A MapReduce Algorithm: HOW-TO approach to the BigData

Antonio Paolacci
e-mail me at: ant.paolacci@gmail.com

***Abstract*—Google MapReduce has become the *best-now* standard to process large data sets in a parallel, high-scalable, fault-tolerant manner on machines cluster. MapReduce can collect, filter and correlate availabe data (structured and not) and derive new rich, high value, relevant information. In this paper I present a MAPREDUCE-based approach to achieve a very simple Java implementation of MINIMUM SPANNING TREE (MST) problem in the MapReduce open source counterpart, HADOOP. Minimum Spanning Tree is one of the most studied combinatorial problems. His fame is due to multiple practical application: VLSI-chip design, electrical and communication networks, content matching and product placement, medical imaging. Furthermore MST in computer science serve as a common introductory example of both graph algorithms and greedy algorithms due to his simplicity. Unfortunately, MST sequential algorithms face the challenge of costing too much time and space when the input becomes huge. A graph can be huge and represented by a file with a batch of millions records. MapReduce can provide help to manage large data sets at the scale of tera-bytes, peta-bytes or higher. The main idea behind MapReduce is to reduce the size of the input in a distributed fashion so that the resulting, much smaller, problem instance can be solved on a single machine, otherwise it recurs with a new round of the MapReduce algorithm.**

***Index Terms*—MapReduce, Hadoop, Minimum Spanning Tree, BigData, Large Scale Distributed Computation, MST**

I. INTRODUCTION

Say a national agency want to avoid a biological/chemical attack such as detecting the spread of toxins through populations, and establishing strategic points that act as an articulation point[1]. Detect articulation point and bridge edge is a minimum spanning tree subproblem. A minimum spanning tree is a subgraph that is a tree(no cycles) with function-weight on edge less than or equal to the weight of every others tree and that connects all the vertices. The minimum spanning tree problem admits an efficient solution only if it is calculated in a parallel way. An excellent contribution to the MST story starts since 1985 in *On the history of the minimum spanning tree problem* [GH85]. MST has many practical application to optimize and build wireless and cabled networks, telco networks, design cmos chip,... [eA73], [JR70], [L.R66], [R.G73], [eA75]. It occurs as subproblem in traveling salesman problem [Nic76], matching problem and k-clustering problem. The authors of [GH85] complain about we make a mistake when we refer MST to the work of **Kruskal** (1956) [J.B56] and **Prim** (1957) [R.C57], the problem was previously solved by **Borüvka** (1926) [Ota26]. All the algorithm are $O(m \log n)$ based on the greedy algorithm technique. Andrew Chi-Chih Yao, David Cheriton e Robert Tarjan in [eRET76] provides a $O(m \log \log n)$ optimization. Michael Fredman and Robert Tarjan in [eRT87] with the introduction of data structure called Fibonacci heaps, decrease the computational complexity to $O(m\beta(m,n))$ where $\beta(m,n) = min\{i|log^i\, n <= m/n\}$. In the worst case $m = O(n)$ the complexity is $O(m \log^* m)$. The execution time is decreased to $O(m log\beta(m,n))$ by Harold N. Gabow, Zvi Galil, Thomas H. Spencer, e Robert Tarjan in the following paper [HG86]. Finding the minimum spanning tree in linear time remains an open problem. Bernard Chazelle in [Cha00] provides a significative step towards a linear-time algorithm, his complexity is $O(m\alpha(m,n))$, where α is the inverse Ackermann function. It's easy to realize that the lowerbound for MST $\Theta(m)$ because each edge must be considered at least once, the best upperbound now is Chezelle $O(m\alpha(m,n))$. These algorithms are beatiful but quite complicated. So engineers have decided to develop distributed algorithms for MST. The classic algorithm is presented in [GHS83] by Gallager et al. based on message-passing, but messages spread across machines introduce bottlenecks and overheads. Now with the emergence of MAPREDUCE, the MST problems could be treated with a novel approch. H. Karloff et al. propose without real implementation a MapReduce MST algorithm in [HK10] (SODA 2010). Recently Vassilvitskii et al. in [LMSV11] (SPAA 2011) propose a MST algorithm based on their MapReduce technique called *filtering*. The Karloff et al.'s practical implementation is shown on this paper to guide the lector discovering MapReduce paradigm.

II. MAPREDUCE

MapReduce is a programming model and an associated implemented framework engine for processing and analyzing large data sets, sponsorized and made famous by Google. Google was author of dozens solution able to manage efficiently huge data, but no one has reached consensus by the community as the MapReduce framework. In recent years Google reported that it processes over 3 petabytes of data using MapReduce in one months for reverse-index for Google Search, article clustering for Google News, data mining, spam detection. The MapReduce abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. Many data computations involve a map operation to each logical record in input to filter, preparing the computation and then applying a reduce operation to

[1]Articulation points keep the network connected, i.e., they are necessary to connect certain subnetworks.

all the values that shared the same information, in order to derive new data appropriately. Hadoop, the Apache Foundation opensource counterpart, is currently the obvious choose used throughout both industry and academia for petabyte scale data analysis. Many company have their own Hadoop cluster installations for analyzing big data. Provides developers capabilities to process, filter, manipulate, large structured and not structured data with distributed algorithm on a cluster of commodity hardware, according to the *on the Cloud* paradigm. The issues of how to parallelize jobs, how distribute the input data, and how handle failures nodes are transparent to the programmer preserving to develop complex code. A MapReduce application splits the total input data into independent chunks which are processed by the MAP FUNCTION in parallel, reliable manner transparent to the programmer. The framework stores and sorts the intermediate data, which are then input to the subsequent REDUCE FUNCTION. The execution of a Map stage plus a Reduce stage performs a MapReduce *job* or *round*. The computational cost of a MapReduce algorithm can be calculated by the number of round that it requires to terminate successfully. In the map phase the input is processed one tuple at time, provided by an adapter (DB row, file reader, socket,...). All <*key,value*> pairs emitted by the mapper in the map phase that have the same key are aggregated by the MapReduce subsystem during the shuffle stage and sent to the reducer. Finally each key, along with all the values associated with it, are processed together, provided like a Java Iterator in the reducer code. Since all the values with the same key end up on the same machine running as a reducer process, one can view the map phase as a kind of routing step that determines commons values, that must be processed together. The basic building block of each MapReduce algorithm are the <*key,value*> pairs. Looking at the computation, on every round each machine of the cluster performs some independent computation on the set of <*key,value*> pairs. HADOOP is an Apache Software Foundation top-level project. It provides an open source implementation of Google MapReduce dictats. The computation must be express with the following functions:

- **Map** user-defined function. Takes an input pair and produces a set of intermediate pairs. The MapReduce subsystem groups together all intermediate values associated with the same intermediate key and passes them to the reduce method. It is crucial that the map operation is *stateless*, i.e., it operates on one pair at a time. This allows for easy parallelization as different inputs for the map can be processed by different machines.
- **Shuffle** automatized by MapReduce engine. The intermediate pairs that share the same key, are grouped and sorted(natural or lexicographical order) and sent to the reducer machine by a partitioning function, that can be appropriately programmly changed.
- **Reduce** user-defined function. Accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values, that represent the input for a new MapReduce round. Output is a multiset of <*key,value*> pairs with the same key k.

An algorithm runs for a constant number rounds, via a continuous iteration of MapReduce rounds, before end with a solution. MapReduce is a powerful computational model that has been proved to enable large scale data mining; Google implemented PageRank [PBMW99] on MapReduce and use it to process huge pieces of web graph.

III. KARLOFF ET AL.

H. Karloff et al. proposed an easily parallelized algorithm IV for MST in MapReduce in [HK10]. The basic idea is to distribute the input large dense[2] graph over the cluster machines, execute parallel MST-computation on partial subgraphs, at the end calculate the final MST on a smaller graph, able to fit in memory of a single cluster machine. This is a common way to proceed when you are involved with a MapReduce; also you have to take care of *shuffle stage* leveraging sorting and grouping functionalities. Denoting the graph, vertex set, edge set by G, V, E, the pseudo algorithm is as follows:

Step 1 partition the vertex set V into k equally sized subsets $V = V_1 \cup V_2 \cup V_3 ... \cup V_k$ with $V_i \cap V_j = \emptyset$ for $i \neq j$ and $|V_i| = \frac{|V|}{k}$ for each i.
For every pair $\{i, j\}$, $E_{i,j} \subseteq E$ be the edge set induced by $V_i \cup V_j$. Formally $E_{i,j} = \{(u, v) \in E \mid u, v \in V_i \cup V_j\}$, $G_{i,j} = (V_i \cup V_j, E_{i,j})$ is the resulting subgraph of partitioning phase.
Step 2 for each of the $\binom{k}{2}$ subgraphs $G_{i,j}$, compute the unique minimum spanning forest $M_{i,j}$, then let H be the graph consisting all of the edges present in $M_{i,j}$, formally $H = (V, \cup_{i,j} M_{i,j})$.
Step 3 compute final M, the minimum spanning tree of H.

In step **3**, the algorithm builds MST on H, where $H = (V, \cup_{i,j} M_{i,j})$ and $M_{i,j}$ is the unique minimum spanning forest of subgraph $G_{i,j}$. Instead, step **1**, note that the resulting subgraphs contains edges overlapped. The vertex set of $G_{i,j}$ is $V_i \cup V_j$ and considering i, V_i goes to any $G_{i,x}$, $x = 0, 1, ..., k-1$. H is the graph consisting of edges present in every $M_{i,j}$, so the edge set of H may have duplicated ones. Thus the number of edges in H can still be very large, and finding MST in H will take a considerable long time or can't be finished due to the limitation of memory. The following proposed algorithm corrects Karloff et al.'s algorithm ill-conceived.

IV. IMPLEMENTATION

The implementation is splitted into only two rounds. This is not a common scenario in MapReduce programming. A MapReduce algorithm is iterative, the number of iterations is just enough to the input to fit in memory of a single

[2] A dense graph is a graph in which the number of edges is close to the maximal number of edges. Given $G(V, E)$ the maximal number of edges is $\frac{|V|(|V|-1)}{2}$.

Algorithm 1 Karloff et al.

1. Random partition vertex set V in k equally sized subsets, $V_1, ..., V_k$
2. Parallel compute MST on resulting subgraphs composed by edges belongs to V_i, V_j, indicates the resulting edge sets with $M_{i,j}$
3. $H \equiv (V, \cup_{i,j} M_{i,j})$. Compute MST on H.

machine. Step **1** and step **2** are completed in the first round and **3** in the second round. Note the k parameter properly set, reduces memory usage on each computational round. With high probability the memory usage is $O(|E|/k)$ calculating $M_{i,j}$ and $O(|V|\,k)$ calculating Minimum Spanning Tree of H.

Input. The input is represented by an undirected graph with no loops. Assume without loss of generality that graph is an input file with the form of *weight firstVertex secondVertex*, each edge per line. The file of a graph G with n vertexes and m edges will have m records.

Output. A file as the input file form, that represents the edges set of MST of G with $|N| = n$. The file will have $|N| - 1$ record lines[3]. In Hadoop algorithms input and output file data are stored in HDFS, a distributed filesystem to support the Hadoop MapReduce operation. HDFS is a unix-like filesystem that provides typical unix command: *put, get, ls, cat, ...* HDFS use a *write-once-read-many* semantic, so file, once written, cannot be modified; this prevents consistency problems in a distributed environment and allows a high *throughput* of access. Therefore the input of a MapReduce computation is a single or multiple files on HDFS directory, the output is a textual file, one for each reducer machine, with name *part-00000, part-00001, part-00002,...*; they represent the input for a new subsequent MapReduce iteration.

The First Round. In step 1, the vertex set V is equally-sized partitioned into k subsets. Every subsets is a MapReduce subproblems. To realize this implementation in MapReduce model, you must answer at the first two step of the HOW-TO 2: Steps **1,2** can be achieved by <key,value> as a form of *<IDsubgraph, weight:firstVertex:secondVertex>* where IDsubgraph is generated by a special hash function. This partioning algorithm represents novel contribute on MST MapReduce literature. The idea is use $umodk$, as $u \in V$, to generate the k subgraphs. Then, an edge (u, v) belongs to $E_{k=i,j}$ subgraph if $u \in V_i$ and $v \in V_j$, so just concat $u\boldsymbol{f}(k) = i$ and $v\boldsymbol{f}(k) = j$ to generate the ik key, that is *IDsubgraph*, and to build <key, weight:firstVertex:secondVertex>. The $\boldsymbol{f}$ function is based on mod hash function but corrects some special case. The

[3]There may be several minimum spanning trees of a graph G, cause not each edge has a distinct weight. If there are $|N|$ vertices in the graph, then each spanning tree has $|N| - 1$ edges.

Algorithm 2 MAPREDUCE HOW-TO

1. Let identify the *intermediate <key,value>* pairs. *<key,value>* pairs are the basic unit of information-exchange of a MapReduce algorithm.
2. Define a mapper method, that:

- **emits *<key,value> pairs*** leveraging shuffle feature to create a small instance *key-based* subproblem. Recall *Iterator* Java interface. Recall natural order key-based calling *.next()* element. Recall that subproblems means *<key, IteratorList<values,...>>* available at the reducer process.
- try to produce **uniformly (equally-sized)** distributed subproblems.

3. At the end, ensure that the reducer code writes lines on file as a form that they can represent new *<key,value> pairs*, facilitating input for a subsequent step.

Java code of the $\boldsymbol{f}$ function is shown at the end of this paper. It's correct to complain that subgraphs $G_{i,j}$ in IV are composed by (u, v) edges such as $u \in V_i$ and $v \in V_j$, but does not preclude $i = j = i$ so these edges appear on each k-permutation that contains the V_i vertex partition. Defining these vertices *duplicates*, the $\boldsymbol{f}$ function presented in this paper remove this inefficiency. The partitioning algorithm establishes that each edge belongs exactly to one subgraph applying $\boldsymbol{f}$ function once per edge.
In MapReduce algorithm the shuffle phase operates as a router of the specific intermediate pairs to the specific reducer process (in Java, an independent virtual machine) running on a cluster machine. The reducers compute MST of their key-based subgraphs, and write output file to the HDFS filesystem with name *part-00000, part-00001, part-00002,....* The reducer apply the Kruskal [J.B56] algorithm to calculate

Algorithm 3 MAPREDUCE HOW-TO

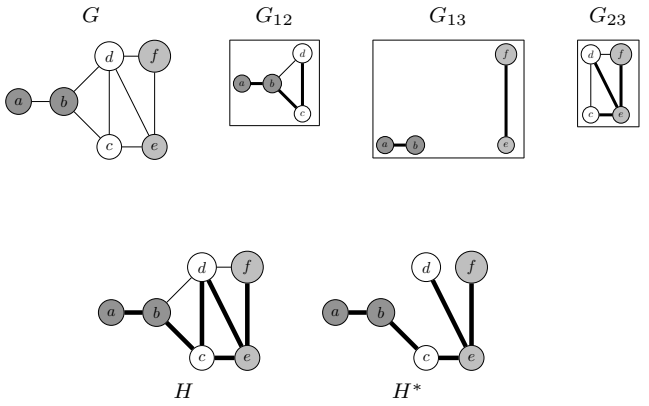
4. If previous round doesn't complete with the definitive solution, **let start a new MapReduce iterative round**. The previous output files on HDFS become the input of new iteration.
5. You can solve an ***indivisible***(not more parallelizable) problem at certain time of your algorithms, only if the input can fit in memory of a single reducer machine.

- If that's the case, recall MapReduce has automatic sorting by keys (shuffle phase), this feature can be used to sort the given data obtaining simply **total order.**

MST, and it writes the result on file. Kruskal algorithm requires a *priority-queue* to sort edges in the order of increasing weight and *union-find* implementation to avoid cycles in the subgraph. The reducer, following step **3** of 2, write one pair per line on file as a form of *null, weight firstVertex secondVertex* for easily treat the iterative input.

The second round. Steps **4,5** of 3 execute the Kruskal

Fig. 1: Execution of Karloff et al.



algorithm to calculate MST of $H \equiv (V, \cup_{i,j} M_{i,j})$, where H is the union of *part-00000, part-00001,...* files. The map function reads a single text line *weight firstVertex secondVertex* , and emits <*key,value*> *pairs* as a form of <*weight, firstVertex:secondVertex*> pairs. The choice of weight as the key allows to take advantage of the automatic sorting feature to order the edges of the graph H, without paying $O(mlogn)$ computational-cost of the first step of Kruskal's algorithm. The edges sorted by weights are considered in the increasing order of the weights. Each edge is taken and, via union-find data structure, if the end nodes of the edge belong to disjoint trees, then they are merged and the edge is considered to be in the MST. Otherwise, if the end nodes of an edge are in the same tree, they will cause cycle so the edge is discarded. The edge weight and the edge information are written to the output as a key-value pair on a single result file that contains the Minimum Spanning Tree of the input large dense graph. This implementation works only with one reducer (**5** of 3). If this algorithm is implemented with multiple reducers, then the edges get split to different reducers. Kruskal's algorithm requires edges to be processed in ascending order of weights. This order cannot be achieved if multiple reducers are used and will result in an incorrect MST answer.

V. Conclusion

In this paper, I focused on Hadoop MapReduce, an open source environment that is comprehensive of all major distributed-system features and which has reached enormous consensus by world community. Easy to install on cluster machine, it provides: automatic parallelization, load balancing, network and disk transfer optimizations via a network filesystem (HDFS), handling of machine failures, scale-out on commodity hardware nodes. Using MST, one of the main example problem in algorithmic and theoretical computer science, I try to introduce to lector the MapReduce framework, via an How-to, a set of recommendations, steps to follow and hidden features relating to the programming model. Map and reduce functions change easy to fit the problem. Many real problems can be reformulated in the MapReduce paradigm. Just think about mapping and reducing methods. Must be clear that MapReduce: reduces the complexity of a distributed environment, read terabytes of data, to be parallel the computation need to run on *independent chunks* and aggregate for a final result, map extract something you care about from each record, *shuffle* and *sort* automatic, reduce method aggregates summarizes filters or transforms, at the end write the consolidated results on *HDFS*, on an input adapter, and so on. However, it's necessary to further investigate to the drawbacks to develop an Hadoop cluster system for complex event processing (prevent DDoS attacks best fits the example)... the current Hadoop is oriented to batch processing, complain the authors of [YL11], not for near real time monitoring. Hadoop would enable us to make sense to the Big Data, extracting, correlating and storing only the relevant facts.

References

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM, 47:1028–1047*, 2000.

[eA73] W.Chou e A.Kershenbaum. A unified algorithm for designing multidrop teleprocessing networks. *Proc. DATACOMM73, IEEE 3d Data Communications Symp.*, pages 148–156, 1973.

[eA75] H.Loberman e A.Weinberger. Formal procedures for connecting terminals with a minimum total wire length. *Proc. Conf. Computer Graphics, Pattern Recognition, and Data Structures*, 1975.

[eRET76] D. Cheriton e R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 1976.

[eRT87] M.L. Fredman e R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 1987.

[GH85] R. L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *IEEE Ann. Hist. Comput.*, 7:43–57, January 1985.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, January 1983.

[HG86] T. Spencer e R.E. Tarjan H.N. Gabow, Z. Galil. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 1986.

[HK10] Sergei Vassilvitskii Howard Karloff, Siddharth Suri. A model of computation for mapreduce. In *Proc. ACM-SIAM SODA*, 2010.

[J.B56] J.B.Kruskal. On the shortest spanning tree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc. 7*, 1956.

[JR70] R.S.Heiser e N.S.King J.A.Dei Rossi. A cost analisys of a minimum distance tv networking for broadcasting medical information. *RM-6204-NLM, Rand Corporation*, 1970.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[L.R66] K.C.Williams L.R.Esau. On teleprocessing system design. *IBM Systems*, 1966.

[Nic76] Christofides Nicos. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[Ota26] Boruvka Otakar. O jistém problému minimalnim. About a certain minimal problem, 1926.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, November 1999.

[R.C57] R.C.Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech.*, 1957.

[R.G73] G.R.Bolotsky e Z.G.Ruthberg R.G.Saltman. Heuristic cost optimization of the federal telepak network. *Tech.Note 787, National Bureau of Standards*, 1973.

[YL11] Youngseok Lee Yeonhee Lee. Detecting ddos attacks with hadoop. Technical report, Chungnam National University, 2011.

Java method 1: f mapping function (first round)

```
public static class Map extends Mapper<Object, Text, Text, Text>{
  public void map(Object key, Text text, Context context) throws Exception{

   int srcPartition, dstPartition, minPartitionValue, maxPartitionValue;

    String[] inToken = text.toString().split(" ");
    srcPartition = Integer.parseInt(inToken[1]) % K_NUMBER_OF_G_SUBSETS;
    dstPartition = Integer.parseInt(inToken[2]) % K_NUMBER_OF_G_SUBSETS;

      if(firstPartition <= secondPartition){
    minPartitionValue = srcPartition;
    maxPartitionValue = dstPartition;
      } else {
    minPartitionValue=dstPartition;
    maxPartitionValue=srcPartition;
       }

      if(minPartitionValue == maxPartitionValue)
   minPartitionValue = 0;

      if(minPartitionValue == 0 && maxPartitionValue == 0)
        maxPartitionValue = 1;

    String keyMapPartOne = String.valueOf(minPartitionValue);
    String keyMapPartTwo = String.valueOf(maxPartitionValue);

    String keyMap = keyMapPartOne.concat(keyMapPartTwo);
    String valueMap = inToken[0]+" "+inToken[1]+" "+inToken[2];

    context.write(new Text(keyMap), new Text(valueMap));
  }
}
```

Antonio Paolacci, received my M.S.degree in Ingegneria Informatica Universita' di Roma Sapienza in 2012. I am currently employed at Reply S.p.A in Business Integration and SOA Oracle middleware area for primary telco companies. My research interests are algorithmics, data analysis, text mining, social analysis and open data.