A Distributed k-Core Decomposition Algorithm on Spark*

Aritra Mandal[1] and Mohammad Al Hasan[2]

***Abstract*— k-core decomposition of a graph is a popular graph analysis method that has found widespread applications in various tasks. Thanks to its linear time complexity, k-core decomposition method is scalable to large real-life networks as long as the input graph fits in the main memory. For graphs that do not fit in the main memory, external memory based approach or distributed solution based on iterative MapReduce platform have been proposed. However, both external memory solution and iterative MapReduce based solution are slow due to their high disk I/O cost. In this paper we propose, Spark-kCore, a distributed k-core decomposition algorithm, which runs on Spark cluster computing platform. Using think-like-a-vertex paradigm, the proposed method utilizes a message passing paradigm for solving k-core decomposition, thus reducing the I/O cost substantially. Experiments on 15 large real-life networks show that our method is much faster than the existing k-core decomposition solutions.**

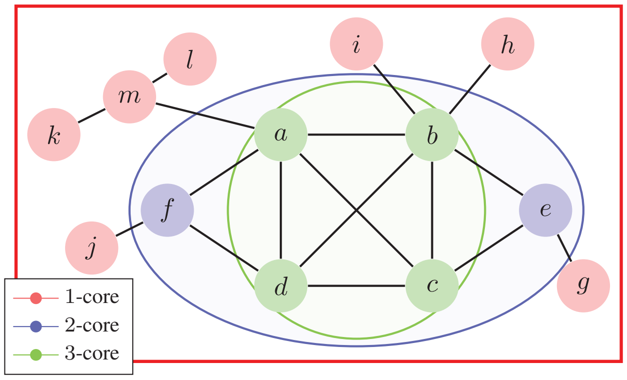


Fig. 1: A toy graph and its k-core decomposition

I. INTRODUCTION

Structural analysis and mining of large and complex graphs is a well studied research direction having widespread applications in graph clustering, classification, and modeling. There are various methods for structural analysis of graphs including, the discovery of frequent subgraphs or network motifs [1], [2], counting triangles or graphlets [3], or finding highly connected subgraphs, such as cliques and quasi-cliques [4]. The above tasks help to identify small subgraphs which are building blocks of large real-life graphs. Besides, they are used for solving tasks such as community discovery, building features for graph indexing or classification, and graph partitioning. Unfortunately, the algorithms for solving the majority of the above tasks are very costly, which makes them not-scalable to large real-life networks. So, scalable tools for structural analysis of massive networks are of high demand to meet the need of today's graphs that have millions of vertices and edges.

In recent years, k-core decomposition of graphs has emerged as an effective and low-cost alternative for structural analysis of large networks. Till date k-core decomposition has been used for studying Internet topologies [5].k-coere also finds usage in study hierarchical, and self-similarity in Internet graph [6]. Lately k-core decomposition is being used for structural composition of brain networks [7], for identifying influential spreaders in complex networks [8], for building data structures for graph clustering [9], and for computing lower bound to prune search space while searching for maximum cliques [10]. The salient feature that enables k-core decomposition as a leading structural analysis tool is its linear runtime which makes it scalable to large real-life networks with millions of vertices and edges.

k-core decomposition of a graph G is partitioning the vertices of G based on its "coreness"; in this partitioning, vertices belonging to the core of a given k value form the k-cores of G. A k-core of G is an induced subgraph of G such that all nodes of that subgraph have a degree at least equal to k. Informally, k-cores can be obtained by removing all vertices of degree less than or equal to k, until the degree of all remaining vertices is larger than or equal to k. By definition, k-core partitions are concentric, i.e., if a node belongs to k-core for a given $k = K$, it also belongs to the k-core for all k values from 1 to K; thus the coreness of a vertex is determined by the largest k value for which the vertex participates in a k-core. Vertices belonging to the largest core value occupy the central position of the network and thus they play a larger role in the composition of a network. See Figure 1 for a graph in its k-core decomposed form. The largest core in this graph is a 3-core consisting of the vertices a, b, c and d.

Initial research on k-core were in graph theory, k-core was studied in relation to the study of the degeneracy of a network. Linear time algorithm to obtain the degeneracy of a network has been developed decades ago [11], using such an algorithm k-cores of a graph can be obtained. However, in recent years, there has been a renewed interest in developing efficient and practical algorithms explicitly for k-core decomposition by researchers in the domain of complex networks, data mining, and life sciences. In this direction, Batagelj et al. [12] authored an influential work; they proposed a $O(m)$ algorithm for core decomposition of

[1]Aritra Mandal is with Department of Computer Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street `amandal at iupui.edu`

[2]Mohammad Al Hasan is with Department of Computer Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street `alhasan at iupui.edu`

a network, where m is the number of edges in the network. This is a sequential algorithm running on a single-memory machine. The algorithm works well as long as the entire input graph fits in the main memory of a network, which unfortunately is not the case for today‘s gigantic networks, such as Internet graph, and social networks. In some cases, the network may fit in the main memory of a machine, but the network can be inherently distributed over a collection of hosts, making it difficult to move the entire graph in a single-memory machine. So, in recent years, there are several works for obtaining effective distributed algorithms for k-core decomposition on various platforms, like Pregel [13], GraphLab [14], and GraphChi [15]. Algorithms that run on external memory, such as, EMCore, has also been proposed [16].

Apache Spark is an open source bigdata processing engine which unifies batch, streaming, interactive, and iterative processing of large and diverse data. Spark uses transformations on in-memory resilient data structures called RDD's. With it's extensions, like SparkSQL, SparkML and GRaphX, Spark can perform a multitude of complex tasks, like executing complex SQL queries, training machine learning models, and processing large complex graph mining methodologies. Specifically, for graph processing, Pregel-like iterative algorithms are very slow on MapReduce based distributed engines due to a high number of disk I/O and slow access speed. On the other hand, Spark is more optimized for iterative processing and is reported to be 100 times faster on such tasks than traditional MapReduce. Due to the benefits of spark and its capability to scale horizontally, the community has demanded for an implementation of k-core decomposition on Spark through Spark feature request [1]. Unfortunately, no k-core decomposition implementation on Spark is available yet.

In this paper, We propose a distributed k-core algorithm and its implementation, Spark-kCore. Spark-kCore runs on top of Apache Spark's GraphX framework. The implementation follows the "think like a vertex" paradigm, which is an iterative execution framework provided by Pregel API of GraphX. We compare Spark-kCore with two other k-core decomposition algorithms: EMCore [16] and Graphlab's k-core implementation [14]. Experimental results on 15 large real-life graphs show that Spark-kCore is substantially superior to the competing algorithms. We also present experimental results which demonstrate the runtime behavior of Spark-kCore over various input graph parameters, such as the number of edges, and the size of maximum k-core. We also made the source of Spark-kCore available on Github for the community to use [2].

[1]`https://issues.apache.org/jira/browse/SPARK-16976`

[2]`https://spark-packages.org/package/DMGroup-IUPUI/Spark-kCore`

II. BACKGROUND

A. *k-Core*

Let $G(V, E)$ is a graph, where V is the set of vertices and E is the set of edges. G is undirected, simple graph with no self-loop. For a vertex $u \in V$, we use $\mathcal{N}(u)$ to represent the set of vertices which are adjacent to u. Also, we use $deg(u)$ to represent the size of $\mathcal{N}(u)$, i.e., $deg(u) = |\mathcal{N}(u)|$. Given G, an undirected, simple graph with no self-loop, k-core of G, denoted by $C_k(G)$, is a maximal connected subgraph $H \subseteq G$ such that $\forall u \in H \ deg(u) \geq k$ if it exists. The core number of a vertex, $core(v)$, is the largest value for k such that $v \in C_k(G)$. The maximum core number of a graph G, $C_{max}(G)$, is defined as $\max_{\forall v \in G} \{core(v)\}$. In graph theory, an undirected graph G is called k-degenerate, if for every induced subgraph $H \subseteq G \ \exists v \in H$ such that $deg(v) \leq k$. If a graph has a (non-empty) k-core, the degeneracy value of that graph is at least k.

B. *Pregel Paradigm*

Pregel [13] paradigm of large scale graph processing was introduced by Goolge. This paradigm has a "think like a vertex" approach for a graph analysis task. Pregel has two different stages of operation. It has an initialization stage which is executed once at the beginning of the execution. The initialization function sets the value of each vertex to a default value. The next stage is an iteration stage; in each iteration, all the nodes execute three operations. Each node collects and merges all the messages it has received from its neighbors; it updates its value based on the messages it has received and sends a message out to all its neighbors. The Pregel paradigm fits very well for a distributed k-core decomposition algorithm, which we will discuss next.

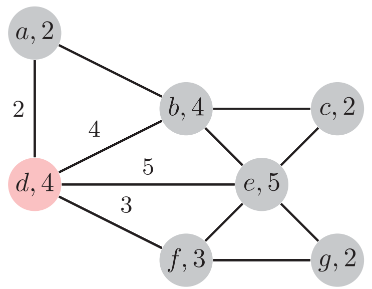
III. METHODS

A. *Distributed k-core algorithm*

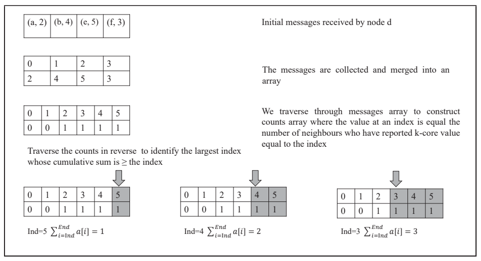
The primary assumption of a distributed k-core decomposition algorithm is that the input graph may or may not fit in the main memory of a single processing unit. Another assumption is that the listing of nodes and edges of the graph are stored in distributed manner across different machines in a cluster. Mostly, all the existing distributed k-core methods follow a vertex centric protocol which was initially presented by Montresor et al. [17]. The distributed algorithm is based on the property of locality of the k-core decomposition method. The property of locality states that for $\forall u \in V$, $core(v) = k$ if and only if

1) there exist a subset $V_k \subseteq \mathcal{N}(u)$ such that $|V_k| = k$ and $\forall u_i \in V_k, \ core(u_i) \geq k$;
2) there exist no subset $V_{k+1} \subseteq \mathcal{N}(u)$ such that$|V_{k+1}| = k+1$ and $\forall u_i \in V_{k+1}, \ core(u_i) \geq k+1$.

Thus, the core value of a vertex u, $core(u)$, is the largest value k such that the vertex u has exactly k neighbors whose core value is greater than or equal to k. The property of locality enables the calculation of core value of a node based on the core value of its neighbors.



(a) Messages addressed to d by its neighbors



(a, 2)	(b, 4)	(e, 5)	(f, 3)

Initial messages received by node d

0	1	2	3
2	4	5	3

The messages are collected and merged into an array

0	1	2	3	4	5
0	0	1	1	1	1

We traverse through messages array to construct counts array where the value at an index is equal the number of neighbours who have reported k-core value equal to the index

Traverse the counts in reverse to identify the largest index whose cumulative sum is ≥ the index

0	1	2	3	4	5
0	0	1	1	1	1

Ind=5 $\sum_{i=Ind}^{End} a[i] = 1$

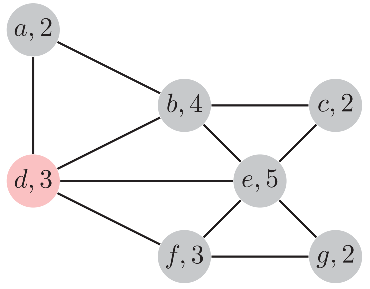
0	1	2	3	4	5
0	0	1	1	1	1

Ind=4 $\sum_{i=Ind}^{End} a[i] = 2$

0	1	2	3	4	5
0	0	1	1	1	1

Ind=3 $\sum_{i=Ind}^{End} a[i] = 3$

(b) Processing at node d



(c) Graph with k-core value of d updated after one iteration

Fig. 2: The k-core update flow for one iteration for the vertex d

An obvious upper bound of the core value of each node is its own degree value. So, in a vertex-centric k-core decomposition algorithm, each node initializes its core value with the degree of itself. Each node (say u) then sends messages to its neighbors $v \in \mathcal{N}(u)$ with the current estimate of its (u's) core value. For an undirected graph with m edges, there can be at most a total of $2m$ messages that have been sent during a message passing session. Upon receiving all the messages from its neighbors, the vertex u computes the largest value l such that the number of neighbors of u whose current core value estimate is l or larger is equal or higher than l, i. e., $l = \arg\max_{1 \leq i \leq core(u)} \left\{ \left(\sum_{v \in \mathcal{N}(u)} \mathbb{I}_{core(v) \geq i} \right) \geq i \right\}$.

The above l value can be computed easily by gathering the current estimate of neighbors' core values from the messages and use those to build a frequency array. In this array, the element indexed by i is the number of u's neighbors for which the current core estimate is exactly i. Then the frequency array is traversed from the largest index; the first index for which the cumulative sum of the array from the end up to (including) that index is greater than or equal to the index value is set as the updated core value of u. Once an updated estimate of the core is obtained, u sends out a message to all its neighbors with its updated core value. This receive-merge-update-broadcast iteration occurs until there are no more messages to process in any node in the graph.

In Figure 2, we show one iteration of update operation on core value estimate of the vertex d for the graph. In this graph, the number associated with the node label is the current estimate of the core value of that node. As we can see, the initial estimate of core value for d is 4 which is d's degree value. In 2(a), we show the messages carrying the current core value of the neighbors being received by d along the edges of the graph. Now, in 2(b), the messages from d's neighbors are arranged in a frequency array and the largest index for which the cumulated sum from the end up to (including) that index is higher than the index value is 3. So, 3 is the updated core value estimate of vertex d, which is correctly reflected in Figure 2(c).

B. Distributed k-core Implementation on Apache Spark

In this section we go into details of the implementation of the distributed k-core algorithm on Apache Spark as was explained in section III-A. We use the GraphX engine of Spark to load and process graphs. We start by explaining a few details about the GraphX engine which is relevant to our implementation.

GraphX is a graph processing engine which allows a graph like manipulation on top of the native Spark RDDs. All Graphs in GraphX are directed. By default, edge direction is from a node with lower nodeId to a node with higher nodeId. The edges are stored in an Spark RDD. For an edge, GraphX also supports triplet view. In this view, an edge is represented as a triplet, which joins two nodes with an edge along with all properties of the nodes and the edges stored into an $RDD[EdgeTriplet[VD, ED]]$. GraphX also provides us Pregel API which takes a custom merge, update, propagate function and iteratively execute them on each node till a user-defined termination condition is met. More details on the GraphX framework can be found here [18].

From the above explanation, we can see that in GraphX engine every edge is directed. But the Pregel framework will process only messages inbound to a node, which will lead to an incorrect k-core algorithm on undirected graphs. This is due to the fact that for k-core computation logic needs the messages to traverse in both directions of an edge. We can handle this problem in two different ways which we discuss below.

For each pair of nodes connected by an edge, we can enforce the creation of an edge in the opposite direction. This will solve the above limitation of Pregel framework in GraphX. But with this approach, we will need twice the amount of memory to store the extra edges. The other approach which we use for the implementation in this paper is using the triplet view of the graph. In the send message function rather than sending the message to all outbound edges, we utilize the triplet to put the message in inbound link of both the nodes in the triplet and thus forcing Pregel framework to pick up the update information of the node irrespective of the direction of the edge.

Algorithm 1, 2, and 3 provides a pseudo-code of the required functions performed by each node. It follows the property of locality that we have discussed above. This property of locality enables the calculation of core value of a node from the core estimate of its neighbors in an iterative fashion, which makes it a think-like-a-vertex based distributed algorithm.

Algorithm 1 KcoreSpark - Merge

```
1: procedure MergeMessage($Str\ msg1, Str\ msg2$)
2:     return $msg1.Concatenate(msg2, delimiter)$
```

Algorithm 2 KcoreSpark - Update

```
1: procedure UpdateNode($Node\ u, Str\ msg$)
2:     $msgArray \leftarrow msg.Split$(delimiter)
3:     for all $m \in msgArray$ do
4:         if $m \leq u.kcore$ then
5:             $count[m] + +$
6:         else
7:             $count[u.kcore] + +$
8:     for $i := k$ to 2 do
9:         $curWeight \leftarrow CurWeight + count[i]$
10:        if $curWeight \geq i$ then
11:            $u.kcore \leftarrow i$
12:            break
13:    return $u$
```

Algorithm 3 KcoreSpark - Propagate

```
1: procedure SendMsg($EdgeTriplet\ triplets$)
2:     $srcVertex \leftarrow triplet.getSrcAttr()$
3:     $destVertex \leftarrow triplet.getDstAttr()$
4:     $I \leftarrow new\ MsgIterator()$
5:     $I.append(triplet.dstId, srcVertex)$
6:     $I.append(triplet.srcId, destVertex)$
7:     return $I$
```

The upper bound of k-core of each node is the degree of the node so to begin with each node is initialized with k-core value equal to its $degree$. Each vertex u runs the procedure MergeMessages followed by the UpdateNode procedure, if the core value of u is changed (reduced), the updated core value estimate is sent to all of u's neighbors by the SendMsg subroutine. In the MergeMessage subroutine, u gathers all messages collected from its neighbors into a single message. The UpdateNode procedure traverses through all the collected messages and keeps a count of each element in the array whose value is smaller than the current core value of u in a counts array (Algorithm 2 Line 3 to 7). The count array is traversed in reverse and counts are summed up. The largest index whose cumulative count is greater than or equal to the index values is set as the updated core value of the node (Algorithm 2 Line 8 to 11). In the third phase of operation, the SendMsg procedure sends out a message to a nodes neighbors if its core value of the node is updated. This receive-merge-update-broadcast iteration occurs until there are no more messages to process in any node in the graph.

The time complexity of this algorithm is bounded by $1 + \sum_{u \in V}[deg(u) - core(u)]$ [17], which is equivalent to the summation of the number of updates that each node makes to reach to its actual core value. For the measurement of this time complexity, we consider the fact that Pregel iterations are synchronous i.e during each iteration each node receives all messages addressed to it, calculates its new core value, and sends its updated core value to all its neighbors.

IV. Experimental Results

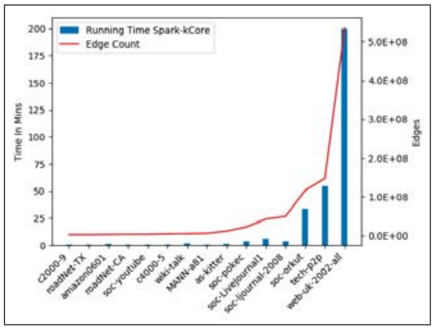
Setup: Spark-kCore is implemented in Scala and the experiments are conducted on a cluster of 8 machines, each having Intel i7, 2.2Ghz CPU, and 16 GB RAM, running CentOS (Linux). The hard disk is Seagate Constellation ST2000NM0033-9ZM 2TB 7200 RPM.

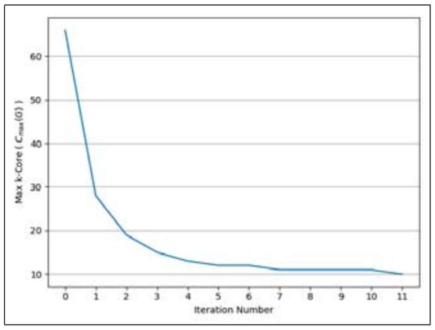
Datasets: We test Spark-kCore on publicly available SNAP datasets (`snap.stanford.edu`) and Network Repository datasets (`networkrepository.com`). We perform our analysis on the following fourteen graph datasets: as-kitter, soc-youtube, Amazon product co-purchasing network (amazon0601), Texas road network (roadNet-TX), California road network (roadNet-CA), Wikipedia Talk network (wiki-Talk), LiveJournal social network (LiveJournal), Soc-orkut, tech-p2p, MANN-a81, c4000-5, c2000-9, soc-Pokec, soc-orkut. The number of vertices, edges and the maximum core number of these graphs are available in Table I.

TABLE I: Table of results showing the No. of vertices, Edges, Maximum k-core, running time and Pregel iterations in Spark-kCore.

Dataset	Vertices	Edges	C_{max}(G)	**T**(mins)	Iters
as-skitter	1.7M	11.1M	111	1.3	26
soc-youtube	1M	3M	51	0.9	46
wiki-talk	2.4M	4.7M	131	1.7	50
amazon0601	0.4M	2.4M	10	1.1	10
roadNet-CA	2.0M	2.8M	3	0.75	10
roadNet-TX	1.4M	1.9M	3	0.6	10
MANN-a81	3.3K	5.5M	3280	0.5	3
c4000-5	4K	4M	1909	0.9	14
c2000-9	2K	1.8M	1758	0.4	8
soc-pokec	1.6M	22M	47	3.8	38
tech-p2p	5.7M	147.8M	856	55	70
soc-orkut	3M	117M	231	34	63
soc-ljournal-2008	5.3M	50M	427	3.9	5
soc-LiveJournal1	4.8M	42.8M	372	6.1	20

Competing Methods for Performance Comparison: For graphs which can fit in main memory we compare Spark-kCore's running time with that of Turi Graphlabs implementation of k-core decomposition which is based on [14]. Note that, our implementation is on distributed platform, but Graphlab implementation runs on a single machine, nevertheless this is an interesting comparison for graphs which are small enough to fit into main memory. In fact, for small files, distributed algorithms have an overhead of distributing and synchronizing, which a single system engine does not have. So, comparison on small graphs is actually unfair for Spark-kCore, yet we make this comparison to show the superiority of Spark-kCore over Graphlab implementation. We also compare our results with the EMCore algorithm presented by J. Cheng et al. [16]. We use the


(a)


(b)

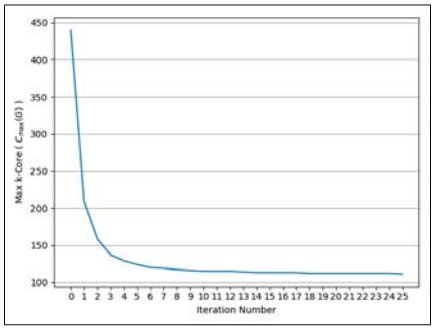

(c)

Fig. 3: (a) shows the comparison of running time with number of edges. (b) and (c) shows the change in k-core value for the amazon0601 and as-kitter graphs respectively

EMCore implementation given in [15]. We cannot compare with MapReduce implementation of k-core decomposition discussed in [19], because neither a publicly available implementation of this algorithm is available, nor the authors could provide their implementation.

A. *Spark-kCore's Runtime Behavior on Various Graph Metrics*

The runtime of Spark-kCore increases almost linearly with the number of edges. This is expected as the number of messages in the initial iterations of the execution of Spark-kCore is almost equal to the number of edges. This is due to the fact that during the initial iterations, for the majority of the vertices, their core value estimations have not yet been settled to their exact value. However, as iteration progresses, the number of messages drops as many nodes have their exact core values and they do not transmit any message. In Figure 3a, we show the execution time of Spark-kCore in a bar chart, where each bar corresponds to one of the graphs. The left Y-axis represents running time in minutes and the right Y-axis represents edge counts. The bars are sorted from left to right based on their running time. The line graph shows the edge count for each of the graphs represented by the bar. As we can see the execution time shows a trend of increasing almost linearly with the number of edges. But there are small variations to this trend for some graphs, which can be attributed to the distribution overhead of the framework.

B. *Convergence of Spark-kCore*

As part of the experiment, we also record the changes in the value of the max k-core ($C_{max}(G)$) with each iteration till the value converges. Initially, the max k-core value is equal to the maximum degree of the graph. Based on our experiment, we see that the value of max k-core drops very steeply in the first few small number of iterations to a value close to the actual max k-core value of the graph. After first few iterations, the rate of change in max k core value is slow till it converges. Figure 3b and 3c shows that change in max k-core value with each iteration for 2 graphs: amazon0601, as-skitter. The X-axis represents the number of iterations and the Y-axis represents the max k-core value of a graph for a given iteration. These results show that although it may take a large number of iterations to converge to the max k-core value, we can get a very close estimate of the max k-core value of the graph in a fraction of these iterations.

TABLE II: Running time comparison between Turi Graphlabs and Spark-kcore

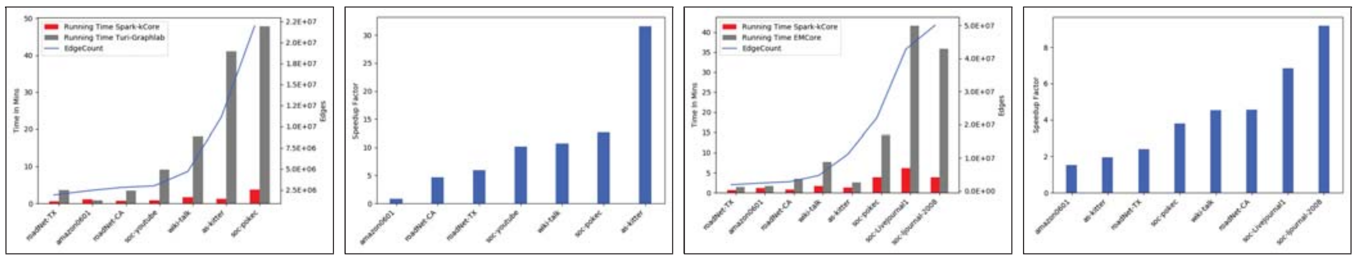
Dataset	$C_{max}(G)$	$\mathbf{T_{spark}}$(mins)	$\mathbf{T_{GraphLabs}}$(mins)
as-skitter	111	1.3	41
soc-youtube	51	0.9	9.1
wiki-talk	131	1.7	18.2
amazon0601	10	1.1	0.9
roadNet-CA	3	0.75	3.5
roadNet-TX	3	0.6	3.5
soc-pokec	47	3.8	47.5

C. *Runtime comparison between Spark-kCore and Turi Graphlab*

As mentioned above for this comparison we use graphs that fit in the main memory. Among the graphs that we use in this paper, 7 graphs qualified. The comparison results are shown in Table II. The results show that Spark-kCore is faster than the Turi Graphlabs, by a wide margin.

We also found out that the difference in running time of algorithms increases with increasing number of edges. We demonstrate this behavior in Figure 4a. In Figure 4a, the bars represents the running time for Spark-kCore and Graphlab. The line graph represents the edge count for the graphs in X-axis. The left Y-axis represents running time in minutes and the right Y-axis represents edge count.

With Spark-kCore we see a speedup of 4 to 32 times. For example k-core decomposition of soc-pokec on Spark-kCore took 3.8 mins and on graphlabs it took 47.5 mins resulting in 13X speedup. Although we have a distribution factor of 8, for large graphs we have a speedup much higher than 8. For smaller graphs the speedup falls to 4 times due to distribution overhead. Figure 4b shows the speed up of Spark-kCore. The Y-axis of the plot represents the speedup factor and the bars represent the speedup for the graphs sorted by the speedup factor.



(a) (b) (c) (d)

Fig. 4: (a) Graphlab Vs Spark-kCore running time comparison, (b) Speedup achieved by Spark-kCore over Graphlab, (c)EMCore Vs Spark-kCore running time comparison and (d) Speedup achieved by Spark-kCore over EMCore

D. Runtime Comparison between Spark-kCore and EMCore

As mentioned above we also compare the the running time of spark-kCore with EMCore implementation given in [15]. We compare the results on graphs which are medium to large in size. We run the comparison on 4 medium size graph like amazon0601, wiki-talk, roadnet-CA, roadnet-TX and two large graphs soc-pokec and soc-livejournal1. The comparison results are shown in Table III. The results show that spark-kcore is faster than EMCore. In Figure 4c the bars represents the running time for Spark-kCore and Graphlab sorted by the number of edges in the graph. The line graph represents the edge count for the graphs in X-axis. The left Y-axis represents running time in minutes and the right Y-axis represents edge count. The difference in execution time is small for medium sized graphs but for larger graphs the difference becomes substantial.

TABLE III: Running time comparison between EMCore and Spark-kcore

Dataset	$C_{max}(G)$	$\mathbf{T_{spark}}$(mins)	$\mathbf{T_{EMCore}}$(mins)
amazon0601	10	1.1	1.68
wiki-talk	131	1.7	7.71
roadNet-CA	3	0.75	3.42
roadNet-TX	3	0.6	1.5
soc-pokec	47	3.8	14.38
soc-livejournal1	372	6.1	41.7

Figure 4d shows the speedup achieved by the Spark-kCore for 7 different graphs. Although we are running on a distributed system with a distribution factor of 8 we don't get a speedup greater than 8 times with the graphs we tested because of the overhead of distribution, but we can see a trend that as the size of graph grows the speedup factor also increases suggesting that with larger files speedup factor will also increase.

V. CONCLUSIONS

In this work, we propose Spark-kCore, a Spark based distributed algorithm for k-core decomposition. The proposed method is scalable, and it runs on graphs that do not fit in the main memory of a computer. Our comparison with existing k-core implementation on other distributed platforms, such as GraphLabs shows that our method is significantly better than the existing methods.

REFERENCES

[1] U. Alon, "Network motifs: theory and experimental approaches," *Nat Rev Genet*, vol. 8, no. 6, pp. 450–461, Jun. 2007.
[2] T. K. Saha and M. A. Hasan, "Finding network motifs using MCMC sampling," in *Complex Networks VI - Proceedings of the 6th Workshop on Complex Networks CompleNet 2015, New York City, USA, March 25-27, 2015*, 2015, pp. 13–24.
[3] M. Rahman, M. A. Bhuiyan, M. Rahman, and M. A. Hasan, "GUISE: a uniform sampler for constructing frequency histogram of graphlets," *Knowl. Inf. Syst.*, vol. 38, no. 3, pp. 511–536, 2014.
[4] J. Pattillo, A. Veremyev, S. Butenko, and V. Boginski, "On the maximum quasi-clique problem," *Discrete Applied Mathematics*, vol. 161, no. 1, pp. 244 – 257, 2013.
[5] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "A model of internet topology using k-shell decomposition," *Proceedings of the National Academy of Sciences*, vol. 104, no. 27, pp. 11 150–11 154, 2007.
[6] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases," *arXiv preprint cs/0511007*, 2005.
[7] P. Hagmann, L. Cammoun, X. Gigandet, R. Meuli, C. Honey, V. Wedeen, and O. Sporns, "Mapping the structural core of human cerebral cortex," *PLoS Biology*, vol. 6, p. e159, 2008.
[8] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *arXiv preprint arXiv:1001.5285*, 2010.
[9] G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis, "Graph clustering and minimum cut trees," *Internet Mathematics*, vol. 1, no. 4, pp. 385–408, 2004.
[10] R. Rossi, D. Gleich, A. Gebremedhin, and M. M. A. Patwari, "Parallel maximum clique algorithms with applications to network analysis and storage," *arXiv:1302.6256v2*.
[11] D. R. Lick and A. T. White, "k-degenerate graphs," *Canadian J. of Mathematics*, vol. 22, pp. 1082–1096, 1970.
[12] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.
[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.
[14] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the visualization of large scale networks," *CoRR*, vol. abs/cs/0504107, 2005.
[15] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.
[16] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 51–62.
[17] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2013.
[18] Apache.org. (2017) Graphx- spark 2.2.0 documentation.
[19] B. Elser and A. Montresor, "An evaluation study of bigdata frameworks for graph processing," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 60–67.