MapReduce-based Distributed *k*-shell Decomposition for online Social Networks

Katerina Pechlivanidou
Electrical & Computer Engineering
CERTH and University of Thessaly
Volos, Greece
kapehliv@inf.uth.gr

Dimitrios Katsaros
Electrical & Computer Engineering
CERTH and University of Thessaly
Volos, Greece
dkatsar@inf.uth.gr

Leandros Tassiulas
Electrical & Computer Engineering
CERTH and University of Thessaly
Volos, Greece
leandros@inf.uth.gr

***Abstract*—Social network analysis comprises a popular set of tools for the analysis of online social networks. Among these techniques, k-shell decomposition of a graph is a popular technique that has been used for centrality analysis, for communities discovery, for the detection of influential spreaders, and so on. The huge volume of input graphs and the environments where the algorithm needs to run i.e., large datacenters, makes none of the existing algorithms appropriate for the decomposition of graphs into shells. In this article, we develop for the first time in the literature, a distributed algorithm based on MapReduce for the k-shell decomposition of a graph. We furthermore, provide an implementation and assessment of the algorithm using real social network datasets. We analyze the tradeoffs and speedup of the proposed algorithm and conclude for its virtues and shortcomings.**

Keywords-Map-Reduce; Hadoop; distributed algorithms; k-shell decomposition; graph algorithms; social networks

I. INTRODUCTION

The tremendous advances in information technologies and hardware, coupled with the omnipresent connectivity have created a frenzied development and popularity of online social networks (OSN) such as Facebook, Twitter, Instagram. All these online social networks store and process colossal volumes of data, mainly in the form of pair wise interactions, thus giving birth to networks, i.e., graphs which record persons' interactions.

Analysis and mining of these graphs offers both operational and business advantages to the OSN owner. Social Network Analysis (SNA) is comprised by tools and algorithms for analyzing social networks. Among the plethora of concepts encountered in SNA, the concept of *k-shell decomposition of a graph* [14] is particularly appealing, because it reveals the internal core structure of a network. If from a given graph we recursively delete all vertices, and lines incident with them, of degree less than k, the remaining graph is the k-core[1]. K-shell has been used as a centrality measure, for detecting influential spreaders [2], for discovering communities [1], for analyzing the Internet structure [4].

Several algorithms have been proposed for the computation of it in diverse computational environments, ranging from single machine main memory [3] to secondary storage [5] and to small clusters comprised by a few machines [12], and for types of networks that are unweighted or weighted [7] and vary from static to slowly changing or to networks whose topology is acquired in a streaming fashion [13]. Of all these diverse settings, the case of static networks (or those whose topology is changing very slowly compared to the time required to run analytics over the network) is the most common and encountered in the majority of today's online social networks. Therefore, we focus our attention on static, unweighted networks and an efficient implementation of the k-shell decomposition of a network.

A. Motivation and contributions

Modern OSNs are comprised by millions of nodes; therefore any algorithm for the computation of the k-shell decomposition that relies on a single machine (*centralized*) – exploiting solely the machine's main memory [3] and/or its disk [5] – is eventually doomed to fail due to lack of resources.

However, developing a distributed solution is a challenging task because it must deal with a highly sequential process, deleting one node (and incident edges) after the other. The algorithms presented in [12] and [11] developed distributed solutions, but they can run only on a small cluster of machines, which is still insufficient, since modern OSN are maintained by Internet giants such as Google, LinkedIn and Facebook who own huge datacenters and operate clusters of several thousand machines. These clusters are usually programmed by high-performance middleware of the MapReduce type [6].

Therefore, solutions for the computation of the k-shell decomposition of a network based on the MapReduce "programming paradigm" would be necessary. This is exactly the gap that the present article fills. In summary, the present article makes the following contributions:

- It develops a distributed algorithm, namely *MR-SD (MapReduce Shell Decomposition)* for the computation of k-shells of a network.
- It presents for the first time in the literature a distributed algorithm based on the MapReduce "paradigm" and therefore it is tailored for datacenter environments.
- It assesses the performance of the proposed algorithm in an experimental fashion using real datasets and analyzes various tradeoffs in its operation.

The rest of the article is organized as follows: In section II, we describe the related work; in section III we provide

[1]We use the terms k-core and k-shell interchangeably.

978-1-4799-5069-0/14 $31.00 © 2014 IEEE
DOI 10.1109/SERVICES.2014.16

background knowledge and the proposed algorithm. Section IV presents the evaluation of the proposed algorithm and finally section V concludes the article.

II. RELATED WORK

Since its initial introduction in [6], the Map-Reduce framework and its implementation in Hadoop[2] have been used in many areas related to huge data processing. MapReduce's most successful application area is Information retrieval [10], but it has also been used for bioinformatics, data mining and databases. The data management community has contributed fundamental ideas in the improvement and the extension of MapReduce [8].

One particularly interesting and significant topic in SNA is that of discovering the most "central" or most "influential" nodes in a social network. Apart from the classic centrality measures (degree, closeness, shortest-path betweenness), the notion of *k-shell* has attracted the attention of "data/network" scientists. It was first proposed in [14] and found wide application in areas such as Internet topology modeling [4], detection of influential spreaders [2], discovery of communities [1], etc.

In a straightforward implementation of the k-core decomposition algorithm, we need to perform recursive deletions of all vertices and edges incident with them, but efficient versions of the basic algorithm do exist for various settings.

There are two categories of algorithms depending on whether the graph is dynamic (slowly or fast changing) or static (known in advance and not changing). The literature on k-shell decomposition for dynamic graphs includes algorithms that are able to handle only slowly changing graphs when they fit entirely in main memory [9], for processing in small clusters with the type of distributed algorithms described in [11] and for graphs whose topology is acquired in a streaming mode [13]. For static networks, when the entire graph can be stored in main memory, the core decomposition of a graph can be done in time *O(num_of_edges)* due to [3]. For larger graphs that have to be stored in secondary storage the techniques described in [5] can be used. Moving on to progressively larger networks that cannot fit into a single machine, the exploitation of a very small cluster for the k-shell decomposition of a network can be done as in [12].

Clearly, none of the aforementioned solutions is appropriate for the type of infrastructure operated by modern Internet companies such as Google, Yahoo, LinkedIn, Facebook and Twitter. These Internet giants are operating huge data centers with clusters comprised of several thousand low-cost machines. These clusters are usually programmed by MapReduce-type frameworks.

[2] http://hadoop.apache.org

III. PROPOSED DISTRIBUTED ALGORITHM

A. Background

K-shell decomposition of a graph is performed iteratively. The first step involves removing all degree-1 nodes, along with their link, and indexing these as k=1. In the resulting graph, all nodes of degree 1 are also considered to have k=1 and are again pruned. The process is repeated until there are no nodes of degree 1. Similarly, all nodes with i or fewer connections are iteratively removed; these nodes are indexed as k=i. The output of the k-shells algorithm is a single number for each node: its core assignment.

Let us mention that a node is said to have k-coreness if it belongs to k-core but not to (k+1)-core. A k-shell is composed of all nodes that have k-coreness. However, it is common to use them alternatively although their definition differs. Figure 1 illustrates a sample graph and its decomposition into shells.

B. The MR-SD algorithm

In this section we present the details of the proposed algorithm (Figure 2). Algorithm 2 (running on the master node) is the driver routine reading the input and deciding if the decomposition reached its end; Algorithm 1 is an auxiliary one, and the MR-SD algorithm (running on slaves) is the routine performing the k-shell decomposition. Our Hadoop cluster consists of a collection of slave nodes and one master node. From here on let $node_c$ denote a node of the cluster, either a slave or the master node. The base idea of our algorithm is that each $node_c$ is responsible for performing the pruning phase of the k-core decomposition of a collection of nodes of a given network graph and only some $nodes_c$ for combining the intermediate results.

Our algorithm computes the k-cores starting by finding out the 1-hop neighborhood of each node. This is achieved by handing an independent and random chunk of the actual graph to each $node_c$.

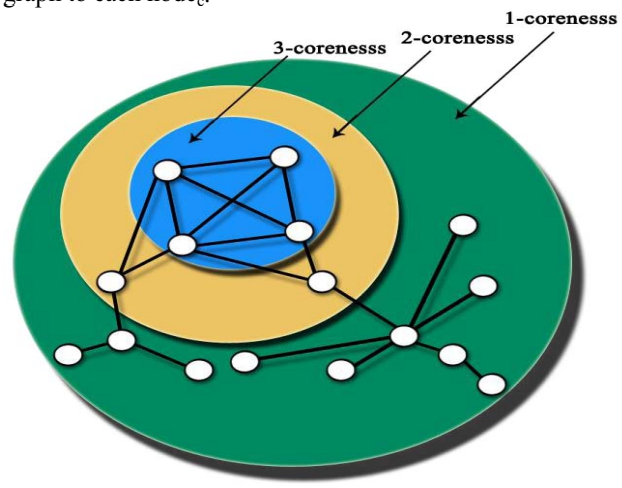

Figure 1k-shell decomposition of a simple graph

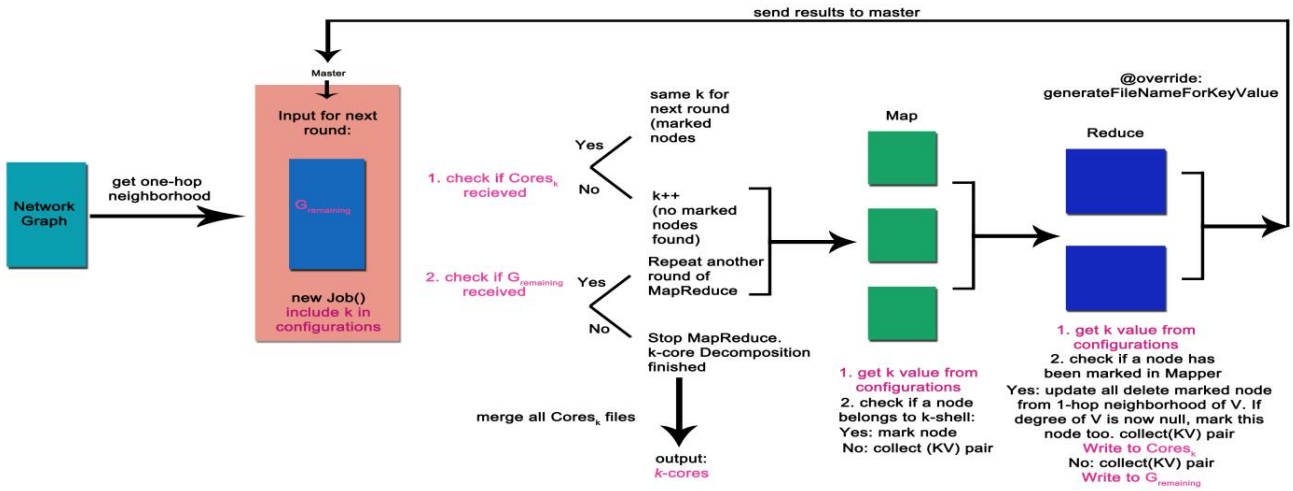

Figure 2 Flow of the MR-SD algorithm

As shown in Algorithm 2, a Map task is now forced. At this point, each node is considered as the output key and each neighbor as a single value. When a Reducer receives a key-value pair, it groups all values (*V*).

Algorithm 1: computeOneHopNeighborHood(NetworkGraph G) A Map-Reduce pair to compute the one-hop neighborhood for each node in G

```
Mapper:
  on map do
      for each KV pair do
          K ←nodeId
          V ←neighbor;
          collect(K, V);
  end map

Reducer:
  on reduce do
      for each KV pair do
          collect(K,V);
  end reduce
```

The intuition behind MR-SD algorithm is that each $node_c$ can only process part of the network graph and that value *k* (value of the *k*-core) must be disseminated efficiently. Described in this way, the master node of the cluster is now responsible for updating the *k* value during the pruning rounds and for its propagation. Initially, the master sets *k* equal to 1, configures a Map-Reduce job, and announces *k* to each worker node. We are now ready to distribute computations to the slaves.

The master node maintains the following variables to ensure the progress of the algorithm and detect termination:

- *k* is an integer representing the value of the *k*-coreness we currently examine.
- $Cores_k$ is a variable containing the nodes that should be included in the current *k*-shell.
- $G_{remaining}$ is a variable that represents the remaining network graph after a pruning round is performed, i.e., one execution of the MR-SD algorithm.
- G_{in} is a variable that contains the actual network graph provided by the user.

Algorithm 2: Driver Routine executed to coordinate the job and detect termination of the k-core decomposition process

```
on initialization do
    configure(Job);
    G_remaining ← computeOneHopNeighborhood(G_in);
    k ←1;
end initialization

repeat until each node has k-coreness
    configure(Job);
    <Cores_k, G_remaining> ← ALGORITHM (G_remaining);

    if Cores_k not received then
        k ← k+1;
    if G_remaining not received then
        k-cores ← mergeIntermediate(Cores_k);
return k-cores;
```

Before a slave starts a Map task, it must first retrieve the *k* value. Since the master node is the only node of the cluster responsible to update *k* and only at the end of each job, we guarantee that the value of *k* that a slave receives is always up-to-date. As a next step, $node_c$ counts the one-hop neighbors (i.e., degree) of a node which are stored in *V* from previous MapReduce process. Since this information is now available, the slave has to check if the degree of the node under consideration has shrunk below the *k* threshold or is equal to *k*. If this should happen, the node is marked by the slave so that the Reducer can include the node in the current *k*-shell and delete it from the remaining network graph. Moreover, some information is attached (*attachedInfo* variable in MR-SD algorithm) to all nodes that are included in its one-hop neighborhood so that the Reducer can exclude the current node from their neighborhood. However, if the preceding check indicates that the degree of the node exceeds the *k* limit then the node's id is collected along with its one-hop neighbors; the node id is the Key and the one-hop neighbors the Value of the *KV* pair collected and send to the Reducer.

At the beginning of each Reduce task, the slave follows the same protocol as the Mapper; *k* value becomes available to the worker node. Whenever a Reduce is performed, *KV* pairs are received. From previous stage of the MR-SD

algorithm *K* represents the node's id and *V* stores its one-hop neighborhood. There are now three possible scenarios we examine below:

1. node *K* was marked in preceding Map task.
2. node *K* is not marked by previous Mapper.
3. node *K* comes coupled with additional information which says that one or more neighbors where pruned in this pruning round.

In Scenario 1, if a Reducer receives a node that was marked during the Map task as *K*, i.e., the degree of the node is lower than *k*, it indicates that this node should be included in the current *k*-shell by the Reducer. In this case, we collect *K* and the *k* value as the *KV* pair in this round. Scenario 2 appears when the node that was examined in Map phase of the MR-SD algorithm does not meet the requirements to be included in the current *k*-shell. Although this seems to require a quite basic collect process of the *KV* pair, there has actually some further work to be done. The Reducer is now obliged to check if there is additional information attached to the *KV* pair received. If the node does not come with additional information then *KV* pair is collected. Scenario 3 appears in case where there is such information attached (*attachedInfo*). The information is about the node that has to be removed from the one-hop neighborhood of node *K*. When receiving this message we exclude the node from the one-hop neighbors set; the degree is therefore decreasing too. It is possible at this point that the node remains with an empty neighborhood; in this case the node has *k*-coreness and is therefore collected along with its *k*-core value. In case there exists at least one neighbor after the pruning, the node id and the neighbors ids are collected as *KV* pair and sent back to the master in the $G_{remaining}$ output file.

Termination and Progress

We need to discuss how we presume that Algorithm 2 both detects termination and converges to the correct *k*-cores or forces correctly another pruning round. There are few situations that appear:

After every node has been examined, i.e., all *KV* pairs have been received and processed, we have reached the end of the MR-SD algorithm for this round. The master, who is also responsible to terminate and progress the *k*-core decomposition as mentioned above, receives now either both output files or just one of them. Let us make this more specific:

- If new additions to the current *k*-shell appeared in Map and Reduce tasks, the master receives the $Cores_k$ file including all node ids that have been removed previously and the $G_{remaining}$ file with the remaining network graph. At this point *k* value stays the same for another round.
- If only $G_{remaining}$ is received, this indicates that *k* value has to be updated since no other node has been added to the current *k*-shell in previous round. The master has to increase *k* value and force another pruning round.

The MR-SD algorithm : A Map-Reduce pair that implements the pruning phase of the *k*-shell decomposition process

```
Mapper:
on map do
    k ← get(k);
    for each KV pair do
        degree ← ||V||;
        if degree ≤ k then
            node ← mark(node);
            for each v∈V do
                collect(v, attachedInfo);
                collect(node, k);
        else
            for each v∈V do
                V ← V + v;
                collect(K, V);
end map
Reducer:
on reduce do
    k ← get(k);
    for each KV pair do
        if attachedInfo received from Mapper then
            for each attachedInfo received do
                oneHopNeiborhood ←{V} – attachedInfo;
        degree ← ||V||;
        if degree == 0 then
            mark(node);
            Cores_k ← collect(K,k);
        else
            V ← oneHopNeiborhood;
            G_remaining ← collect(K,V);
end reduce
```

- Finally, if only the first file ($Cores_k$) is gathered by the master node, then termination is detected as no other nodes are left for examination. As a last step, all intermediate files that have been generated during previous rounds are now merged into one final output file (referred as *k*-cores in Algorithm 2).

In order to understand the way our proposed algorithm works, we conclude with an example of a pruning round performed with our proposed algorithm. The example graph is depicted in Figure 3 and the running of the algorithm in Figure 4.

Let us assume that we want to calculate the *k*-cores of the network in Figure 3. It is a small graph, so *k*-cores can be retrieved easily and obviously and one does not need a cloud to do this, but it is suitable for our example in order to understand the concept of Algorithm 2 and the MR-SD algorithm.

It is evident that the first job run should maintain to exclude nodes 6, 7, 8, 10 and 12 from the network and include them in the 1-shell.

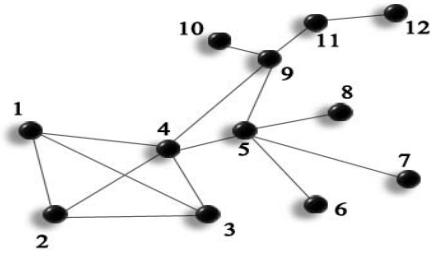

Figure 3 An example 12 node network

First, the master sets k value to 1 and the $G_{remaining}$, which is the description of the network in Figure 3, is split into independent chunks; assume that the calculation of one-hop neighborhood preceded the initialization. In map task of Mapper 1 there is no node that has degree less or equal to 1. So Mapper 1 collects simply the *KV* pairs. Mapper 2 and Mapper 3 on the other hand, find nodes with degree equal to one; nodes 8, 10 and12 and nodes 6 and 7 respectively. At this point they mark them so that the Reducer includes them in the 1-shell. Moreover, Mapper 2 and 3 attach information to nodes 5, 9 and 11 to let the Reducers know that they have to delete nodes 6, 7 and 8 from one-hop neighborhood of node 5, node 10 from one-hop neighborhood of 9 and node 12 from one-hop neighborhood of 11; the pink boxes represent the *attachedInfo*. When Reducers 1 and 2 receive *KV* pairs they check if the Key, which represents a node id, is marked. In case of nodes 1, 2 and 3 and 4 the reducers simply collect the *KV* pairs as they appear. For nodes 5, 9 and 11 all nodes that are mentioned in the additional attached information are deleted from their one-hop neighborhood; here nodes 6, 7 and 8 are removed from node's 5, node 10 from node's 9 and node 12 from node's 11 one-hop neighborhood. Now the Reducers check if node 5, 9 or 11 has a non-empty one-hop neighborhood. In case of node 6, 7, 8, 10 and 12 the Reducers collect the Key and k value as Value.

IV. Experimental Evaluation

In this section we provide the results of the experimental analysis; we describe the hardware, the real complex networks we have used and the obtained results.

A. The evaluation platform

We tested our algorithm on a cluster which consists of five nodes, one master node and four slaves. Each node is equipped with a disk space of 42GB and a 12GB RAM. Each node is an 8-core Intel CPU based blade running CentOS. The network switch which connects our network storages supports a 10-gigabit Ethernet connection. During each experiment there was no significant interference from other workloads.

B. The experimental setting

We have already mentioned that the proposed algorithm is the first one in the literature of k-shell decomposition that is based on the Map-Reduce paradigm; therefore, there are no competitors. We initially used the algorithms reported in [3] and [5], but they soon run out of memory for the large graphs and never terminated. Therefore, we do not present results for them. For the evaluation of the proposed algorithm, we used eight real social network graphs, which are described in Table 1. They were retrieved from https://snap.stanford.edu/ and from wiki.gephi.org/index.php/Datasets. Thus, we have used both small (a few thousands of nodes) and very large networks (a few millions of nodes).

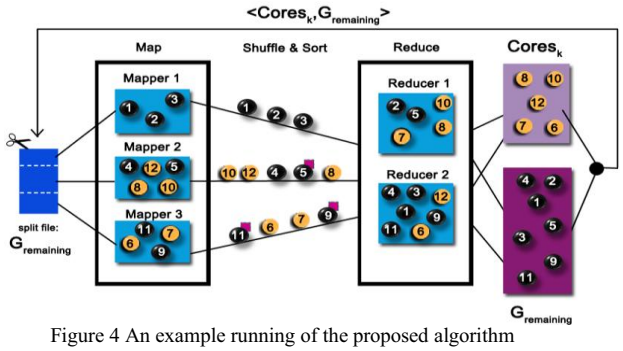

Figure 4 An example running of the proposed algorithm

Data sets				
Exper	**Social Network name**	**Num of nodes**	**Num of edges**	**Num of Jobs**
1	**Autonomous systems AS-733**	6474	13895	61
2	**DBLP collaboration network**	317080	1049866	360
3	**Autonomous systems by Skitter**	1696415	11095298	1305
4	**LiveJournal online Social Network**	3997962	34681189	3363
5	**Orkut online social network**	3072441	117185083	5918
6	**Amazon product co-purchasing network**	334863	925872	87
7	**Deaseasome**	7533	22052	118
8	**Protein Interaction Network in budding Yeast**	2361	7182	74

Table 1 Real social networks used for the evaluation

We have performed eight experiments, one for each dataset as shown in the first column of Table 2. These are also the datasets used in [1].

As performance measures of the efficiency of the proposed algorithm, we have used the following quantities:

- [Average CPU time spent (msec)]. It is the time spent *solely* by the CPU to perform the calculations. The obtained results are averaged over all Map-Reduce tasks.
- [Average Total Execution Time (sec)]. It is the total execution time of the k-shell decomposition process. It differs from the previous time in that it includes also the communication among master and slaves, the time required to store intermediate and final results and any other latency incurred. The obtained results are averaged over all Map-Reduce tasks.
- [Average Total committed heap usage (Bytes)]. It is the memory footprint of the algorithm.

For clarity of presentation purposes, we give the results concerning each dataset in a different plot.

	Average CPU time spent (msec)			Average Total Execution Time (sec)			Average Total committed heap usage (Bytes)		
Experiment	**Map**	**Reduce**	**Total (Job)**	**Map**	**Reduce**	**Total (Job)**	**Map**	**Reduce**	**Total (Job)**
Autonomous systems AS-733	2061.31	2715.74	4777.05	99.95	9.13	16.18	379584512	189792256	569376768
DBLP collaboration network	7696.08	5768.33	13464.42	104.79	10.30	19.33	381046693	192004460	573051153
Autonomous systems by Skitter	61377.72	61982.36	123360.08	158.55	71.48	90.83	397844599	199193282	597037881
LiveJournal online Social Network	79122.92	28758.76	107881.68	159.53	33.63	56.77	486663048	201195325	687858373
Orkut online social network	505243.72	304598.03	809841.74	311.50	299.78	339.79	1695512575	157003260	1852515835
Amazon product co-purchasing network	9803.56	6339.20	16142.76	200.13	10.75	20.06	376372495	191817092	568189587
Deaseasome	2223.05	2718.64	4941.69	175.80	8.87	16.26	379584512	189792256	569376768
Protein Interaction Network in budding Yeast	1559.32	2256.49	3815.81	156.41	8.82	15.00	379584512	189792256	569376768
Autonomous systems AS-733	2061.31	2715.74	4777.05	99.95	9.13	16.18	379584512	189792256	569376768

Table 2 Average resource consumption (Top) Average CPU time, (Middle) Average total execution time, (Bottom) Memory footprint

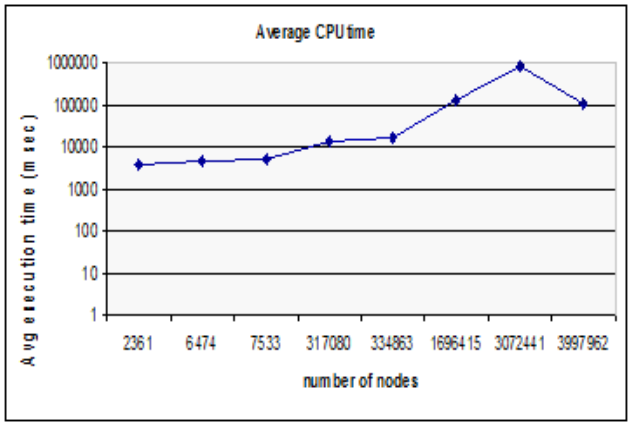

Figure 5 Average CPU time vs. number of nodes

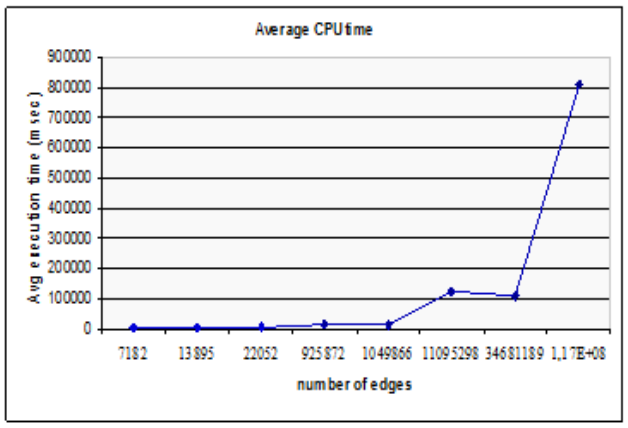

Figure 6 Average CPU time vs. number of edges

C. *The obtained results*

Now we are ready to present and comment the obtained results. First we will present the averages of the three aforementioned measures, and then present plots that concern their precise distribution.

1) *Average execution times and memory footprint*

Table 2 depicts the average values of CPU time, total execution time and memory footprint per Map job, per Reduce job and per dataset. Figure 5 and Figure 6 show that the execution time depends on the number of edges and not on the number of nodes. This is expected because the number of edges is related to the density of the networks, which defines the number of rounds that the algorithm will execute.

2) *Impact of the number of VMs*

Recall that we experimented with a small cluster. This has consequences on the performance of each worker node. We observed that when running jobs on a small cluster combined with very large-sized network graphs some of the $nodes_c$ are obliged to run more than one map task. This case for instance occurred when we decomposed the social network in experiment 5; here the max number of Map tasks exceeded 30 at the beginning. This result corresponds to more than 7 map tasks per slave during the first rounds of Algorithm 2 and the MR-SD algorithm and therefore a greater time. For small to medium sized graphs no significant workload appeared. However, our proposed approach manages every time, even with great workload, to retrieve the *k*-cores. It is obvious that having a larger infrastructure a better performance can be achieved since

map jobs are more efficiently placed over the cluster. Results can be seen in Figure 6 (averages) and Figure 7 (per experiment). In Figure 7, the x-axis in each plot depicts the total committed heap usage per job (in MBs), and the y-axis depicts the total CPU time spent per job (in msecs). In general, the reduce tasks cost far less in computation time than the respective map tasks, which is expected since the reducers perform mainly aggregation functions.

3) Impact of Network density

The performance of our algorithm depends on the network density, i.e., the number of network edges. Figure 6 shows that the density impacts exponentially the execution time. This is expected because the number of jobs increases with the number of edges.. In Table 2 we can see in details the performance of the algorithm for each data set. In particular, we emphasize the impact of the network density when comparing the results of CPU time of dense networks (e.g., experiment 5) with those of more sparse social networks (e.g., experiment 8).

4) Impact of the machine load

The processing of a vast network is difficult until a large number of nodes are pruned. This means that during the first pruning rounds, we observe greater resource demands. Indeed, CPU time measured is greater and a larger heap size is required. As shown in Figure 8, the workload demands for large network graphs require an increasing processing power while the heap size remains actually almost stable during this interval. The same can be observed for small to medium networks but to a much lesser extent. Another interesting outcome is that reduce tasks tend to be more time consuming in the first 20% to 45% of the pruning rounds than in the last ones. In fact, the execution time seems to follow a decreasing exponential trend for very large graphs. Contrary to large networks, small or medium graphs have a rather flat execution time during the reduce phase. The latter observation is depicted in Figure 8.

V. CONCLUSIONS & FUTURE WORK

The problem of k-shell decomposition of an online social network is a significant task involved in social network analysis; it can be used for the discovery of influential spreaders, for community detection, etc. In this article, motivated by the unsuitability and lack of existing solutions to deal with this problem when the size of network is large and the computation takes place in huge clusters such as those deployed by current Internet giants (Google, Yahoo, LinkedIn), we designed a MapReduce-based distributed k-shell decomposition algorithm for social networks. We addressed the challenges involved in the design of a parallel/distributed version of a graph decomposition technique, which is highly sequential in its nature, and provided an effective and efficient algorithm able to scale to millions of graph nodes and edges. We implemented the proposed algorithm in the Hadoop middleware, and assessed its performance for eight real social networks of varying size and density. We investigated the performance of the algorithm in terms of pure CPU time, of total execution time and memory footprint. We recognized its virtues and suitability for modern distributed environments.

As a future work, we plan to work on some aspects of it that involve the communication among slaves so as to optimize it and also to develop a variation of the algorithm for annotated networks, e.g., weighted. Finally, we plan to perform experimentation of our proposed algorithm on a MapReduce environment of a major cloud service provider in order to highlight its scalability and achieve further speedup.

ACKNOWLEDGMENT

The authors acknowledge the support of the THALES project "Optimal Control of Self Organized Wireless Networks", co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF).

REFERENCES

[1] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, O. Ulusoy, "Multi-resolution social network community identification and maintenance on big data platform", *Proc. of IEEE BigData*, pp. 102-109, 2013.

[2] P. Basaras, D. Katsaros, L. Tassiulas, "Detecting influential spreaders in complex, dynamic networks", *IEEE Computer magazine*, vol. 46, no. 4, pp. 26-31, 2013.

[3] V. Batagelj and M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks", University of Ljubljana, Department of Theoretical Computer Science, Ljubljana, Slovenia, *Preprint series*, vol. 40, 2002. Available at http://arxiv.org/abs/cs.DS/0310049.

[4] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, E. Shir, "A model of Internet topology using *k*-shell decomposition", *Proceedings of the National Academy of Sciences*, vol. 104, no. 27, pp. 11150-11154, 2007.

[5] J. Cheng, Y. Ke, S. Chu, M.T. Ozsu, "Efficient core decomposition in massive networks", *Proc. of IEEE ICDE*, pp. 51-62, 2011.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", *Proc. Of USENIX OSDI*, pp. 137-150, 2004.

[7] A. Garas, F. Schweitzer, S. Havlin, "A k-shell decomposition method for weighted networks", *New Journal of Physics*, vol. 14, 2012.

[8] A. Kala Karun and K. Chitharanjan, "A review on Hadoop — HDFS infrastructure extensions", *Proc. of IEEE ICT*, pp. 132-137, 2013.

[9] R.-H. Li, J.X. Yu, R. Mao, "Efficient core maintenance in large dynamic graphs", *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2014. Available at http://arxiv.org/abs/1207.4567

[10] J. Lin and C. Dyer, "*Data-Intensive Text Processing with MapReduce*", Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers, 2010.

[11] D. Miorandi and F. de Pellegrini, "K-shell decomposition for dynamic complex networks", *Proc. of WiOpt*, pp. 488-496, 2010.

[12] A. Montesor, F. de Pellegrini, D. Miorandi, "Distributed *k*-core decomposition", *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 2, pp. 288-300, 2012.

[13] A.E. Sariyuce, B. Gedik, G. Jacques-Silva, K.-L. Wu, U.V. Catalyurek, "Streaming algorithms for k-core decomposition", *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433-444, 2013.

[14] S.B. Seidman, "Network structure and minimum degree", *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

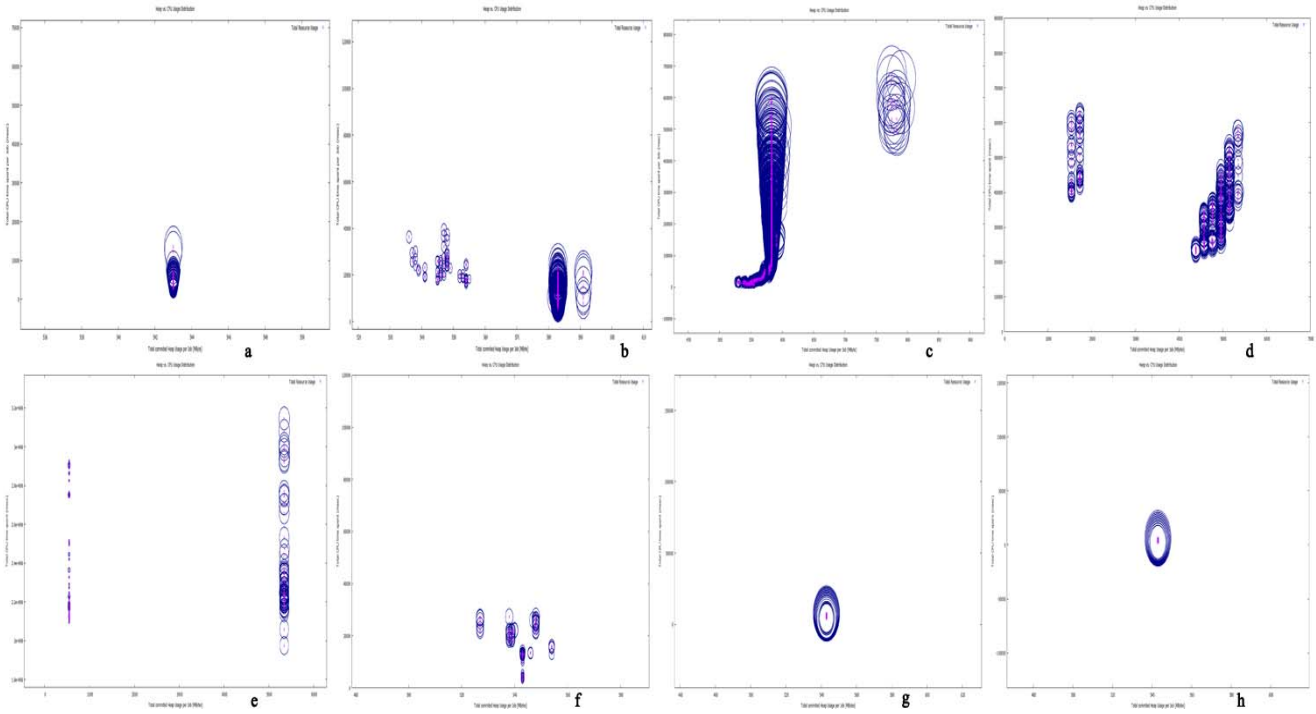

Figure 7 Heap vs. CPU usage distribution. The size of the radius of each circle is relative to the square root of the ratio of CPU and Heap usage multiplied by a constant number c. a) Experiment 1, c=0.1, b) Experiment 2, c=0.1, c) Experiment 3, c=0.8, d) Experiment 4, c=5, e) Experiment 5, c=5, f) Experiment 6, c=0.2, g) Experiment 7, c=2, h) Experiment 8, c=2

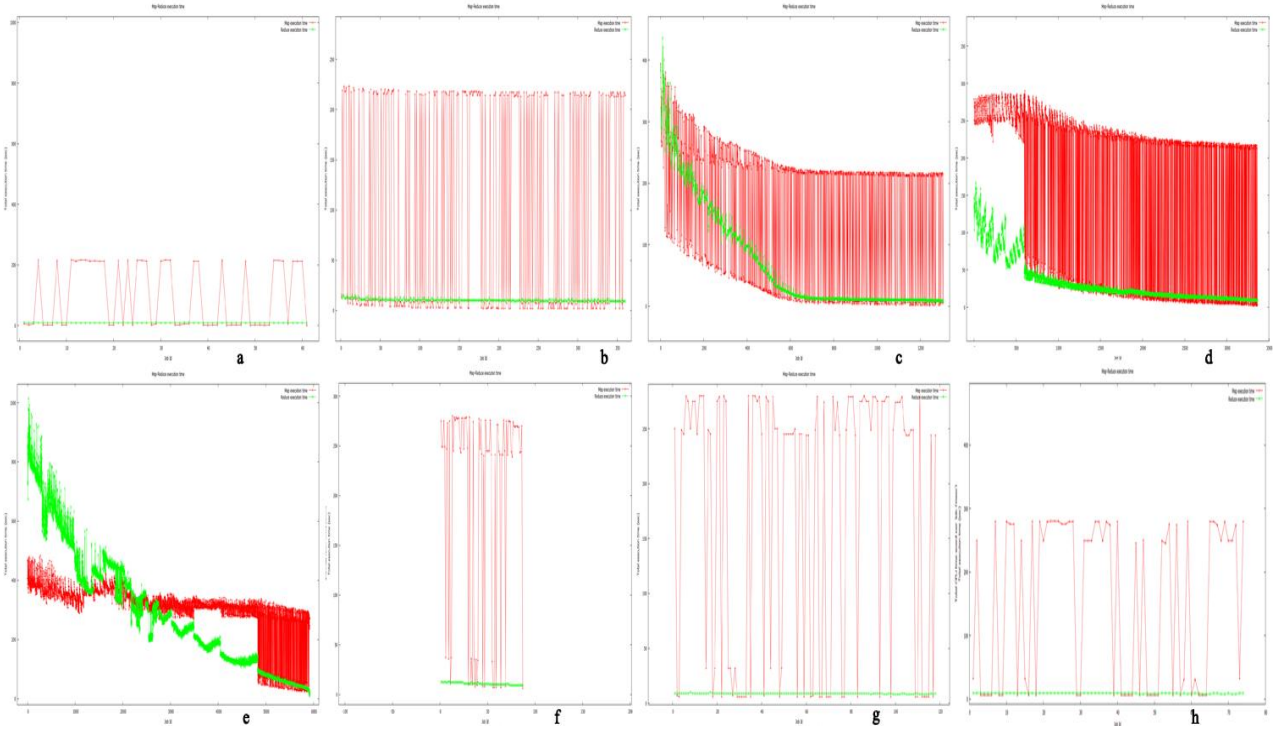

Figure 8 Map and Reduce task execution time (red is used for Map tasks and green for Reduce tasks) a) Experiment 1, b) Experiment 2, c) Experiment 3, d) Experiment 4, e) Experiment 5, f) Experiment 6, g) Experiment 7, h) Experiment 8