Efficient Core Decomposition in Massive Networks

James Cheng[1], Yiping Ke[2], Shumo Chu[3], M. Tamer Özsu[4]

[1,3]*School of Computer Engineering*
Nanyang Technological University
Nanyang Avenue, Singapore
[1]j.cheng@acm.org
[3]shumo@pmail.ntu.edu.sg

[2]*Shenzhen Institutes of Advanced Technology*
Chinese Academy of Science, China
[2]yp.ke@siat.ac.cn

[4]*David R. Cheriton School of Computer Science*
University of Waterloo
Waterloo, Ontario, Canada
[4]tamer.ozsu@uwaterloo.ca

***Abstract*—The k-*core* of a graph is the largest subgraph in which every vertex is connected to at least k other vertices within the subgraph. Core decomposition finds the k-core of the graph for every possible k. Past studies have shown important applications of core decomposition such as in the study of the properties of large networks (e.g., sustainability, connectivity, centrality, etc.), for solving NP-hard problems efficiently in real networks (e.g., maximum clique finding, densest subgraph approximation, etc.), and for large-scale network fingerprinting and visualization. The k-core is a well accepted concept partly because there exists a simple and efficient algorithm for core decomposition, by recursively removing the lowest degree vertices and their incident edges. However, this algorithm requires random access to the graph and hence assumes the entire graph can be kept in main memory. Nevertheless, real-world networks such as online social networks have become exceedingly large in recent years and still keep growing at a steady rate. In this paper, we propose the first external-memory algorithm for core decomposition in massive graphs. When the memory is large enough to hold the graph, our algorithm achieves comparable performance as the in-memory algorithm. When the graph is too large to be kept in the memory, our algorithm requires only $O(k_{max})$ scans of the graph, where k_{max} is the largest core number of the graph. We demonstrate the efficiency of our algorithm on real networks with up to 52.9 million vertices and 1.65 billion edges.**

I. Introduction

Given a graph G, the k-*core* of G is the largest subgraph of G in which every vertex has degree of at least k within the subgraph [1]. The problem of *core decomposition* in G is to find the k-core of G for all k.

Core decomposition has been shown to be an important concept in the study of graph properties and has many significant applications in network analysis. It was first introduced to simplify graph topology to aid in the analysis and visualization of networks [1], [2]. It was then recognized as an important tool for visualization of complex networks and interpretation of cooperative processes in them [3], [4]. It was used to analyze complex networks [5], in particular their hierarchies, self-similarity, centrality, and connectivity. It was employed to find structural fingerprints of large-scale networks and design effective visualization tools [6]. It was used to describe the architecture of randomly damaged uncorrelated networks [7]. It was applied to predict protein functions based on the k-cores of protein-protein interaction networks and amino acid sequences [8]. It was used to analyze the static structure of large-scale software systems [9]. In addition, hierarchical degree core tree [10] was proposed for summarizing the structure of massive graphs and performing model validation.

In graph theory, the concept of k-core has been extensively studied in random graphs to understand various graph properties [11], [12], [13], [14], [15]. The k-cores can be used as heuristics for maximum clique finding since a clique of size k is guaranteed to be in a $(k-1)$-core, which can be significantly smaller than the original graph. Moreover, core decomposition can be applied to give a (1/2)-approximation algorithm for the densest subgraph problem [16] and a (1/3)-approximation algorithm for the densest at-least-k-subgraph problem [17] in linear time. It can also be used as an approximation of betweenness score [10] and to discover dense clusters in noisy spatial data [18].

Compared with the computation of other similar concepts of *cohesive groups* in a network [19], such as *cliques*, n-*cliques*, n-*clans*, k-*plexes*, f-*groups*, n-*clubs*, *lambda sets*, most of which are algorithmically difficult (NP-hard or at least quadratic), there exists a simple and efficient algorithm for computing k-cores.

Given a graph G, we can compute the k-core of G by recursively deleting all the vertices (together with their incident edges) with degree less than k. Batagelj and Zaversnik [20] propose a linear algorithm for core decomposition, which uses bin-sort to order the vertices and recursively deletes the vertex with the lowest degree.

This algorithm, however, requires random accesses to the graph and thus assumes that the whole graph can be kept in main memory. Unfortunately, many real-world networks have grown exceedingly large in recent years and are continuing to grow at a steady rate. For example, the Web graph has

over 1 trillion webpages (Google), most social networks (e.g., Facebook, MSN) have hundreds of millions to billions of participants, many citation networks (e.g., DBLP, Citeseer) have millions of publications, other networks such as phone-call networks, email networks, stock-market networks, etc., are also massively large.

In this paper, we develop the first *external-memory* algorithm for core decomposition in a massive network. The result of this research can be readily applied to design an efficient solution or approximation of many important but high-complexity problems in massive real networks which cannot fit in main memory. Quite a few of these problems are described above, and all of these problems assume the existence of a linear in-memory algorithm to find the k-cores.

All existing in-memory algorithms are *bottom-up* approaches that compute a k-core before a $(k+1)$-core. However, a k-core is a supergraph of a $(k+1)$-core and the 0-core is essentially the entire graph. Therefore, the bottom-up approach cannot be adopted to design an efficient external-memory algorithm.

We devise a novel *top-down* approach that recursively computes the k-cores from larger values of k to smaller ones, and progressively reduces search space and disk I/O cost by removing the vertices in each computed k-core. Our algorithm, **EMcore**, consists of three key components: an efficient strategy for graph partitioning, an effective mechanism for estimating the upper bound of the core number of the vertices, and a recursive top-down core decomposition procedure. Our graph partitioning cuts the original graph into small blocks by only one scan of the graph. This allows core decomposition to load only the relevant blocks into main memory at each recursive step. The estimation of the upper bound on the core number of vertices ensures the correctness of our top-down approach as well as reducing the overall search space. We develop an effective mechanism for estimating the upper bound and progressively refining it at each recursive step. Based on the upper bound, our top-down strategy identifies a range of k-cores to be computed in main memory by loading only the relevant blocks.

We prove that our algorithm uses only $O(k_{max})$ scans of the graph in the worst case, where k_{max} is the largest possible value of k for any k-core in the graph. In practice, however, our experimental results show that the actual amount of I/Os required is significantly less than that required for k_{max} scans of the entire graph.

Our experimental results on various massive real networks verify that our algorithm is both CPU-efficient and I/O-efficient. When main memory is sufficient to hold the entire network, our top-down algorithm achieves comparable performance to the state-of-the-art in-memory algorithm for core decomposition [20]. When main memory is not sufficient to keep the entire network, our algorithm is able to perform core decomposition efficiently for networks with up to 52.9 million vertices and 1.65 billion edges, while the in-memory algorithm fails in these large graphs.

TABLE I
NOTATIONS

Symbol	Description
M	Size of main memory (internal memory)
B	Size of disk (external memory) block
n	Number of vertices in graph $G = (V, E)$
m	Number of edges in graph $G = (V, E)$
$\|G\|$	Size of G, defined as $\|G\| = \|E\| = m$
$nb(v)$; $nb(v, G')$	The set of neighbors of a vertex v in G / G'
$d(v)$; $d(v, G')$	The degree of v in G / G'
$C_k = (V_{C_k}, E_{C_k})$	The k-core of G
k_{max}	The maximum core number of any vertex in G
$\psi(v)$	Core number of v in G, the largest k s.t. $v \in V_{C_k}$
$\overline{\psi}(v)$	The upper bound on the core number of v in G
Ψ_k	The k-class of G, $\Psi_k = \{v : v \in V, \psi(v) = k\}$
$\overline{\psi}_{max}(X)$	The maximum $\overline{\psi}$ value among all vertices in X
$deposit(v, k_u)$	The number of v's neighbors with $\psi > k_u$

Organization. Section II formally defines the problem and gives the basic notations. Section III describes the in-memory algorithm. Section IV presents the overall framework of our solution. Section V details the EMcore algorithm. Section VI reports the experimental results. Section VII discusses the related work. Section VIII gives the conclusion.

II. NOTATIONS AND PROBLEM DEFINITION

In this paper, we focus on large networks that are modeled as graphs. For simplicity of presentation, we discuss our algorithm for undirected graphs only. The algorithm can be extended to handle directed graphs in a way similar to the adaption of the in-memory algorithm for directed graphs [20].

Table I shows the notations used throughout the paper. For external-memory algorithm analysis, we use the standard I/O model [21] with the following parameters: M is the main memory size and B is the disk block size, where $1 \ll B \leq M/2$. In most practical cases, we consider the internal memory as the main memory and the external memory as the disk, though other memory hierarchies are also possible.

Let $G = (V, E)$ be an undirected and unlabeled graph. We define $n = |V|$ and $m = |E|$. We define the *size* of G, denoted as $|G|$, as the number of edges in G, i.e., $|G| = m$. We define the set of *neighbors* of a vertex v in G as $nb(v) = \{u : (u, v) \in E\}$, and the *degree* of v in G as $d(v) = |nb(v)|$. Similarly, for a subgraph $G' = (V_{G'}, E_{G'})$ of G, we define $nb(v, G') = \{u : (u, v) \in E_{G'}\}$ and $d(v, G') = |nb(v, G')|$.

The *k-core* [1] of G is the largest subgraph $C_k = (V_{C_k}, E_{C_k})$ of G such that $\forall v \in V_{C_k}$, $d(v, C_k) \geq k$. The *core number* of a vertex $v \in V$, denoted as $\psi(v)$, is defined as the largest k such that v is in C_k, i.e., "$\psi(v) = k$" means that $v \in V_{C_k}$ and $v \notin V_{C_{k+1}}$. We further define the notion of k-class as follows.

Definition 1 (k-CLASS): Given a graph $G = (V, E)$, the *k-class*, Ψ_k, of G is defined as $\Psi_k = \{v : v \in V, \psi(v) = k\}$.

In particular, the 0-class, Ψ_0, is the set of vertices in G with degree of 0.

The problem of **core decomposition** is: *given a graph G, find the k-core of G for $k = 0, 1, \ldots, k_{max}$, where k_{max} is the maximum core number of any vertex in G.* Equivalently, the problem is to *find the k-class of G for $k = 0, 1, \ldots, k_{max}$*. In this paper, we propose an external-memory algorithm that finds the k-classes of G when main memory is not large enough to hold the whole graph G. From the k-classes, we can easily obtain any k-core as the induced subgraph of G by the vertex set $V_k = \bigcup_{k \le i \le k_{max}} \Psi_i$.

The following example illustrates the concept of core decomposition.

Example 1: Figure 1 shows a graph G that contains 14 vertices, $a, \ldots, n$. The 0-class, Ψ_0, of G is an empty set since there is no isolated vertex in G. The 1-class $\Psi_1 = \{a, b, c, j\}$, the 2-class $\Psi_2 = \{d, e, f, g, m\}$, and the 3-class $\Psi_3 = \{h, i, k, l, n\}$. In this example, we have $k_{max} = 3$. The 0-core is the induced subgraph of G by vertex set $\bigcup_{0 \le i \le 3} \Psi_i$. Since $\Psi_0 = \emptyset$ in this example, the 0-core is the same as the 1-core, which is simply the entire graph. The 2-core is the induced subgraph by $(\Psi_2 \cup \Psi_3)$ and the 3-core is the induced subgraph by Ψ_3. One can easily verify that in the k-core, every vertex is connected to at least k other vertices, where $0 \le k \le 3$. □

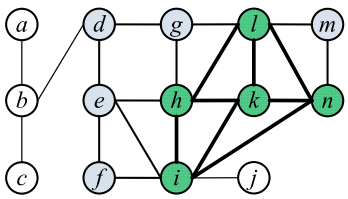


Fig. 1. A Graph and Its k-classes

III. In-Memory Core Decomposition

In this section, we describe the existing in-memory algorithm for core decomposition and explain why it is not suitable to be extended to an external-memory algorithm. We also discuss the advantage of the top-down approach over the bottom-up approach for core decomposition in large graphs.

Algorithm 1 describes the skeleton of the existing in-memory algorithm for core decomposition [20]. It is a bottom-up approach as the algorithm starts the computation of the k-class from smaller values of k and moves to larger values of k.

The algorithm first sorts the vertices in ascending order of their degree (among the vertices with the same degree, the ordering can be arbitrary). Then, it starts to compute the d-class Ψ_d, where d is the minimum vertex degree in G. It removes from G all vertices with degree of d, together with all the edges incident to them, and puts these vertices into Ψ_d. After removing these vertices and their edges, the degree of some vertices that are previously connected with the removed vertices decreases. If any vertices remaining in G now have

Algorithm 1 *BottomUp*

Input: $G = (V, E)$.
Output: The k-class, Ψ_k, of G for all k.

```
1. order the vertices in G in ascending order of their degree;
2. while (G is not empty)
3.     let d be the minimum vertex degree in G;
4.     Ψ_d ← ∅;
5.     while (there exists a vertex v with degree of at most d)
6.         Ψ_d ← Ψ_d ∪ {v};
7.         remove v and all edges incident to v from G;
8.         re-order the remaining vertices in G
               in ascending order of their degree;
9. output all Ψ_k;
```

degree of d or less, they cannot be in a k-class where $k > d$, and thus must be in Ψ_d. Therefore, they are added to Ψ_d and removed from G. This process continues until all vertices remaining in G have degree greater than d. Then, the algorithm moves on to the next iteration to compute the next k-class where $k > d$. The algorithm terminates when all vertices, and their edges, are removed from G.

The following example further explains the bottom-up computation.

Example 2: Suppose that we compute the k-classes of the graph in Figure 1 by Algorithm 1. Line 1 sorts the vertices as follows: $\{a(1), c(1), j(1), f(2), m(2), b(3), d(3), g(3), e(4), k(4), n(4), h(5), l(5), i(6)\}$, where the number in the parentheses indicates the degree of the vertex in the current G. Then, in the first iteration (Lines 3-8) that computes the 1-class, the vertices a, c and j are first removed from G together with their incident edges since their degree is 1. After that, the vertices are re-ordered as $\{b(1), f(2), m(2), d(3), g(3), e(4), k(4), n(4), h(5), l(5), i(5)\}$. Then vertex b is also removed because after removing a and c, the degree of b becomes 1. At this point, the ordered vertex list becomes $\{f(2), m(2), d(2), g(3), e(4), k(4), n(4), h(5), l(5), i(5)\}$. The first iteration completes and obtains $\Psi_1 = \{a, c, j, b\}$. In the second iteration that computes the 2-class, the vertices removed are (in the order): f, m, d, g, and e. In the third iteration that computes the 3-class, the vertices removed are (in the order): n, k, h, l, and i. □

When main memory is sufficient to hold the entire input graph, the most costly step of the in-memory algorithm is sorting the vertices according to their degree at each iteration (Line 8). Batagelj and Zaversnik [20] use bin-sort to order the vertices to achieve $O(m + n)$ time complexity.

When the graph is too large to be placed in main memory, the bottom-up approach fails because it requires memory space of $\Omega(m + n)$. The bottom-up approach is not suitable for designing an efficient external-memory algorithm since it starts core decomposition over the entire graph to compute the k-class with the smallest k. Although sorting can be done in $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ I/Os (assuming that we store separately the

vertices and their degree on consecutive disk blocks), it is non-trivial to address the problem of random accesses to vertices and edges on disk during the computation.

In this paper, we propose a top-down approach, which first computes the k-class for larger k and then moves down to smaller values of k. The top-down approach is more suitable for designing an external-memory algorithm. Another advantage of the top-down approach over the bottom-up approach is that most applications and algorithm designs using k-cores only need the k-cores with larger values of k, and sometimes, only the k_{max}-core (e.g., for defining the nucleus of a communication network [4], for predicting protein functions [8], for approximating the densest subgraph [16], etc.). The top-down approach can obtain the k-cores with larger k without computing those with smaller k. This can result in huge savings of computational resources for handling massive networks, because the smaller the value of k, the larger is the graph from which the k-core is computed.

IV. Overall Framework

In this section, we first give the overall framework of our external-memory algorithm, called **EMcore**, for core decomposition, and then discuss the details of each part in the next section.

1) Graph partitioning:

   The first step of EMcore is to partition the vertex set V of G, so that core decomposition by EMcore in later steps can be processed by loading only the corresponding subgraphs of the relevant subsets of vertices (instead of the whole graph) in main memory. We describe an efficient partitioning strategy that uses only one scan of G.

2) Estimation of the upper bound on the core number:

   We develop heuristics to estimate the upper bound on the core number of the vertices in each subset in the partition. The upper bound is refined progressively at each step of core decomposition.

3) Recursive top-down core decomposition based on the upper-bound core number:

   The main step of EMcore is a recursive procedure of core decomposition. Based on the upper bound on the core number of the vertices, EMcore recursively computes the k-classes on a set of vertices whose core number falls within a range such that the corresponding relevant subgraph can be kept in main memory. Our algorithm is top-down, meaning that we compute the k-classes with ranges of larger core numbers before those of smaller core numbers.

   At the end of each recursive step, we remove from G the k-classes already computed, as well as their incident edges, to reduce the search space and disk I/O cost for the next recursion of core decomposition.

Algorithm 2 *PartitionGraph*

Input: $G = (V, E)$, memory size M, disk block size B.
Output: A partition of disjoint vertex sets, $\mathscr{U} = \{U_1, \ldots, U_l\}$.

```
1.  create an empty partition 𝒰;
2.  𝒰_mem = 𝒰;      // the part of 𝒰 that are currently in memory
3.  for each vertex, v ∈ V, do
4.      if(v is not connected to a vertex in any vertex set in 𝒰_mem)
5.          create a new set U = {v} and add U to 𝒰_mem;
6.      else
7.          find the vertex set U ∈ 𝒰_mem to which
                v has the most connections;
8.          add v to U;
9.          if(∑_{u∈U} d(u) = B)  // memory used for U reaches B
10.             WriteBlock(U);
11.     if(∑_{U∈𝒰_mem} ∑_{u∈U} d(u) = M)
12.         let U_max be the vertex set in 𝒰_mem s.t.
                ∀U ∈ 𝒰_mem, ∑_{u∈U_max} d(u) ≥ ∑_{u∈U} d(u);
13.         WriteBlock(U_max);
```

Procedure 3 *WriteBlock*(U)

```
1. let H be the subgraph of G that consists of all edges
       connected to the vertices in U;
2. estimate ψ̄(v) for each vertex v in H;
3. write H to the disk and remove U from 𝒰_mem;
```

V. External-Memory Core Decomposition

In this section, we discuss in detail the three parts in the overall framework, followed by a detailed analysis of the algorithm.

A. Graph Partitioning

The purpose of graph partitioning is to decompose the original large graph into a set of smaller subgraphs, so that core decomposition can be processed by loading only those relevant subgraphs into main memory. There are many existing graph partitioning algorithms [22], [23], but they are in-memory algorithms. For core decomposition in massive networks that cannot fit into main memory, we need an efficient algorithm that partitions a large graph with limited memory consumption. To this end, we devise a simple graph partitioning algorithm that requires only one scan of the original graph and has linear CPU time complexity.

The outline of the partitioning algorithm is given in Algorithm 2. The algorithm reads the input graph G from external memory once and partitions V into a set of disjoint vertex sets, $\mathscr{U} = \{U_1, \ldots, U_l\}$, where $\bigcup_{1 \le i \le l} U_i = V$ and $U_i \cap U_j = \emptyset$ for all $i \neq j$.

For each vertex v, the algorithm processes v as follows. There are two cases. The first case (Lines 4-5 of Algorithm 2) is that v is not connected to any vertex in any vertex set in $\mathscr{U}_{mem}$ (the part of $\mathscr{U}$ that are currently in main memory). In this case, we simply create a new vertex set U with a single vertex v and add U to $\mathscr{U}_{mem}$.

The second case (Lines 6-10 of Algorithm 2) is that v is connected to some vertex(es) in some existing vertex set(s) in

$\mathscr{U}_{mem}$. The algorithm finds the vertex set $U \in \mathscr{U}_{mem}$ with which v has the most connections; that is, $\forall U' \in \mathscr{U}_{mem}$, $|nb(v) \cap U| \geq |nb(v) \cap U'|$. This vertex set U can be found in $O(d(v))$ expected time using a hashtable. Note that we do not keep all vertices in G in the hashtable, but only those in a vertex set in $\mathscr{U}_{mem}$. After finding U, we add v to U. If the memory used for U is as large as the block size B, we write U to disk by Procedure 3.

After processing each vertex v, we also check (Lines 11-13 of Algorithm 2) if the memory used by the current partition reaches the available memory space assigned. When this is the case, we reclaim the memory by writing the vertex set currently with the largest memory usage to disk.

The writing of a vertex set U to the disk as shown in Procedure 3 proceeds as follows. When we read the vertices in U from the disk (during the scanning of G), we read their edges as well. Let H be the corresponding subgraph; that is, $H = (V_H, E_H)$, where $V_H = U \cup \{v : u \in U, (u, v) \in E\}$ and $E_H = \{(u, v) : u \in U, (u, v) \in E\}$. We write H to disk and, at the same time, release the memory occupied by H (also U). We delay the discussion of the estimation of the upper-bound core number, $\overline{\psi}(v)$, of each v in H to Section V-B.

Before we move on to present our main algorithm for core decomposition, we ask whether we can devise a divide-and-conquer solution that computes core decomposition on each subgraph obtained by the partition, and then merges the results to find the k-cores in the original graph. We note that the divide-and-conquer method does not work because at each conquer phase, this approach still needs to perform search in the merged problem space to determine the core number of each vertex, and hence it demands as much memory as does an in-memory algorithm.

B. Upper Bound on Core Number

Our top-down core decomposition algorithm requires the selection of a relevant set of vertices based on the upper bound of their core number. In this subsection, we develop heuristics to estimate this upper bound.

We use $\overline{\psi}(v)$ to denote the upper bound on $\psi(v)$ of a vertex v. The following lemma gives a coarse estimation on $\overline{\psi}(v)$.

LEMMA *1:* $\overline{\psi}(v) = d(v)$.

Proof: It is trivial to show that $\overline{\psi}(v) = d(v)$ because $\psi(v) \leq d(v)$. ■

Lemma 1 serves as an initialization of $\overline{\psi}(v)$ for each v. After the initialization, we can further refine $\overline{\psi}(v)$. The basic idea is to refine $\overline{\psi}(v)$ based on the $\overline{\psi}$ values of v's neighbors. Intuitively, v's neighbors that have $\overline{\psi}$ values lower than $\overline{\psi}(v)$ definitely have the true core number lower than $\overline{\psi}(v)$. These neighbors cannot contribute to the degree of v in the $\overline{\psi}(v)$-core and thus if v has many of such neighbors, $\overline{\psi}(v)$ can be further tightened.

We first define a notation $\overline{\psi}_{max}(X)$ for a non-empty vertex set X to denote the maximum $\overline{\psi}$ value among all the vertices in X. That is, $\overline{\psi}_{max}(X) = max\{\overline{\psi}(u) : u \in X\}$, where $X \neq \emptyset$.

The following lemma describes how to refine $\overline{\psi}(v)$.

LEMMA *2:* Given a vertex $v \in V$, where the current $\overline{\psi}(v) > 0$, let Z contain all neighbors of v with $\overline{\psi}$ values lower than $\overline{\psi}(v)$, that is, $Z = \{u : u \in nb(v),\ \overline{\psi}(u) < \overline{\psi}(v)\}$.

If $(d(v) - |Z|) < \overline{\psi}(v)$, then $\overline{\psi}(v)$ can be refined as $max\{d(v) - |Z|,\ \overline{\psi}_{max}(Z)\}$.

Proof: Based on the definition of $\overline{\psi}_{max}(Z)$, we know that all vertices in Z can only exist in the k-cores for $k \leq \overline{\psi}_{max}(Z)$, which means that the edges connecting v and its neighbors in Z cannot exist in the $(\overline{\psi}_{max}(Z) + 1)$-core. Therefore, if v exists in the $(\overline{\psi}_{max}(Z) + 1)$-core, the degree of v is at most $(d(v) - |Z|)$.

If $(d(v) - |Z|) > \overline{\psi}_{max}(Z)$, we have $\psi(v) \leq (d(v) - |Z|)$ because it is possible that in the $(d(v) - |Z|)$-core, v has a degree of $(d(v) - |Z|)$. Otherwise, we have $(d(v) - |Z|) \leq \overline{\psi}_{max}(Z)$, which means that v cannot exist in the $(\overline{\psi}_{max}(Z) + 1)$-core, and thus we have $\psi(v) \leq \overline{\psi}_{max}(Z)$. In both cases, $\psi(v) \leq max\{d(v) - |Z|,\ \overline{\psi}_{max}(Z)\}$.

Finally, for the current value of $\overline{\psi}(v)$, we have $(d(v) - |Z|) < \overline{\psi}(v)$ (given) and $\overline{\psi}_{max}(Z) < \overline{\psi}(v)$ (by the definition of Z). Therefore, $max\{d(v) - |Z|,\ \overline{\psi}_{max}(Z)\}$ is a tighter value of $\overline{\psi}(v)$, which completes the proof. ■

Lemma 2 can be applied iteratively to replace the current value of $\overline{\psi}(v)$ whenever a tighter value is possible for $\overline{\psi}(v)$. However, simply applying Lemma 2 might not be good enough since it is likely that $\overline{\psi}_{max}(Z) > (d(v) - |Z|)$ especially if Z is not small. As a result, we often choose $\overline{\psi}_{max}(Z)$ to refine $\overline{\psi}(v)$, which weakens the effectiveness of Lemma 2 in obtaining a good estimation of $\overline{\psi}(v)$.

To get a better estimation of $\overline{\psi}(v)$, we apply a finer refinement based on the following corollary of Lemma 2.

COROLLARY *1:* Given a vertex $v \in V$, where the current $\overline{\psi}(v) > 0$, let $Z = \{u : u \in nb(v),\ \overline{\psi}(u) < \overline{\psi}(v)\}$.

For any $Z' \subseteq Z$, $Z' \neq \emptyset$, if $(d(v) - |Z'|) < \overline{\psi}(v)$, then $\overline{\psi}(v)$ can be refined as $max\{d(v) - |Z'|,\ \overline{\psi}_{max}(Z')\}$.

Proof: Similar to proof of Lemma 2. ■

Based on Corollary 1, we can select the subset Z' of Z that attains the tightest estimation of $\overline{\psi}(v)$, rather than restricting to the whole set Z for the estimation.

Procedure 4 describes our algorithm to estimate $\overline{\psi}(v)$. Lines 1-2 first initialize $\overline{\psi}(v)$ to be $d(v)$ by Lemma 1. Then, a refinement is applied to obtain a tighter bound on the core number of each vertex in V (Lines 3-7), as explained in Corollary 1.

Note that Line 5 of Procedure 4 uses the condition "$d(v) - |Z| < \overline{\psi}(v)$", while Corollary 1 uses "$d(v) - |Z'| < \overline{\psi}(v)$". An incorrect value of $\overline{\psi}(v)$ may be assigned if Lines 6-7 of Procedure 4 chooses some Z' with $(d(v) - |Z'|) \geq \overline{\psi}(v)$.

Procedure 4 *Estimate-$\overline{\psi}$*

1. **for each** $v \in V$ **do**
2. initialize $\overline{\psi}(v)$ as $d(v)$;
3. **for each** $v \in V$ **do**
4. let $Z = \{u : u \in nb(v),\ \overline{\psi}(u) < \overline{\psi}(v)\}$;
5. **if**$(d(v) - |Z| < \overline{\psi}(v))$
6. let $f(X) = max\{d(v) - |X|,\ \ \overline{\psi}_{max}(X)\}$;
7. $\overline{\psi}(v) \leftarrow min\{f(Z') : Z' \subseteq Z, Z' \neq \emptyset\}$;
8. repeat Steps 3-7;

However, this Z' cannot be selected to refine $\overline{\psi}(v)$, since Line 7 selects the subset of Z with the lowest f value and Z with $(d(v) - |Z|) < \overline{\psi}(v)$ is obviously a better refinement. Therefore, Lines 5-7 of Procedure 4 refines $\overline{\psi}(v)$ as stated in Corollary 1 and the correctness of the refinement of $\overline{\psi}(v)$ is ensured.

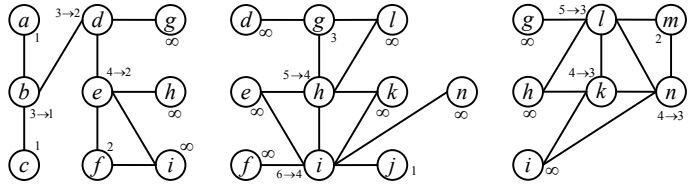
In each refinement process as described in Lines 3-7, we always refine the vertices with smaller $\overline{\psi}$ values before those with higher ones. This strategy allows the refinement to tighten the bounds more quickly since only the neighbors with low $\overline{\psi}$ values are useful in tightening the bounds (by Corollary 1). The smaller $\overline{\psi}$ values refined are thus particularly effective in tightening the $\overline{\psi}$ values for other vertices in the remaining process.

To tighten the value of $\overline{\psi}(v)$, we do not need to compute $f(Z')$ for all subsets of Z in Line 7 of Procedure 4. The definition of $f(Z')$ in Line 6 of Procedure 4 actually implies a dilemma: the larger the size of Z', the smaller is the value of $(d(v) - |Z'|)$ but the larger may be the value of $\overline{\psi}_{max}(Z')$; while a smaller Z' implies the opposite. Moreover, for subsets of the same size, clearly those vertices in Z that have a smaller $\overline{\psi}$ help more in obtaining a smaller $\overline{\psi}_{max}(Z')$ than other vertices. Therefore, we can sort the vertices in Z in ascending order of their $\overline{\psi}$ values. Then, we select Z' of size from 1 to $|Z|$, where the size-i Z' simply contains the first i vertices in the ordered Z, until we find that $f(Z')$ reaches its minimum. In this way, we process at most $|Z|$ number of subsets of Z instead of $2^{|Z|}$ subsets.

We apply Procedure 4 to estimate $\overline{\psi}(v)$ for each v in H in Line 2 of Procedure 3. Since H is only a small subgraph of G, the refinement process (Lines 3-7 of Procedure 4) converges quickly. To handle the worst-case scenario, in our algorithm, we set a limit on the number of iterations for the refinement process. We bound the cost of the refinement by the cost of a disk I/O. That is, we terminate the refinement process once the accumulated time for refinement reaches the time for a disk I/O. Therefore, the total cost of our algorithm is always bounded by the number of disk I/Os.

In Procedure 3, the subgraph H is associated with a vertex set U in partition $\mathscr{U}$. However, there may be some vertices in H that are not in U but are connected to some vertices in U. Since we may not have the degree of these vertices to initialize their $\overline{\psi}$ values, we consider their $\overline{\psi}$ values to be ∞ when we refine $\overline{\psi}(v)$ for $v \in U$. The following example illustrates how we refine $\overline{\psi}$ by Procedure 4.

Example 3: Assume that the vertex set of the graph in Figure 1 is partitioned into three sets: $U_1 = \{a, b, c, d, e, f\}$, $U_2 = \{g, h, i, j\}$, and $U_3 = \{e, k, m, n\}$. Figure 2 shows the three corresponding subgraphs[1]. The first number next to each vertex v is the initial $\overline{\psi}(v) = d(v)$, while $x \rightarrow y$ indicates that the $\overline{\psi}$ value is tightened from x to y by Procedure 4.



(a) U_1={a, b, c, d, e, f} (b) U_2={g, h, i, j} (c) U_3={k, l, m, n}

Fig. 2. Vertex Set Partition and Their Subgraphs

Let us consider vertex b in Figure 2(a). We have $Z = \{a, c\}$ since $\overline{\psi}(a) = \overline{\psi}(c) = 1 < \overline{\psi}(b) = 3$ at this moment. It is easy to see $min\{f(Z') : Z' \subseteq Z, Z' \neq \emptyset\} = 1$ in this case and thus we obtain $\overline{\psi}(b) = 1$. Similarly for vertex d, we compute $Z = \{b\}$ since now $\overline{\psi}(b) = 1 < \overline{\psi}(d) = 3$. Thus, we obtain $\overline{\psi}(d) = 2$. Note that we cannot use g to compute $\overline{\psi}(d)$ since we do not have the degree or $\overline{\psi}$ information of g at the current stage, which will become available in a later stage of core decomposition. □

C. Recursive Top-Down Core Decomposition

We now discuss our algorithm **EMcore**, as outlined in Algorithm 5. EMcore first invokes *PartitionGraph* (i.e., Algorithm 2) to partition G. Then, it invokes Procedure 6, *EmCoreStep*, to compute the k-classes with k in the range $[k_l, k_u]$, where $k_l \leq k_u$, recursively from higher value ranges to lower value ones.

In order to bound the memory usage, we determine each range $[k_l, k_u]$ as follows. We first define $\overline{\Psi}_{k_l}^{k_u} = \{v : v \in V,\ k_l \leq \overline{\psi}(v) \leq k_u\}$; that is, $\overline{\Psi}_{k_l}^{k_u}$ is the set of vertices whose $\overline{\psi}$ values fall within the range $[k_l, k_u]$. At each recursive step, we construct a subgraph that is relevant for computing the real core number of the vertices in $\overline{\Psi}_{k_l}^{k_u}$ in main memory. Let $b = \frac{M}{B}$. Then, b is the maximum number of blocks that can be held in main memory. We use b and a given k_u to determine k_l as follows.

Let K be a set of k values such that the vertices in $\overline{\Psi}_k^{k_u}$ are scattered in at most b vertex sets in $\mathscr{U}$. That is,

$$K = \Big\{k : 1 < k \leq k_u,\ \ \big|\{U : U \in \mathscr{U}, U \cap \overline{\Psi}_k^{k_u} \neq \emptyset\}\big| \leq b\Big\}.$$

In this paper, we assume K is not empty. We then define

$$k_l = \begin{cases} min\{k : k \in K\}, & \text{if } K \neq \emptyset, \\ k_u, & \text{otherwise.} \end{cases} \quad (1)$$

[1]We use an adjacency list to store $nb(v)$ of a vertex v. In each subgraph, we only need to keep the adjacency list of those vertices in U_i, where $1 \leq i \leq 3$. Thus, an edge between two vertices in the same U_i is counted twice, and the three subgraphs have 14, 15, and 15 edges, respectively.

Algorithm 5 *EMcore*

Input: $G = (V, E)$, memory size M, disk block size B.
Output: The k-class, Ψ_k, of G for all k.

1. invoke *PartitionGraph*;
2. compute k_l by setting $k_u = \infty$ and $b = \frac{M}{B}$;
3. invoke *EmCoreStep*(k_l, k_u);

Procedure 6 *EmCoreStep*(k_l, k_u)

1. $W \leftarrow \overline{\Psi}_{k_l}^{k_u}$;
2. let $\mathcal{H}$ be the set of subgraphs of the vertex sets in $\mathscr{U}$ that contain any vertex in W;
3. read $\mathcal{H}$ into main memory to construct the induced subgraph G_W by W;
4. invoke *ComputeCore*(G_W, k_l, k_u);
5. refine $\overline{\psi}(v)$ for any vertex v currently in main memory;
6. *deposit* any vertex connected to a vertex in Ψ_k ($k_l \leq k \leq k_u$);
7. remove all vertices in Ψ_k ($k_l \leq k \leq k_u$) and their edges from the subgraphs in $\mathcal{H}$, and merge any of these subgraphs if their combined size is less than B;
8. **if**($k_l > 2$)
9. compute k_l' in $\overline{\Psi}_{k_l'}^{k_u'}$ by setting $k_u' = (k_l - 1)$;
10. write the subgraphs in $\mathcal{H}$ that consist of no vertex in $\overline{\Psi}_{k_l'}^{k_u'}$ to the disk, unless all the subgraphs can now be kept in main memory;
11. invoke *EmCoreStep*(k_l', k_u');
12. **else**
13. add all the remaining vertices to Ψ_1 and output Ψ_1;

Intuitively, $k_l = min\{k : k \in K\}$ means that we load as many parts of the graph as possible into main memory (bounded by the memory size M) for core decomposition at each recursive step. However, in the rare case when $|K| = \emptyset$ (i.e., the vertices in $\overline{\Psi}_{k_u}^{k_u}$ are in more than b vertex sets in $\mathscr{U}$), we simply assign $k_l = k_u$ and load b vertex sets in $\{U : U \in \mathscr{U}, U \cap \overline{\Psi}_{k_u}^{k_u} \neq \emptyset\}$ each time into main memory to construct the subgraph for core decomposition.

To obtain k_l, we keep a heap in main memory to maintain the highest $\overline{\psi}$ in each vertex set $U \in \mathscr{U}$. The highest $\overline{\psi}$ among the vertices in each $U \in \mathscr{U}$ is inserted into the heap before U is written to disk during graph partitioning. Given b and k_u, we can easily determine the corresponding value of k_l by querying the heap.

We now discuss how EmCoreStep computes the k-classes, where $k_l \leq k \leq k_u$, as outlined in Procedure 6.

Let $W = \overline{\Psi}_{k_l}^{k_u}$. In other words, W is the set of candidate vertices whose exact core number is to be computed at this recursive step. The first step is to construct the induced subgraph G_W of G by W. We construct G_W from the set of subgraphs of the vertex sets in $\mathscr{U}$ that contain a vertex in W, since all vertices in W and their inter-connection edges are contained in these subgraphs. During the construction, we can discard those vertices that are not in W, together with their edges.

After G_W is constructed, we invoke *ComputeCore*, as shown in Procedure 7, to compute the k-classes Ψ_k from G_W, where $k_l \leq k \leq k_u$. However, an immediate question raised here is: since G_W is the induced subgraph, is the core number computed from G_W correct? We will prove the correctness of our algorithm in Theorem 1, which will also become clearer as we continue to unwrap the details of our algorithm.

Procedure 7 *ComputeCore*(G_W, k_l, k_u)

1. initialize $\psi(v)$ to be -1 for each $v \in W$;
2. $k \leftarrow k_l$;
3. **while** ($k \leq k_u$)
4. **while** ($\exists v \in W$, $(d(v, G_W) + deposit(v, k_u)) < k$)
5. remove v and all edges incident to v from G_W;
6. update $\psi(v)$ to be k for each remaining vertex v in G_W;
7. $k \leftarrow k + 1$;
8. **for each** k, where $k_l \leq k \leq k_u$, **do**
9. $\Psi_k \leftarrow \{v : v \in W, \psi(v) = k\}$;
10. output Ψ_k;

One key concept to ensure the correctness of our algorithm is the concept of *deposit* assigned to each vertex (Line 6 of Procedure 6). We define the deposit of a vertex as follows.

*Definition 2 (*DEPOSIT*):* Given a vertex $v \in V$ and an integer k_u, where $k_u > \overline{\psi}(v)$, the *deposit* of v with respect to k_u is defined as $deposit(v, k_u) = |\{u : u \in nb(v), \psi(u) > k_u\}|$.

The concept of deposit is used to ensure that the number of edge connections to a vertex v in the k-core, where $k_l \leq k \leq k_u$, can be correctly counted even if the vertices and edges that belong to all k'-core are removed, where $k' > k_u$. After we compute the k-classes, where $k \geq k_l$, we can deposit a "coin" to each neighbor of a vertex in the k-classes if the neighbor's core number has not been determined (i.e., the neighbor's $\overline{\psi}$ is less than k_l and its core number will be computed in a later recursive step). This "coin" accounts for an edge connection to this neighbor vertex in the computation of its core number.

Using the deposit, we can safely remove all the vertices and their edges after the core number of these vertices in the current recursive step is computed (Line 7), without worrying about the correctness of the core decomposition in the later recursive steps. In this way, we can save considerable disk I/O costs for reading and writing these large-core-number vertices and their edges again and again in computing the k-classes with a smaller core number.

Procedure 7 computes the correct core number of the vertices in the induced subgraph G_W as follows. The algorithm iteratively deletes the lowest degree vertex and its edges as in an in-memory algorithm. However, in selecting the lowest degree vertex, the algorithm considers the sum of the local degree in G_W and the deposit of the vertex. We remark that the initial value of the deposit of all vertices with respect to $k_u = \infty$ is set to 0. We prove the correctness of Procedure 7 in Lemma 3.

After we compute Ψ_k, where $k_l \leq k \leq k_u$, we return to Lines 5-6 of Procedure 6. We need to refine the upper bound on the core number of remaining vertices as well as updating

their deposit. We first refine $\overline{\psi}(v)$ for any vertex v that is currently in main memory but does not have its core number computed. The refinement of $\overline{\psi}(v)$ involves two steps: (1) We first set $\overline{\psi}(v)$ to be $(k_l - 1)$ if currently $\overline{\psi}(v) \geq k_l$. This is because if $\overline{\psi}(v) \geq k_l$, then v must be processed in the current recursive step. However, since v's core number is not determined, v is not in any Ψ_k, where $k_l \leq k \leq k_u$, which implies that $\psi(v) \leq (k_l - 1)$. (2) We then refine all v by Procedure 4, since now we have the exact core number of more vertices computed, which can be used to tighten the $\overline{\psi}$ value of the remaining vertices.

After the refinement, we have $\overline{\psi}(v) < k_l \leq \psi(u)$ for all v whose core number has not been determined and any u in Ψ_k, where $k \geq k_l$. Then, Line 6 updates the deposit of a vertex v with respect to $k'_u = (k_l - 1)$. The $deposit(v, k_l - 1)$ can be calculated as $(deposit(v, k_u) + |\{u : u \in nb(v), u \in \bigcup_{k_l \leq k \leq k_u} \Psi_k\}|)$, where the first part is the number of edge connections to v that were removed at previous recursive steps, while the second part is the number of edge connections to v that are to be removed at the current recursive step. Therefore, $deposit(v, k_l - 1)$ ensures the correct computation of $\psi(v)$ at later recursive steps.

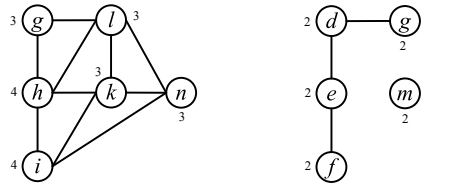
After the update of the deposit, we can safely remove all vertices in Ψ_k, where $k_l \leq k \leq k_u$, and their edges from the subgraphs in $\mathcal{H}$ (Line 7 of Procedure 6). Then, we merge any of these subgraphs if their combined size is less than B, so as to further reduce the disk I/O cost. Furthermore, we keep those subgraphs that will be used in the next recursive step in main memory and only write the rest to disk, or keep all the subgraphs in main memory if they are now small enough to be all kept in main memory (Line 10 of Procedure 6).

The recursive step goes on to process the next range of k, $[k'_l, k_l - 1]$, until $k_l = 2$ (Lines 8-11 of Procedure 6). Note that the upper bound of the range, k'_u, for the next recursive step now changes to $(k_l - 1)$. This is because, after the refinement in Line 5 in the current recursive step, we have $\overline{\psi}(v) \leq (k_l - 1)$ for any vertex v whose core number has not been determined. When $k_l = 2$, the remaining vertices must be all in the 1-class, Ψ_1 (Lines 12-13).

There is still a small problem: are there any vertices in the 0-class? The case of $k = 0$, i.e., the 0-class, means that no vertex in this class is connected to any other vertices. Thus, the 0-class contains only isolated vertices in the graph, which can be obtained trivially by one scan of the graph. To do this, we add a check in Algorithm 2: as we scan the input graph, if a vertex has a degree of 0, we output it as a 0-class vertex.

The following example explains how the top-down core decomposition is processed.

Example 4: Let us consider the three subgraphs given in Figure 2 and suppose $b = 2$. Then, we obtain $k_l = 3$ and $W = \overline{\Psi}_3^{\infty} = \{g, h, i, k, l, n\}$. We load the subgraphs in Figures 2(b) and 2(c) into main memory to construct the induced subgraph by W, G_W, which is shown in Figure 3(a), where the number next to each vertex v is the current $\overline{\psi}(v)$. It is then easy to obtain the 3-class, $\{h, i, k, l, n\}$, by Procedure 7.



(a) W={g, h, i, k, l, n} (b) W={d, e, f, g, m }

Fig. 3. Induced Subgraph G_W

After computing the 3-class, we can refine $\overline{\psi}(g)$ from 3 to 2. We also update the deposit of any vertex that is connected to a vertex in the 3-class for $k'_u = (k_l - 1) = 2$, as shown in the following table.

TABLE II
$deposit(v, 2)$ WHERE $v \in \{e, f, g, j, m\}$

v	e	f	g	j	m
$deposit(v, 2)$	2	1	2	1	2

Now we can remove all the edges connected to any vertex in $\{h, i, k, l, n\}$. After removing the edges, we only have 1 edge (d, g) and 3 isolated vertices e, f and j in Figure 2(b), and only 2 isolated vertices g and m in Figure 2(c).

Next we compute for $k'_u = 2$ and we load the subgraph in Figure 2(a) (and also Figure 2(c) which is still in main memory) to construct the induced subgraph by $W = \{d, e, f, g, m\}$, as shown in Figure 3(b). By using the deposit, it is easy to obtain the 2-class, $\{d, e, f, g, m\}$, by Procedure 7.

After computing the 2-class, all the remaining vertices, $\{a, b, c, j\}$, belong to the 1-class. □

D. Analysis of Algorithm

In this subsection, we provide the theoretical analysis regarding the correctness as well as the complexity of our algorithm EMcore.

LEMMA 3: Given $W = \Psi_{k_l}^{k_u}$ and G_W, for any $v \in W$, if $\psi(v) \geq k_l$, then ComputeCore (i.e., Procedure 7) correctly computes $\psi(v)$.

Proof: If $\psi(v) \geq k_l$, to compute $\psi(v)$ we only need the k_l-core of G, C_{k_l}. In G_W, the largest core number that a vertex can have is k_u. Thus, in computing $\psi(v)$ using G_W instead of C_{k_l}, we are missing the edge set $\{(u, v) : v \in W,\ \psi(u) > k_u,\ (u, v) \in E\}$, where $k_l \leq \psi(v) \leq k_u$.

Since our algorithm follows a top-down approach, $\forall v \in W$, if $\exists u \in nb(v)$, $\psi(u) > k_u$, then $\psi(u)$ must have already been computed in some previous recursive step and deposited to $deposit(v, k_u)$. Thus, we have $(d(v, G_W) + deposit(v, k_u)) = d(v, C_{k_l})$ for all $v \in W$. Since ComputeCore removes a vertex v only if $(d(v, G_W) + deposit(v, k_u)) < k$, or equivalently, $d(v, C_{k_l}) < k$, it correctly computes $\psi(v)$ for all $v \in W$, $\psi(v) \geq k_l$. ■

Using Lemma 3, we prove the correctness of our main algorithm EMcore.

THEOREM *1:* Given a graph G, EMcore (i.e., Algorithm 5) correctly computes the k-class of G for all k, where $0 \leq k \leq k_{max}$ and k_{max} is the maximum core number of any vertex in G.

Proof: Procedure 6 recursively computes a range of k-classes with $k \in [k_l, k_u]$ each time, from $k_u = \infty$ to $k_l = 2$. Let $[k_l, k_u]$ and $[k'_l, k'_u]$ be the ranges of k of two succussive recursive steps. Procedure 6 assigns $k'_u = k_l - 1$; thus, since Lemma 3 proves the correctness of each recursion, the recursive procedure correctly computes all the k-classes for $k \geq 2$. After Procedure 6 computes the 2-class, the remaining vertices must belong to the 1-class, since the 0-class is already outputted by a simple checking on the degree of each vertex being read as we scan G in Algorithm 2, by the fact that a vertex v is in the 0-class iff $d(v) = 0$. ■

We now analyze the complexity of EMcore. For external-memory algorithms, the complexity analysis is mainly based on the I/O model, which focuses on the importance of disk I/Os over CPU computation. We remark that our algorithm is not CPU intensive, it takes $O(k_{max}(m+n))$ CPU time, since graph partitioning takes linear time and each recursion of k-class computation takes at most $O(m+n)$ time. Note that the refinement of the $\overline{\psi}$ value follows a similar strategy as the in-memory core decomposition algorithm and thus its time complexity is also bounded by that of the in-memory algorithm, i.e., $O(m+n)$ time. The amortized CPU time should be close to $O(m+n)$ since each recursion operates on an induced subgraph relevant for the range $[k_l, k_u]$ only, which is verified by our experimental results as compared with the in-memory algorithm. Thus, disk I/O time dominates the overall complexity.

THEOREM *2:* Given a graph G, EMcore computes the k-classes of G using $O(\frac{k_{max}(m+n)}{B})$ I/Os and $O(\frac{m+n}{B})$ disk block space in the worst case.

Proof: The graph partitioning process scans the graph only once and the total size of the subgraphs written to disk is the same as the size of the original graph. Thus, Algorithm 2 uses $O(\frac{m+n}{B})$ I/Os. Each subsequent recursive step reads/writes only those subgraphs relevant for computing the k-classes that fall into the range $k \in [k_l, k_u]$. Thus, the number of disk I/Os at each recursive step is bounded by $O(\frac{m+n}{B})$. Thus, the total number of disk I/Os is $O(\frac{k_{max}(m+n)}{B})$ in the worst case, since there are at most k_{max} recursive steps.

The algorithm uses $O(\frac{m+n}{B})$ disk blocks since the total size of the subgraphs written to disk is $O(m+n)$. ■

We further remark that the actual I/O cost of the algorithm can be significantly smaller for the following reasons: (1) each recursive step reads only the relevant subgraphs from the disk, which takes $O(b) = O(\frac{M}{B})$ instead of $O(\frac{m+n}{B})$ disk I/Os; (2) we cache some subgraphs in main memory for reuse in the next recursive step; and (3) the size of each subgraph reduces after each recursive step. In our experiments, we show that the actual I/O is close to $\frac{m+n}{B}$ instead of $\frac{k_{max}(m+n)}{B}$.

The worst-case complexity is given in terms of k_{max}. In the following, we give an estimation on the value of k_{max}.

We focus our analysis on real-world networks whose degree distribution follows a *power law*. We note that similar analysis can also be applied for graphs with other degree distributions. However, extensive studies [24], [25], [26], [27], [28], [29], [30] have shown that graphs with power-law degree distributions are prevalent in real-world applications, such as the WWW, many social networks, neural networks, genome and protein networks. More importantly, most real-world networks that are too large to be kept in main memory follow a power-law degree distribution [27], [31].

Faloutsos et al. [26] give the following power law degree distribution for real-world networks:

$$d(v) = \frac{1}{n^{\mathcal{R}}}(r(v))^{\mathcal{R}}, \tag{2}$$

where $r(v)$ is the *degree rank* of a vertex v, i.e., v is the $(r(v))$-th highest-degree vertex in G, and $\mathcal{R} < 0$ is a *constant* called the *rank exponent*.

Given a graph G with the above vertex degree distribution, we now try to obtain the maximum core number k_{max} that G can have. k_{max} is essentially the maximum value of k such that the top-$(k+1)$ highest-degree vertices in G are pairwisely connected (i.e., of degree at least k). Therefore, by substituting $r(v)$ by $(k_{max}+1)$ in Equation (2) and having $d(v)$ of at least k_{max}, we have

$$d(v) = \frac{1}{n^{\mathcal{R}}}(k_{max}+1)^{\mathcal{R}} \geq k_{max}. \tag{3}$$

An approximation for the solution of the inequality stated in Equation (3) by putting $(k_{max}+1) \simeq k_{max}$ gives the following upper bound on k_{max}.

$$k_{max} \leq n^{\frac{\mathcal{R}}{\mathcal{R}-1}}. \tag{4}$$

The value of $\mathcal{R}$ measured in [26] for three snapshots of the internet graph is between -0.8 and -0.7. For a graph of 1 million vertices, we have $k_{max} \leq 296$ when $\mathcal{R} = -0.7$. Note that this is the worst case. Some existing studies on large scale networks [6], [3], [5] show that k_{max} is less than 50 approximately. Our experimental results also show that k_{max} is small and the maximum is 372 among the datasets used (see Table III). But suppose that k_{max} is close to its upper bound given by Equation (4) in the worst case, we have the following analysis on our algorithm.

Let S be the subgraph of G such that one or both end vertices of every edge in S are in the k_{max}-class; that is, $S = (V_S, E_S)$, where $V_S = \Psi_{k_{max}} \cup \{v : u \in \Psi_{k_{max}},\ (u,v) \in E\}$ and $E_S = \{(u,v) : (u,v) \in E, u \in \Psi_{k_{max}}\}$. By Equation (2), we have the following lower bound for $|S|$.

$$|S| \quad \geq \quad \sum_{r=1}^{k_{max}+1} (\frac{r}{n})^{\mathcal{R}} - \frac{k_{max}(k_{max}+1)}{2}. \tag{5}$$

The equality holds when there are only $(k_{max}+1)$ vertices in the k_{max}-core. The first component in the right-hand side of Equation (5) is the sum of degrees of the $(k_{max}+1)$ vertices. Since they are pairwise connected in this case, their inter-edges are double counted and hence deducting the number of these edges from the degree sum gives the lower bound of $|S|$.

By Equation (2), we also obtain the size of the entire graph G, which is half of the degree sum of all vertices in V. Since all edges are counted twice in this case, the degree sum needs to be halved.

$$|G| = \frac{1}{2}\sum_{r=1}^{n}(\frac{r}{n})^{\mathcal{R}}. \quad (6)$$

In our top-down approach, after computing the k_{max}-core, we can remove S from G. According to Equation (5) and Equation (6), for a network with $\mathcal{R} = -0.7$ and 1 million vertices, $|S|$ is at least 12% of the entire network, which is a huge reduction on the search space and disk I/O cost.

The value of k_{max} is in general small for large scale real-world networks, in which case EMcore has fewer number of recursive steps. On the other hand, if k_{max} is larger, we have greater reduction on the search space and I/O cost. Thus, in both cases, the analysis shows the effectiveness of our top-down approach for core decomposition.

E. Algorithm Optimization

There is one optimization that may considerably improve the performance of the algorithm. Before we write a subgraph H to the disk in Procedure 3, we can further trim H to reduce both the search space and the disk I/O cost by pre-computing a *partial* 1-class. Such 1-class vertices in H can be found by iteratively deleting vertices with degree of 1 and the edges incident to them. Note that the 1-class so computed may not be complete, since some inter-edges may exist between different subgraphs in the partition and the computation here only considers each local subgraph H.

Since vertices with a smaller core number, together with their edges, cannot be in k-cores with a greater core number, we can safely remove all these vertices and their edges from H. If there are many such vertices and edges in H, we can continue to keep H in main memory and put into H more vertices and edges from G. We write H to the disk until its size reaches B.

We remark that if after removing the partial 1-class vertices and their edges, the graph becomes small enough to be kept in main memory, then our algorithm simply performs core decomposition in main memory, as does an in-memory algorithm.

VI. Experimental Evaluation

In this section, we evaluate the performance of our algorithm **EMcore**, comparing it with the state-of-the-art in-memory core decomposition algorithm by Batagelj and Zaversnik [20], denoted as **IMcore** in our experiments. We implement our algorithm in C++ (as does IMcore). We ran all experiments on a machine with an Intel Xeon 2.67GHz CPU and 6GB RAM, running CentOS 5.4 (Linux).

Datasets. We use the following four datasets in our experiments: *road-network-California* (*RNCA*), *LiveJournal* (*LJ*), *Billion Triple Challenge* (*BTC*), and *Web*. The *RNCA* dataset is a road network of California, where vertices represent intersections and endpoints, and edges represent the roads connecting these intersections or road endpoints. The *LJ* dataset is the free on-line community called Livejournal (www.livejournal.com), where vertices are members and edges represent friendship between members. The *RNCA* and *LJ* datasets can be downloaded from the Stanford Network Analysis Platform (http://snap.stanford.edu/data/index.html). The *BTC* dataset is a semantic graph converted from the Billion Triple Challenge 2009 RDF dataset (http://vmlion25.deri.ie/), where each node represents an object such as a person, a document, and an event, and each edge represents the relationship between two nodes such as "has-author", "links-to", and "has-title". The *Web* graph is obtained from the YAHOO webspam dataset (http://barcelona.research.yahoo.net/webspam), where vertices are pages and edges are hyperlinks. We give the number of vertices and edges, as well as the storage size in the disk, of the datasets in Table III.

TABLE III
DATASETS (M $= 10^6$, G $= 10^9$)

	RNCA	*LJ*	*BTC*	*Web*
$n = \|V\|$	2.0M	4.8M	148.5M	52.9M
$m = \|E\|$	5.5M	69.0M	673.3M	1.65G
Disk size (byte)	67.4M	809.1M	8.7G	17.2G
k_{max}	6	372	54	307

Experiment Settings. We test our algorithm for two scenarios: (1) *when main memory can hold the entire input graph;* and (2) *when main memory cannot hold the entire input graph.* We use Case (1) to demonstrate that our external-memory algorithm does not process a large graph at the expense of significantly increased CPU time complexity. We use Case (2) to verify that our algorithm computes core decomposition efficiently when the existing in-memory algorithm becomes infeasible.

A. When Main Memory is Sufficient

We first assess the performance of our algorithm when main memory is sufficient to hold the entire graph, by mainly comparing the running time with the in-memory algorithm. We use two medium size networks, *RNCA* and *LJ*.

Table IV reports the partitioning time, the recursive core decomposition time (and the time to compute the k_{max}-class Ψ_k), the total running time, and the memory consumption of EMcore and IMcore. All time taken is the elapsed wall-clock time. The total running time of EMcore is comparable to that of IMcore. For EMcore, the partitioning time occupies a large portion of the total running time. Thus, the actual time taken by the recursive core decomposition procedure (Procedure 6)

is considerably shorter, as shown in the second column of Table IV.

TABLE IV
PARTITIONING TIME, CORE DECOMPOSITION TIME (AND TIME TO COMPUTE k_{max}-CLASS), TOTAL RUNNING TIME (ALL TIME IS ELAPSED WALL-CLOCK TIME), AND MEMORY CONSUMPTION FOR *RNCA* AND *LJ*

	Partitioning Time	Core Decomposition (Compute Ψ_k) Time	Total Running Time	Memory Consumption
IMcore (*RNCA*)	–	9.6 (9.6) sec	9.6 sec	136 MB
EMcore (*RNCA*)	5.4 sec	5.3 (3.4) sec	10.7 sec	80 MB
IMcore (*LJ*)	–	76 (76) sec	76 sec	765 MB
EMcore (*LJ*)	30.9 sec	52.5 (42.4) sec	83.4 sec	643 MB

From Table IV, we can also calculate the total time taken to compute the k_{max}-class, i.e., the partitioning time plus the time shown inside the parentheses in second column of Table IV, which are 8.8 sec and 73.3 sec, respectively. This result shows that the total time taken to compute the k_{max}-class by our top-down approach, though an external-memory algorithm, is shorter than the in-memory bottom-up algorithm.

Table IV also shows that EMcore consumes less memory than IMcore. This is because IMcore keeps the entire graph in main memory, while EMcore does not keep the entire graph in the running process but writes a block to disk whenever it is full.

The result thus suggests that our external-memory algorithm is as efficient as the state-of-the-art in-memory algorithm for the case when main memory is sufficient to keep the entire input graph.

Table V further shows the percentage of the amount of disk I/Os used by EMcore during graph partitioning and the top-down recursive core decomposition, respectively. The result shows that the entire recursive process of top-down core decomposition only uses almost the same amount of disk I/Os as in the process of graph partitioning. Since graph partitioning reads and writes the whole graph approximately once, this result implies that the recursive procedure actually uses only $O(\frac{m+n}{B})$ disk I/Os in practice, instead of the theoretical bound of $O(\frac{k_{max}(m+n)}{M})$ disk I/Os as given in Theorem 2.

TABLE V
PERCENTAGE OF DISK I/OS FOR *RNCA* AND *LJ*

	Graph Partitioning	Top-Down Core Decomposition
EMcore (*RNCA*)	50%	50%
EMcore (*LJ*)	48%	52%

B. When Main Memory is Insufficient

We now assess the performance of our algorithm when main memory is not sufficient to hold the entire graph. We use two larger networks, *BTC* and *Web*.

Table VI reports the partitioning time, the recursive core decomposition time (and the time to compute the k_{max}-class Ψ_k), and the total running time, in seconds. All time taken is the elapsed wall-clock time. We only report the results of EMcore because IMcore ran out of all available memory before it can finalize its computation.

TABLE VI
PARTITIONING TIME, CORE DECOMPOSITION TIME (AND TIME TO COMPUTE k_{max}-CLASS), AND TOTAL RUNNING TIME (ALL TIME IS ELAPSED WALL-CLOCK TIME) FOR *BTC* AND *Web*

	Partitioning Time	Core Decomposition (Compute Ψ_k) Time	Total Running Time
EMcore (*BTC*)	667 sec	445 (115) sec	1112 sec
EMcore (*Web*)	544 sec	1315 (537) sec	1859 sec

Table VI shows that the total running time of EMcore is very competitive because it is only about two to three times more than the partitioning time, while we know that the partitioning time is linear. EMcore uses more time in partitioning than in the recursive core decomposition for the *BTC* dataset, while it is the opposite for the *Web* dataset. This is because the *Web* dataset has a much larger k_{max} than the *BTC* dataset (see Table III). Therefore, in processing the *Web* dataset, more recursions are needed in the top-down core decomposition process. Table VI also reports that the time taken to compute the k_{max}-class Ψ_k is a small portion of the overall core decomposition time, which shows an advantage of our top-down approach for applications that only require the k_{max}-class.

Table VII reports the CPU time, the disk I/O time, and the total running time (e.g., elapsed wall-clock time) in seconds. The result shows that the disk I/O time of EMcore is larger than the CPU time for processing the *BTC* dataset, while it is the opposite for the *Web* dataset. In general the disk I/O time dominates, as it is for the *BTC* dataset. However, in the case of processing the *Web* dataset, the number of recursions is much larger due to a larger k_{max}, which gives rise to a larger CPU time. This result also indicates that our external-memory algorithm is I/O-efficient, as further verified by the results in Table VIII that the amount of disk I/Os needed in the recursive core decomposition phase is almost the same as that needed in the graph partitioning phase, which reads the input graph only once and writes an amount of data at most as large as the input graph. Thus, the result again justifies our conclusion in Section VI-A that the recursive procedure uses only $O(\frac{m+n}{B})$ disk I/Os instead of $O(\frac{k_{max}(m+n)}{M})$ disk I/Os in practice.

TABLE VII
CPU TIME, DISK I/O TIME, AND TOTAL RUNNING TIME (ELAPSED WALL-CLOCK TIME) FOR *BTC* AND *Web*

	CPU Time	Disk I/O Time	Total Running Time
EMcore (*BTC*)	355 sec	757 sec	1112 sec
EMcore (*Web*)	1245 sec	614 sec	1859 sec

TABLE VIII
PERCENTAGE OF DISK I/OS FOR *BTC* AND *Web*

	Graph Partitioning	Top-Down Core Decomposition
EMcore (*BTC*)	50%	50%
EMcore (*Web*)	47.7%	52.3%

VII. Related Work

A considerable amount of research has been done on studying the existence of a non-empty k-core in a random graph [11], [12], [13], [14], [15], usually with the assumption of certain degree distribution or structure properties. There is also a study on the structural characteristics of k-cores in damaged random networks and the nature of the k-core percolation in complex networks [7]. We have discussed in Section I quite a number of studies that apply k-cores in different areas, such as network analysis [1], [2], [6], [5], network visualization [6], [3], [4], protein function prediction [8], structure analysis of software systems [9], and graph model validation [10], as well as many graph problems such as clique finding, dense subgraph problems [16], [17], dense cluster discovery [18], and approximation of between-ness score [10].

From the algorithmic perspective, Batagelj and Zaversnik [20] apply bin-sort to design an $O(m + n)$ algorithm for core decomposition. However, the algorithm requires memory space of $\Omega(m + n)$, which is not feasible for many massive real networks today.

VIII. Conclusions

We propose the first external-memory algorithm, **EMcore**, for core decomposition in massive networks that cannot be kept in main memory. To limit memory usage, we devise a novel top-down approach for core decomposition. Compared with the conventional bottom-up approach, which requires enough memory to hold the entire graph since the 0-core is the entire graph, our top-down approach starts from the smallest-size core, i.e., the k_{max}-core, and recursively reduces the search space and disk I/O cost for each k-core computed. We prove that EMcore requires only $O(k_{max})$ scans of the graph in the worst case and we give detailed analysis on the value of k_{max} for scale-free networks. Our experimental results on various real networks verify that EMcore is efficient for core decomposition in massive networks with up to 52.9 million vertices and 1.65 billion edges, which cannot be processed by the in-memory algorithm. In particular, the amount of disk I/Os required by EMcore is only about two times (one for graph partitioning, one for the recursive core decomposition) that of a graph scan in practice, instead of $O(k_{max})$ scans in the worst case. The results also show that our algorithm, in spite of all the mechanisms (e.g., graph partitioning) used to achieve I/O efficiency, is as efficient as the state-of-the-art in-memory algorithm [20] when main memory is sufficient to keep the entire input graph.

Acknowledgment

This research is supported in part by the AcRF Tier-1 Grant (M52020092) from Ministry of Education of Singapore.

References

[1] S. B. Seidman, "Network structure and minimum degre," *Social Networks*, vol. 5, pp. 269–287, 1983.

[2] B. Bollobas, "The evolution of sparse graphs, in graph theory and combinatorics," *Proc. Cambridge Combinatorial Conf. in honor of Paul Erdos, Academic Press*, pp. 35–57, 1984.

[3] 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.

[4] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "Medusa - new model of internet topology using k-shell decomposition," *PNAS*, vol. 104, pp. 11 150–11 154, 2007.

[5] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "How the k-core decomposition helps in understanding the internet topology," in *ISMA Workshop on the Internet Topology*, 2006.

[6] ——, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *NIPS*, 2005.

[7] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "k-core organization of complex networks," *Phys. Rev. Lett.*, vol. 96, no. 4, p. 040601, 2006.

[8] Md. Altaf-Ul-Amin et al., "Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences," *Genome Informatics*, vol. 14, pp. 498–499, 2003.

[9] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou, "Using the k-core decomposition to analyze the static structure of large-scale software systems," *Journal of Supercomputing*, 2009.

[10] J. Healy, J. Janssen, E. E. Milios, and W. Aiello, "Characterization of graphs using degree cores," in *WAW*, 2006, pp. 137–148.

[11] T. Luczak, "Size and connectivity of the k-core of a random graph," *Discrete Mathematics*, vol. 91, no. 1, pp. 61–68, 1991.

[12] B. Pittel, J. Spencer, and N. Wormald, "Sudden emergence of a giant k-core in a random graph," *J. Comb. Theory Ser. B*, vol. 67, no. 1, pp. 111–151, 1996.

[13] C. Cooper, "The cores of random hypergraphs with a given degree sequence," *Random Struct. Algorithms*, vol. 25, no. 4, pp. 353–375, 2004.

[14] M. Molloy, "Cores in random hypergraphs and boolean formulas," *Random Struct. Algorithms*, vol. 27, no. 1, pp. 124–135, 2005.

[15] S. Janson and M. J. Luczak, "A simple solution to the k-core problem," *Random Struct. Algorithms*, vol. 30, no. 1-2, pp. 50–62, 2007.

[16] G. Kortsarz and D. Peleg, "Generating sparse 2-spanners," *J. Algorithms*, vol. 17, no. 2, pp. 222–236, 1994.

[17] R. Andersen and K. Chellapilla, "Finding dense subgraphs with size bounds," in *WAW*, 2009, pp. 25–37.

[18] Y. Qian, G. Zhang, and K. Zhang, "FaÇade: A fast and effective approach to the discovery of dense clusters in noisy spatial data," in *SIGMOD Conference*, 2004, pp. 921–922.

[19] R. A. Hanneman and M. Riddle, "Introduction to social network methods," *University of California, Riverside*, 2005.

[20] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[21] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[22] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning for irregular graphs," *SIAM Review*, vol. 41, no. 2, p. 278300, 1999.

[23] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *IPDPS*, 2006.

[24] D. J. de Solla Price, "Networks of scientific papers," *Science*, vol. 149, no. 3683, pp. 510–515, 1965.

[25] R. Albert, H. Jeong, and A.-L. Barabási, "The diameter of the world wide web," *Nature*, vol. 401, pp. 130–131, 1999.

[26] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM*, 1999, pp. 251–262.

[27] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, pp. 167–256, 2003.

[28] S. N. Dorogovtsev and J. F. F. Mendesand, "Evolution of networks: From biological nets to the internet and www," *Oxford University Press*, 2003.

[29] V. Boginski, S. Butenko, and P. M. Pardalos, "Statistical analysis of financial networks," *Computational Statistics & Data Analysis*, vol. 48, no. 2, pp. 431–443, 2005.

[30] G. Bianconi and M. Marsili, "Emergence of large cliques in random scale-free networks," *Europhysics Letters*, vol. 74, no. 4, pp. 740–746, 2006.

[31] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks by h*-graph," in *SIGMOD Conference*, 2010, pp. 447–458.