DISTRIBUTED GRAPH DECOMPOSITION ALGORITHMS ON APACHE SPARK

A Thesis

Submitted to the Faculty

of

Purdue University

by

Aritra Mandal

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2018

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Mohammad Al Hasan, Chair

Department of Computer and Information Science

Dr. George Mohler

Department of Computer and Information Science

Dr. Fengguang Song

Department of Computer and Information Science

Approved by:

Dr. Shiaofen Fang

Head of the Graduate Program

I dedicate this work to my beloved parents and dearest friends and amazing siblings.
This humble work is my tribute to you all.

ACKNOWLEDGMENTS

The success of this research is a result of constant support of faculty advisors, friends and family. I would like to take this opportunity to thank those few who were paramount in making this research a success. First and foremost, I would like to thank my thesis advisor, Dr. Mohammad Al Hasan, for his guidance, encouragement and valuable lessons in machine learning throughout my research. Dr. Hasan with is immense patience, guided me with my various publication and research work. He gave me the liberty to find my own research direction while giving me constant guidance so that I don‘t stray from my goal. His capacity to explain concepts with clarity and simplicity helped me cross many of my research hurdles. He mentored me in gaining a good research aptitude.

I am also grateful to the thesis committee members, Dr. George Mohler, and Dr. Fengguang Song for giving me the opportunity to pursue my thesis.

I would like to thank my parents for their constant support and motivation throughout my Master‘s education. They have always been the roots of all my achievements. I would also like to express my gratitude to my friends for their constant support and inspiration.

I would also like to thank my lab mates for making my work here fun and for the constructive and collaborative environment. They collaborated with me in several research work and provide me useful advise and direction as and when required.

Lastly, I feel indebted to the opportunities given to me by IUPUI, through the knowledge I acquired from courses and the guidance I received from professors throughout my Master‘s education.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

SYMBOLS

$\forall$ For all elements in set

$\exists$ Their exist an element in set

$\nexists$ Their does not exist an element in set

$\cup$ Union of sets

$\cap$ Intersection of sets

$\setminus$ Difference of two sets

ABBREVIATIONS

RDD	Resilient Distributed Datasets
HDD	Hard Disk Drive
SSD	Solid State Drive
MR	Map-Reduce

GLOSSARY

Map-Reduce	A distributed programming paradigm which uses a combination of Map and reduce functions do transformation on data.
Pregel	A programming paradigm oriented toward graph-based algorithms.
Hadoop	An open source programming framework for processing and storage of large data in distributed manner.
Spark	A engine for handling large-scale in memory data processing

ABSTRACT

Mandal, Aritra M.S., Purdue University, May 2018. Distributed Graph Decomposition Algorithms on Apache Spark. Major Professor: Md. Al Hassan.

Structural analysis and mining of large and complex graphs for describing the characteristics of a vertex or an edge in the graph have widespread use in graph clustering, classification, and modeling. There are various methods for structural analysis of graphs including the discovery of frequent subgraphs or network motifs, counting triangles or graphlets, spectral analysis of networks using eigenvectors of graph Laplacian, and finding highly connected subgraphs such as cliques and quasi-cliques. Unfortunately, the algorithms for solving most of the above tasks are quite costly, which makes them not-scalable to large real-life networks.

Two such very popular decompositions, k-core and k-truss of a graph give very useful insight about the graph vertex and edges respectively. These decompositions have been applied to solve protein functions reasoning on protein-protein networks, fraud detection and missing link prediction problems.

k-core decomposition with is linear time complexity is scalable to large real-life networks as long as the input graph fits in the main memory. k-truss on the other hands is computationally more intensive due to its definition relying on triangles and their is no linear time algorithm available for it.

In this paper, we propose distributed algorithms on Apache Spark for k-truss and k-core decomposition of a graph. We also compare the performance of our algorithm with state-of-the-art Map-Reduce and parallel algorithms using openly available real-world network data. Our proposed algorithms have shown substantial performance improvement.

1. INTRODUCTION

Structural analysis and mining of large and complex graphs is a well studied and prominent research direction having wide-spread 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], counting triangles or graphlets [2], spectral analysis of networks using eigenvectors of graph Laplacian [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 networks, and hence, these subgraphs can, subsequently, be used for solving tasks such as community discovery, building features for graph indexing or classification, and graph partitioning.

Structural analysis of graphs can also be used for obtaining graph metrics pertaining to a vertex or an edge by utilizing the number of distinct structural patterns to which a vertex or an edge participates. For instance, we can label a vertex or an edge by the number of cliques, graphlets, k-cores, k-trusses or k-plexes, it touches. Such labeling enables us to identify relatively important vertices in a large network—a knowledge which has many applications in real life. For instance, in viral marketing, the important vertices can be used to facilitate or block a viral diffusion process over a network [5, 6]. Important vertices are also instrumental for solving the problem of group formation for impromptu activities [7, 8]. Important edges identified by k-plex structural patterns have been shown to capture the dynamics of a social network [9].

In massive networks, identification of cohesive subgraphs is often more fruitful, and more feasible as it emphasizes the focus on smaller but more important areas of the network. It helps find subgraphs that can be used to study, important properties of the network such as connectivity, robustness, self similarity, and centrality [10].

Cliques [11] and *maximal cliques* [12] are some of the basic cohesive subgraph measures identified, but both these measures have very rigid definitions, and produce cohesive subgraphs which are small and scattered. The produced subgraphs may either largely overlap each other or are disconnected from the rest. More relaxed definitions of cohesive subgraphs like n-clique [13] which relaxes the distance between nodes in a clique from 1 to n, k-plex [14] which relaxes the degree requirement of a vertex in a clique from $(n-1)$ to $(n-k)$, n-clan and n-club [15] have been identified. The quasi-clique methods of identifying cohesive subgraphs impose relaxation on either density or the degree [16] [17].

Unfortunately, the algorithms for solving the majority of the above tasks are computationally NP-hard 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.

k-core and k-truss are two such cohesive subgraph decompositions of graph which is computationally more efficient than other quasi-clique decomposition techniques. k-core is a decomposition technique based on the neighbors of an vertex and can be viewed as an quasi-clique achieved by degree relaxation on the vertex. k-truss is a decomposition of a graph which is based on edge property, this decomposition can be thought of as a quasi-clique with density relaxation on the decomposed subgraph. k-core was described by Seidman [14] as a seedbed within which cohesive subgraphs may precipitate. In the following subsections, I will provide a brief discussion of k-core, k-truss, and GraphX on Apache Spark.

1.1 k-core

In recent years, k-core (also called k-shells) decomposition of graphs has emerged as an effective and low-cost alternative for structural analysis of large networks. To date, k-core decomposition has been used for studying Internet topology [18], and also to study hierarchy, and self-similarity in Internet graph [19], for studying structural

composition of brain networks [20], for identifying influential spreaders in complex networks [21], for building data structures for graph clustering [22], and for computing lower bound to prune search space while searching for maximum cliques [23]. 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 has also been successfully applied to the task of large network visualization [24] [25], protein functions reasoning from protein-protein networks [26], fraud detection [27] and missing link prediction [28] [29]. k-core decomposition of a network has also proved to be be successful in approximating the densest subgraph, and the densest at-least-k-subgraphs problems for community detection [30].

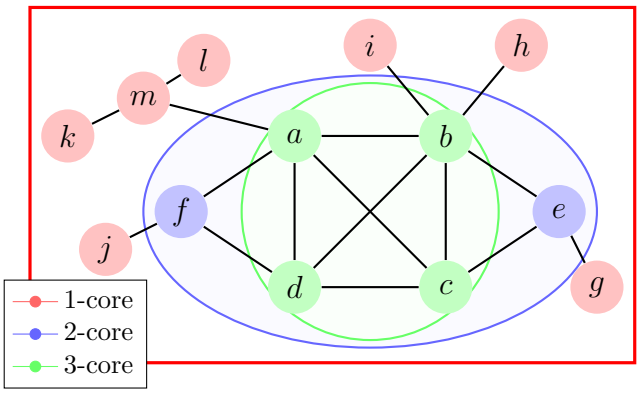


Figure 1.1.: A toy graph and its k-core decomposition

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 (in that subgraph) 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 1.1 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.

1.2 k-truss

Similar to k-core, k-truss is also a cohesive subgraph decomposition of a graph, but unlike k-core, which is a vertex centric decomposition, k-truss is an edge-centric decomposition of a graph G. In other words, for the case of k-core we compute the core-value of a vertex, but for the case of k-truss we compute the truss value of an edge. The subgraph induced by the edges of truss value k (or more) forms the k-truss of a graph.

k-truss generates cohesive sub graphs which are hierarchical in nature and gives cohesive structure of varying granularity in a graph. In this view graph decomposition using k-truss is similar to k-core explained in the previous section, in fact k-truss is a $(k-1)$-core but not the other way around. By definition k-trusses are much more rigorous in nature as it is based on triangles which are fundamental structures in a graph [31] [32] [33] [12].

k-truss decomposition of a graph G is the partitioning of the graph G where every edge in a partition is contained in $(k-2)$ triangles formed with other edges in the partition. k-truss is a connected subgraph of a graph in which every edge is contained in at least $(k-2)$ triangles. The problem of truss decomposition in G is to find the (non-empty) k-trusses of G for all k [34]. Truss value of an edge e is defined as the largest value of k such that e is contained in the largest possible subgraph S of G and e is contained in $(k-2)$ triangles with other edges in S. Figure 1.2 shows the truss decomposition of a toy graph. Edges $\langle ab, ac, ad, bd, bc, cd\rangle$ forms a 4-truss, edges $\langle af, df, ai, bi, be, ce\rangle$ forms a 3-truss as all edges in this subgraph form triangles with

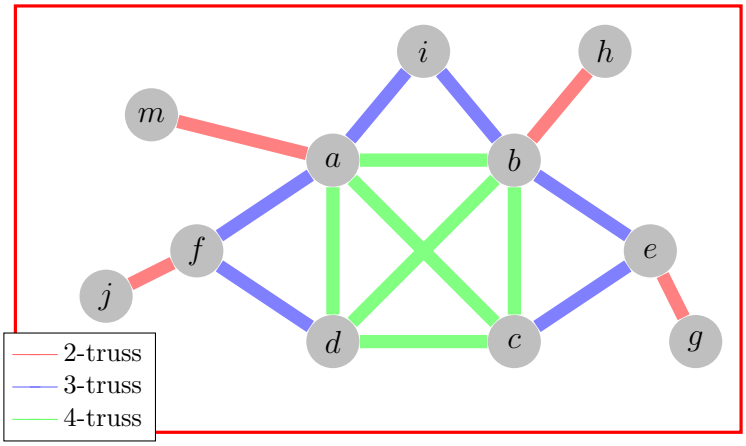


Figure 1.2.: A toy graph and its k-truss decomposition

edges whose truss value is at least 3. All k-truss subgraphs are also part of $(k-1)$-truss subgraphs and this leads to the hierarchical structure of k-truss decomposition.

1.3 GraphX on Apache Spark: Implementation Framework

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. 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. Unfortunately, no k-core decomposition or k-truss decomposition implementation on Spark is available yet.

1.4 Contribution of this Thesis

In this thesis, I have proposed scalable k-core and k-truss based graph decomposition methodologies, which are built on GraphX and Apache Spark platform. Both of these decompositions enable labeling a vertex or an edge with a core value or a truss value, facilitating many downstream graph analysis tasks. Spark based implementation makes the proposed methods very scalable, and efficient.

I propose a distributed k-core algorithm and its implementation, titled Spark-kCore, which runs on top of Apache Spark's GraphX framework. Spark-kCore follows "think like a vertex" paradigm, which is an iterative execution framework provided by Pregel API of GraphX. I compare Spark-kCore with two other k-core decomposition algorithms: EMCore [35] and Graphlab's k-core implementation [36]. Experimental results on 15 large real-life graphs show that Spark-kCore is substantially superior to the competing algorithms. I 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. I have made the source of Spark-kCore available on Github for the community to use [1].

I also propose a distributed k-truss algorithm and its implementation, titled Spark-kTruss on Apache Spark and GraphX. Spark-kTruss is an graph parallel edge-local computation of k-core decomposition of k-truss. It follows an "think like and edge" paradigm in computing truss values of an edge. I compare the performance of our proposed method in terms of running time and iteration count with a iterative map-reduce k-truss decomposition algorithm [37] implemented on Apache Spark. I also compare scalability of my algorithm using synthetic graphs of varying scale and controlled truss values. I report the convergence behavior of our algorithm for different real-life graphs.

[1] `https://github.com/AriMand/Spark-kCore`

2. RELATED WORKS

In this chapter we will discuss the research done in the field of graph decomposition and graph partitioning. The chapter contains a broad overview of the research interest in the field of graph decomposition with special focus on research related to k-core and k-truss decomposition.

2.1 Reseach Reated to Graph Decomposition

Graph decomposition or partitioning of a graph into subsets of vertices and edges has attracted researchers from multiple domains like Computer Science, Bio-Informatics, Sociology, and many more. The problem of graph decomposition can be seen as the problem of calculating graph measures to create partitions.

The partitioning of graph can also be viewed as a graph bisection problem. The initial approaches on graph partitioning was based on graph cuts. Graph cut is the process of partitioning the vertexes of a graph into disjoint subsets. A cut is defined by a set of edges that have endpoints in each of the vertex subsets formed by the cut, the cut is said to be made along these edges [38]. Kernighan et al. [39] were one of the earliest to suggest a heuristic based greedy approach on partitioning a graph which will minimize the cost of the edges in a cut. Their work compares different heuristics to perform a cut in large networks. The other main approach in traditional bisection based graph partitioning is using Laplacian and eigenvectors [40] [41]. Liu et al. [42] proposed another heuristic based approach which ensures maximum node separation among partitions. Another algorithm suggested by Spielman et al. [43] performs graph partitioning in near linear time. Ford and Fulkerson presented a pioneering work in the field of graph cuts [44]. In their research they suggested the Max-flow-Min Cut algorithm for graph cuts. The algorithm uses the flow of information in a

network to make cuts which minimizes the flow of information along cuts.

Most of the recent works in graph decomposition is done in the perspective of community identification [45] [46] [46]. Newman in his research [47] compared methods to identify dense structures in graph and used it for solving a community detection problem. He also stated why the traditional graph cut based algorithms are not suitable for modern era graphs.

Graph partitioning has also been studied by Sociologists, their studies revolve around the study of a specific social behavior from interaction and relationship graphs. The majority of these studies involve some form of hierarchal clustering [48]. The hierarchical clustering approach uses either the single link method or the complete link method. The single link method identifies communities hierarchically as new edges are added with decreasing similarity causing components to merge and form larger components. The single link can construct the dendogram using tree-based union and find algorithm of Fischer [49] [50]. The complete link method also starts with an empty graph and build the partition hierarchically but rather than using single links, communities are identified by maximal cliques [51]. Some complete linkage algorithms rather than using maximal clique use k-component communities. A pair of vertices in a k-component community have k independent paths between them [52] [53].

The recent algorithms for community detection like the one suggested by Girvan and Newman [54] features decisive way of identifying communities based on natural division among the vertices. Their method in contrast to agglomerative approach of hierarchical clustering, works on repeated removal of edges based on edge betweenness property. Tyler et al [55] in their study modified the algorithms suggested by Girvan and Newman to speed up cluster identification. Radicchi et al. [56] on the other hand proposed a different measure other than the betweenness measure to partition graphs into communities. Newman in a later research introduced a modularity function over all possible divisions of a network which can identify partitions in a graph more efficiently than existing methods of that time [57]. In another recent work Andersen et al. have used pagerank of vertices to achieve local partitioning of graphs [58].

With growth in the size of graphs more and more researchers have shown interest in parallel and distributed graph partitioning and decomposition. Gilbert et al. [59] produced one of the earliest works on parallel graph processing using message passing algorithm on multiprocessor. Gonzalez et al. [60] proposed a new framework called PowerGraph for distributed graph parallel calculation of graph measures which can be used for a partitioning task. Stanton et al. [61] in their research proposed a stream based algorithm for graph partitioning of large graphs. Karypis et al. [62] created a library, ParMETIS for parallel graph partitioning and sparse matrix ordering.

In the following section we will discuss in detail work related to graph decomposition and partitioning techniques using k-core, distributed triangle enumeration and counting, and k-truss.

2.2 Research related to k-core decomposition

Vladimir Batagelj et al. [63] presented a sequential k-core decomposition method which runs in linear time with respect to the number of edges. Their technique traverses through a list of vertices sorted by the node degrees and updates the position of each vertex based on degree of its neighbors as it traverses. The authors used a form of insertion sort to make the sorting and update process to run in linear time. However, their method can operate only when the whole graph can be stored in the main memory of a computer. Another noticeable work on k-core decomposition by using a single computer was done by Khaouid et al. [64]. A salient feature of their work is that although their proposed method runs on a single machine, it follows the vertex centric approach which is, more often, seen in distributed frameworks such as, Pregel. Along with the k-core algorithm, they also proposed a single memory graph processing engine, called Graphchi. Besides Graphchi, Khaouid et al.'s method also runs on Graphlabs platform. The algorithm proposed by Batagelj and Zaversnik is highly efficient but they are not suitable for parallel processing. Dasari et al. [65] proposed ParK, a parallel k-core decomposition algorithm, which runs on multi-core

processors. Authors show that ParK is significantly faster than the existing sequential algorithms as long as the entire graph can be loaded into the main memory.

Graphs in real life can get very dense and huge in size so there has been a considerable research on identifying algorithms which can compute k-core decomposition of a graph in distributed manner. Alberto Montresor et al. provided the pioneering algorithm in this direction. In their work [66], they provided a good overview of solving k-core decomposition in a distributed paradigm. They also proved the correctness of an iterative message passing algorithm to calculate the k-core decomposition. Their research also draws a conclusion on the upper bound on the number of iterations needed to get an accurate k-core value of each node. Later Montresor et al. also gave a comparison between distributed k-core algorithm on de-facto standard Hadoop Map-Reduce with fine tuned implementation of Map-Reduce like Pregel [67], Graphlab [68], Stratosphere, and Apache Giraph. To this date, no k-core decomposition method is available for the Spark computation platform, which is one of the focuses of my thesis.

For large graphs that do not fit in main memory, an alternative approach for k-core decomposition is to use an algorithm that runs seamlessly over limited memory by using external memory (EM) as needed. The EMCore algorithms developed by James Cheng et al. [35] in one such method. Since it has been proposed, this algorithm and its implementation has been a very popular k-core decomposition algorithm for very large graphs.

With the increase in large graphs whose structures vary over time, interest in developing algorithms which incrementally updates the k-core value grew. The work done by Ahmet Erdem et al. [69] is one of the earliest which provides an efficient algorithm to calculate and maintain k-core values for nodes in a graph where new nodes and edges were added over time. In a later work [70], Ahmet provided another algorithm which maintains k-core values for nodes in graphs with both addition and deletion of nodes and edges over time.

In next two section we will briefly discuss on related works on distributed triangle enumeration and then follow it by the works on k-truss decomposition.

2.3 Research related to distributed triangle enumeration and counting

The definition of k-truss is based on edge local triangle count so before discussing on k-truss decomposition work we start with research done on distributed triangle enumeration and counting. Hasan et.al [71] provided an extensive study of the different approaches used for triangle enumeration and counting on graphs of different scale.

Triangles are one of the most commonly studied network structures, finding application in a variety of use-cases starting with clustering [72], spam detection [73], community detection etc. Triangles have also been used as meta information for other decomposition tasks like k-truss and k-cliques. Triangle counts have also been applied to database query plan optimization [74].

With growing size of network datasets which can not be held in memory, research interest grew in solving the triangle counting and enumeration problem with a graph represented as stream of edges. Yossef et.al [74] were one of the first to propose an algorithm on streaming graph data with a method called stream-reduction, although their idea was theoretically innovative but was not applicable to real-world network datasets. Another interesting work was done by Buriol et al. [75] they proposed a three pass algorithm which uses the Chernoffs inequality to calculate probabilistic bound on the triangle count approximation task. Later they also proposed a one pass algorithm which performs the actions of each of the three passes in one go. Jha et al. [76] came up with another one-pass streaming triangle counting algorithm which is a variant of triple sampling algorithm adapted for streams. Lim et. al [77], proposed a way of local triangle enumeration for each node of a network—this idea can be easily extended to edge-local triangle enumeration. They proposed a system called *MASCOT*, a memory-efficient and accurate method for local triangle estimation in a

graph stream based on edge sampling.

Stream algorithms are good for map-reduce kind of environment but streaming algorithms generally provide approximation. For exact counts a distributed algorithm is more suitable. Suri and Vassilvitskii et. al. [78] proposed a map-reduce based algorithm. The algorithm proposed by them is based on node iterator algorithm. The algorithm makes a two round execution, first round generates all length-two paths in the graph from the edge list, in parallel. In the second pass the algorithm counts how many of the length two paths are closed by the edges in edge list. Later Suri and Vassilvitskii [78] proposed a partition based map-reduce algorithm. The algorithm partitions the graph first and then runs an exact triangle counting method on each partition, in parallel. Later Park and Chung [79] improved on Suri et al.s method and proposed a partitioning logic called *Triangle Type Partitioning*. Another innovative approach was proposed by Arifuzzaman et. al [80] using distributed memory based message passing to count triangles. The algorithm partitions the graph into disjoint subsets of nodes, and generates induced subgraph in each partition with nodes in the partition and their neighborhood. Triangles are calculated for each of these subgraphs in an distributed manner. Finally, the counts from all the machines are collected to get the final count.

Avron [81] in his research suggested a matrix-based approximation algorithm for triangle counting using Monte-Carlo simulation and the trace of the cube of the adjacency matrix. Later Ariful Azad et al. [82] improved the algorithm to a simple exact triangle counting parallel algorithm that is based on matrix algebra on sparse adjacency matrices.

Considerable research interest is also shown on disk based algorithms on multi core CPU for large graphs. Kim et al. [83] in his research proposed a disk-based algorithm for triangle counting on a multi-core CPU using *openMP*. The Algorithm uses the notion of internal triangles to represent adjacency list of two connected nodes already in memory, and external triangles to represent adjacency list of two connected nodes out of which only one is in memory. In a different approach Rahman and Al

Hasan [84] proposed a multi-core parallel variant of the node/edge iterator algorithm for triangle counting, where the loop of node/edge iterator algorithm are distributed across multiple cores. A shared memory multi core algorithm was proposed by Shun and Tangwongsan [85]. The algorithm has two phases, first a parallel ranked adjacency list creation based on degree and in the second phase a local triangle counting is done, finally counts are summed to get the total triangle count.

2.4 Research related to k-truss decomposition

k-truss is a measure of structural cohesiveness of a network [34] which has got attention of researchers in recent years. Cohen [34] was one of the first to suggest in his research that, k - truss is a effective network reachability measure which is computationally more relaxed as compared to cliques. He also provided an algorithm which requires random access to the whole network resident in memory. Cohen [86] later proposed a parallel distributed algorithm for k-truss decomposition based on map-reduce framework which solves the problem of keeping the entire graph in memory. Wang and Cheng in there research [10] suggested a method for k-truss decomposition which uses an I/O efficient algorithm. The algorithm is lower bounded by the worst-case complexity of in-memory triangle listing algorithm [87]—by optimizing I/O it tries to address the problem of limited memory in a single machine for large graphs.

Huang et al. [88] proposed a query based method to identify k-truss communities in large real-life dynamic graphs which have a frequent addition and removal of nodes, vertices or both. Huang et al. [89] also proposed a way of computing a k-truss on probabilistic graphs. Both these techniques identify the k-trusses given a value of k. These techniques do not attempt to perform a k-truss decomposition and identify the maximal k-truss value in a graph.

With advancement in high performance computing and advanced high throughput architectures new techniques have been proposed for parallel and faster computation of k-truss decomposition. Kabir and Madduri in their work have proposed an al-

gorithm which uses parallelization on a multi-core system to solve all HPEC 2017 Static Graph Challenge datasets under a minute [90]. Smith et al [91]. have further extended the work, they use the hierarchical nature of k-truss decomposition. Their algorithm breaks the efficient serial algorithm into a bulk synchronous parallel steps which does not rely on atomic updates and synchronization.

Pei-Ling Chen et al. [37] extended the work done by Cohen [86] to propose distributed k-truss decomposition algorithm on the map-reduce framework. The algorithm eliminates the repeated triangle enumeration from Cohen's [86] work. They also further provided a proof suggesting the locality of k-trusses, leading to a graph parallel computation of k-truss decomposition.

3. BACKGROUND

In this chapter we will provide mathematical definitions of k-core and k-truss decomposition problem and related terminologies, which we will be using in the following chapters.

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., $deg(u) = |\mathcal{N}(u)|$. k-core and k-truss of a graph and its related terminologies are defined below. We will use the example graph G shown in figure 3.1 to elaborate on the definitions given below.

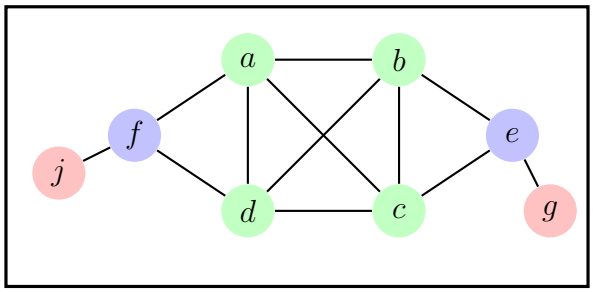


Figure 3.1.: A example graph G

3.1 Definitions related to k-Core

Definition 3.1.1 (k-core) *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. In the above example graph $\langle a, b, c, d \rangle$ forms a 3-core as each vertex in the subgraph has 3 neighbors with core have greater than equal to 3*

Definition 3.1.2 (Core number) *The core number of a vertex, core(v), is the largest value for k such that $v \in C_k(G)$. By definition the core number of $\langle a, b, c, d\rangle$ in the sample graph G are all equal to 3, i.e $core(a) = 3$*

Definition 3.1.3 (maximum core) *The maximum core number of a graph G, $C_{max}(G)$, is defined as $\max_{\forall v \in G}\{core(v)\}$. By definition the maximum core value of any vertex in the given graph G is 3, therefore $C_{max}(G) = 3$*

Definition 3.1.4 (k-degenaracy) *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 $deg(v) \leq k$.*

Lemma 3.1.1 *If a graph has a (non-empty) k-core, the degeneracy value of that graph is at least k.*

3.2 Definition related to k-Truss

Definition 3.2.1 (Triangle) *Triangles incident on an edge $e = (u, v)$ denoted by $Triangles(u, v)$ is defined as a set of vertices S such that $\forall z \in S | \exists (u, z), (v, z) \in E$. In the above sample graph the edge (a, b) forms triangle (a, b, c) and (a, b, d) so $Triangles(a, b) = \{c, d\}$*

Definition 3.2.2 (Edge Support) *The support of an edge $e = (u, v) \in E$ $sup(e, G)$ is given by $|\mathcal{N}(u) \cap \mathcal{N}(v)|$ This can also be defined as number of triangles incident on an edge i.e $|Triangles(u, v)|$. For example $sup(e = (a, b), G) = |Triangles(a, b)| = 2$ where G is the sample graph given above.*

Definition 3.2.3 (Edge neighbor) *The neighbors of an edge $e = (u, v)$ are defined as $nb(e) = \{e_i = (x, y) : \{x, y\} \cap \{u, v\} = 1, e_i \in E\}$ i.e edge e_i shares a vertex with edge e. By definition we can see that in the example graph given in figure 3.1 that edge neighbor of (a, b) are $\langle (a, d), (a, c), (a, f), (b, c), (b, d), (b, e)\rangle$ as all these edges share either the vertex a or the vertex b with (a, b).*

Definition 3.2.4 (Edge Triangle Relationship) *The edge triangle relationship is defined as $R(e)$, set of all edge neighbors of an edge e such that e forms a triangle with two members of $R(e)$. For an edge e $R(e) = \{r(e) \in nb(e) : r(e), r_i(e) \in nb(e)$ and e forms a triangle. In figure 3.1 the edge triangle relationship of (a, b) are $\langle (a, d), (a, c), (b, c), (b, d) \rangle$ as all these edges share either the triangle (a, b, c) or the triangle (a, b, d) with edge (a, b).*

Definition 3.2.5 (Line Graph) *This is a transformation of graph G into a graph $LG(V_{LG}, E_{LG})$ such that a vertex $v_{LG} \in V_{LG}$ represents an edge $e \in E$, an edge $e_{LG} = (u_{LG}, v_{LG}) \in E_{LG}$ is created if and only if u_{LG} and v_{LG} share a edge triangle relationship with each other in G*

Definition 3.2.6 (k-truss) *A k-truss, $K(G, k)$ is defined as a connected subgraph $S_K(V_K, E_K)$ of G such that $\forall e = (u, v) \in E_K | sup(e, R_K) \geq k - 2$. In figure 3.1 a 4-truss is defined by $\{(a, b), (a, d), (a, c), (b, c), (b, d), (c, d)\}$. The subgraph formed by these edges is the maximal subset of edges such that each edge forms 2 triangles with other edges in the subset.*

Definition 3.2.7 (maximum k-truss) *The maximum k-truss of a graph G, $K_{max}(G)$, is defined as $\max_{\forall S_K(V_K, E_K) \subset G} K(G)$. The maximum size of a truss in the graph shown in figure 3.1 is 4, therefore $K_{max}(G) = 4$*

4. K-CORE

We propose a distributed k-core algorithm and its implementation, Spark-kCore. Spark-kCore runs on top of Apache Sparks 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 [35] and Graphlabs k-core implementation [36]. 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.

4.1 Methods of computing k-Core

In this section we will discuss the "think like a vertex" algorithm for calculating the core value of each node. The algorithm uses the definition of core value given in the background section to update the nodes in each super step until a state of global equilibrium is reached across all nodes in the graph. We then explain the details of implementing this algorithm for both directed and undirected graph using GraphX on Apache Spark.

4.1.1 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. [66]. 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. $\exists V_k \subseteq \mathcal{N}(u)$ such that $|V_k| = k$ and $\forall u_i \in V_k,\ core(u_i) \geq k$;

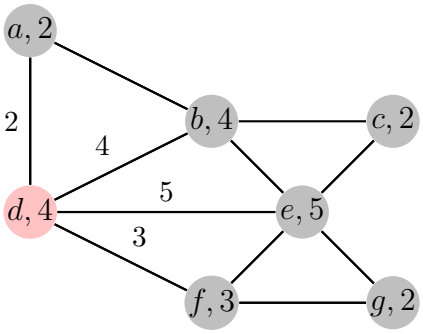
2. $\nexists V_{k+1} \subseteq \mathcal{N}(u)$ such that$|V_{k+1}| = k+1$ and $\forall u_i \in V_{k+1},\ core(u_i) \geq k+1$.

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.

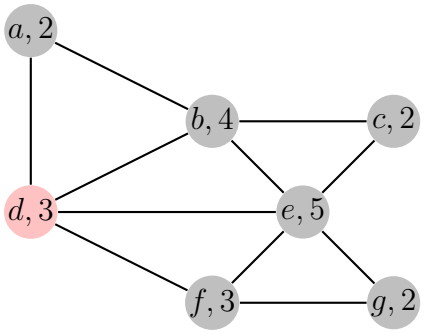
An obvious upper bound of the core value of each node is its own degree value. So, in a vertex-centric k-core decomposition algorithm, each node initializes its core value with the degree of itself. Each node (say u) then sends messages to its neighbors $v \in \mathcal{N}(u)$ with the current estimate 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 easily computed 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-

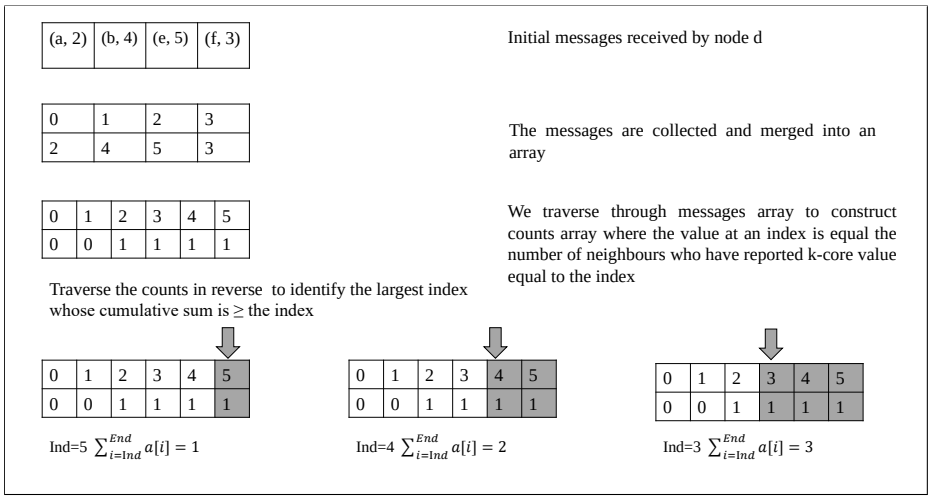
update-broadcast iteration occurs until there are no more messages to process in any node in the graph.



(a) Messages addressed to d by its neighbors



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



(c) Processing at node d

Figure 4.1.: The k-core update flow for one iteration for the vertex d

In Figure 4.1, 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 4.1(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 4.1(c), 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 4.1(b)

4.1.2 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 4.1.1. We use the GraphX engine of Spark to load and process graphs. We start by explaining a few details about the GraphX engine which are 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 [92].

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 ← msg.Split(delimiter)
3:     for all m ∈ msgArray do
4:         if m ≤ u.kcore then
5:             count[m] + +
6:         else
7:             count[u.kcore] + +
8:     for i := k to 2 do
9:         curWeight ← CurWeight + count[i]
10:        if curWeight ≥ i then
11:            u.kcore ← i
12:            break
13:    return u
```

Algorithm 3 KcoreSpark - Propagate

```
1: procedure SendMsg(EdgeTriplet triplets)
2:     srcVertex ← triplet.getSrcAttr()
3:     destVertex ← triplet.getDstAttr()
4:     I ← 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 counts array is traversed in reverse and the 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 the node's neighbors if the 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 $1+\sum_{u\in V}[deg(u)-core(u)]$ [66], which is equivalent to the summation of number of updates made by each node. For the measurement of this time complexity, we consider the fact that Pregel iterations are synchronous i.e. in a iteration each node receives messages addressed to it, updates core value, and shares updated core value with neighbors.

4.2 Experiments

We perform experiments using Spark-kTruss on real-life datasets to determine its performance. For comparison purposes we also run test on EMCore, an external memory k-core decomposition algorithm, and Graphlab, a parallel k-core decomposition library. We use running time as a measure of performance, lower running time indicates better performance.

4.2.1 Experiment Setup

Cluster Configuration

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.

4.2.2 Experimental Results

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

Table 4.1.: Table of results showing the No. of vertices, Edges, Maximum k-core, running time and the number of Pregel iterations in Spark-kCore.

Dataset	**Vertices**	**Edges**	$C_{max}(G)$	$\mathbf{T_{mins}}$	**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

implementation of k-core decomposition which is based on [36]. 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. [35]. We use the Emcore implementation given in [64]. We cannot compare with MapReduce implementation of k-core decomposition discussed in [93], because neither a publicly available implementation of this algorithm is available, nor could the authors provide their implementation.

Spark-kCore's Runtime Behavior on Various Graph Metrics

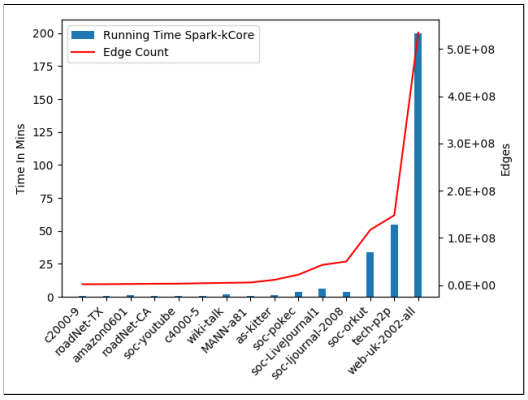
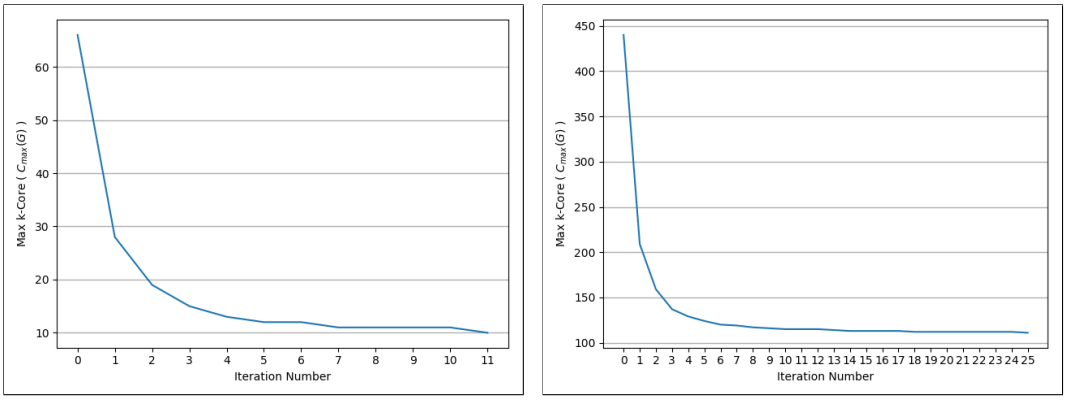


Figure 4.2.: Comparison of running time with number of edges fro Spark-kCore

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 4.2, 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.

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 4.3 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.



(a) amazon0601 (b) as-skitter

Figure 4.3.: Change in k-core value

Runtime comparison between Spark-kCore and Turi Graphlab

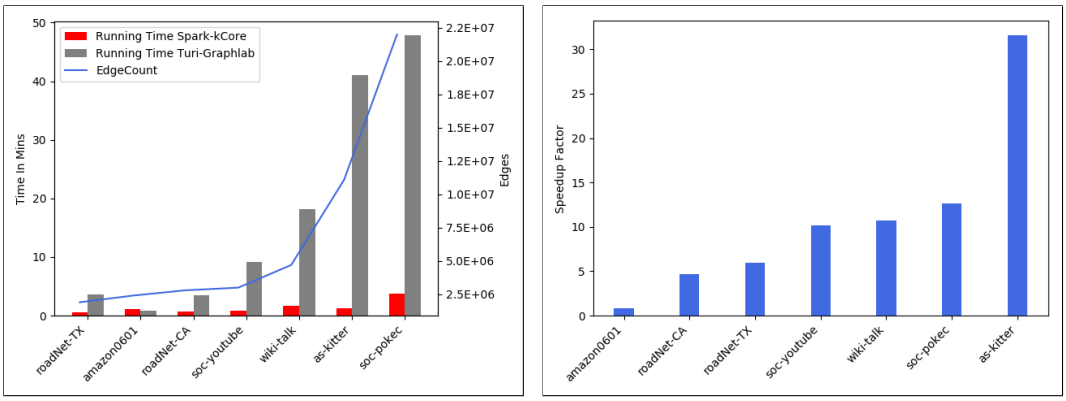
As mentioned above for this comparison we use graphs that fit in the main memory. Among the graphs that we use in this paper, 7 graphs qualified. The comparison

results are shown in Table 4.2. 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

Table 4.2.: 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

time of algorithms increases with increasing number of edges. We demonstrate this behavior in Figure 4.4(a). In Figure 4.4(a) 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. 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 4.4(b) 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) running time vs number of edges (b) Speedup achieved

Figure 4.4.: Comparison between Spark-kCore and Graphlab

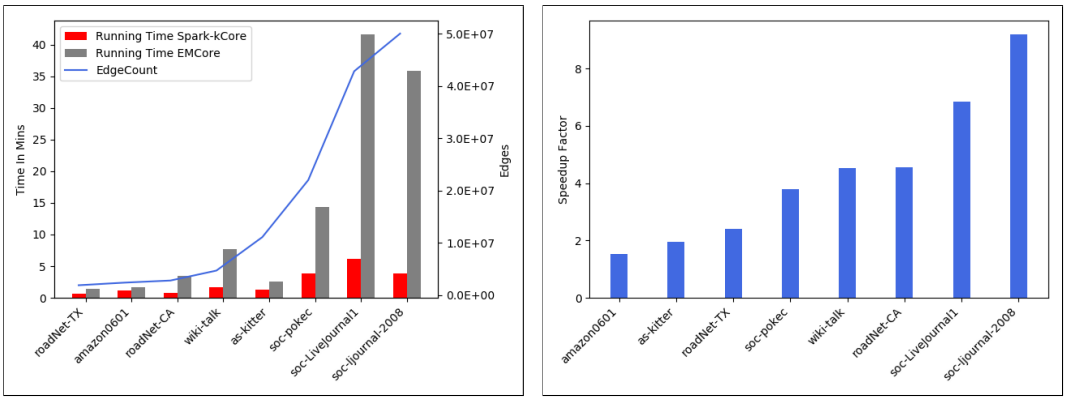
Runtime Comparison between Spark-kCore and EMCore

As mentioned above we also compare the the running time of spark-kCore with EMCore implementation given in [64]. We compare the results on graphs which are medium to large in size. We run the comparison on 6 graph amazon0601, wiki-talk, roadnet-CA, roadnet-TX, soc-pokec and soc-livejournal1. The comparison results are shown in Table 4.3. The results show that spark-kcore is faster than EMCore. In Figure 4.5(a) the bars represents the running time for Spark-kCore and EMCore 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.

Figure 4.5(b) 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 see a trend that as the size of graph grows the speedup factor increases suggesting that with larger files speedup factor will also increase.

Table 4.3.: 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



(a) Running time Vs number of edges

(b) Speedup achieved

Figure 4.5.: Comparison between Spark-kCore and EMCore

5. DISTRIBUTED TRIANGLE ENUMERATION AND COUNTING

In this chapter we will discuss different triangle enumeration and counting algorithms on distributed frameworks. We also propose an algorithm to enumerate edge local triangles using in-memory distributed processing framework of Apache Spark. The proposed algorithm is optimized for partitioned in-memory processing with the requirement of resource sharing. We also compare the performance of this algorithms with Suri and Vassilvitskii [78] Node iterator based map-reduce implementation. An efficient triangle enumeration is essential to achieve efficient k-truss decomposition.

5.1 Distributed triangle enumeration algorithm

The baseline map-reduce algorithm was initially proposed by Cohen [86]. The algorithm is a two phase map-reduce operation. In the first phase the map task reads in the input file and emits an edge keyed by the lower degree vertex. If two vertices of an edge has the same degree the tie is broken by employing an ordering among the vertex-ids. The reduce task of first phase takes as input a key value pair where the key is a vertex and value is a bin of edges adjacent to the vertex. The reducer emits for each pair of edges in the bin an open triad with the key as the center vertex of the triad.

The Second map-reduce phase takes as input both the original edge list file and the output of the first map-reduce phase. The second map task reads both the input files and combine the records from both the sources and change the edge record's keys so they are keyed by the vertices that the edges join. The second and final reduce task takes input key, value pair such that the key is a vertex pair and bin of edges and/or open triads. If a bin contains both open triads and edges from original graph

then that bin produces triangle. A bin will contain maximum one edge from original graph and any number of open triads. If bin contain an edge and k open triad then it will result in k triangles.

Algorithm 4 TriangleEnumMR

Map1: *Input:*$\langle(u; deg(u), v; deg(v))\, \phi\rangle$

1: **if** $deg(u) < deg(v)$ **then**

2: **emit** $\langle u; (u, v)\rangle$

Reduce1: *Input:* $\langle v; S \subset E\rangle$

3: **for** $(u, w) : u, w \in S \wedge u, w \neq v$ **do**

4: **emit** : $\langle (u, w); S\rangle$

Map2:

5: **if** Input of type $\langle (u, v); S \subset E \cup \{(a, b) : a, b \in V\}\rangle$ **then**

6: **emit** $\langle (u, v); S\rangle$

7: **if** Input of type $\langle (u; deg(u), v; deg(v)); \phi\rangle$ **then**

8: **if** $deg(u) < deg(v)$ **then**

9: **emit** $\langle (u, v); (u, v)\rangle$

10: **else**

11: **emit** $\langle (v, u); (u, v)\rangle$

Reduce2: *Input:*$\langle (u, v); S \subset E \cup \{(a, b) : a, b \in V\}\rangle$

12: **if** $(u, v) \in S$ **then**

13: **for** $(x, w) \in S : x = u \vee x = v$ **do**

14: **emit** : $\langle (u, v) \cup (x, w)\rangle$

5.1.1 Distributed node iterator based algorithm

The baseline algorithm explained above requires a degree information appended to it which is not easily available for large graphs. In this section we will explain a

different map-reduce approach proposed by Suri et al. [78]. This algorithm is based on node iterator algorithm and will be referred to as NodeIteratorMR going forward. This algorithm is one of the most commonly used map-reduce based Triangle counting algorithm in use for large real world graphs. We also use this algorithm for comparison with our proposed algorithm.

Similar to map-reduce Algorithm suggested above, NodeIteratorMR is a two phase algorithm.

1. first phase reads in edge list and emits all unique length two paths.

2. second phase reads in both the input edge file as well as the output of the first phase checks how many of the open triads in phase one are closed by an edge, and emits the closed triads as triangles.

Algorithm 5 shows the pseudo code for the NodeIteratorMR Algorithm. The Mapper for phase 1 reads in an edge list file with key as line number and value as a pair of nodes that form an edge. To avoid duplicate counting of edges ordering is imposed on edges if the source vertex is smaller than the destination vertex of an edge the mapper emits a key value pair with key equal to the source vertex-id and value equal to destination vertex-id. The reducer in first phase takes as input key value pair where the key is a vertex-id and value is the adjacency list of the vertex. The reducer emits a key value pair representing a triad where the key is the center vertex of the triad and the value is pair of vertex from the adjacency list of the center vertex representing an edge which may close the triad into a triangle.

In second phase mapper takes as input the output file generated in firts phase as well as the original input file. This mapper emits two different kind of records based on the input. If the record is from the edge file it emits a key value pair where key is the edge and value is chosen to be a special character. For records from the output file of first phase the mapper emits a key value pair where the key is the edge (u, v) that closes the triad and value is the center vertex of the triad. The reducer takes as input a key value pair where the key is an edge and the value is a list of vertices

Algorithm 5 NodeIteratorMR

Map1: *Input:*$\langle(u, v)\, \phi\rangle$

1: **if** $u < v$ **then**

2: **emit** $\langle u; v\rangle$

Reduce1: *Input:* $\langle v; S \subset \eta(v)\rangle$

3: **for** $(u, w) : u, w \in S$ **do**

4: **emit** : $\langle v; (u, w)\rangle$

Map2:

5: **if** Input of type $\langle v; (u, w)\rangle$ **then**

6: **emit** $\langle(u, w); v\rangle$

7: **if** Input of type $\langle(u, v); \phi\rangle$ **then**

8: **emit** $\langle(u, v); \alpha\rangle$

Reduce2: *Input:*$\langle(u, v); S \subset V \cup \{\alpha\}\rangle$

9: **if** $\alpha \in S$ **then**

10: **for** $w \in S \cap V$ **do**

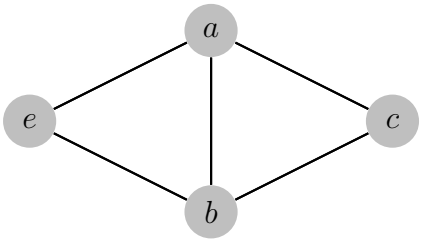
11: **emit** : $\langle u, v, w\rangle$

and a special character that indicates the edge was present in the original graph. The reducer checks the presence of special character in the list of values. If the special character is present it indicates the closure of an open triad. In this case the open triad given by the key forms a triangle with all other values in the list.

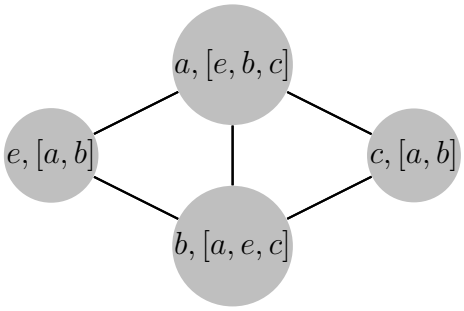
5.1.2 Proposed distributed algorithm on GraphX

The MapReduce algorithm suggested above both suffer from the problem of multiple disk I/O as they needs to make multiple pass over the same data. In Spark we can minimize the disk I/O using resilient storage in memory but we still cant avoid the multiple pass over the edge list. We propose an algorithm which takes advantage of

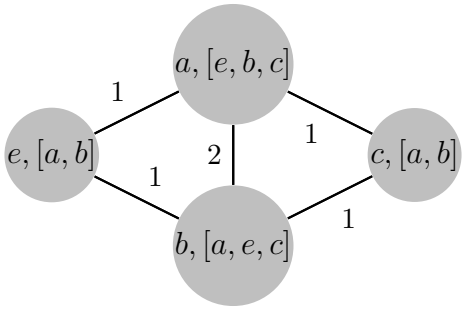
the graph abstraction provided by GraphX on Apache Spark to reduce the amount of disk I/O as well as the multiple passes over the data. The proposed algorithm makes edge-local calculation of triangles and finally combine the results to get the count. The Algorithm relies of partitioning the graph into multiple edge centric overlapping subgraphs, where vertices at the end points of an edge are aware of their neighbors.



(a) Original Sample Graph G



(b) Graph G updated with node neighbor information



(c) Updated edge local Triangle Count

Figure 5.1.: Spark-Edge-Triangle operation example

Figure 5.1 shows the GraphX based triangle counting locally for each edge. The first step of the process is to generate a list of records, where each record is a pair of values containing a vertex-id and the adjacency list of the vertex. Then using the node update procedure of GraphX each node u is updated with its adjacency list in the graph G, as shown in figure 5.1(b). Now in the updated graph we use the triplet view to count triangles incident on each triplet. The triplet view consists of $[edge - attribues,$ source-vertex-arrtribute, and destination-vertex-arrtribute]. We

take the intersection of source vertex neighbors and the destination vertex neighbors of an triplet to find the number of triangles incident on an edge as shown in Figure 5.1(c). Algorithm 6 gives a pseudo code for edge-local triangle counting using Apache Spark.

Algorithm 6 EdgeLocalTriangle

```
1: procedure EnumerateEdgeLocalTriangle(Graph : G)
2:     nei ← G.CollectNeighbors()   ▷ nei : RDD[vertexID, List[Neighbours]]
3:     G.MapNodes(nei)              ▷ Given G(V,E) ∀u ∈ V u.nb = nei[u]
4:     for all e ∈ G.edges do
5:         u ← e.src
6:         v ← e.dest
7:         e.triangles ← u.nb ∩ v.nb
8:         e.tcount ← Size(e.triangles)
9:     return G
```

5.2 Experiments

We perform experiments on real-life datasets to test the performance of Spark-Edge-Triangle, a partitioning based edge-local triangle enumeration and counting algorithm on Apache Spark. We compare the performance of our proposed method against traditional defacto hadoop map-reduce method of triangle enumeration and counting. We use running time as a measure of performance, lower running time indicates better performance.

5.2.1 Experimental Setup

Cluster Configuration

Spark-Edge-Triangle 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-edge-triangle on publicly available SNAP datasets (`snap.stanford.edu`) and Network Repository datasets (`networkrepository.com`). We perform our analysis on the following five graph datasets: Gowalla-edges, as-kitter, com-youtube,LiveJournal social network,soc-Pokec.

5.2.2 Experimental Results

We compare our proposed algorithm Spark-Edge-Triangle with the performance of NodeIteratorMR (node-iterator based map-reduce proposed by Suri et al. [78]). Table 5.1 shows the results of running Spark-Edge-Triangle and NodeIteratorMR on 5 different graphs. The columns of the table represent the name of the dataset, the size of the vertex set, size of edge set. T_{count} represents total number of triangles found, T^E_{max}; the maximum number of triangles incident on any given edge, $Time_{MR}$ and $Time_{SET}$ represent the time taken by the map-reduce algorithm and Spark-Edge-Triangle algorithm to count triangles respectively.

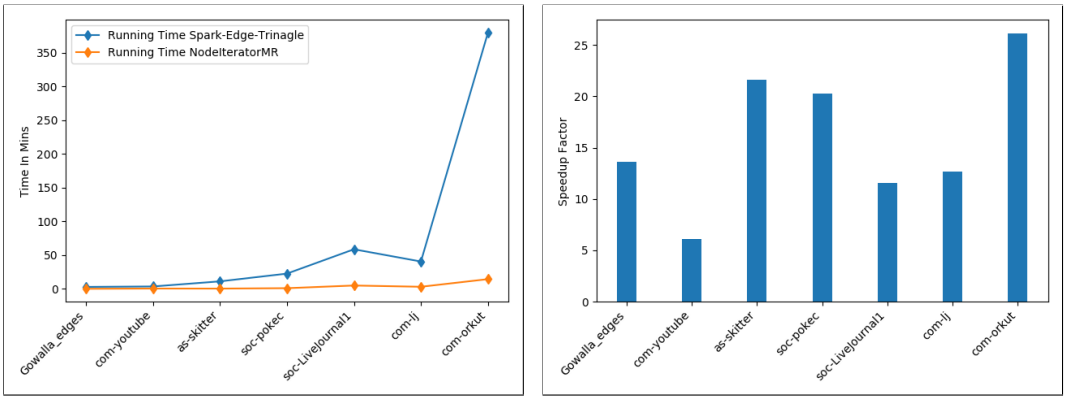
Speedup achieved by Spark-Edge-Triangle over NodeIteratorMR

We compare the performance of our proposed algorithm with NodeItertorMR on 7 different real life datasets. We observe that we get a speedup ranging from 5 times to around 20 times for different networks. The speedup factor increases with the size of network in general. One more interesting observation is that the speed up increases but as the size of graph becomes large enough that they cannot be held in memory we see a drop in performance. This drop in performance can be addressed by using faster disks like SSD.

Table 5.1.: Table showing running time in minutes of Spark-Edge-Triangle and NodeIteratorMR

Dataset	**Vertex**	**Edge**	$\mathbf{T_{count}}$	$\mathbf{T^{E}_{max}}$	$\mathbf{Time_{MR}}$	$\mathbf{Time_{SET}}$
Gowalla-edges	0.2M	0.9M	2273138	1297	3	0.22
com-youtube	1M	3M	3056386	4034	3.67	0.6
as-skitter	1.7M	11.1M	28769868	28654	11.24	0.52
soc-pokec	1.6M	22M	32557458	5566	22.72	1.12
com-lj	5.3M	50M	177820130	1393	40.64	3.21

Figure 5.2 shows the running time comparison of Spark-Edge-Triangle with NodeIteratorMR on different real-life graphs with increasing magnitude of edge. The X-axis represents the different graphs arranged in increasing order of edges, and Y-axis represents the time taken. Figure 5.2(b) Shows the speedup achieved by Spark-Edge-Triangle over NodeIteratorMR.



(a) Running time of Spark-Edge-Triangle and NodeIteratorMR

(b) Speedup achieved

Figure 5.2.: Comparison spark-edge-triangle with NodeIteratorMR

6. K-TRUSS

6.1 Methods of Computing k-Truss

In this section we will describe methods of k-truss decomposition for a graph G using two different distributed algorithms. The first approach uses the traditional map-reduce approach with iterative pruning of edge represented as a list of key value tuple. The second approach is a more robust graph parallel computation model which takes advantage of the locality of k-trusses. Both these decomposition models use the count of triangles incident on an edge for initialization. For enumerating triangles on an edge we use the algorithm described in previous section.

Spark is suitable over traditional map-reduce for iterative algorithms. Both these algorithms are iterative in nature so we compare the performance of these competing methods on an framework suitable for iterative algorithm. We also study the convergence of the graph parallel algorithm to see if it follows the same pattern as the convergence of k-core.

6.1.1 Distributed k-truss algorithm based on map-reduce

Here we explain the iterative map-reduce based approach of calculating the truss value of each edge. Distributed k-truss decomposition on a map-reduce framework was proposed in [86] and [37]. The Algorithm works by iteratively filtering the edges which do not satisfy the minimum trussness criterion and updates the trussness of remaining edges until no more edges are left.

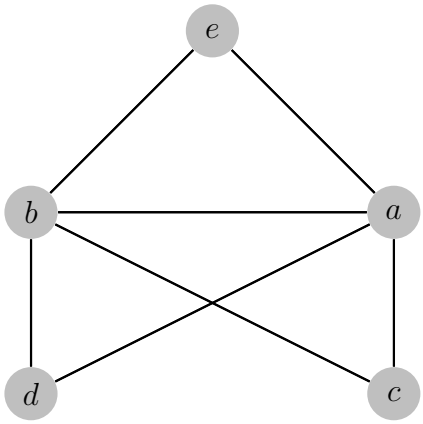
The Algorithm consists of three main tasks. 1) Get the set of vertices V_T^e that from triangles with a given edge e; 2) Create an edge centric view of the graph where each record is a $key, value$ pair; key is the edge e and $value$ is the current truss

value of e denoted by $cTruss(e)$ along with all the edges that share an edge triangle relationship with e denoted by $R(e)$; 3) iteratively update and maintain $cTruss(e)$ according to the information passed on from the last iteration.

We use c to denote the current threshold of minimum truss value. This value signifies that all edges with truss value less that c have already been discovered in the graph. The minimum value of c for any graph is 2 as 0 is the minimum number of triangles that can be incident on an edge. In this implementation we do not filter the edges, rather we use the threshold to decide which edge participate in a particular iteration. The edges which do not satisfy the threshold condition pass through without any processing in all future iterations.

Figure 6.1 shows one iteration of the map-reduce approach for the edge (a, b) in graph G. In the first step the graph is transformed in to an edge centric key value view, where the key is the edge itself and the value is a tuple consisting of potential truss value of the edge and set of edge triangle relationship of $e = (a, b)$. This transformation of graph to edge centric view with triangle information is performed only once. As shown in figure 6.1(b) the edge (a, b) has three triangles (a, b, c), (a, b, d), (a, b, e) incident on it, therefore the edge centric view of the edge (a, b) states that the current truss value is 5 (3+2) and the edge triangle relationship contains edges $[ae, be, ac, bc, ad, bd]$. This initial edge centric view is then passed on to another map and reduce function pair. The map function uses the threshold c, for each record in the edge centric view it emits a record of the type $\langle e, cTruss(e), r_1(e), r_2(e)...r_{|R(e)|}(e)\rangle$ along with a set of key-value pairs of the form $\langle r_i(e), 0, e\rangle$ where $r_i(e)$ is an edge which shares edge triangle relationship with e i.e $r_i(e) \subset R(e)$. This step means e informs $r_i(e)$ that it still exists in the graph in the current iteration. Figure 6.1(c) shows the map phase operation on the edge ab with threshold value $c = 3$.

The output of the map phase is passed to a reduce operation where a collection of values of the form $\langle cTruss(e), e\rangle$ is received corresponding to an edge, the reduce task also takes as input the threshold c. The records gathered by the reduce function are of two types. As $cTruss(e)$ can never be equal to 0 for a edge record we use



(a) Original Sample Graph G

ab	5	ae	be
		ac	bc
		ad	bd
ae	3	ab	be
be	3	ab	ae
ac	3	ab	bc
bc	3	ab	ac
ad	3	ab	ac
bc	3	ab	ac

(b) Edge centric view

ab	5	ae	be
		ac	bc
		ad	bd

Input

ab	5	ae	be
		ac	bc
		ad	bd
bc	0	ab	
ae	0	ab	
.....			
ad	0	ab	

(c) Map Task on the edge 'ab'

ab	3	ae	be
		ac	bc
		ad	bd

Output

bc	0	ab	
.....			
ad	0	ab	
bc	3	ab	ac
ab	5	ae	be
		ac	bc
		ad	bd

(d) Reduce Task on the edge 'ab'

Figure 6.1.: One iteration of Spark-kTruss-MR on the edge (a, b)

this value to separate the two different types of records, if the first value in the tuple is 0 then this record shows the presence of an edge in this iteration generated from the map phase and not an original edge, in this case the second value in the tuple is appended to the list V_E otherwise the value is appended to the list V_T; $V_T = R(e)$. If $cTruss(e) < c$ we don't need to do anything as this edge is already in its optimal state. For $cTruss(e) \geq c$ we can validate the existence of the edges in $R(e)$ using the operation $V_T \cap V_E$ where V_E contains set of all edges which have not been filtered in the previous iteration. Truss value of e is updated as $\frac{|V_T \cap V_E|}{2} + 2$, if the updated $cTruss(e)$ doesn't satisfy the trussness threshold c, final truss value is set to $c - 1$. Figure 6.1(d) shows the reduce operation on edge (a, b).

Algorithm 7 Spark-kTruss - MapReduce

```
1:  procedure KTrussMR(Graph : G)
2:      G' ← EnumerateEdgeLocalTriangle(G)
3:      emap ← MapEdges(G')
4:      stop ← False
5:      c ← 2
6:      while ¬stop do
7:          repeat
8:              emap ← UpdateTrussnessMR(emap, c)
9:          until ∃e ∈ G.edges|e.oldTrussness ≠ e.trussness
10:         if ∃e ∈ G.edges|e.trussness > c then
11:             c ← c + 1
12:         else
13:             stop ← True
14:     return emap
```

Algorithm 7 gives a pseudo-code for the map-reduce approach explained above. In line 1 and 2 we enumerate the triangles incident on each edge using the EnumerateEdgeLocalTriangle routine and then map edges to form an RDD of the

form $RDD[\langle e, trussness(e), R(e)\rangle]$. Line 6 through 13 shows the iterative process of updating the truss value of each edge using the UpdateTrussnessMR subroutine. After each update process the status variable is checked to see if any of the edges have been update. If non of the edges get updated the threshold value is incremented by one as shown in line 10 - 11.

Algorithm 8 Spark-kTruss - MapReduce - UpdateTrusness

Input: The set of $I_2 : (e, trussness, R(e) \in edgeneighbour(e)) \in E$ and threshold c

Output: Same as the input format with updated trussness

1: **procedure** UpdateTrussMap($(edge, trusness, R(edge), c)$
2: **emit** (K=$edge$,val=$trusness, R(edge)$)
3: **for** $e_i \in R(edge)$ **do**
4: **emit** (K=e_i,val=$u, [edge]$)
5: **procedure** UpdateTrussReduce($e_m : edge, V : Iterable[(trussness, N)], c$)
6: $L \leftarrow$ EmptyList() and $L_0 \leftarrow$ EmptyList()
7: **for** $val \in V$ **do**
8: **if** $val.trusness == 0$ **then**
9: $L_0.append(val.N)$
10: **else**
11: $L.append(val.N)$ ▷ $N := \{r_1(e_m)...r_m(e_m) \in R(e_m)\}$
12: $s \leftarrow val.trussness$
13: **if** $s < c$ **then**
14: **Output** (e_m, s, L)
15: **else**
16: $T \leftarrow \{t : t \in L \cap L_0\}$
17: **if** $|T|/2 + 2 < c$ **then**
18: **Output** $(e_m, c - 1, L)$
19: **else**
20: **Output** $(e_m, |T|/2 + 2, L)$

Algorithm 8 provides the pseudo code for the map and reduce procedures which perform the update operation on each edge record at every iteration as shown in Algorithm 7. The UpdateTrussMap function makes a pass through the edge records and emits them, if the record satisfies the truss threshold; along with the edge record, the edge triangle relationships of the edge are also emitted, line 2 - 5 shows these operations. The UpdateTrussReduce function starts by separating the two kinds of records collected from the map phase using the truss value of a record as shown in line 6 through 12. In line 16 through 18 the algorithm checks how many of the edge triangle relationship of an edge remains in the current iteration and update truss value accordingly.

In Apache Spark we achieve the map operation of algorithm 8 using the flatMap routine and reduce operation is achieved by the combination of GroupByKey and Map routine. These functions are all applied on an of type $RDD[\langle e, trussness(e), R(e)\rangle]$, the functions transform the RDD iteratively until termination condition is meet. We also append a state variable with each edge record in an RDD to check if the record has been modified in a given iteration, this information is used to terminate the code i.e when no record in an RDD has been updated the process is terminated. On termination each record contains the exact truss value of the edge represented by the record.

6.1.2 Distributed graph parallel k-truss algorithm

The primary assumption of a distributed k-truss decomposition algorithm is that the input graph may or may not fit in the main memory of a single processing unit and the list of nodes and edges of the graph are stored in distributed manner across different machines in a cluster. The graph parallel computation is based on an edge centric abstraction suggested by [37]. We use an algorithm which uses the locality property of k-trusses for decomposition. The locality property of k-truss suggests that $\forall e \in E$, $truss(e) = k$ if and only if

1. $\exists E_k \subseteq nei(e)$ such that $|E_k| = 2(k-2)$, edges in E_k forms total $(k-2)$ triangles with e, and $\forall e' \in E_k, truss(e') \geq k$;

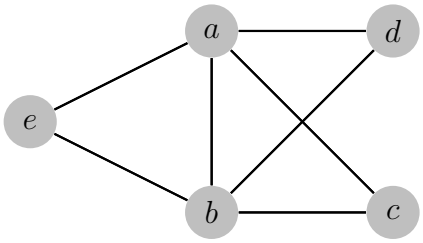
2. $\nexists E_{k+1} \subseteq nei(e)$ such that $|E_{k+1}| = 2(k-1)$, edges in E_{k+1} forms total $(k-1)$ triangles with e, and $\forall e' \in E_{k+1}, truss(e') \geq k+1$;

Thus, the truss value of an edge e, $truss(e)$, is the largest value k such that the edge e has exactly $2(k-2)$ neighboring edges whose truss value is greater than or equal to k. The property of locality enables the calculation of truss value of an edge based on the truss value of its edge-neighbors. This property of locality reduces the amount of large expensive communication between edge partitions which are located in different machines in a cluster.

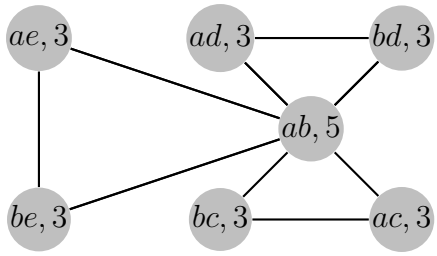
Most common graph parallel paradigms like Pregel are all vertex centric and k-truss is an edge centric computation. GraphX in Apache Spark also supports only vertex centric graph parallel abstraction there is no provision for edge centric graph parallel computation. So to achieve the graph parallel computation of k-truss we convert the graph to a line graph based on the definition given earlier. As defined earlier all edges in a graph is converted to vertexes in a line graph and two vertexes in a line graph are connected with each other if and only if they share an edge triangle relationship with each other.

The graph parallel computation runs on line graph(nodes and edges of the line graph will be marked with a subscript LG) and on convergence a vertex of the line-graph contains the truss value of an edge in the original graph. We use the triangle enumeration algorithm given above to construct the line graph. In a vertex-centric k-truss decomposition algorithm on the line graph, each node initializes its truss value to $(t+2)$ where t is the count of triangles incident on the edge (represented by the vertex in LG) in the original graph. Each node (say u_{LG}) then sends messages to its neighbors $v_{LG} \in N(u_{LG})$ with the current estimate of its (u_{LG}'s) truss value. For an undirected line 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

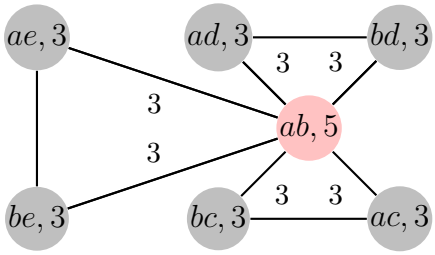
from its neighbors, vertex u computes the largest value p such that u has atleast p neighbors whose current truss estimate is p or more.



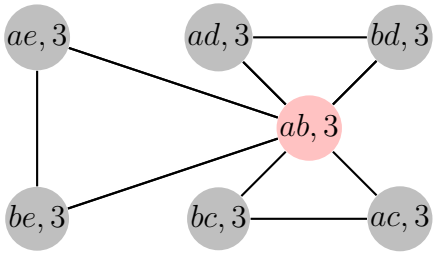
(a) Original Sample Graph G



(b) The line graph LG of G



(c) Messages direcetd to node 'ab'



(d) Node 'ab' updated after one iteration

Table for 'ab'

c	3
d	3
e	3

Truss Counter for 'ab'

Truss Size	1	2	3	4	5
Frequency	3	3	3	0	0

(e) Processing at node 'ab'

Figure 6.2.: One iteration of Spark-kTruss on the edge (a, b)

In Figure 6.2 we show one iteration of the truss update process for a node ab in the line graph LG representing an edge between a and b in original graph G. Figure 6.2(a) shows the original graph and Figure 6.2(b) shows the transformed line graph with

each node being an edge in the original graph initialized with the initial truss value of the edges which is two more than the number of triangles incident on the edge. Each node shares their current trussness value with its neighbors. Figure 6.2(c) shows neighbors sharing messages with node *ab*. On receiving a messages from neighbors, node *ab* constructs a mapping table, the table lists all uncommon vertices (from original graph) in the messages and all their corresponding minimum trussness as shown in Figure 6.2(d). We use the table to construct a frequency array and the largest index for which the cumulative sum from the end up to (including) that index is higher than the index value is the new truss value for the node (edge in original graph), the value is 3 in Figure 6.2(d).

Algorithm 9 KtrussSpark - Merge

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

Algorithm 9, 10, and 11 provide a pseudo-code of the required functions performed by each node in the line graph LG. It follows the property of locality that we have discussed above. This property of locality enables the calculation of truss value of a node in the line graph from the truss estimate of its neighbor nodes in the line graph in an iterative fashion, which makes it a think-like-an-edge distributed algorithm.

The upper bound of truss value of each node is the triangle incident on the edge in the original graph G so to begin with each node in line graph LG is initialized with k-truss value equal to two more than the count of triangles incident on the edge in graph G. Each vertex uv in LG runs the procedure MERGEMESSAGESKT followed by the UPDATENODEKT procedure, if the truss value of uv is changed (reduced), the updated truss value estimate is sent to all of uv's neighbors by the SENDMSGKT subroutine. In the MERGEMESSAGEKT subroutine, uv gathers all messages collected from its neighbors into a single message. The UPDATENODEKT procedure traverses through all the collected messages and creates a table where the key is given by $(w, v) \setminus (u, v)$ for message received from wv, and the value is the minimum truss value

Algorithm 10 KtrussSpark - Update

```
1: procedure UpdateNodeKT(Node u, Str msg)
2:     msgArray ← msg.Split(delimiter)
3:     T ← EmptyTable()
4:     for all m ∈ msgArray do
5:         (co, un) ← getCommonUncommon(m.edge.src, m.edge.dst)
6:         if un ∉ T ∨ T(un) > m.trusness then
7:             T(un) ← m.trusness
8:     for (un, ctrussness) ∈ T do
9:         j ← Min(u.trusness, ctrussness)
10:        count[j] + +
11:    for i := u.trussness to 3 do
12:        curWeight ← curWeight + count[i]
13:        if curWeight ≤ i − 2 then
14:            u.trussness ← i
15:            break
16:    return u
```

Algorithm 11 KtrussSpark - Propagate

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

for this key seen so far, as shown in line 4 to 7 in Algorithm 10. Line 8 to 10 in Algorithm 10 shows the construction of a frequency array which keeps a count of

each element in the table whose value is smaller than the current truss value of uv in. The frequency array is traversed in reverse and counts are summed up. The largest index whose cumulative frequency is greater than or equal to the index values is set as the updated truss value of the node (Edge in original graph); shown in line 11 to 16. In the third phase of operation, the SendMsgKT procedure sends out a message to a nodes neighbors if truss value of the node is updated. In Apache Spark the graphs are directed and messages only propagate in one direction so for undirected graphs rather than creating reverse edges to facilitate bi-directional communication we used a modified send message function which puts a message in input of both source and destination node of a directed edge. This receive-merge-update-broadcast iteration occurs until there are no more messages to process in any node in the graph.

6.2 Experiments

We perform experiments on both real-life datasets and synthetic datasets to test the performance of Spark-kTruss-MR, an iterative map-reduce based truss computation on Apache Spark, and Spark-kTruss, a graph parallel truss computation algorithm on Apache Spark. We compare the performance of both the methods in terms of running time and number of iterations required for convergence.

6.2.1 Experiment Setup

Cluster Configuration

Spark-k-truss 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-kTruss and Spark-kTrussMR on publicly available SNAP datasets (`snap.stanford.edu`). We perform our analysis on the following eight graph datasets: as-kitter, com-youtube, Amazon product co-purchasing network (amazon0601), Texas road network (roadNet-TX), California road network (roadNet-CA), Wikipedia Talk network (wiki-Talk).

To check the scaling of the algorithms we also use artificially generated datasets. We use Kronecker graph generation model [94]to generate our synthetic datasets. The Kronecker model is known to satisfy most of the properties shown by real world graphs. We use an identical Kronecker matrix $\{0.90.328; 0.350.39\}$ to generate 4 datasets of different scales. Using the same matrix for the different scale graphs ensure that all of them have the same attribute and they only vary in scale which helps us perform scalability study of our algorithms.

6.2.2 Experimental Results

Map-Reduce Implementation of k-truss on Apache Spark

In this section we will discuss the performance of the iterative map-reduce based k-truss decomposition algorithm on Apache Spark. We will test Spark-kTruss-MR on real life datasets mentioned above as well as on datasets generated by Kronecker‘s method. Table 6.1 summaries the results of Spark-kTruss-MR on 6 real world graphs which vary in size and structure. We also report the number of iterations taken by the algorithm to converge. We observe that the amount of time required for the decomposition to converge is proportional to the number of iterations and the number of edges in the graph. The algorithm in each iteration makes a scan over the entire edge RDD so this behavior is expected.

For synthetically generated graphs this behavior remains consistent. The synthetically generated graphs are controlled to have max k-truss value of 4. We perform this

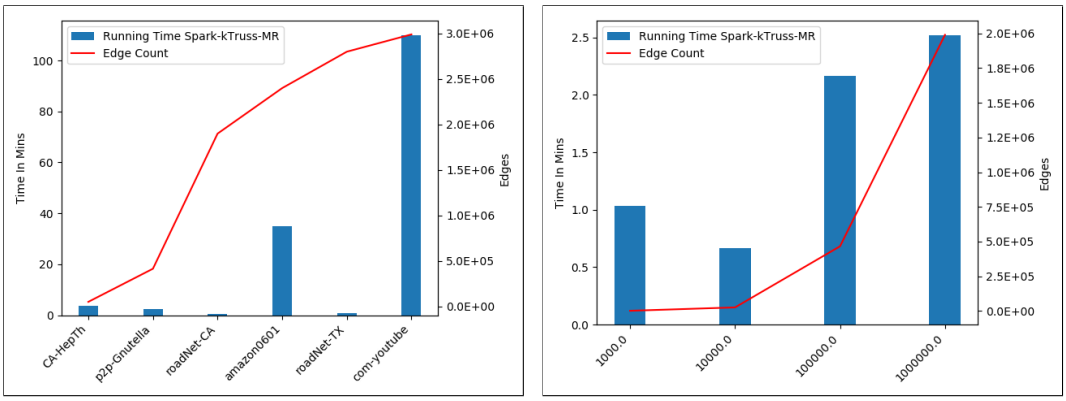
test with synthetic graphs to ascertain the scalability of the algorithm with growing size of graphs. These graphs take a small number of iteration to converge and the running time is mostly proportional to the number of edges $|E|$. Table 6.2 summarizes the results of Spark-kTruss-MR on 4 synthetic datasets of different scale.

Table 6.1.: Table of results showing the No. of vertices, Edges, Maximum k-truss, running time and the number of MapReduce iterations on real life graphs.

Dataset	**Vertices**	**Edges**	$K_{max}(G)$	**Iters**	$\mathbf{T_{mins}}$
CA-HepTh	9.9K	52K	32	64	3.6
p2p-Gnutella	6.3K	0.42M	5	23	2.28
roadNet-CA	1.4M	1.90M	4	7	0.44
amazon0601	0.4M	2.4M	11	130	35
roadNet-TX	2M	2.8M	4	7	0.67
com-youtube	1M	3M	19	400	110

Table 6.2.: Table of results showing the No. of vertices, Edges, Maximum k-truss, running time and the number of MapReduce iterations on 4 synthetic graphs.

DataSet Scale	**Vertices**	**Edges**	$K_{max}(G)$	**Iters**	$\mathbf{T_{mins}}$
10^3	1K	1.41K	4	5	1.03
10^4	16.4K	25.6K	4	4	0.67
10^5	0.26M	0.46M	4	10	2.1
10^6	1M	2M	4	12	2.5



(a) Real World Graphs (b) Synthetic Graphs

Figure 6.3.: Running time Vs number of edges for Spark-kTruss-MR

Graph Parallel Implementation of k-truss on Apache Spark

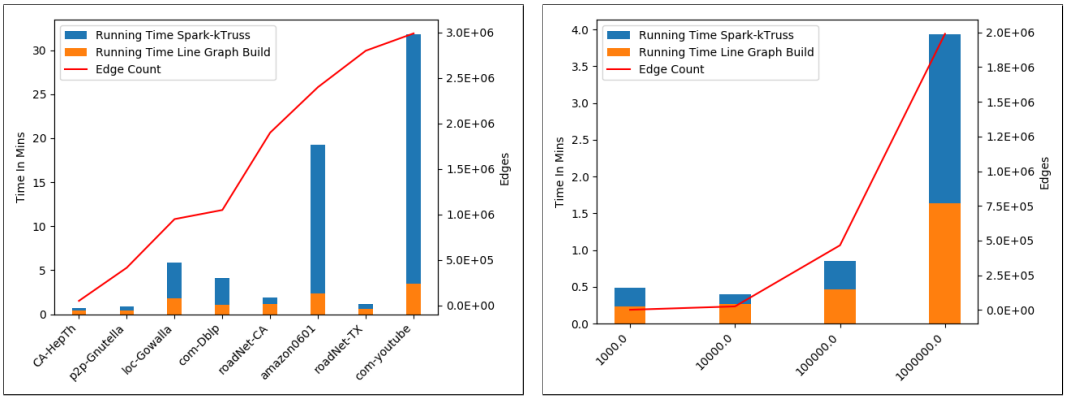
Here we provide the results related to the performance of the graph parallel implementation, Spark-kTruss. We also compare the performance of Spark-kTruss against the performance of Spark-kTruss-MR mentioned above. Table 6.3 shows execution summary of Spark-kTruss on 9 different real world large graph datasets, the table lists the total time taken by the implementation as well as the time taken to construct the line graph and the time taken for the pregel computation. The line graph computations is a one-time function and can be computed and stored. So for comparisons with the map-reduce implementation we will use the time taken by the pregel iterations. We use StorageLevel.MEMORY_ AND _ DISK persistence for nodes and edges of the linegraph. So with increasing graph size and iteration number spark will fall back to disk when it cannot hold an RDD completely in memory; this causes disk I\O resulting in performance drop.

For the real world datasets the running time show no trend with the number of edges. There are other factors like the density of the graph and the value of $K_{Max}(G)$ which also affect the running time. The running time is a function of the number of updates required for each edge.

For synthetic graphs shown in table 6.4 where the density and the $K_{Max}(G)$ value is controlled we see a clear trend of execution time increasing with the number of edges. We also observe that the ratio of line graph construction time to the pregel based decomposition time drops substantially with increasing graph size.

Table 6.3.: Table of results showing the No. of vertices, Edges, Maximum k-truss, running time and the number of Pregel iterations in Spark-kTruss on real life graphs.

Dataset	**Vertices**	**Edges**	$K_{max}(G)$	**Iters**	$\mathbf{T_{mins}}$
CA-HepTh	9.90K	52K	32	2	0.3
p2p-Gnutella	63K	0.41M	5	3	0.45
loc-Gowalla	0.19M	0.95M	29	7	4.05
com-Dblp	0.317M	1.05M	114	4	3.05
roadNet-CA	1.4M	1.9M	4	3	0.8
amazon0601	0.4M	2.4M	11	5	16.87
roadNet-TX	2M	2.8M	4	3	0.55
com-youtube	1.1M	3M	19	45	28



(a) Real World networks

(b) Synthetic networks

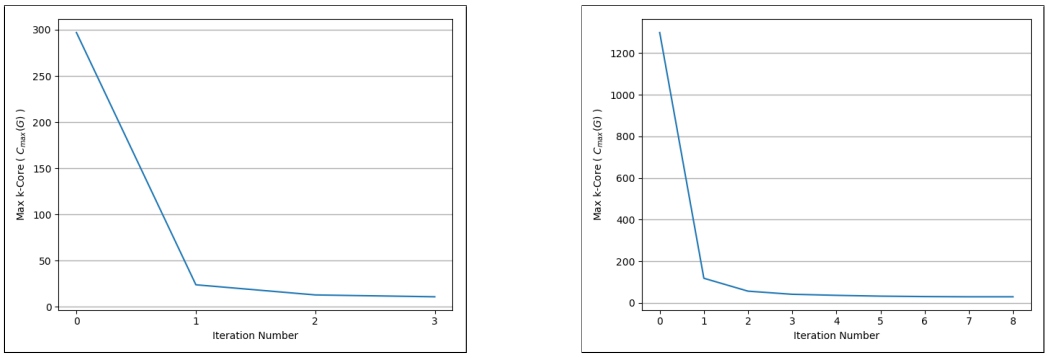
Figure 6.4.: Running time Vs number of edges for Spark-kTruss

Table 6.4.: Table of results showing the No. of vertices, Edges, Maximum k-truss, running time and the number of Pregel iterations in Spark-kTruss on on 4 synthetic graphs.

DataSet Scale	**Vertices**	**Edges**	$K_{max}(G)$	**Iters**	$\mathbf{T_{mins}}$
10^3	1K	1.41K	4	5	0.48
10^4	16.4K	25.6K	4	5	0.4
10^5	0.26M	0.46M	4	8	0.85
10^6	1M	2M	4	9	2.3

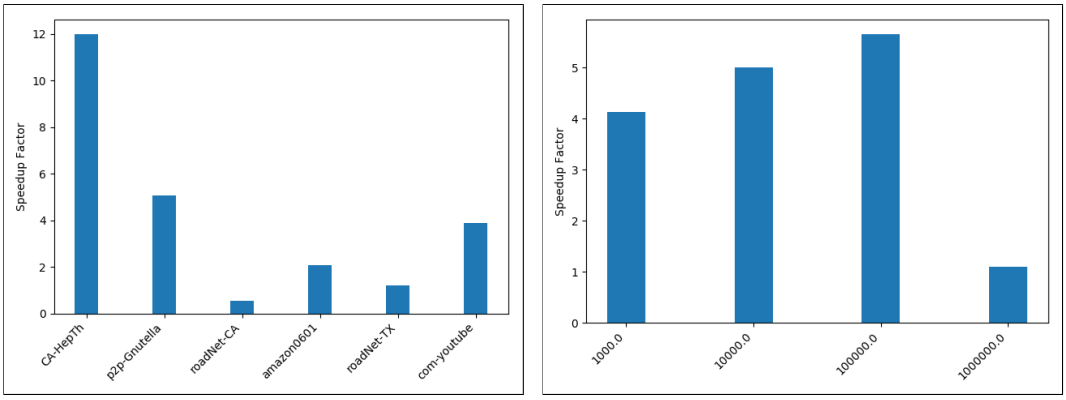
Convergence of Spark-kTruss

As part of the experiment, we also record the changes in the value of the max k-truss ($K_{max}(G)$) with each iteration till the value converges. Initially, the max k-truss value is equal to the maximum number of triangles incident on the edge + 2. Based on our experiment we see that the value of $K_{max}(G)$ drops very steeply in the first few iterations to a value close to the actual max k-truss value of the graph. After first few iterations, the rate of change in max k-truss value is slow till it converges. Figure 6.5 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-truss 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-truss value, we can get a very close estimate of the max k-truss value of the graph in a fraction of these iterations.



(a) amazon0601 (b) loc-Gowalla

Figure 6.5.: Convergence of max k-truss value.



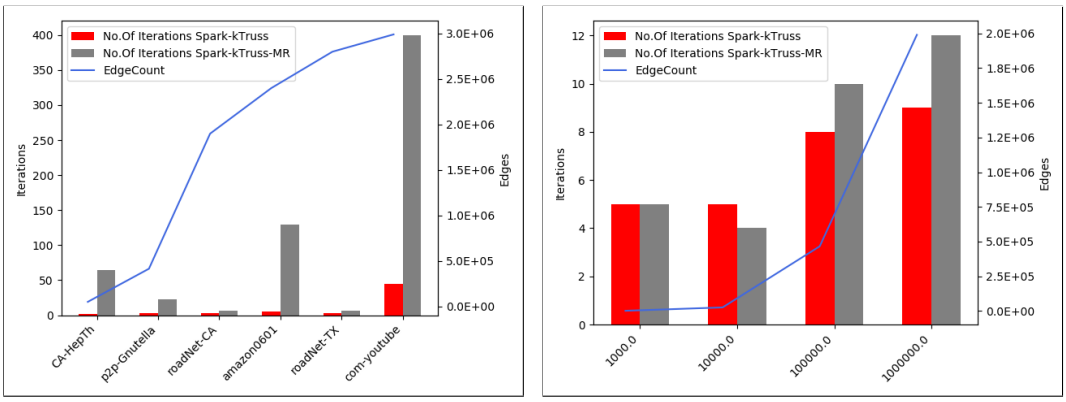
(a) Speedup achieved on real world networks (b) Speedup achieved on synthetic graphs

Figure 6.6.: Comparison between speedup of Spark-kTruss and Spark-kTruss-MR

Comparison between Spark-kTruss and Spark-kTruss-MR

The speedup achieved by Spark-kTruss over Spark-kTruss-MR generally grows with the number of edges in the graph but as the size of graph grows and spark flushes large RDD's to disk we see a drop in the speedup factor. With our experimental setup we noticed that graphs with edges in the scale of 10^6 spark starts flushing RDD's to disk.

Figure 6.6(a) shows the speed up achieved by Spark-kTruss on different real word graphs. Most of the real world graphs are large for our experimental setup and suffer from flush to disk effect. Figure 6.6(b) shows the speedup achieved by synthetic graphs of different scales and as stated above the speedup achieved increases with the size of graph but as scale becomes larger than 10^6 we see a drop in the speedup due to disk I/O



(a) Real World Graphs (b) Synthetic Graphs

Figure 6.7.: Comparison between Iterations of Spark-kTruss and Spark-kTruss-MR

We also compare the number of iterations taken by Spark-kTruss with Spark-kTruss-MR. Figure 6.7(a) shows the number of iterations taken by each of the real world graphs for both Spark-kTruss and Spark-kTruss-MR. We observed that Spark-kTruss always takes lesser number of iterations as compared to Spark-kTruss-MR. One interesting obeservation is that for road network graphs like roadNet-CA and roadnet-TX the difference is very small these graphs by virtue of there structure dont get much benefit from the graph parallel computations as compared to much denser graphs like amazon co-purchasing graph. Figure 6.7(b) shows the iteration comparison of the synthetically generated graphs, we observed that as we grow in scale the difference in number of iterations of Spark-kTruss and Spark-kTtruss-MR increases, with larger graphs the graph parallel computation leads to faster propagation of edge information to edge neighbors leading to faster convergence as compared to the Spark-kTruss-MR.

7. SUMMARY

In this work, we propose Spark-kCore, and Spark-kTruss two different decomposition methods of a graph on Apache Spark framework.

Spark-kCore is a Spark based distributed algorithm for k-core decomposition. This algorithm assigns characteristics to each vertex in a graph which can be used to partition the graph. 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 has significant improvement in terms of speedup and scalability. We also show that the graph parallel method of k-core computation proposed by us converges to a very close estimate of k-core in a small fraction of iteration, for large graphs this estimate can be used for most practical tasks.

Spark-kTruss is a Spark based distributed algorithm for k-truss decomposition. Unlike k-core k-truss is edge characteristic based partitioning of a graph. The proposed implementation is a graph parallel computation of k-truss relying on the locality property of k-truss. The proposed implementation scales horizontally as shown in the above section. We also compare our proposed algorithm with an iterative map-reduce based algorithm on Apache Spark and show that the graph parallel implementation takes lesser number of iterations to converge and is thus faster that the bottom up iterative map-reduce algorithm.

Both these suggested methods are good upper bounds to maximal cliques of a graph. These methods are computationally more efficient than maximal clique computation and can be used as a good partitioning measure for downstream tasks like community detections. These can also be used as a graph pruning measure to make the computation for maximal cliques more efficient.

8. FUTURE WORK

We have achieved some considerable speedup with the graph parallel algorithms of k-truss and k-core decomposition. We can further extend this work to perform intensive experiments of distribution on spark and effect of disk and network I/O.We can see how network latency, and disk I/O latency affect the performance of graph parallel algorithms on Apache Spark.

As a future scope we plan on deriving a mathematical upper bound on the number of iterations a bulk synchronous graph parallel message passing algorithm like Spark-kCore and Spark-kTruss, will take for all its nodes to converge. As of now we have a very loose upper bound which is equal to the maximum degree of a vertex in the graph.

k-core and k-truss are upper bounds to maximum clique in a graph. We can use k-core and k-truss to prune the vertexes and edges of a graph and then perform the maximum clique computation. This pre-processing of graph can speedup the computation of a maximum clique on that graph.

This research can be applied fast distributed community identification task using k-truss and k-core decomposition on large graphs.

In recent times a substantial interest has been shown in fast construction of hierarchical dense subgraphs. k-core and k-trusses are both hierarchical and are good measures of density. These algorithms for k-truss and k-core decomposition can be further extended to perform fast distributed hierarchical dense subgraph identification.

In a large database of graphs the proposed distributed algorithms can be used for identification of frequent subgraphs which uses the edge and vertex properties derived by k-core and k-truss decomposition respectively, to achieve partitioning.

REFERENCES

REFERENCES

[1] U. Alon, "Network motifs: theory and experimental approaches," *Nat Rev Genet*, vol. 8, no. 6, pp. 450–461, Jun. 2007.

[2] 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.

[3] M. E. J. Newman, "Spectral methods for network community detection and graph partitioning," *arXiv:1307.7729*, 2013.

[4] J. Pattillo, A. Veremyev, S. Butenko, and V. Boginski, "On the maximum quasi-clique problem," *Discrete Applied Mathematics*, vol. 161, no. 1-2, pp. 244–257, 2013.

[5] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 137–146.

[6] X. He, G. Song, W. Chen, and Q. Jiang, "Influence blocking maximization in social networks under the competitive linear threshold model," in *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM, 2012, pp. 463–474.

[7] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen, "On socio-spatial group query for location-based social networks," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 949–957.

[8] D.-N. Yang, Y.-L. Chen, W.-C. Lee, and M.-S. Chen, "On social-temporal group query with acquaintance constraint," *Proceedings of the VLDB Endowment*, vol. 4, no. 6, pp. 397–408, 2011.

[9] B. Balasundaram, S. Butenko, and I. V. Hicks, "Clique relaxations in social network analysis: The maximum k-plex problem," *Operations Research*, vol. 59, no. 1, pp. 133–142, 2011.

[10] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[11] R. D. Luce and A. D. Perry, "A method of matrix analysis of group structure," *Psychometrika*, vol. 14, no. 2, pp. 95–116, 1949.

[12] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[13] R. D. Luce, "Connectivity and generalized cliques in sociometric group structure," *Psychometrika*, vol. 15, no. 2, pp. 169–190, 1950.

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

[15] R. J. Mokken, "Cliques, clubs and clans," *Quality and quantity*, vol. 13, no. 2, pp. 161–173, 1979.

[16] H. Matsuda, T. Ishihara, and A. Hashimoto, "Classifying molecular sequences using a linkage graph with their pairwise similarities," *Theoretical Computer Science*, vol. 210, no. 2, pp. 305–325, 1999.

[17] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 228–238.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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*, 2013.

[24] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Advances in neural information processing systems*, 2006, pp. 41–50.

[25] M. Á. Serrano, M. Boguná, and A. Vespignani, "Extracting the multiscale backbone of complex weighted networks," *Proceedings of the national academy of sciences*, vol. 106, no. 16, pp. 6483–6488, 2009.

[26] M. Altaf-Ul-Amin, Y. Shinbo, K. Mihara, K. Kurokawa, and S. Kanaya, "Development and implementation of an algorithm for detection of protein complexes in large interaction networks," *BMC bioinformatics*, vol. 7, no. 1, p. 207, 2006.

[27] S. Pandit, D. H. Chau, S. Wang, and C. Faloutsos, "Netprobe: a fast and scalable system for fraud detection in online auction networks," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 201–210.

[28] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Physica A: statistical mechanics and its applications*, vol. 390, no. 6, pp. 1150–1170, 2011.

[29] N. Korovaiko and A. Thomo, "Trust prediction from user-item ratings," *Social Network Analysis and Mining*, vol. 3, no. 3, pp. 749–759, 2013.

[30] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, "A survey of algorithms for dense subgraph discovery," in *Managing and Mining Graph Data*. Springer, 2010, pp. 303–336.

[31] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[32] S. Wasserman and K. Faust, *Social network analysis: Methods and applications*. Cambridge university press, 1994, vol. 8.

[33] M. E. Newman, D. J. Watts, and S. H. Strogatz, "Random graph models of social networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl 1, pp. 2566–2572, 2002.

[34] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, vol. 16, 2008.

[35] 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.

[36] 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.

[37] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 471–480.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms second edition," 2001.

[39] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[40] A. Pothen, "A. pothen, h. simon, and k.-p. liou, siam j. matrix anal. appl. 11, 430 (1990)." *SIAM J. Matrix Anal. Appl.*, vol. 11, p. 430, 1990.

[41] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak mathematical journal*, vol. 23, no. 2, pp. 298–305, 1973.

[42] J. W. Liu, "A graph partitioning algorithm by node separators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 15, no. 3, pp. 198–219, 1989.

[43] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, 2004, pp. 81–90.

[44] L. R. Ford Jr and D. R. Fulkerson, *Flows in networks*. Princeton university press, 2015.

[45] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[46] R. Guimera, L. Danon, A. Diaz-Guilera, F. Giralt, and A. Arenas, "Self-similar community structure in a network of human interactions," *Physical review E*, vol. 68, no. 6, p. 065103, 2003.

[47] M. E. Newman, "Detecting community structure in networks," *The European Physical Journal B*, vol. 38, no. 2, pp. 321–330, 2004.

[48] J. Scott, "Social network analysis: A handbook. sage london," *2nd edtion*, 2000.

[49] M. J. Fischer, "Efficiency of equivalence algorithms," in *Complexity of Computer Computations*. Springer, 1972, pp. 153–167.

[50] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 215–225, 1975.

[51] ——, "Decomposition by clique separators," *Discrete mathematics*, vol. 55, no. 2, pp. 221–232, 1985.

[52] J. E. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM Journal on Computing*, vol. 2, no. 3, pp. 135–158, 1973.

[53] D. R. White and F. Harary, "The cohesiveness of blocks in social networks: Node connectivity and conditional density," *Sociological Methodology*, vol. 31, no. 1, pp. 305–359, 2001.

[54] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[55] J. R. Tyler, D. M. Wilkinson, and B. A. Huberman, "Email as spectroscopy: Automated discovery of community structure within organizations," in *Communities and technologies*. Springer, 2003, pp. 81–96.

[56] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. 9, pp. 2658–2663, 2004.

[57] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[58] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 2006, pp. 475–486.

[59] J. R. Gilbert and E. Zmijewski, "A parallel graph partitioning algorithm for a message-passing multiprocessor," *International Journal of Parallel Programming*, vol. 16, no. 6, pp. 427–449, 1987.

[60] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[61] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 1222–1230.

[62] G. Karypis, K. Schloegel, and V. Kumar, "Parmetis: Parallel graph partitioning and sparse matrix ordering library," *Version 1.0, Dept. of Computer Science, University of Minnesota*, p. 22, 1997.

[63] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[64] 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.

[65] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 9–16.

[66] 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.

[67] 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.

[68] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: https://doi.org/10.14778/2212351.2212354

[69] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.

[70] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: algorithms and evaluation," *The VLDB Journal*, vol. 25, no. 3, pp. 425–447, 2016.

[71] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, 2018.

[72] C. Aggarwal and K. Subbian, "Evolutionary network analysis: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, p. 10, 2014.

[73] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 16–24.

[74] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2002, pp. 623–632.

[75] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2006, pp. 253–262.

[76] M. Jha, C. Seshadhri, and A. Pinar, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 589–597.

[77] Y. Lim and U. Kang, "Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 685–694. [Online]. Available: http://doi.acm.org/10.1145/2783258.2783285

[78] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 607–614.

[79] H.-M. Park and C.-W. Chung, "An efficient mapreduce algorithm for counting triangles in a very large graph," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 539–548.

[80] S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: A parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 529–538.

[81] H. Avron, "Counting triangles in large graphs using randomized matrix trace estimation," in *Workshop on Large-scale Data Mining: Theory and Applications*, vol. 10, 2010, pp. 10–9.

[82] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 804–811.

[83] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu, "Opt: a new framework for overlapped and parallel triangulation in large-scale graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 637–648.

[84] M. Rahman and M. Al Hasan, "Approximate triangle counting algorithms on multi-cores," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 127–133.

[85] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 149–160.

[86] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[87] S. Chu and J. Cheng, "Triangle listing in massive networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 4, p. 17, 2012.

[88] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1311–1322.

[89] X. Huang, W. Lu, and L. V. Lakshmanan, "Truss decomposition of probabilistic graphs: Semantics and algorithms," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 77–90.

[90] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," *arXiv preprint arXiv:1707.02000*, 2017.

[91] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–6.

[92] Apache.org. (2017) Graphx- spark 2.2.0 documentation.

[93] 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.

[94] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

VITA

VITA

Aritra Mandal

EDUCATION

- Masters of Science, May 2018
  Purdue University, Indianapolis
  Department of Computer and Information Science
- Bachelor of Engineering, May 2013
  Birla Institute of Technology, Mesra, India
  Department of Computer Science and Engineering

TEACHING EXPERIENCE

- CSCI 53600 - Teaching Assistant - Data Communication & Computer Networks

CORPORARE EXPERIENCE

- PayPal Inc. - May 2017 to Aug 2017 - Machine Learning Engineer (Intern)
- Genpact - Aug 2013 to July 2016 - Machine Learning and Big Data Engineer

PUBLICATIONS

2017 IEEE International Conference on Big Data (BIGDATA)

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.**

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.

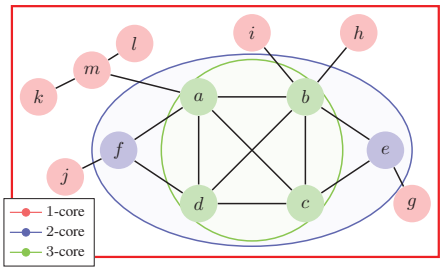


Fig. 1: A toy graph and its k-core decomposition

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

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

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.

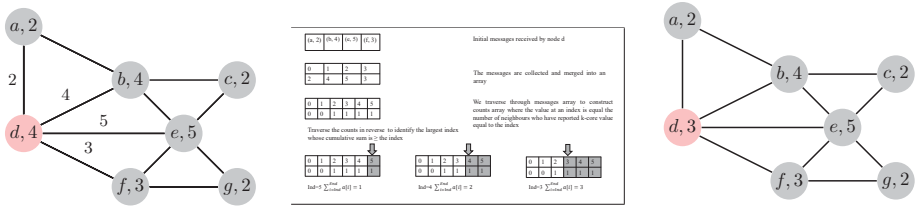
III. METHODS

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

(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 ← msg.Split(delimiter)
3:     for all m ∈ msgArray do
4:         if m ≤ u.kcore then
5:             count[m] + +
6:         else
7:             count[u.kcore] + +
8:     for i := k to 2 do
9:         curWeight ← CurWeight + count[i]
10:        if curWeight ≥ i then
11:            u.kcore ← i
12:            break
13:    return u
```

Algorithm 3 KcoreSpark - Propagate

```
1: procedure SENDMSG(EdgeTriplet triplets)
2:     srcVertex ← triplet.getSrcAttr()
3:     destVertex ← triplet.getDstAttr()
4:     I ← 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

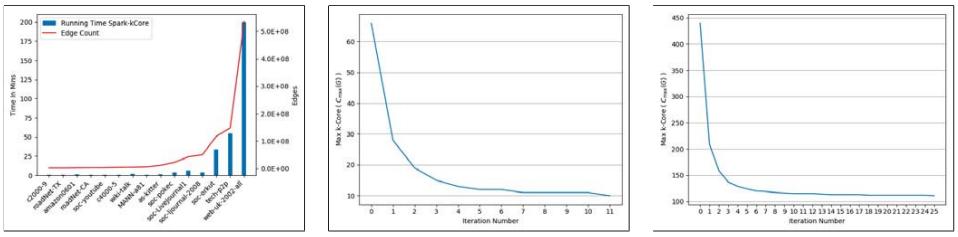
Setup: Spark-kCore is implemented in Scala and the experiments are conducted on a cluster of 8 machines, each having Intel i7, 2.2Ghz CPU, and 16 GB RAM, running CentOS (Linux). The hard disk is Seagate Constellation ST2000NM0033-9ZM 2TB 7200 RPM.

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

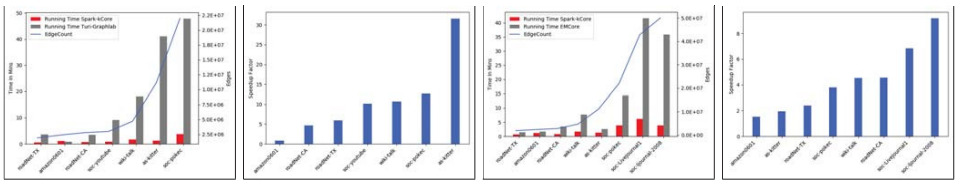
Dataset	$C_{max}(G)$	$\mathbf{T_{spark}}$(mins)	$\mathbf{T_{GraphLabs}}$(mins)
as-skitter	111	1.3	41
soc-youtube	51	0.9	9.1
wiki-talk	131	1.7	18.2
amazon0601	10	1.1	0.9
roadNet-CA	3	0.75	3.5
roadNet-TX	3	0.6	3.5
soc-pokec	47	3.8	47.5

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.