$$

returns 55 (the sum of the square elements)

Added value:

1. There is an implicit "group" phase on intermediate keys, so that reducers can assume data is ordered;
2. Intermediates are transient, only in memory and/or local disk caches (no distributed I/O)

Hello World: Word Count

Count the number of occurrences of the words in a document:

Input: a (set of) document(s)

Map: read documents, emit $(key, value)$, where Key is a word and value is 1, e.g. $(w1, 1), (w2, 1), (w1, 1)$

Grouping: build lists of values with same key

$$(w1, [1, 1]), (w2, [1]) \ldots$$

Reduce: add elements from the lists $(w1, 2), (w2, 1) \ldots$

Output: Print $(key, reduced\, values)$ pairs.

Spark

- A framework built on top of Hadoop;
- Many speed optimizations;
- Extensions for stream processing.

Spark essentials

- `SparkContext`: an object created at the start of the Spark program, defining the access to the cluster. Variable `sc` in pyspark. Its `master` variable controls how to connect: all examples in the lab have been run with `local`, but in principle:
  - Connects to a cluster manager;
  - Acquires execution instances on the cluster nodes;
  - Distributes code to the workers;
  - Distributes tasks to the workers;
- Resilient Distributed Datasets (RDD): the primary data abstraction. A collection of elements, either from local memory or from a Hadoop file system (hence, fault tolerant);

Spark essentials: Transformations

Operations that take a dataset and produce another:

- `map` Streams each element through a function;
- `filter` Selects a subset;
- `flatMap` Like `map`, but to each input there can be 0 or more outputs;
- `sample` (with or without replacement)
- `union`, `distinct`, `groupByKey`, `reduceByKey`, `join`, `cogroup`, `cartesian`.

Spark essentials: Actions

Operations that take a dataset and produce a value:

- `reduce` Aggregate with an associative function to produce a value;
- `collect` Return the dataset as an array;
- `count`
- `first`, `take(n)` Return the first element, or an array with the first n elements;
- `takeSample` return an array with a random sample.
- `saveAsTextFile`, `saveAsSequenceFile` name says all
- `countByKey`
- `foreach`

Full details at: `https://spark.apache.org/docs/latest/api/python/pyspark.html`

Hello world

```
from __future__ import print_function

import sys
from operator import add

from pyspark import SparkContext


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print("%s: %i" % (word, count))

sc.stop()
```

Perron-Frobenius theory of nonnegative Matrices:

Let Q be a stocastic matrix, e.g. state transitions of a Markov chain

$$e^T Q = e^T, \qquad Q \geq 0, \qquad e^T = (1, 1, \ldots, 1)$$

if Q is irreducible, then

1. $\lambda = 1$ *is the dominant eigenvalue;*
2. *To the dominant eigenvalue there is associated a unique positive eigenvector* $r > 0$ *;*
3. r *gives the steady-state probability distribution of the Markov chain*

Perron-Frobenius theory of nonnegative Matrices:

Let Q be a stocastic matrix, e.g. state transitions of a Markov chain

$$e^T Q = e^T, \qquad Q \geq 0, \qquad e^T = (1, 1, \ldots, 1)$$

if Q is irreducible, then

1. $\lambda = 1$ *is the dominant eigenvalue;*
2. *To the dominant eigenvalue there is associated a unique positive eigenvector* $r > 0$ *;*
3. r *gives the steady-state probability distribution of the Markov chain*

Which application are we thinking of?

Pagerank

$$\text{Google} = \text{crawling} + \text{matching} + \text{PageRank}$$

$$\text{Google} = \text{crawling} + \text{matching} + \text{PageRank}$$

Ranking of a WEB page:

> *The relevance of page i is the weighted average of the relevance of all pages j linking into page i*

$$r_i = \sum_j \frac{r_j}{N_j}$$

Note that this is a recursive definition!

Pagerank

$$\text{Google} = \text{crawling} + \text{matching} + \text{PageRank}$$

Ranking of a WEB page:

> *The relevance of page i is the weighted average of the relevance of all pages j linking into page i*

$$r_i = \sum_j \frac{r_j}{N_j}$$

Note that this is a recursive definition! PageRank algorithm:

- Create a list of all WEB pages and their links (WWW connectivity graph);
- Assign a weight to each link and build matrix Q;
- Find the eigenvector r;

The value in eigenvector r for page i is the measure of its authoritativeness.

Building the matrix:

$$Q_{ij} = \begin{cases} 1/N_j & \text{if page } J \text{ links into page } I \\ 0 & \text{otherwise} \end{cases}$$

In summary: at the heart of search engines there lies the computation of a humongous eigenvector

$$r^T Q = r^T$$

The power iteration

To compute the solution to $Ar = \lambda r$:

$r^{(0)} \leftarrow r_0$
for $k = 1, \dots$ **do** until convergence
  $q^{(k)} \leftarrow Ar^{(k-1)}$
  $r^{(k)} \leftarrow q^{(k)}/\|q^{(k)}\|$! Redundant for $A = Q$ stocastic
end for

a few tens of iterations.

Note that if A is stocastic and $\|z\|_1 = 1$, then $\|Az\|_1 = 1$, because

$$\|y\|_1 = e^T y = e^T Az = e^T z = 1$$

hence the normalization is redundant.

What do you do when a page has no outlinks?

You add a jump to any other page, with a uniform probability distribution

This is called "teleportation"
Defining

$$d_j = \begin{cases} 1 & \text{if} N_j = 0; \\ 0 & \text{otherwise.} \end{cases}$$

we modify the matrix as

$$P = Q + \frac{1}{n} e d^T$$

This would be sufficient

Moreover, we rewrite

$$A = \alpha P + (1-\alpha)\frac{1}{n}ee^T = \alpha(Q + \frac{1}{n}ed^T) + (1-\alpha)\frac{1}{n}ee^T$$

because the matrix P would otherwise be reducible, i.e. there is at least a subgraph in which you can remain trapped (and this plays havoc with the eigenvector).
You would *not* want to write A explicitly (it's full!). But you can do the following:

$$y = Az = \alpha(Q + \frac{1}{n}ed^T)z + (1-\alpha)\frac{1}{n}ee^Tz = \alpha Qz + \beta\frac{1}{n}e$$

where

$$\beta = \alpha d^Tz + (1-\alpha)\frac{1}{n}e^Tz$$

But if we know that

$$\|Az\|_1 = 1 \Rightarrow 1 = \beta + e^T(\alpha Q z)$$

and you do not need to know d.
The vector e gives a uniform teleportation, but you can add any vector

$$v, \quad \|v\|_1 = 1$$

in the equation

$$A = \alpha P + (1-\alpha) v e^T$$

to have a personalized teleportation which will favour certain nodes (those nodes corresponding to web sites that are willing to pay!).
Among the Spark predefined examples, you will find a version that does not personalize, and uses $\alpha = 0.85$ and $\beta = 0.15$ with a simple formula that does not rescale by $1/n$ because it starts from $\|z\|_1 = n$. You can use it unmodified