Distributed Breadth-First Search on Amazon EC2's server

Sandro Cavallari
161956

Luca Masera
163025

Simone Melchiori
161957

I. Introduction

The aim of this project is to develop and test a graph exploration algorithm based on a Breadth First Search (BFS) in Hadoop MapReduce. This will run on the Amazon's EC2 servers where we installed a copy of the Hadoop framework. In the following sections we will take an overview over the big data problem and the solutions that the distributed systems offer, then we will see how the map-reduce works. We will enunciate the BFS problem and a centralized solution. After that we'll show our implementation and our results.

A. Big Data and Distributed Systems

Nowadays "Big Data" are an issue that must be faced. Modern society produces an enormous amount of information. For example Google grew from processing 100 terabytes of data a day with MapReduce in 2004 to processing 20 petabytes a day with MapReduce in 2008. Analogous data comes from Twitter, Facebook and the most IT companies. Storage capacity has grown fast keeping low cost and reliability, bandwidth and latency have improved relatively little instead. To process all this data is required a lot of computational power. Relying on ad-hoc computing solution is expensive and not really scalable, therefore we are moving towards robust and stable cluster architecture using frameworks for distributed calculus.

B. Hadoop and MapReduce

MapReduce is both a programming model for expressing distributed computations on huge amounts of data and an execution framework for large-scale data processing.
Map Reduce was initially developed by Goolge who don't release it for public users. So in the 2007 Hadoop was released under the Apache project, it's a opens source implementation of Map Reduce framework that become very popular and used by eBay, IBM, Yahoo!, Facebook and Twitter.

Typically this framework is useful for:

- Scaling
- Managing common failure automatically
- Move processing operation on the same processor where data are
- Providing an abstraction that isolates the developer from system-level details

The two main concept of this framework comes from the functional languages. The map, in functional languages, is a function that applies a given function to each element of a list. The reduce takes a list and a function as input and give as output a single value. The MapReduce framework is based on this two main function that have a Key-Value structure.
Map function takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$Map(k1, v1) \rightarrow list(k2, v2)$$

The Map function can be applied in parallel on different node, after his computation the framework collects all pairs with the same key from all lists and groups them together, creating one group for each key. Once it has generated the groups of key they are subdivided to the Reducer that are applied in parallel. The Reduce function produces a collection of values in the same domain:

$$Reduce(k2, list(v2)) \rightarrow list(v3)$$

The result of the framework is usually a key-value pairs from each reducer that are written persistently onto the distributed file system. Usually Map Reduce run on big cluster formed by low-end machines so the framework have to take care of:

- Scheduling;Each MapReduce job is divided into smaller units called tasks. Large jobs could be composed by thousands of tasks that has to be assigned to nodes in the clusters.Generally there are more job-tasks than tasks that can be performed concurrently, so the scheduler must manage a task queue and track the progress of running tasks,
- Process location: As consequence of the "Move processing to data" the scheduler starts tasks on the node that holds a particular block of data needed by the task. If this is not possible, is the date that are moved elsewhere(on a new node that hold the task, the transfer is done using the network and this will downgrade the performance)
- Synchronization: In MapReduce, the main synchronization task is run between the map and reduce phases.The synchronization guarantees that the reduce phases start only when all mappers finish his work, so certainly all key-value pairs are detected,shuffled and sorted.
- Fault Handling:In the running environment errors and faults are the norm.MapReduce is explicitly designed around low-end commodity servers: the framework must be particularly resilient against hardware failures as well as software bugs potentially contained in the end-user code.

C. The Breadth-First Search

Graphs are a very common data structure in Computer Science and the algorithms that works on them are fundamental in this field. The Breadth-First Search is one of the simplest search algorithm on graphs and the backbone of many important algorithms that works on graphs. Given a graph G = (E,V) and a starting node s, the BFS starts inspecting all the neighbours of s, then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. To achieve this goal in centralized algorithm is used a queue that allow to maintain an order.

Algorithm 1: Breadth-First Search algorithm

```
function BFS (Graph G, Node s)
    foreach $u \in G$ do
        $dist[u] \leftarrow \infty$;
        $father[u] \leftarrow nil$;
    $dist[s] \leftarrow 0$;
    queue $Q$;
    $Q$.enqueue($s$);
    while $Q \neq \emptyset$ do
        Node $u \leftarrow Q$.dequeue();
        foreach $v \in neighbor(u)$ do
            if $(dist[u] + 1) < dist[v]$ then
                $dist[v] \leftarrow dist[u] + 1$;
                $father[v] \leftarrow u$;
                $Q$.enqueue($v$);
```

The complexity of this algorithm is $O(|V| + |E|)$ because before starting is required a initialization of every node in G to set the distance to infinity and the father to nil. After that it explores all the edges of the graph reachable from s. Iterate this algorithm on the "non-discovered" nodes is a possible technique for the connected components identification. There are further application from the characteristic of advancing level-by-level. Is possible to obtain the spanning tree and the distances of the nodes from a chosen starting node. These informations are contained in the variables father[] and dist[] of the code [Algorithm 1].

II. MapReduce Implementation

A. Input

The input of this implementation is a text file with each line formatted as follows:

$< id >< tab >< list(adj) > | < dist > | < colour >$

Where id represent the unique identification number of a node, tab is a tabulation, list(adj) represent the list of adjacent nodes separated by a comma and described with the id, dist is the distance from the starting node and is initialized at 0 for the starting node and infinite (MAXINT) for the other nodes. At last the tag colour can assume the value BLACK, GRAY and WHITE respectively for the visited node, the in queue node and the not yet visited node. At the beginning is set as GREY the node with the highest degree.

B. Exploring phase

1) Map function: The Map function receive as an input a line of the text input file, the class Node is able to generate itself from a text line formatted as explained above. If the node has colour equals to GRAY (hence is enqueued) then the algorithm emits all the neighbours with the distance incremented by one and setting the colour to GREY. Then the explored node's colour is set to BLACK. Note that if the input node's colour is WHITE or BLACK the node is emitted unchanged.

2) Reduce function: During the map phase is possible that the same node is emitted by different neighbour, the goal of the reducer is to iterate over this duplicate

Algorithm 2: Map algorithm

```
function MAP (int key, Text value)
    Node $u \leftarrow$ Node($value$);
    if $n.colour = GRAY$ then
        foreach $u.id \in neighbor(n)$ do
            $u.dist \leftarrow n.dist + 1$;
            $u.colour \leftarrow GRAY$;
            emit($u.id$,$u$.toText()));
        $u.colour \leftarrow BLACK$;
    emit($u.id$,$u$.toText()));
```

and emit a new node that is obtained collecting the maximum distance of the duplicate nodes and the darkest colour. A node in the list will also contain the list of the neighbours, that is saved.

Algorithm 3: Reduce algorithm

```
function REDUCE (int key, Text[] values)
    List $edges$;
    colour $c \leftarrow WHITE$;
    int $dist \leftarrow +\infty$ ;
    foreach $value \in values$ do
        Node $n \leftarrow$ Node($value$);
        if $!n.edges.isEmpty()$ then
            $edges \leftarrow n.edges$;
        if $n.dist < dist$ then
            $dist \leftarrow n.dist$;
        if $n.colour > c$ then
            $c \leftarrow n.colour$;
    Node $u \leftarrow$ Node($key, edges, c, dist$);
    emit($key$,$u$.toText()));
```

C. Removing phase

When a connected component has been completely visited, after several iterations, then there are only BLACK and WHITE nodes in the system. To continue, the algorithm, removes all the BLACK nodes and turns a WHITE node into GRAY. As is possible to see in Algorithm 4 the mapper emits only the WHITE nodes, and the reducers check if the GRAY node has been set. A WHITE node turns into GRAY, and the algorithm proceed as before, exploring the new connected components.

Algorithm 4: Remove Map algorithm

```
function MAP (int key, Text value)
    Node $u \leftarrow$ Node(value);
    if $u.colour = WHITE$ then
        emit($u.colour$,$u$.toText()));
```

Algorithm 5: Remove Reduce algorithm

```
function REDUCE (int key, Text[] values)
    foreach $value \in values$ do
        Node $u \leftarrow$ Node(value);
        if $grayCounter = 0$ then
            $u.colour \leftarrow GRAY$;
            $u.dist \leftarrow 0$;
            $grayCounter++$;
    emit(key,$u$.toText()));
```

D. Complexity

The complexity of this algorithm in term of iteration is $O(\sum_{i=0}^{|CC|}(\varnothing cc_i))$. Regarding the complexity of each iteration the results are shown in Figure 3 and in Figure 4. The introduction of the combiner reduces the load on the network processing part of the information before they are sent to the "shuffler and sorter".

E. Hadoop counters and Stop Condition

When operating with distributed system is often useful to have a way to communicate between the machines. One possible method provided by the Hadoop framework are the counters. Every counter is identified by an Enumerator, and using that is possible to share information between the machines. In this case, for example, we needed a way to count the number of nodes of each colour. As explained before, in our project the colours are implemented as enumerators, hence we used a counter for each colour. To switch between the "exploring" phase and the "removing" phase, we counted the BLACK nodes after each iteration and if the number does not increase then it means that all the nodes of a connected components has been visited. Hence the removing phase begins, in which we use the GREY counter to colour exactly one node. At last, when the BLACK counter is equal to the overall node counter the protocol stops.

III. Results

The experiments has been conducted on three Amazon EC2 c1.medium machines with Ubuntu 12.04 64 bit. On each of them we installed the JDK7 (OpenJDK) and Hadoop_1.1.2. One machine was dedicated to the jobManager (master node) and two to the computational work (slave nodes).
We analysed the Slahdot0902 graph[1] which has 82168 vertices and 948464 edges. The results has been taken from just a graph because we noticed that the execution time was not related to the graph, it was dominated from the framework overhead. So we decided to take into account just this graph and to show how the combiner impacts on the network load.
In Figure 2 can be noticed that the use of the combiner reduce significantly the number of messages sent over the network. The peak of active nodes is between iteration 2 and 4 when, as shown in Figure 1, there is the maximum number of grey nodes.

IV. Conclusion

The goal al this project was to develop and test a distributed implementation of the breath first search in with the Hadoop framework. We took inspiration from an

[1]http://snap.stanford.edu/data/soc-Slashdot0902.html

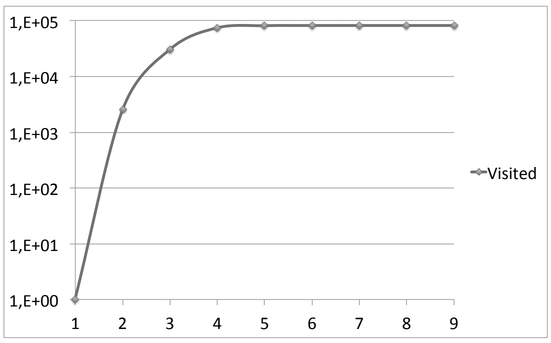

Figure 1. Growth of the visited nodes after each iteration. The chart is logarithmic scale

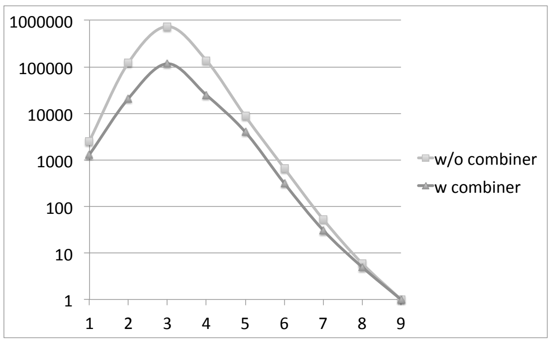

Figure 2. Comparison between the active nodes with and without the combiner. The chart is logarithmic scale

implementation found on-line[2] on top of which we added the stop condition, the possibility to restart the exploration on a new connected component and the combiner to improve the performance.

The first attempts to run our algorithm on AWS have been done with the ElasticMapReduce provided by Amazon. It was easy to configure, but the long start-up times of each execution and the interface with the S3 buckets made us decide to move the entire project on three EC2 machines manually configured. The experience with with EC2 was good, it needs just a little confidence with SSH and Unix commands to set up and use the machines.

The execution-times of our algorithm are strictly related to the number and the diameter of connected components of the graph taken in exam. For its nature the framework reads and writes always the whole graph, so there is no significantly difference between the time of each iteration. Moreover we noticed that even changing graph the run-time was not affected. We can conclude that, for the examined graphs, the overhead introduced by the framework is larger of the algorithm work-load. Maybe with even larger graphs the growth of the computation needs will impact on the overall performance of the execution.

[2]http://www.johnandcailin.com/blog/cailin/breadth-first-graph-search-using-iterative-map-reduce-algorithm

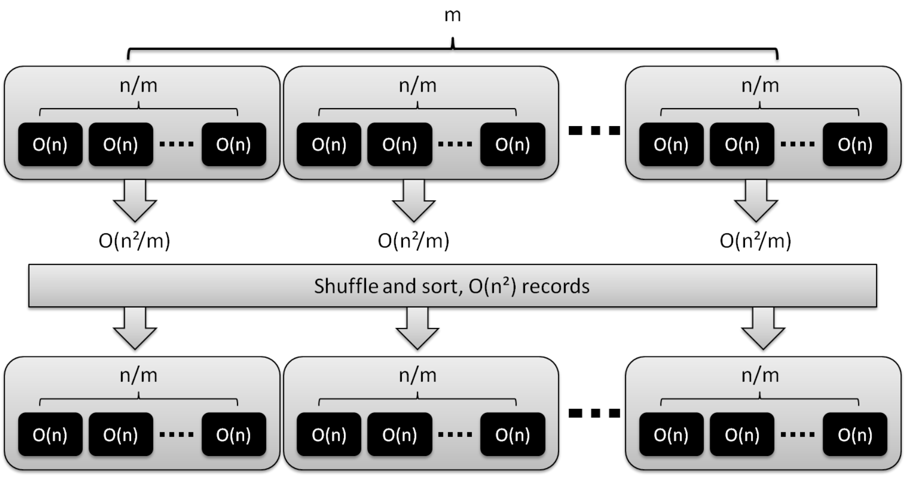

Figure 3. Complexity scheme without combiner

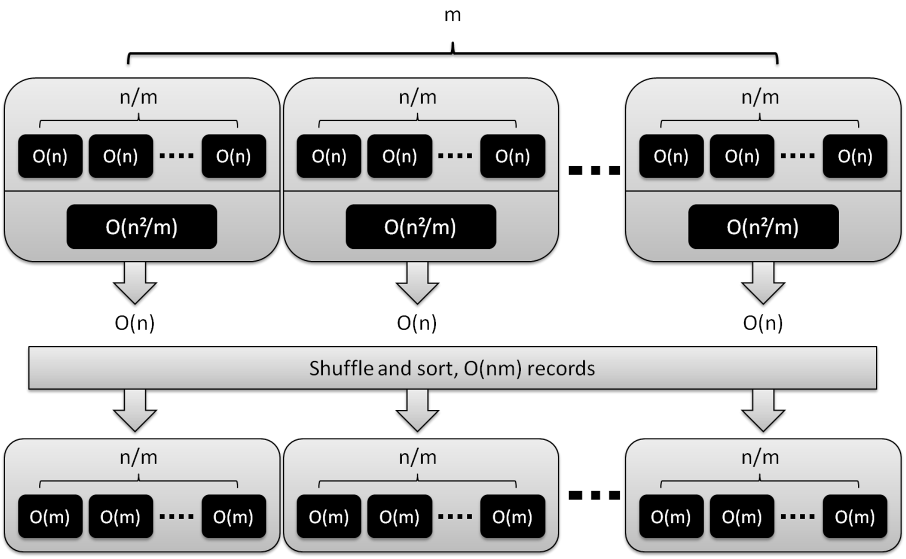

Figure 4. Complexity scheme with combiner