K-Core Decomposition of Large Networks on a Single PC

Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, Alex Thomo
University of Victoria, BC, Canada
{wissamkhaouid,mgbarsky}@gmail.com, {srinivas,thomo}@uvic.ca

ABSTRACT

Studying the topology of a network is critical to inferring underlying dynamics such as tolerance to failure, group behavior and spreading patterns. k-core decomposition is a well-established metric which partitions a graph into layers from external to more central vertices. In this paper we aim to explore whether k-core decomposition of large networks can be computed using a consumer-grade PC. We feature implementations of the "vertex-centric" distributed protocol introduced by Montresor, De Pellegrini and Miorandi on GraphChi and Webgraph. Also, we present an accurate implementation of the Batagelj and Zaversnik algorithm for k-core decomposition in Webgraph. With our implementations, we show that we can efficiently handle networks of billions of edges using a single consumer-level machine within reasonable time and can produce excellent approximations in only a fraction of the execution time. To the best of our knowledge, our biggest graphs are considerably larger than the graphs considered in the literature. Next, we present an optimized implementation of an external-memory algorithm (EMcore) by Cheng, Ke, Chu, and Özsu. We show that this algorithm also performs well for large datasets, however, it cannot predict whether a given memory budget is sufficient for a new dataset. We present a thorough analysis of all algorithms concluding that it is viable to compute k-core decomposition for large networks in a consumer-grade PC.

1. INTRODUCTION

Connections between people or entities are modeled as graphs, where vertices represent the people or entities, and edges represent the connections. Many big graphs have been constructed this way coming from a multitude of systems and applications, such as social and web networks, product co-purchases, and protein interaction networks, to name a few. Analyzing the graph structure has been shown to be highly beneficial in targeted advertising [36], fraud-detection [27], missing link prediction [23, 18], locating functional modules of interacting proteins [33, 16], identifying new emerging trends in scientific disciplines [5], and so on. One of the most important tasks in analyzing graphs for these applications is detecting communities of graph nodes that have close ties with each other [13, 14, 28, 34].

The nodes in the aforementioned networks often exhibit a crucial property: their utility increases or decreases depending on the number of connections they have with other nodes in their community [8]. This is especially pronounced in social networks, where the engagement of users is more likely if many of their friends are engaged. From the opposite point of view, the disengagement of users is a serious concern for social network providers. If sparsely connected users drop out, then their friends in turn might become sparsely connected and drop out too, thus causing a possibly dangerous cascade of disengagement. Therefore, a social network provider needs to know what part of the network will remain active after such an iterated disengagement for different levels of a parameter k, which represents the least number of friends a user needs to remain in the network. This corresponds to a well-known concept in graph theory, that of k-core of a graph G, which is the largest induced subgraph of G in which every vertex has degree at least k. The coreness of a vertex v in G is the largest value of k such that there is a k-core of G containing v. In the k-core decomposition problem, the goal is to compute the coreness of each vertex in G.

k-Core decomposition has many other applications. It is extensively used in aiding the visualization of the network structure [3, 30], understanding and interpreting cooperative processes in social networks [15, 12], capturing structural diversity in social contagion [32], analyzing complex networks in terms of node hierarchies, self-similarity, and connectivity [2], describing protein functions based on protein-protein networks [1, 21], exploring collaboration in software teams [35], facilitating network approaches for large text summarization [4], and approximating hard to compute network centrality measures [17].

In the theory community, the concept of k-core has enjoyed much popularity because it can be used as subroutine for harder problems, such as computing cliques of size k, or provide an approximation for the densest subgraph problem, and the densest at-least-k-subgraph problem [20].

In contrast to computing other kinds of cohesive groups in networks, e.g. s-plexes, n-cliques, s-defective cliques [29], k-cores can be computed in polynomial time. As such, k-core decomposition is more amenable to use in network analysis of large graphs. Our goal in this paper is to engineer existing k-core algorithms to scale to large graphs of billions of

Proceedings of the VLDB Endowment, Vol. 9, No. 1

edges. The largest graph we are able to handle in this paper, Twitter 2010, consists of 41.7 million nodes and 2.4 billion edges. To the best of our knowledge, this is considerably bigger than the graphs considered in literature. Furthermore, we are able to compute the core decomposition using only a consumer grade PC. The algorithms we engineer, our objectives and contributions are described in detail below.

1.1 Algorithms, Objectives, and Contributions

Montresor, De Pellegrini and Miorandi in [25] present a distributed k-core decomposition algorithm following the "vertex-centric" model of computation. It operates on the premise that the input graph is spread across multiple cluster nodes or hosts. In the case where the large graph resides on a single machine's disk, resorting to the distributed approach for k-core decomposition can be challenging. Although distributed resources are readily available (e.g. Amazon EC2), the task of partitioning the graph across computational nodes in a way that optimizes communication amongst them is a hard problem.

We ask the following question: Is there a viable approach for efficiently computing the k-core decomposition of large networks without resorting to distributed computation, and preferably using a consumer-grade PC? We turn to GraphChi [19] and Webgraph [9]. GraphChi is a modern, general-purpose, graph engine which employs a novel technique for processing large data from disk and uses the "vertex-centric" model of computation. Webgraph is a graph compression framework that provides an API for accessing a compressed graph using lazy techniques that delay the decompression until it is actually necessary. In this paper, we present implementations of the protocol introduced in [25] on GraphChi and Webgraph. We perform our experimental analysis on a variety of real-world graph datasets ranging from a few thousands to billions of edges in size including those used in [25]. We show that our implementations scale to significantly larger datasets than those in [25] using only a single PC. This is in contrast to a cluster of 16 machines employed by [25].

We also present an accurate implementation of the Batagelj and Zaversnik (BZ) algorithm for k-core decomposition in Webgraph. When the graph can fit in memory, the BZ algorithm outperforms all the others. Due to the truly impressive compression ratio that Webgraph can achieve, it is possible to fit many large graphs in memory. The BZ algorithm is a very good choice in such a case. The real power of BZ comes not from its high-level idea (for which it is often cited), but from a careful engineering of its data structures.

When the graph does not fit in memory, in terms of algorithms for a single machine, the main competition is with external-memory (EM) algorithms. A well-known EM algorithm for k-core decomposition of massive networks is *EMcore* proposed by Cheng, Ke, Chu, and Özsu in [11]. Another question that we ask is: How do our implementations on GraphChi and Webgraph compare with a special-purpose algorithm, such as EMcore? To answer this question we provide an optimized implementation of the EMcore algorithm. We observe that the Webgraph implementation is faster, whereas the GraphChi implementation is slower than EMcore. However, the Webgraph implementation and EMcore have certain requirements on the available memory budget, whereas GraphChi does not.

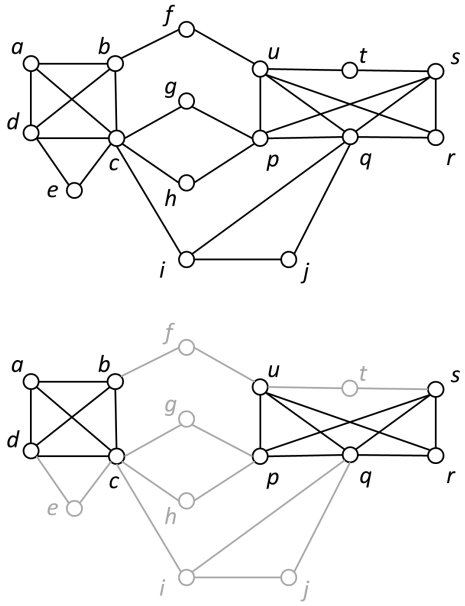

Figure 1: [Top] A graph. [Bottom] The 3-core of the graph.

An advantage of using our GraphChi or Webgraph implementations is their simplicity, facilitating extension and maintenance. This is because GraphChi and Webgraph are general-purpose graph systems providing a simple API for processing the graph. On the other hand, EMcore is a special-purpose algorithm, where the programmer needs to implement many details related to the graph storage.

The rest of the paper is organized as follows. In Section 2 we give the necessary notation and definitions. In Section 3 we describe the setup of our experiments. In Section 4 we explain our GraphChi implementation. In Section 5 we give details about our implementations of the BZ algorithm and Montresor et. al. using Webgraph. In Section 6 we describe our optimized EMcore implementation. Finally, Section 7 concludes the paper.

2. PRELIMINARIES

We represent networks, for the purpose of computing the k-core decomposition, using undirected graphs.

We denote an undirected graph by $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We set n and m to be $|V|$ and $|E|$, respectively.

Given a vertex v, we denote the set of its neighbors, $\{u : (u, v) \in E\}$, by $N_G(v)$. The degree of v, $|N_G(v)|$, is denoted by $d_G(v)$. We set $d_{\max}(G) = \max\{d_G(v) : v \in V\}$.

Let $K \subseteq V$ be a subset of vertices of a graph $G = (V, E)$. We have the following definitions.

DEFINITION 1. *Graph $G(K) = (K, E_K)$, where $E_K = \{(u, v) \in E : u, v \in K\}$ is called* the subgraph of G induced by K.

DEFINITION 2. *$G(K)$ is a* k-core *if and only if the following conditions are true.*

k-Degree *For each $v \in K$, $d_{G(K)}(v) \geq k$.*

Maximality *For each K', such that $K \subset K' \subseteq G$, there exists $u \in K' \setminus K$, such that $d_{G(K')}(u) < k$.*

Name	Abbrev.	$\|V\|$	$\|E\|$	d_{max}	k_{max}	k_{avg}	i
Astro Physics (*ca-astroph*)	AP	18.7 K	198.1 K	504	56	12.62	53
Condensed Matter (*ca-condmat*)	CM	23.1 K	93.5 K	280	25	4.90	15
Gnutella P2P network (*p2p-gnutella31*)	GN	62.6 K	147.9 K	95	6	2.52	32
Slashdot 1 (*soc-sign-Slashdot-090221*)	S1	82.1 K	500.5 K	2,548	54	6.23	17
Slashdot 2 (*soc-Slashdot0902*)	S2	82.2 K	543.4 K	2,553	56	7.22	20
Amazon product co-purchasing network (*amazon0601*)	AM	0.4 M	2.4 M	2,752	10	7.22	27
Berkeley-Stanford web graph (*web-BerkStan*)	BS	0.7 M	6.6 M	84,230	201	11.11	243
Texas road network (*roadNet-TX*)	TX	1.4 M	1.9 M	12	3	1.79	122
California road network (*roadNet-CA*)	CA	2.0 M	2.8 M	12	3	1.81	72
Wikipedia Talk network (*wiki-Talk*)	WT	2.4 M	4.7 M	100,029	131	1.96	25
LiveJournal network (*LiveJournal*)	LJ	4.8 M	43.1 M	20,334	373	9.48	53
UK 2005 web crawl (*uk-2005*)	UK	39.5 M	790 M	1,776,858	589	24.20	250
Twitter 2010 followers network (*twitter-2010*)	TW	41.7 M	2,405 M	2,997,487	2,488	29.04	63

Table 1: Dataset name, abbreviation, number of vertices, number of edges, maximum degree, maximum coreness, average coreness, number of GraphChi iterations until completion. Dashed lines divide the datasets into small, medium, and big.

From the maximality condition in the above definition the following proposition follows.

PROPOSITION 1. *For each $k = 1, 2, \ldots, d_{\max}(G)$, there exists exactly one k-core in G (which could possibly be empty).*

Given $k \in [1, d_{\max}(G)]$, we denote the k-core of G by $C_k(G)$. Finally, we have

DEFINITION 3. *A vertex $v \in G$ has coreness k if and only if it is a vertex in graph $C_k(G)$.*

For an example, see Fig. 1. On the top is a graph and at the bottom is its 3-core.

Vertices e, f, g, h, and j have a degree of two, so they cannot be in the 3-core.

Observe that vertex i, even though it has a degree of three, does not belong in the 3-core of the graph. This is because node j, one of the neighbors of i, has a degree of two.

There are two connected components in the 3-core, a, b, c, d, and p, q, r, s, u. Each of these vertices has at least three neighbors in same connected component.

The 2-core, in this example, happens to be the whole graph.

Finally, the 4-core is empty. Even though we have several vertices with a degree of four, b, c, d, p, q, s, and u, once their neighbors with degree three or less are removed from consideration, their degree becomes less than four. Hence they do not belong in the 4-core.

The cores of the graph form an inclusion relationship. For a graph G, we have

$$C_{d_{\max}}(G) \subset C_{d_{\max}-1}(G) \subseteq \ldots \subseteq C_1(G).$$

3. EXPERIMENTAL METHODOLOGY

Setup. Our implementations are in Java and the experiments are conducted on a machine with Intel i7, 2.2Ghz CPU, and 8Gb RAM, running Ubuntu 14.03 (Linux). The hard disk is Seagate Barracuda ST31000524AS 1TB 7200 RPM.

Datasets. We perform our analysis on the following thirteen graph datasets: Astro Physics (*ca-astroph*) and Condensed Matter (*ca-condmat*) collaboration networks; Gnutella P2P network (*p2p-gnutella31*); two datasets captured from the *Slashdot social network*: *soc-sign-Slashdot-090221*, and *soc-Slashdot0902*; Amazon product co-purchasing network (*amazon0601*); Berkeley-Stanford web graph (*web-BerkStan*); Texas road network (*roadNet-TX*); California road network (*roadNet-CA*); Wikipedia Talk network (*wiki-Talk*); LiveJournal social network (*LiveJournal*); 2005 crawl on the *.uk* domain (*uk-2005*); 2010 snapshot of the Twitter network (*twitter-2010*).

The dataset characteristics are given in Table 1. We show the name and abbreviation, number of vertices, number of edges, maximum degree, maximum coreness, average coreness, and number of GraphChi iterations until completion.

We have divided the datasets into three groups, small, medium, and big. The first five datasets (small) have less than 1 million edges. The next six datasets (medium) have from 2.4 to 43.1 million edges. Finally, the last two datasets (big) have 790 million and 2.4 billion edges, respectively.

All graphs, with the exception of the last two, are obtained from `http://snap.stanford.edu/data/index.html`.

The last two are obtained from `http://law.di.unimi.it/webdata`.

Undirected graphs have been converted to directed graphs by replacing each edge with two directed edges.

4. GRAPHCHI IMPLEMENTATION

Our implementation of the algorithm of [25] is written for GraphChi [19]. It is motivated by the fact that GraphChi adheres to the "think-like-a-vertex" paradigm, introduced in Pregel [24], the framework also used in [25].

In this model, a computation on a graph runs an *update(v)* function operating on a vertex v by accessing and modifying its value along with its attached edges. Function *update(v)* is carried out for each scheduled vertex iteratively.

Similar to Pregel, GraphChi is based on communication between vertices. However, instead of using explicit messaging, a GraphChi program implements the sending of a message to adjacent vertices by writing to out-edges, and the receiving of an incoming message by reading in-edge values.

GraphChi implements a novel method, Parallel Sliding Windows (PSW), for processing very large graphs from external memory. PSW reduces random access to a negligible amount, and thus performs well on data stored in disk.

A nice feature of GraphChi is selective scheduling. Using this feature we only schedule for update those vertices that have a chance to change their state.

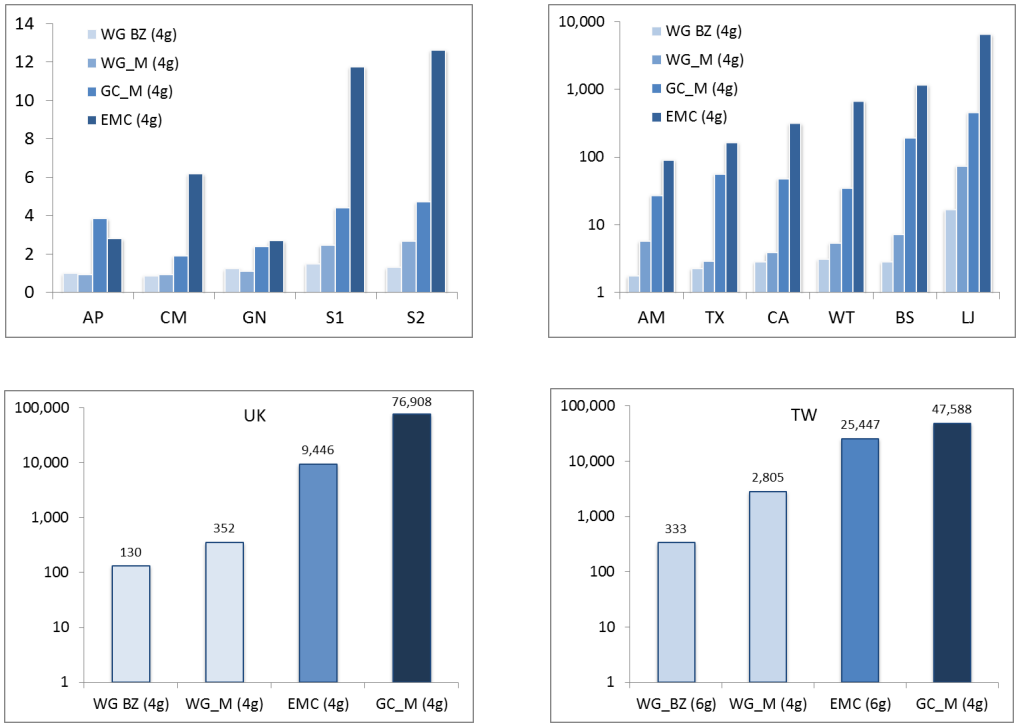

Figure 2: Running times in **seconds** for the small, medium, UK, and TW datasets (from left to right). Algorithm labels are: WG_BZ for Batagelj and Zaversnik [6] on Webgraph, WG_M and GC_M for Montresor et.al. [25] on Webgraph and GraphChi respectively, and EMC for EMcore [11]. Memory allocated is in parenthesis.

The main idea of the algorithm of [25] is to maintain an estimate, called *vertex value*, for each vertex that is an upper-bound on its coreness. This estimate is initialized to be the vertex degree. As the execution progresses, this upper-bound is tightened further in each iteration using an *update()* step that we describe below, gradually improving the estimate for each vertex until it reaches the true core value before termination.

The *update()* step computes a tighter upper-bound on the vertex's coreness based on values read from in-edges. In each iteration, vertices broadcast their current upper-bound to their neighbors by writing to out-edges. Vertices read the in-edge values coming from its neighbors and create a *count array* c indexed by the upper-bound values they read.

For example, if a vertex v has three neighbors with upper-bound value of 7, then $c[7] = 3$. Let n_v be the length of c for v. Then, v determines the largest index i such that

$$i \leq \sum_{k=i}^{n_v} c[k]. \tag{1}$$

The new upper-bound for v will be i; vertex v cannot have core greater than i because there are fewer than i neighbors with core greater than i. See algorithms 1 and 2 for more details.

In lines 2–5, the vertex value is initialized to its degree and is broadcast to the neighbors. Also the vertex is scheduled for the next iteration. This only happens in iteration 0.

Lines 6–17 execute in the other iterations. In line 7, we set a local estimate (*localEstimate*) to be the computed upper bound on the coreness of the vertex. If the local estimate is less than the current vertex value, the latter is updated to the former, and the new vertex value is broadcast to all the neighbors of the vertex.

Here we also contribute a simple optimization using selective scheduling in GraphChi. Lines 12–16 of Algorithm 1 schedule (for the next iteration) only those neighbors with an estimate of coreness that is greater or equal to the estimate of the current vertex.

A vertex will be scheduled only if it has a neighbor with a lower estimate. This neighbor will perform a

$$scheduler.\text{addTask}(inEdge.vertexid)$$

action (see line 14).

A vertex can be scheduled by several neighbors, but this is not a problem as addTask is idempotent and does not cause any performance penalty.

If a vertex is not scheduled, it does not have a neighbor with lower estimate, so it does not have a chance to lower its core estimate.

Algorithm 2 implements the idea of inequality 1. We initialize to zero all the elements of array c. The array only needs to have up to the current vertex value (coreness estimate) elements. Observe in line 6 that any neighbor with a value greater than the vertex value contributes to incrementing the last element of the array. In lines 10–15, we find the largest index in the array that satisfies inequality 1. Variable *cumul* corresponds to i in inequality 1.

For an example see the graph in Fig. 1. Initially, in iteration 0, each node sets it local estimate of coreness to its

degree. Then, the message exchange starts in iteration 1.

Bound tightening is quite effective. If we assume that the vertices are processed in alphabetic order, then, in this example, all the vertices get a local estimate for their core value, which happens to be exactly their true core value. In the following table, we show the array of counts for each vertex and the bound obtained by Algorithm 2.

Vertex	Array c	Bound
a	$[2:0, 3:3]$	3
b	$[2:1, 3:1, 4:2]$	3
c	$[2:3, 3:2, 4:2, 5:0, 6:0, 7:0]$	3
d	$[2:1, 3:3, 4:0]$	3
e	$[2:2]$	2
f	$[2:2]$	2
g	$[2:2]$	2
h	$[2:2]$	2
i	$[2:1, 3:2]$	2
p	$[2:2, 3:2, 4:0, 5:2]$	3
q	$[2:2, 3:2, 4:1, 5:1, 6:0]$	3
r	$[2:0, 3:3]$	3
s	$[2:1, 3:3, 4:0]$	3
t	$[2:2]$	2
u	$[2:1, 3:3, 4:0, 5:0]$	3

For instance, consider vertex c. It has a degree of seven, hence the number of elements in its array of counts is seven. Now, what is the greatest index, such that, the sum of array elements starting from this index to the end of the array is greater or equal to the value of the index? It is three in this instance, hence the upper bound for vertex c is three.

GraphChi implements the asynchronous model of computation [19]. In this model, the update function can use *the most recent values* of the edges and the vertices. The asynchronous model of computation has been shown to be more efficient than synchronous computation for many purposes [7, 22, 19]. We observe the benefit of the asynchronous computation even in this small example. Vertex q, for instance, being processed after p, sees that the latter has currently a core estimate of 3, not 5 (that it had in the previous iteration). Therefore, p contributes to incrementing the third element of the array of q, not the fifth as it would have been under a synchronized computation discipline.

Algorithm 1 Update function running at a vertex

```
1: function UPDATE(Vertex vertex)
2:     if iteration = 0 then
3:         vertex.value ← vertex.numOutEdges
4:         broadcastValueToNeighbors(vertex)
5:         scheduler.addTask(vertex.id)    ▷ schedules itself
6:     else
7:         localEstimate ← computeUpperBound(vertex)
8:         if localEstimate < vertex.value then
9:             vertex.value ← localEstimate
10:            broadcastValueToNeighbors(vertex)
11:        end if
12:        for all inEdge in vertex.inEdgeList do
13:            if vertex.value ≤ inEdge.value then
14:                scheduler.addTask(inEdge.vertexid)
15:            end if
16:        end for
17:    end if
18: end function
```

Algorithm 2 computeUpperBound function for a vertex

```
1: function COMPUTEUPPERBOUND(Vertex vertex)
2:     for all i ← 1 to vertex.value do
3:         c[i] ← 0
4:     end for
5:     for all inEdge in vertex.inEdgeList do
6:         j ← min{inEdge.value, vertex.value}
7:         c[j]++
8:     end for
9:     cumul ← 0
10:    for all i ← vertex.value down to 2 do
11:        cumul ← cumul + c[i]
12:        if cumul ≥ i then
13:            return i
14:        end if
15:    end for
16: end function
```

4.1 Results and Analysis

Our results are summarized in Table 1. Columns k_{max} and k_{avg} denote the maximum and average coreness for each graph. The algorithm runs for i iterations.

We observe that the algorithm outputs the k-core decomposition in a few tens of iterations for most of the datasets we consider which is comparable with the convergence results reported in [25].

We encounter a relatively lengthy execution for *web-berkStan* and *uk-2005*. This can be attributed to the problematic computation of certain cores within the graph due to high diameter, as suggested in [25]. Figure 2 shows the running time for each dataset (see the most dark-blue bars).

The experiments show that our implementation on GrapChi terminates in the order of minutes for most graph datasets. For large-scale graphs, namely *uk-2005* and *Twitter-2010*, it took several hours on our consumer-grade machine.

In contrast, [25] needed a cluster of 16 machines employed for just the medium and small datasets. They do not provide results for big graphs.

To further investigate the execution of the algorithm as it unfolds with time, we look at the percentage of updated vertices and maximum difference from the true core value over the sequence of iterations (see Figure 3). We recall that, starting in iteration 1, a vertex's value is updated whenever its current core estimate is recomputed and improved. The plot on the left shows that, for all the graphs we consider, the percentage of updated vertices drops below 1% before iteration 20 is reached. The plot on the right shows that the maximum difference from the true core value (max error) drops quickly and becomes 1 for all the graphs we consider in only a fraction of the total number of iterations.

In summary, our analysis shows that we can compute the k-core decomposition for large social and web networks in reasonable time using a consumer-level computer. Moreover, in applications where exact coreness values are not required, the algorithm converges quickly producing excellent approximations, and thus can be halted after the first few tens of iterations.

Parallel sliding windows and number of shards. GraphChi implements the idea of Parallel Sliding Windows (PSW) [19]. It can process a graph quite efficiently from disk using only a negligible number of random accesses. In PSW,

the vertices of a graph are split into P disjoint intervals. For each interval, a *shard* is associated with it, which stores all the edges that have destination in the interval. Intervals are created so as to balance the number of edges in each shard. Also, the number of intervals, P, is such that that any one shard can be loaded completely in main memory. To process an interval, PSW reads one shard, and slides a window over each of the other shards. In total, PSW needs only P sequential disk reads to process each interval. Each of these sequential reads is, of course, preceded by a random access to determine the start of the read. So, there are P random accesses as well to process an interval. This amount of random reads is, however, negligible when P is in the order of a few tens.

For our datasets, we used $P = 1$ (number of shards and intervals) for the small datasets, $P = 2$ for the medium datasets, and $P = 20$ for the big datasets. As long as any one shard can fit in memory, changing their number, while being under 100, does not show any appreciable difference in the running time.

5. WEBGRAPH IMPLEMENTATION

Webgraph [9] is a framework for graph compression. The compression ratios are truly impressive and all our graphs with the exception of TW fit in less than 4GB of memory. TW runs fine with 6GB, which is still possible for a middle tier consumer-grade PC. Webgraph provides a fast API for random access of graphs that fit in memory.

5.1 Batagelj and Zaversnik Algorithm

We implemented the Batagelj and Zaversnik (BZ) algorithm [6] using the Webgraph API for random access. We denote our implementation by WG_BZ.

At a high level, the BZ algorithm computes the core decomposition by recursively deleting the vertex with the lowest degree. The deletions are not physically done on the graph; a bin array is used to capture (logical) deletions.

BZ is somewhat inaccurately identified with the pseudocode in Algorithm 3 (see for example [10, 11]).

Algorithm 3 k-core computation using a set array D

```
1: function K-CORES(Graph G)
2:     L ← 0, d ← 0, D ← [∅, ..., ∅]
3:     for all i ← 0 to n do
4:         d[i] ← d_G(i)
5:         D[d[i]].insert(i)
6:     end for
7:     for all k ← 0 to d_max(G) do
8:         while not D[k].empty() do
9:             i ← D[k].remove()
10:            L[i] ← k
11:            for all j ∈ N_G(i) do
12:                if d[j] > k then
13:                    D[d[j]].remove(j)
14:                    D[d[j] − 1].insert(j)
15:                    d[j]−−
16:                end if
17:            end for
18:        end while
19:    end for
20:    return L
21: end function
```

There are three arrays in Algorithm 3, L which will eventually hold the core value of each vertex, d which holds the degree of each vertex, and D which holds, for each possible degree value, the set of vertices with that degree. Lines 2–6 initialize these arrays. Lines 7-20 implement the main idea. The smallest degree vertex, i, is located in the first non-empty set $\mathbf{D}[k]$. The core value of i is k and this is recorded in L. Next, the algorithm logically deletes i from the graph and deals with the neighbors of i, whose degree is decremented by one. A neighbor j is moved from $\mathbf{D}[\mathbf{d}[j]]$ to $\mathbf{D}[\mathbf{d}[j]-1]$. Finally, the algorithm returns L, which contains the core value for each vertex.

The practical challenge with Algorithm 3 is how to implement sets $\mathbf{D}[k]$. One way is to use a hash-table (hash-set) for each $\mathbf{D}[k]$.

There are several options for implementing a hash-table. The most common hash-maps are those which incur collisions (e.g. most chaining and open addressing methods). Such hash-maps have *expected constant time* (ECT) for lookup and deletions. In practice, ECT for lookup and deletions was not good enough when processing our big graphs, uk2005 and twitter2010; the computation could not be performed in our machine in a reasonable time. We stopped the program after days of computation. This was even though both graphs fitted in memory after storing them in Webgraph.

Specifically, we experimented with the hash-map implementations in (1) Java standard library[1], (2) Google's Guava collections library[2], and (3) Trove high-performance collections for Java[3]. None of them was able to handle our two big graphs.

Cuckoo-hashing is another hashing method. It has worst-case constant time for lookups and deletions. However, it has ECT for insertions. We experimented with Keith Schwarz's implementation of Cuckoo-hashing[4]. The original algorithm for this implementation is from [26] and the family of universal hash functions it needs is based on the functions of [31]. This is a modern hashing method and implementation, however, insertions turned out to be a serious bottleneck; we were unable to populate $\mathbf{D}[k]$'s in reasonable time.

In contrast to the above hashing methods, our WG_BZ implementation of the real BZ algorithm took only 130 seconds and 333 seconds for uk2005 and twitter2010, respectively.

The real power of the BZ algorithm comes from a simple, but quite effective idea of flattening array $\mathbf{D}$. Array $\mathbf{d}$ has the same form as before. There are two additional arrays now, $\mathbf{b}$ and $\mathbf{p}$. Array $\mathbf{b}$ has dimension $d_{\max}(G)$ and stores the index boundaries of the vertex blocks having the same degree in $\mathbf{D}$. For an example, see Fig. 4 and Table 2. In Fig. 4, we show the graph of Fig. 1, where the vertex labels have been replaced by numbers 1 to 16. (Vertices in BZ are assumed to be labeled by numbers 1 to n.) We see in Table 2 arrays $\mathbf{d}$, $\mathbf{b}$, $\mathbf{D}$ and $\mathbf{p}$. For instance, vertex 1 has a degree of 3, so $\mathbf{d}[1] = 3$. We have colored $\mathbf{D}$ in shades of blue. The first block in $\mathbf{D}$ contains the vetices with degree 2; the second block contains vertices with degree 3, and so on.

[1] `http://docs.oracle.com/javase/8/docs/api`
[2] `https://github.com/google/guava`
[3] `http://trove.starlight-systems.com`
[4] `http://www.keithschwarz.com/interesting/code/cuckoo-hashmap`

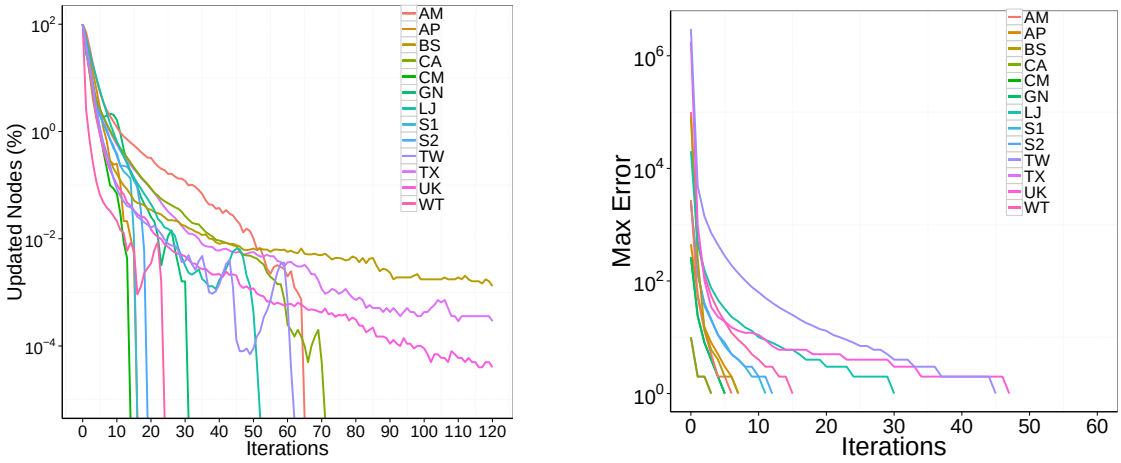

Figure 3: Percentage of updated vertices (left) and max difference from the true coreness (right).

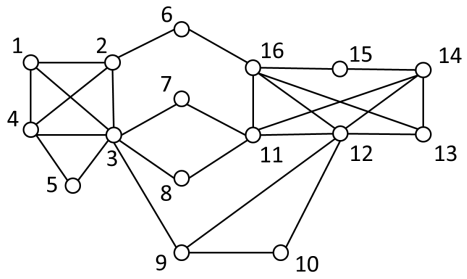

Figure 4: The graph of Fig. 1 where letters have been replaced by numbers.

index	**d**	**b**	**D**	**p**
1	3	0	5	7
2	4	1	6	10
3	7	7	7	16
4	4	10	8	11
5	2	13	10	1
6	2	15	15	2
7	2	16	1	3
8	2		9	4
9	3		13	8
10	2		2	5
11	5		4	13
12	6		14	15
13	3		11	9
14	4		16	12
15	2		12	6
16	5		3	14

Table 2: Arrays **d**, **b**, **D**, and **p** in the BZ algorithm for the graph in Fig. 4 .

Now see array **b**. We have, for instance, $\mathbf{b}[2] = 1$ and $\mathbf{b}[3] = 7$. This is because the block of vertices with degree 2 starts at index 1 in **D**, and the block of vertices with degree 3 starts at index 7 in **D**.

Finally, array **p** stores for each vertex i its position in **D**. For instance, vertex 1 is in position 7 in **D**, thus, $\mathbf{p}[1] = 7$.

Now, the BZ algorithm is given in Algorithm 4.

In line 2, arrays **d**, **b**, **D**, and **p** are initialized. The main algorithm is in lines 3–17.

The top for-loop runs for each vertex, 1 to n, scanning array **D**. The coreness of current vertex v is the current degree of v, i.e. $\mathbf{d}[v]$. Now v is logically deleted. For this, we process each neighbor u of v with a higher degree.

Vertex u needs to have its degree decremented (see line 13). However before that, u needs to be moved to the block on the left in **D** since its degree will be one less. This is achieved in constant time (see lines 7-12). Specifically, u is swapped with the first vertex, w, in the same block in **D**. Also, the positions of u and w are swapped in **p**. Then, the block index in **b** is updated incrementing it by one (line 13), thus losing the first element of the block, u, which becomes the last element of the previous block.

The complexity of the main algorithm is $O(m)$.

The initialization of **d**, **b**, **D**, and **p** in [6] is done using a somewhat complicated procedure in order to keep the initialization in $O(n)$. A much simpler way is the procedure

Algorithm 4 k-core computation using a flat array D

```
1: function K-CORES(Graph G)
2:     initialize(d, b, D, p, G)
3:     for all i ← 1 to n do
4:         v ← D[i]
5:         for all u ∈ N_G(v) do
6:             if d[u] > d[v] then
7:                 du ← d[u], pu ← p[u]
8:                 pw ← b[du], w ← D[pw]
9:                 if u ≠ w then
10:                    D[pu] ← w, D[pw] ← u
11:                    p[u] ← pw, p[w] ← pu
12:                end if
13:                b[du]++, d[u]−−
14:            end if
15:        end for
16:    end for
17:    return d
18: end function
```

given in the next paragraph with a complexity of $O(n \log n)$. This is not a problem in practice; n is much smaller than m, and $O(n \log n)$ is easily absorbed by the $O(m)$ complexity of the main algorithm.

Specifically, the initialization can be done by first populating array **d** by the degrees of each vertex, and array **D** by numbers 1 to n. Then, **d** is sorted with **D** tagging along, i.e. whenever two elements $\mathbf{d}[i]$, $\mathbf{d}[j]$ are swapped during the sort, $\mathbf{D}[i]$, $\mathbf{D}[j]$ are swapped as well. Finally, **p** is populated by scanning **D** and setting $\mathbf{p}[\mathbf{D}[i]]$ equal to i, for each $i \in [1, n]$.

5.2 A Vertex-Centric Algorithm

For the case when the graph does not fit in main memory, but a vertex array does, we provide a second, "vertex-centric", implementation based on the Montresor et. al. protocol. We denote this implementation by WG_M.

We denote the vertex array by C. It is initialized by the degrees of vertices and will contain their core values at the end of the computation. The pseudo-code is given in algorithms 5, 6, and 7.

The implementation in Algorithm 5 iterates over the vertices and calls the update function for those vertices that are scheduled. The vertex update function is given in Algorithm 6 and the upper bound function is given in Algorithm 7.

Algorithms 6 and 7 adapt Algorithms 1 and 2, respectively. Instead of broadcasting the value of a vertex v, we only store the value in $C[v]$ (see line 9, Algorithm 6). Also, instead of accessing *inEdge* values, we access array C (line 12, Algorithm 6).

Algorithm 7 also accesses array C. For instance now, $j \leftarrow \min\{inEdge.value,\ vertex.value\}$ (line 6, Algorithm 2) becomes $j \leftarrow \min\{C[v], C[u]\}$ (line 6, Algorithm 7).

In both Algorithm 6 and 7, instead of *vertex.value* we use $C[v]$.

Selective scheduling of vertices is achieved by keeping a Boolean array *scheduled* with a flag for each vertex. If the flag is set (line 13, Algorithm 6), the vertex is scheduled for the next iteration (lines 10–11, Algorithm 5).

We make a clone, *scheduledNow*, of *scheduled*; then we reinitialize *scheduled* to the all-false vector (lines 7–8, Algorithm 5). The clone is needed to be used in the for loop (lines 10, Algorithm 5). We cannot use *scheduled* to decide whether to invoke the update function on a vertex because *scheduled* is modified continuously during the calls to the update function.

Variable *change* is global and is used to register whether there is some vertex with its value changed or not. The computation terminates when $change = false$.

Webgraph Memory-Mapped Access. WG_M only needs sequential access to the graph file. However when employing selective scheduling we need to efficiently skip many vertices and continue forward in the file. Webgraph provides memory-mapped (MM) access to the file, which we use in our implementation. MM access is ideal in our case of access with skips in the forward direction.[5]

[5]Webgraph also uses back-references that can make the on-the-fly decompression do a few reads in the backward direction as well. We observed though that the benefit of efficiently skipping large portions of the file far outweighs the penalty for going backwards a few hops occasionally.

Algorithm 5 VC main function

```
1: function K-CoreCompute(Graph G)
2:     scheduled ← True                          ▷ all-true vector
3:     C ← 0                                     ▷ array of core values
4:     change ← true
5:     iter ← 0
6:     while true do
7:         scheduledNow ← scheduled.clone()
8:         scheduled ← False                     ▷ all-false vector
9:         for all v ← 1 to n do
10:            if scheduledNow[v] = true then
11:                update(G, v, scheduled, C, iter)
12:            end if
13:        end for
14:        iter++
15:        if change = false then
16:            break
17:        else
18:            change ← false
19:        end if
20:    end while
21:    return cores
22: end function
```

Algorithm 6 VC vertex update function

```
1: function update(G, v, scheduled, C, iter)
2:     if iter = 0 then
3:         C[v] ← d_G(v)
4:         scheduled[v] ← true
5:         change ← true
6:     else
7:         localEstimate ← computeUpperBound(G, v)
8:         if localEstimate < C[v] then
9:             C[v] ← localEstimate
10:            change ← true
11:            for all u ∈ N_G(v) do
12:                if C[v] ≤ C[u] then
13:                    scheduled[u] ← true
14:                end if
15:            end for
16:        end if
17:    end if
18: end function
```

Algorithm 7 VC upper bound function

```
1: function computeUpperBound(G, v, C)
2:     for all i ← 1 to C[v] do
3:         c[i] ← 0
4:     end for
5:     for all u ∈ N_G(v) do
6:         j ← min{C[v], C[u]}
7:         c[j]++
8:     end for
9:     cumul ← 0
10:    for all i ← C[v] down to 1 do
11:        cumul ← cumul + c[i]
12:        if cumul ≥ i then
13:            return i
14:        end if
15:    end for
16: end function
```

5.3 Results and Analysis

The running times of WG_BZ were impressive, with UK and TW completing in only 130 sec and 333 sec, respectively (see Fig. 2). The running times of WG_M were also quite good, being better than those of GraphChi and EMcore by an order of magnitude.

For the small and medium datasets, both WG_BZ and WG_M produced results in less than 3 sec for the small and less then 100 sec for the medium datasets.

WG_BZ requires that the graph vertices and edges fit in memory, whereas WG_M requires that a vertex array fits in memory.

The first condition may well be satisfied for moderate graphs given the compression ratios achieved by Webgraph.

The second condition is easier to satisfy even for big graphs, such as TW. Remarkably, WG_M completes for all graphs using only 4GB. Nevertheless, there can be graphs with more vertices than what a memory budget can accommodate. In that case, we should opt for the GraphChi implementation, which will be able to run, albeit for a long time.

6. EMCORE IMPLEMENTATION

The original EMcore algorithm consists of two main parts. In the first step, partitioning, all vertices are distributed among buckets. The buckets are initially created in memory. Each new vertex v is placed in a bucket which contains the maximum number of vertices adjacent to v (if such a bucket exists). The idea is to collect in the same bucket as many connected vertices as possible, with the goal of tightening an upper bound on the core class of vertices. Once a bucket is full, it is sent to disk. A bucket, along with the edges connected to its vertices, comprises a partition.

The EMcore algorithm works on the partitioned graph by performing several iterations. It tries to find vertices of the largest core class k_{max} and proceeds down to the vertices of core class 2. In each iteration, it loads into memory a sub-graph containing all vertices with an estimated core class in an interval $[k_i, k_j]$ $(i \leq j)$. These vertices are obtained by a scan of all disk-resident buckets. Then, the k-core decomposition of this sub-graph follows the traditional bottom-up in-memory core decomposition approach (see [6]). After the vertices belonging to the $[k_i, k_j]$ cores are extracted from this sub-graph, they are removed from the input graph and the adjacency lists of other vertices. If we remove a processed vertex from the adjacency list of a remaining vertex v_r, we deposit a token to v_r, to account for an adjacent vertex with larger core. This process of updating the input graph is performed by another scan of the disk-based buckets. Then, the next group of k-core candidates is loaded, and the process is repeated. However, if we created an in-memory sub-graph for all candidate vertices in the core interval $[k_i, k_j]$, and none of these vertices actually belongs to any of these cores, then the iteration is wasted. This is a substantial drawback of the EM Core algorithm; it starts from the largest possible core value and can perform multiple fruitless iterations without finding the vertices of the sought class. Thus, the performance was poor, and we improved it with a simple optimization, described below.

We observe that if the total number of vertices in a sub-graph is less than k_j, there is no vertex in this sub-graph that is connected to at least k_j other vertices of the same core. Towards this goal, we maintain the counts of candidates for each k-core class. We collect these counts during the partitioning phase. If the count of candidates for the currently highest core class k_j is less than k_j, we do not consider this value of k, but transfer its count to the lower level, class $k_j - 1$.

6.1 Results and Analysis

In general, the time for partitioning in our implementation is negligible compared to the running time of the entire algorithm. In most graphs used in the experiments, more than 20% of vertices have their bound tightened (see Table 3). For datasets with very large max degree (d_{max}), there is no improvement in u_{max} because the vertices with degree d_{max} did not fit in a bucket with their neighbors.

In the core computation phase, by applying our optimization, we avoid multiple unnecessary scans of the input, and the performance of the algorithm improves considerably. For example, for Twitter-2010, if we follow the original algorithm, we would test unnecessarily all core classes from 2,997,487 (highest tightened upper bound) to 8,925, when in fact only core class 8,925 had sufficiently many candidates to deserve testing (see Table 3).

The main problem of EMcore, however, is that it relies on the assumption that a sub-graph for each core class fits in main memory. For big datasets, such as Twitter-2010, this is not always possible for a given memory budget. Let us take a closer look at the behavior of EMcore on Twitter-2010 for a memory budget of 4GB. First, EMcore processed for core classes from 8,925 down to 2,488, and despite the tightening and the afore-mentioned optimization, none of these core classes were discovered because the actual highest core class is 2,488. When the program reached the moment of creating a sub-graph for k-core class 340, the allocated memory could not accommodate the sub-graph and the program crashed. We were able to run EMcore on Twitter-2010 when allocating 6GB. However, it is not possible to predict beforehand the amount of memory needed for a new dataset.

The running times are reported in Fig. 2 (see medium dark-blue bars). For the big datasets, UK and TW, we ran EMcore using 4GB, 6GB, and 8GB of memory. We did not observe significant differences in running times. For example the running times (in sec) for UK were 4GB: 9,446; 6GB: 9,264; 8GB: 8,968 and for TW were 6GB: 25,447; 8GB: 22,593GB. The $[k_i, k_j]$ intervals become wider with more memory, which means fewer passes, however, the processing time for each interval increases. We show the times using 4GB for UK, and 6GB for TW.

7. CONCLUSIONS

We presented and evaluated several implementations for computing the k-core decomposition of massive networks using a single consumer-grade PC. The Batagelj and Zaversnik algorithm on Webgraph (WG_BZ) is the fastest, running in the order of minutes for our largest datasets UK and TW. WG_BZ needs the whole graph, both vertices and edges, to fit in memory. As such, it could not be run for TW for all memory budgets considered. The second fastest is the Montresor et. al. algorithm on Webgraph (WG_M), still running in the order of minutes for UK and a few tens of minutes for TW. WG_M needs the vertices of the graph to fit in memory. This is a much easier constraint to satisfy even for big datasets, such as TW. WG_M was able to run for all memory budgets considered. Third came EMcore. However, it

	BT%	k_{max}	u_{maxopt}	u_{max}	d_{max}
TX	45%	3	6	9	12
CA	37%	3	6	6	12
GN	25%	6	29	59	95
CM	25%	25	61	249	280
AP	25%	56	116	315	504
S1	63%	54	257	2,469	2,548
S2	36%	56	258	2,315	2,553
AM	2%	10	178	2,742	2,752
LJ	21%	373	969	20,314	20,334
BS	47%	201	564	83,430	84,230
WT	49%	131	1,032	100,029	100,029
UK	34%	589	3,798	1,776,858	1,776,858
TW	28%	2,488	8,925	2,997,487	2,997,487

Table 3: EMcore bound tightening. Columns are as follows. BT%: percentage of nodes with tightened bound, k_{max}: max core, u_{maxopt}: max upper bound after our optimization, u_{max}: max original upper bound, d_{max}: max degree.

can fail when the subgraph of vertices for some core number does not fit in main memory. As such, it could not be run for TW for all memory budgets considered. The Montresor et. al. algorithm on GraphChi (GC_M) does not need the vertices or edges to fit in main memory. It is the slowest implementation compared to the other ones. Nevertheless, it can yield satisfactory approximate results using only a fraction of the iterations.

Source Code. Our implementations can be accessed at:
`https://github.com/athomo/kcore`
`https://github.com/mgbarsky/emcore`

8. REFERENCES

[1] 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*, 7(1):207, 2006.

[2] 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.

[3] 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*, pages 41–50, 2005.

[4] L. Antiqueira, O. N. Oliveira, L. da Fontoura Costa, and M. d. G. V. Nunes. A complex network approach to text summarization. *Information Sciences*, 179(5):584–599, 2009.

[5] A.-L. Barabási and J. Frangos. *Linked: the new science of networks science of networks*. Basic Books, 2014.

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

[7] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[8] K. Bhawalkar, J. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma. Preventing unraveling in social networks: the anchored k-core problem. In *Automata, Languages, and Programming*, pages 440–451. Springer, 2012.

[9] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th Int. Conference on World Wide Web*, pages 595–602. ACM, 2004.

[10] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1316–1325. ACM, 2014.

[11] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *Data Engineering (ICDE), 2011 IEEE 27th Int. Conference on*, pages 51–62. IEEE, 2011.

[12] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. K-core organization of complex networks. *Physical review letters*, 96(4), 2006.

[13] N. Du, B. Wu, X. Pei, B. Wang, and L. Xu. Community detection in large-scale social networks. In *Proceedings of the 9th WebKDD workshop*, pages 16–25. ACM, 2007.

[14] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[15] A. V. Goltsev, S. N. Dorogovtsev, and J. Mendes. k-core (bootstrap) percolation on complex networks: Critical phenomena and nonlocal effects. *Physical Review E*, 73(5), 2006.

[16] T. Gutierrez-Bunster, U. Stege, A. Thomo, and J. Taylor. How do biological networks differ from social networks? In *Advances in Social Networks Analysis and Mining (ASONAM), 2014 IEEE/ACM Int. Conference on*, pages 744–751. IEEE, 2014.

[17] J. Healy, J. Janssen, E. Milios, and W. Aiello. Characterization of graphs using degree cores. In *Algorithms and Models for the Web-Graph*, pages 137–148. Springer, 2008.

[18] N. Korovaiko and A. Thomo. Trust prediction from user-item ratings. *Social Network Analysis and Mining*, 3(3):749–759, 2013.

[19] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

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

[21] X. Li, M. Wu, C.-K. Kwoh, and S.-K. Ng. Computational approaches for detecting protein complexes from protein interaction networks: a survey. *BMC genomics*, 11(Suppl 1):S3, 2010.

[22] 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. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[23] L. Lü and T. Zhou. Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150–1170, 2011.

[24] 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 Int. Conference on Management of data*, pages 135–146. ACM, 2010.

[25] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *Parallel and Distributed Systems, IEEE Transactions on*, 24(2):288–300, 2013.

[26] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[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 Int. conference on World Wide Web*, pages 201–210. ACM, 2007.

[28] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24(3):515–554, 2012.

[29] J. Pattillo, N. Youssef, and S. Butenko. On clique relaxation models in network analysis. *European Journal of Operational Research*, 226(1):9–18, 2013.

[30] M. Á. Serrano, M. Boguná, and A. Vespignani. Extracting the multiscale backbone of complex weighted networks. *Proceedings of the national academy of sciences*, 106(16):6483–6488, 2009.

[31] M. Thorup. String hashing for linear probing. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 655–664. Society for Industrial and Applied Mathematics, 2009.

[32] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences*, 109(16):5962–5966, 2012.

[33] J. Wang, M. Li, J. Chen, and Y. Pan. A fast hierarchical clustering algorithm for functional modules discovery in protein interaction networks. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 8(3):607–620, 2011.

[34] L. Weng, F. Menczer, and Y.-Y. Ahn. Virality prediction and community structure in social networks. *Scientific reports*, 3, 2013.

[35] T. Wolf, A. Schroter, D. Damian, L. D. Panjer, and T. H. Nguyen. Mining task-based social networks to explore collaboration in software teams. *Software, IEEE*, 26(1):58–66, 2009.

[36] W.-S. Yang and J.-B. Dia. Discovering cohesive subgroups from social networks for targeted advertising. *Expert Systems with Applications*, 34(3):2029–2038, 2008.