Core Decomposition in Large Temporal Graphs

Huanhuan Wu[*], James Cheng[*], Yi Lu[*], Yiping Ke[#], Yuzhen Huang[‡*], Da Yan[*], Hejun Wu[‡]

[*]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*
{hhwu,jcheng,ylu,yzhuang,yanda}@cse.cuhk.edu.hk

[#]*School of Computer Engineering, Nanyang Technological University*
ypke@ntu.edu.sg

[‡]*Department of Computer Science, Sun Yat-sen University*
{hyuzhen2@mail2.sysu.edu.cn,wuhejun@mail.sysu.edu.cn}

***Abstract*—Core decomposition has been applied widely in the visualization and analysis of massive networks. However, existing studies of core decomposition were only limited to non-temporal graphs, while many real-world graphs can be naturally modeled as temporal graphs (e.g., the interaction between users at different time in online social networks, the phone call or messaging records between friends over time, etc.). In this paper, we define the problem of core decomposition in a temporal graph, propose efficient distributed algorithms to compute the cores in massive temporal graphs, and discuss how the technique can be used in temporal graph analysis.**

I. Introduction

In recent years, a lot of attention has been paid on graph analysis due to the ubiquity of graph data in many application domains such as online social networks, mobile communication networks, the Web, online e-commerce networks, etc. One important problem in graph analysis is to identify cohesive subgraphs, for example, (maximal) cliques, quasi-cliques, n-cliques, k-plexes, n-clans, k-cores [19], k-trusses [20], and other types of densest subgraphs.

In this paper, we study the cohesive subgraphs, k-cores, which are a type of hierarchical graph substructures. Since each k-core is a subgraph of the $(k-1)$-core, k-cores are widely used in the visualization of massive graphs at different granularities [1]. There are also studies that used k-cores to analyze the hierarchies, self-similarity, centrality, and connectivity in large networks [2], as well as to interpret cooperative processes in complex networks [1], [6].

The problem of computing k-cores, also called *core decomposition*, is to compute the largest subgraph (of the input graph) in which every vertex has degree at least k within the subgraph, for all k that the k-core is not an empty graph. The problem has been extensively studied [5], [7], [15], [19] in the literature. However, existing studies focused on the analysis of *non-temporal graphs*, including incremental updates on non-temporal graphs [12], [18]. The study of core decomposition, or other visualization tools, in temporal graphs is still missing, even though temporal graphs are very common in real world [9], [22]. In view of this, we propose the problem of *core decomposition in a temporal graph* in this paper.

A *temporal graph* is a graph in which two vertices may communicate with each other at multiple time instances/intervals. For example, in Figure 1(a), the 3 edges between a and b indicate that there is communication between a and b at time 1, 3 and 4 (e.g., Day 1, Day 3, and Day 4). Many real-world graphs can be naturally modeled as temporal graphs, e.g., users call or send messages to each other at different time in phone call networks and messaging networks, users comment on others' postings at different time in social networks, etc.

Temporal graphs are much more difficult to handle than non-temporal graphs due to the extra time information and the existence of multiple temporal edges between two vertices. One may discard the time information on edges and condense the multiple edges between any two vertices into a single edge, thus obtaining a non-temporal version of the graph, called a *de-temporal graph*. But a de-temporal graph loses all temporal information and even presents erroneous information that leads to serious incorrect understanding of the graph or relationship between objects [10], [22].

The definition of k-core, however, is not well formulated in a temporal graph. In fact, as we will show later in Section III, there is information loss by simply considering k-core in a temporal graph. Instead, we define the (k, h)-core, where h accounts for the number of multiple temporal edges between two vertices. Given a temporal graph G, the (k, h)-core of G is the largest subgraph H of G such that every vertex v in H has at least k neighbors, where there are at least h temporal edges between v and each of these neighbors in H. We list a few applications of (k, h)-core as follows.

- **Visualization.** Effective visualization can be useful for analyzing a large graph. Apart from trivially breaking down a temporal graph into snapshots of non-temporal graphs, visualization methods have rarely been studied for temporal graphs. Note that the number of snapshots is often too large for efficient analysis of a temporal graph (e.g., one dataset used in our experiment has 134,074,906 snapshots), as it is difficult to analyze the temporal relationship of vertices across a large number

of snapshots. The (k, h)-cores offer a natural way of visualizing a temporal graph at different granularities in two different dimensions, where k controls the connectivity among vertices while h controls the intensity of temporal activity between any two vertices. The results can then be used to analyze the hierarchies, centrality, connectivity and evolution of networks over time [2].

- **Evolution of important vertices and their connections.** The (k, h)-cores can be used to measure the importance of vertices in a temporal graph, where a vertex appearing in a (k, h)-core with larger values of k and h is considered more important because of the higher connectivity within the core and the stronger intensity of communication between pairs of vertices. Consider that the temporal graph as a data stream, modeled as a landmark window, where edges are added at the time when they become active. We can compute the (k, h)-cores for the temporal graph in each current landmark window, so that we can compare and study the changes of important vertices over time (e.g., their emergence and life cycle as important vertices).
- **Densest subgraph.** Identifying subgraphs with high density has many applications in social network analysis, e.g., community detection, link spam detection, etc. The densest at-least-k subgraph is to find an induced subgraph with the highest density among all subgraphs with at least k vertices [3]. For a temporal graph, we consider the multiple temporal edge connections by requiring at least h temporal communications between two vertices in the subgraph. The problem is NP-hard, but we can apply the (k, h)-cores to obtain a $(1/3)$-approximation solution for any given h.

Note that other applications of k-core in non-temporal graphs (see more related work on applications of k-core in [7]) can also be generally transformed into corresponding applications of (k, h)-core for temporal graphs.

We first propose a distributed algorithm based on Pregel's vertex-centric computing model [14] for core decomposition in large temporal graphs. We highlight the performance bottleneck in the vertex-centric algorithm, and improve the algorithm by devising an efficient block-centric distributed algorithm based on Blogel's computing model [23]. Experimental results show that our block-centric distributed algorithm is efficient and scalable. We also show how (k, h)-cores can be used to analyze temporal graphs.

Paper organization. Sections II and III give the notations and problem definition. Section IV presents the distributed algorithms. Section V reports experimental results. Section VI discusses related work and Section VII concludes the paper.

II. NOTATIONS

Let $G = (V, E)$ be an undirected temporal graph, where V is the set of vertices of G and E is the set of edges of G. An edge $e \in E$ is a triplet (u, v, t), where $u, v \in V$, t is the time that e is *active*, or the *active time* of e (e.g., u communicates/interacts with v during t). The active time t can be expressed as a period $t = [t_s, t_e]$ that starts at time t_s and ends at time t_e, with the *duration* $\lambda = t_e - t_s$. For simplicity, we assume that λ is a fixed time unit for all edges. Note that longer duration can be expressed as a sequence of consecutive time units (e.g., an edge $(u, v,$ [day-1, 2 days]$)$ can be represented by two edges $(u, v,$ day-1$)$ and $(u, v,$ day-2$)$, with the time unit λ equal to 1 day specified for all edges). In this paper, we focus on undirected temporal graphs.

Two vertices, u and v, may communicate with each other at multiple times. We denote the set of temporal edges between u and v in G by $\Pi(u, v)$, i.e., $\Pi(u, v) = \{(u, v, t) : (u, v, t) \in E\}$. We denote the number of temporal edges between u and v in G by $\pi(u, v)$, i.e., $\pi(u, v) = |\Pi(u, v)|$. We also define the maximum number of temporal edges between u and v, for any u and v in G, by $\pi = \max\{\pi(u, v) : (u, v) \in (V \times V)\}$. The value of π can be large for some real world temporal graphs (e.g., in one of the temporal graphs used in our experiments, $\pi = 285, 521$).

Given two temporal edges $e_1 = (u_1, v_1, t_1) \in E$ and $e_2 = (u_2, v_2, t_2) \in E$, we have $e_1 = e_2$ iff $(u_1 = u_2 \wedge v_1 = v_2 \wedge t_1 = t_2)$ or $(u_1 = v_2 \wedge v_1 = u_2 \wedge t_1 = t_2)$.

If we remove the temporal information from G and condense each $\Pi(u, v)$ into a single edge (u, v), we obtain the *de-temporal graph* of G, denoted by $G^- = (V^-, E^-)$, where $V^- = V$ and $E^- = \{(u, v) : (u, v, t) \in E\}$. Let $G_i = (V_i, E_i)$ be a subgraph of G^-, where $V_i = V$ and $E_i = \{(u, v) : (u, v, t) \in E, \pi(u, v) \geq i\}$ for $i \geq 1$. Note that $G_1 = G^-$.

We define the number of vertices in G and G^- as $n = |V| = |V^-|$, and the number of edges in G as $M = |E|$ and in G^- as $m = |E^-|$. We define the number of vertices and edges in G_i as $n_i = |V_i|$ and $m_i = |E_i|$, respectively. We define the set of *neighbors* of a vertex u in G, G^- or G_i as $\Gamma(u, G) = \Gamma(u, G^-) = \{v : (u, v, t) \in E\} = \{v : (u, v) \in E^-\}$, or $\Gamma(u, G_i) = \{v : (u, v) \in E_i\}$. We define the *degree* of u in G as $d(u, G) = \sum_{v \in \Gamma(u,G)} \pi(u, v)$, in G^- as $d(u, G^-) = |\Gamma(u, G^-)|$, and in G_i as $d(u, G_i) = |\Gamma(u, G_i)|$.

Figure 1(a) shows a temporal graph G and its de-temporal graph G^- is shown in Figure 1(b). The numbers on the edges are the active time of the edges. We have $\Gamma(b, G) = \Gamma(b, G^-) = \{a, c, e, f\}$, $\Pi(a, b) = \{(a, b, 1), (a, b, 3), (a, b, 4)\}$ and thus $\pi(a, b) = 3$, $d(b, G) = 3 + 1 + 1 + 1 = 6$ and $d(b, G^-) = 4$. Also, $\Gamma(b, G_3) = \{a\}$, and $d(b, G_3) = 1$.

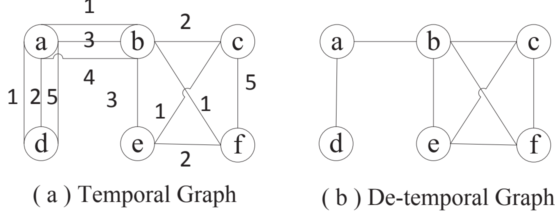


Figure 1. A temporal graph G and its de-temporal graph G^{-}

III. Definitions of Temporal Core

In a de-temporal graph (or any non-temporal graph), $G^{-} = (V^{-}, E^{-})$, a ***k*-core** is the largest subgraph, H^{-}, of G^{-} such that every vertex of H^{-} has at least k neighbors in H^{-}. We define the **core number** of a vertex $v \in V^{-}$ as $\phi(v) = k$ such that v is in the k-core of G^{-} but v is not in the $(k+1)$-core of G^{-}.

Core decomposition in G^{-} is to compute every non-empty k-core of G^{-} for $k \geq 1$. Alternatively, we can compute $\phi(v)$ for every vertex $v \in V^{-}$, as the k-core is simply the subgraph of G^{-} induced by $S = \{v : v \in V^{-}, \phi(v) \geq k\}$.

Technically speaking, isolated vertices, i.e., vertices with degree 0, are in the 0-core. We do not consider isolated vertices in this paper as they can be trivially handled.

We can define the k-core of a temporal graph in a similar way.

Definition 1 (Temporal k-core): The k-core of a temporal graph $G = (V, E)$ is the largest subgraph, H, of G such that for every vertex v in H, $d(v, H) \geq k$. □

The following example shows the concept of temporal k-core and also reveals its weakness.

Example 1: Figure 1(a) shows a temporal graph G. According to Definition 1, the k-core of G is just G itself for $k \in \{1, 2, 3\}$, since $d(v, G) \geq k$ for every vertex v in G, while the 4-core of G does not exist.

We can see that a is closely connected to b and d as there are more temporal edges between them, while $\{b, c, e, f\}$ forms a different community. However, such differences are not distinguished in the k-cores of G since the k-cores are the same for $1 \leq k \leq 3$, nor can the core numbers of the vertices reveal any difference between the vertices since $\phi(a)$=$\phi(b)$=$\phi(c)$=$\phi(d)$=$\phi(e)$=$\phi(f)$=3. ■

The problem of Definition 1 is that it does not consider the temporal connections between two vertices, i.e., the number of temporal edges between the two vertices. To fix this problem, we define temporal core as follows.

Definition 2 (Temporal (k,h)-core): The (k, h)-core of a temporal graph $G = (V, E)$ is the largest subgraph, H, of G such that for every vertex v in H, $|\{u : u \in \Gamma(v, H), \pi(u, v) \geq h\}| \geq k$. □

In Definition 2, on top of the number of neighbors, we enforce that every vertex v in H must have at least k neighbors, where each such neighbor u must be connected to v with at least h temporal edges (u and v communicate with each other for at least h times).

Let H be the (k, h)-core of a temporal graph G. Note that there can be vertices, u, v, in H such that $\pi(u, v) < h$. These edges between u and v can be excluded from H such that for every v in H, we still have $|\{u : u \in \Gamma(v, H), \pi(u, v) \geq h\}| \geq k$. In our work, we simply preserve these edges in H.

Example 2: Based on Definition 2, a and d are in the $(1, 3)$-core; c, e and f are in the $(3, 1)$-core; and b is in the $(1, 3)$-core, as well as in the $(3, 1)$-core. Thus, the $(1, 3)$-core and the $(3, 1)$-core of G clearly distinguish the close temporal relationship between a and b, from a different community which b forms with c, e and f. This example shows that the temporal (k, h)-cores are useful for understanding the temporal structures in graphs. ■

We define the **core numbers** of a vertex $v \in V$ in a temporal graph $G = (V, E)$ as $\Phi(v) = \{(k, h) : v$ is in the (k, h)-core of G but v is not in the (k', h')-core of G, where $k' \geq k$ and $h' \geq h$ and $(k' \neq k$ or $h' \neq h)\}$. Let $S = \{v : v \in V, \exists(k', h') \in \Phi(v), k' \geq k, h' \geq h\}$. The (k, h)-core of G is simply the subgraph of G induced by S.

The problem we study in this paper is to compute $\Phi(v)$ for every vertex v in a temporal graph G. Due to space limitation, we only present distributed algorithms in this paper, while the sequential algorithms can be found in in [21].

We have following lemmas for $\Phi(v)$ (proofs can be found in [21]).

Lemma 1: Let $(k, h) \in \Phi(v)$. Then, $\phi(v) = k$ for v in G_h.

Based on Lemma 1, we can simply compute the core numbers in $\Phi(v)$ from G_i for each i, where $1 \leq i \leq \pi$.

Let $\phi_i(v)$ be the core number of v in G_i, where $1 \leq i \leq \pi$. If $\phi_j(v) = \phi_i(v)$, where $j < i$, we say $(\phi_i(v), i)$ dominates $(\phi_j(v), j)$ and hence we can remove $(\phi_j(v), j)$ from $\Phi(v)$. Note that for any vertex v in G_j and G_i, $\phi_j(v) \geq \phi_i(v)$ since G_j is a supergraph of G_i.

Lemma 2: The total size of $\Phi(v)$ for all $v \in V$ is $O(n + m)$.

IV. Distributed Algorithms

Massive graphs have become common today and sequential algorithms become unsuitable due to limitation in both memory and CPU resources. To process large graphs, distributed algorithms offer a good recourse [15]. We first present a vertex-centric distributed algorithm, and then we identify the computational bottlenecks in the vertex-centric algorithm, and propose an efficient block-centric algorithm as a solution.

Algorithm 1: Vertex-centric distributed core decomposition in a non-temporal graph

Input : An undirected non-temporal graph $F = (V_F, E_F)$
Output : The core number, $\phi(v)$, for each vertex $v \in V_F$

```
1  Each vertex v ∈ V_F keeps four fields: Γ(v,F), Γ'(v,F),
   d(v), and φ(v);
2  Initially, all vertices are active, the master initializes k = 1
   and calls v.compute() for each v ∈ V_F to start the
   computation;
3  When all vertices vote to halt and there is no message
   pending for the next superstep, the master sets k = k + 1,
   sets all vertices to be active, and calls v.compute() for each
   vertex;
4  Each vertex v sends the master a value of 1 when φ(v) ← k
   is executed (or if d(v,F) = 0, i.e., v is an isolated vertex),
   the master aggregates the values and terminates the program
   when the aggregated value is equal to n;

5  v.compute(messages):
6  begin
7      if superstep # = 1 then // executed when k = 1
8          if d(v,F) = k then
9              φ(v) ← k;
10             Send a message < v > to each vertex in
               Γ(v,F);
11         else if d(v,F) > k then
12             Γ'(v,F) ← Γ(v,F);
13             d(v) ← d(v,F);
14             φ(v) ← −1;
15     else if superstep # > 1 and φ(v) = −1 then
16         foreach message, < u >, received in messages do
17             Remove u from Γ'(v,F);
18         d(v) ← d(v) − i, where i is the number of messages
           in messages;
19         if d(v) ≤ k then
20             φ(v) ← k;
21             Send a message < v > to each vertex in
               Γ'(v,F);
22     v votes to halt;
```

A. Vertex-Centric Distributed Algorithm

The vertex-centric distributed algorithm for core decomposition is based on Pregel's computing model [14] (see Section VI-A for a review). We first discuss the distributed algorithm for core decomposition in a non-temporal graph F, as shown in Algorithm 1. The main idea is: starting from $k = 1$, recursively delete all the vertices with degree less than or equal to k, along with their incident edges, from the graph. The details are as follows.

The algorithm proceeds in rounds, and the k-th round computes the core number for any vertex v where $\phi(v) = k$. The master activates all vertices and the workers execute $v.compute()$ for each vertex $v \in V_F$ in parallel. In $v.compute()$, we assign $\phi(v)$ to be k if $d(v) \leq k$, where $d(v)$ is the current degree of v in the k-core of F ($d(v) = d(v, F)$ initially), since v is currently in the k-core of F and v cannot be in the $(k+1)$-core of F. After $\phi(v)$ is determined, we can delete v and all edges incident to v. We do not explicitly delete v since this incurs extra cost and is not necessary, instead we use "$\phi(v) = k \neq -1$" as a mark that v is implicitly deleted. To remove the edges incident to v, we send a message $< v >$ to each vertex $u \in \Gamma'(v, F)$, where $\Gamma'(v, F)$ is the set of neighbors of v in the k-core of F ($\Gamma'(v, F) = \Gamma(v, F)$ initially); and when u receives the message $< v >$, u removes v from $\Gamma'(u, F)$ and at the same time decrements $d(u)$. The process repeats for each active vertex until all vertices vote to halt and there is no more message pending for the next superstep (note that a vertex becomes active in the next superstep if it receives a message). Then, the master increments k to $k + 1$ and activates all vertices to start the $(k+1)$-th round, until $\phi(v)$ is determined for all $v \in V_F$.

The correctness of Algorithm 1 follows directly from the definition of k-core.

Algorithm 2: Vertex-centric distributed core decomposition in a temporal graph

Input : An undirected temporal graph $G = (V, E)$
Output : The core numbers, $\Phi(v)$, for each vertex $v \in V$

```
1  Each vertex v ∈ V keeps two fields: Γ(v,G) and Φ(v);
2  Initially, all vertices are active, the master calls v.compute()
   for each v ∈ V to compute the de-temporal graph G⁻ of G.
   During the process, the master receives C(v) from each
   v ∈ V, and computes C = ⋃_{v∈V} C(v). The master then
   sorts C in ascending order;
3  foreach i ∈ C in order do
4      The master calls v.compute() for each v ∈ V to
       compute G_i (note that G_1 is computed already in Line 9,
       while G_i is to be computed by each v in Line 12);
5      Then the master activates each v in G_i to compute φ(v)
       by Algorithm 1 with F = G_i, where F = G_i is
       constructed in Line 13 of Algorithm 2. For each v in G_i,
       if φ(v) ≥ 1, v adds (φ(v), i) to Φ(v), and remove any
       (φ'(v), j) from Φ(v) if φ'(v) = φ(v) and j < i;

6  v.compute(messages):
7  begin
8      if i = 1 then
9          Γ(v,G⁻) ← {(u, π(v,u)) : u ∈ Γ(v,G)};
10         Send C(v) ← {π(v,u) : u ∈ Γ(v,G)} to the master;
11     else if i > 1 then
12         Γ(v,G⁻) ← {(u, π(v,u)) : (u, π(v,u)) ∈
           Γ(v,G⁻), π(v,u) ≥ i};
13     Γ(v,F) ← {u : (u, π(v,u)) ∈ Γ(v,G⁻)};
14     v votes to halt;
```

The distributed algorithm for core decomposition in a temporal graph G, as shown in Algorithm 2, makes use of Algorithm 1 to compute $\phi(v)$ for each vertex v in each G_i of G. The algorithm first computes the de-temporal graph G^- of G, which is done in parallel by the workers

calling $v.compute()$ for each $v \in V$ to obtain $(v, u, \pi(v, u))$ for each $u \in \Gamma(v, G)$, as stored in $\Gamma(v, G^-)$ (Line 9). Meanwhile, the set $C = \{\pi(u, v) : (u, v, t) \in E\}$ is also computed and the elements in C are sorted in ascending order of their values.

Then, the algorithm proceeds in $|C|$ rounds. For each $i \in C$ in order, the round consists of two phases: (1)compute G_i from G_j (Line 12), where j is the element ordered before i in C, i.e., G_j is the graph processed in the previous round (except for $G_1 = G^-$); and (2)call Algorithm 1 with $F = G_i$ to compute $\phi(v)$ for each v in G_i, and then update $\Phi(v)$ with $(\phi(v), i)$ as in Line 5.

Theorem 1: Algorithm 2 correctly computes $\Phi(v)$ for each $v \in V$.

Performance bottleneck of the vertex-centric algorithm. There are four main costs in running an algorithm in Pregel-like systems [17], [25]: (1)communication cost per superstep; (2)computation cost per superstep; (3)memory used per superstep; and (4)the total number of supersteps. Costs (1)-(3) of Algorithm 2 are linear in the size of the input graph. However, the total number of supersteps required by Algorithm 2 can be large, since Algorithm 1 may take $O(n)$ supersteps in the worst case. Thus, large superstep number becomes the bottleneck of Algorithm 2, which is also verified in our experiments.

B. Block-Centric Distributed Algorithm

To eliminate the performance bottleneck in the vertex-centric algorithm, we adopt Blogel's block-centric model [23] (see Section VI-A for a review). The main idea is to partition the vertex set of each G_i into disjoint partitions, construct a subgraph of G_i for each partition, and then apply the Blogel framework for core decomposition on each subgraph in parallel. We first present the algorithm in details, and then discuss how the block-centric algorithm eliminates the performance bottleneck in the vertex-centric algorithm.

We first define the notion of extended subgraph. Let $F = (V_F, E_F)$ be a non-temporal graph. Given a subset of vertices, $V_B \subseteq V_F$, the *extended subgraph* of F w.r.t. V_B is defined as $B = (V_B \cup V_{B^+}, E_B)$, where V_B is the set of *internal vertices* of B, $V_{B^+} = (\bigcup_{v \in V_B} \Gamma(v, F)) \setminus V_B$ is the set of *extended vertices* of B, $E_B = \bigcup_{v \in V_B} \{(v, u) : u \in \Gamma(v, F)\}$ is the set of edges of B.

Given a non-temporal graph $F = (V_F, E_F)$, we partition V_F into p disjoint partitions $V_F = \{V_{F_1}, \ldots, V_{F_p}\}$, and construct the extended subgraph B_i of F w.r.t. V_{F_i}. For each extended subgraph B_i, we call Algorithm 3 to compute the core number for each vertex v in B_i locally in a worker. We use $\varphi(v)$ to denote the core number of v computed by Algorithm 3, which is an upper bound of the real core number $\phi(v)$.

Since Algorithm 3 does not operate on the complete graph, we need extra information from the extended vertices,

Algorithm 3: Core decomposition in an extended subgraph

Input : The extended subgraph, $B = (V_B \cup V_{B^+}, E_B)$, of a non-temporal graph $F = (V_F, E_F)$ w.r.t. V_B; and the upper-bound core number, $\varphi(v)$, for every vertex $v \in V_{B^+}$
Output : The upper-bound core number, $\varphi(v)$, for each $v \in V_B$

```
1  Initialize d(v) = d(v, B) for each v ∈ V_B;
2  S ← V_B, S+ ← V_B+;
3  Sort the vertices v ∈ S in ascending order of d(v), let
   d_min = min{d(v) : v ∈ S};
4  Sort the vertices v ∈ S+ in ascending order of φ(v), let
   φ_min = min{φ(v) : v ∈ S+};
5  while |S| > 0 do
6      while φ_min < d_min do
7          Let u be the vertex ordered at the first position in
           S+;
8          foreach v ∈ Γ(u, B) do
9              d(v) ← d(v) − 1;
10             reorder the vertices in S;
11         Remove u from S+;
12         Update φ_min and d_min;
13     Let v be the vertex ordered at the first position in S;
14     φ(v) ← d(v);
15     foreach u ∈ Γ(v, B) and u ∈ S do
16         if d(u) > d(v) then
17             d(u) ← d(u) − 1;
18             reorder the vertices in S;
19     Remove v from S;
20     Update d_min;
```

i.e., V_{B^+}. Specifically, the algorithm requires $\varphi(u)$ for every vertex $u \in V_{B^+}$. Note that if $\varphi(u) = \phi(u)$ for every $u \in V_{B^+}$, then $\phi(v)$ of each internal vertex $v \in V_B$ can be computed from the extended subgraph B alone based on the following property.

Property 1: Given a non-temporal graph F=(V_F, E_F) and an edge $(u, v) \in E_F$, let $\phi_{min} = \min\{\phi(u), \phi(v)\}$, then (u, v) is in the ϕ_{min}-core but not in the $(\phi_{min} + 1)$-core of F. □

Let $v \in V_B$ and $u \in V_{B^+}$, where $u \in \Gamma(v, F)$, i.e., $(u, v) \in E_B$. If $\phi(v) > \phi(u)$, we can remove u since (u, v) is not in the $\phi(v)$-core. If $\phi(v) \leq \phi(u)$, then (u, v) is in the $\phi(v)$-core according to Property 1. Since we compute $\phi(v)$ from the $\phi(v)$-core, the presence of such edge (u, v) in the $\phi(v)$-core means that the extended subgraph B contains all the information needed for the computation of $\phi(v)$ for each internal vertex $v \in V_B$.

Algorithm 3 recursively removes extended vertices from V_{B^+} whose $\varphi(.)$ value is smaller than d_{min}, where d_{min} is the smallest degree of an internal vertex in V_B, because these extended vertices are not in the d_{min}-core according to Property 1. Upon the removal of each extended vertex

Algorithm 4: Block-centric distributed core decomposition in a non-temporal graph

Input : An undirected non-temporal graph $F = (V_F, E_F)$
Output : The core number, $\phi(v)$, for each vertex $v \in V_F$

1 The vertex set V_F is divided into p disjoint partitions $V_F = \{V_{F_1}, V_{F_2}, \dots, V_{F_p}\}$;
2 Construct the extended subgraph $B_i = (V_{B_i} \cup V_{B_i^+}, E_{B_i})$ of F for $1 \leq i \leq p$, where $V_{B_i} = V_{F_i}$; initialize $\phi(v) \leftarrow d(v, F)$ for each $v \in V_{B_i}$ and the upper-bound core number $\varphi(u) \leftarrow d(u, F)$ for each $u \in V_{B_i^+}$;
3 Initially, all blocks B_i for $1 \leq i \leq p$ are *active*, the master calls $B_i.compute()$ for each B_i to start the computation (which terminates when all blocks vote to halt and there is no message pending for the next superstep);

4 B.**compute(*messages*)**:
5 **begin**
6   **foreach** *message, $< u, k >$, received in* ***messages****, if any* **do**
7   **if** $k < \varphi(u)$ **then**
8             $\varphi(u) \leftarrow k$;
9   Call Algorithm 3 with input B to compute an upper-bound core number $\varphi(v)$ for each $v \in V_B$;
10   **foreach** $v \in V_B$ **do**
11   **if** $\varphi(v) < \phi(v)$ **then**
12             $\phi(v) \leftarrow \varphi(v)$;
13   **foreach** $u \in \Gamma(v, B)$ *and* $u \in V_{B+}$ **do**
14   **if** $\phi(v) < \varphi(u)$ **then**
15   Let B' be the extended subgraph where $u \in V_{B'}$;
16   Send a message $< v, \phi(v) >$ to the block B';
17   B votes to halt;

Algorithm 5: Block-centric distributed core decomposition in a temporal graph

Input : An undirected temporal graph $G = (V, E)$
Output : The core numbers, $\Phi(v)$, for each vertex $v \in V$

1 Each vertex $v \in V$ keeps three fields: $\Gamma(v, G)$, $\phi(v)$, and $\Phi(v)$;
2 Compute $C = \{\pi(u, v) : (u, v, t) \in E\}$, where elements in C are sorted in ascending order, as in Line 2 of Algorithm 2;
3 **foreach** $i \in C$ *in order* **do**
4   Compute G_i as in Lines 8-14 of Algorithm 2;
5   Call Algorithm 4 with input $F = G_i$; but starting from the second call of Algorithm 4, replace the initialization "$\phi(v) \leftarrow d(v, F)$" in Line 2 of Algorithm 4 by "$\phi(v) \leftarrow \min\{\phi(v), d(v, F)\}$". Meanwhile, update $\Phi(v)$ as in Line 5 of Algorithm 2;

u, we also decrement the degree of each of u's neighbors in B. If the original graph B should not be modified, we can make a copy S^+, S and $d(v)$ for V_{B+}, V_B and $d(v, B)$, respectively, and make changes on the copies.

When there is no extended vertex whose $\varphi(.)$ value is smaller than d_{min}, it implies that all the vertices are now in the d_{min}-core. Thus, each vertex $v \in V_B$, where $d(v) = d_{min}$, has $\varphi(v) = d(v)$. We recursively remove each vertex v, when $\varphi(v)$ is determined, and decrement the degree of each neighbor u of v, if u is also an internal vertex in B and $d(u) > d(v)$. Note that if $d(u) = d(v)$, u is also in the d_{min}-core and will be removed next, and hence $d(u)$ does not need to be updated. The above process repeats until $\varphi(v)$ for all $v \in V_B$ is determined.

The following lemma (see proof in [21]) is vital for Algorithm 3 to be applied in block-centric distributed core decomposition.

Lemma 3: Given $\varphi(u)$, where $\varphi(u) \geq \phi(u)$, for each extended vertex $u \in V_{B+}$, $\varphi(v)$ computed by Algorithm 3 is an upper bound of the true core number $\phi(v)$, for each internal vertex $v \in V_B$.

We now present our block-centric algorithm for distributed core decomposition in a non-temporal graph $F{=}(V_F, E_F)$, as shown in Algorithm 4. The master calls $B_i.compute()$ for each block, i.e., each extended subgraph B_i, and the workers operate on each B_i in parallel. Within $B_i.compute()$, the algorithm calls Algorithm 3 to compute an upper-bound core number for every vertex in V_{B_i}.

For each block B, Line 2 of Algorithm 4 initializes $\phi(v)$ to be $d(v, F)$ for each $v{\in}V_B$ and $\varphi(u)$ to be $d(u, F)$ for each $u{\in}V_{B+}$. Within $B.compute()$, whenever $\varphi(v)$ returned by Algorithm 3 is smaller than the current $\phi(v)$, for any $v \in V_B$, we update $\phi(v)$ to be $\varphi(v)$. Note that v may be in $V_{B'+}$ for another extended subgraph B', and Algorithm 3 with B' as input will require $\varphi(v)$. Thus, in Lines 13-16 of Algorithm 4 we refine $\varphi(v)$ in B' with a smaller value, i.e., the newly updated $\phi(v)$. But note that, let u be the neighbor of v in B', if the newly updated $\phi(v)$ is not smaller than $\varphi(u)$ in this block B, then it means that u is removed before v during the core decomposition in B' regardless of whether $\varphi(v)$ in B' is updated or not, and hence we can skip the update.

The above process repeats until the algorithm converges, i.e., when $\phi(v)$ is not changed for all $v \in V_{B_i}$, for $1 \leq i \leq p$. In this case, no more message will be sent in Line 16 of Algorithm 4, and no block will be activated again for the next superstep.

We devise a distributed block-centric algorithm for core decomposition in a temporal graph, as shown in Algorithm 5, by calling Algorithm 4 to compute $\phi(v)$ in each G_i, for $i \in C = \{\pi(u, v) : (u, v, t) \in E\}$, and then update $\Phi(v)$ with $(\phi(v), i)$.

We can speed up the convergence of the algorithm as follows. Given two elements $(\phi_i(v), i)$ and $(\phi_j(v), j)$ in $\Phi(v)$, according to the definition of $\Phi(v)$, if $i > j$, then $\phi_i(v) < \phi_j(v)$. We utilize this property of $\Phi(v)$ as follows. Let $\phi_i(v)$ and $\phi_j(v)$ be the value of $\phi(v)$ to be computed from G_i and G_j, respectively, where $i > j$ and $\nexists k \in C$ s.t.

Table I
DATASETS

Dataset	$\lvert V \rvert$	$\lvert E^- \rvert$	$\lvert E \rvert$	$d_{avg}(u, G^-)$	$d_{avg}(u, G)$	π	$\lvert T_G \rvert$
amazon	2,146,057	11,486,264	11,553,320	5.35	5.38	28	3,329
arxiv	28,093	6,296,894	9,193,606	224.14	327.25	262	2,337
dblp	1,103,412	8,451,372	11,957,392	7.66	10.84	38	70
delicious	4,535,197	163,985,560	439,161,184	36.16	96.83	1,070	1,583
edit	21,504,191	244,130,836	533,503,390	11.35	24.80	285,521	134,074,906
flickr	2,302,925	45,676,552	49,394,886	19.83	21.45	2	134
wikiconf	118,100	4,055,742	5,835,548	34.34	49.41	562	273,909
wikipedia	1,870,709	73,065,062	78,446,030	39.06	41.93	2	2,198

$i>k>j$. We initialize $\phi_i(v) = \phi_j(v)$ in Line 2 of Algorithm 4 when we compute $\phi_i(v)$ from G_i. This initialization of $\phi(v)$ can speed up the convergence of the computation on G_i.

The proofs to the correctness of Algorithm 4 and Algorithm 5 can be found in [21].

V. EXPERIMENTAL EVALUATION

We ran our experiments on a cluster of 15 machines, where each machine has 24 cores (two Intel Xeon E5-2620 CPU) and 48GB RAM, running 64-bit CentOS 6.5 with Linux kernel 2.6.32. The connectivity between any pair of nodes in the cluster is 1Gbps.

Datasets. We used 8 real temporal graphs, which are from the Koblenz Large Network Collection (http://konect.uni-koblenz.de/), and we selected one large temporal graph from each of the following categories: `amazon-ratings` (`amazon`) from the Amazon online shopping website; `arxiv-HepPh` (`arxiv`) from the arxiv networks; `dblp-coauthor` (`dblp`) from the DBLP coauthor networks; `delicious-ut` (`delicious`) from the network of 'delicious'; `edit-enwiki` (`edit`) from the edit network of the English Wikipedia; `flickr-growth` (`flickr`) from the social network of Flickr; `wikiconflict` (`wikiconf`) indicating positive and negative conflicts between users of Wikipedia; `wikipedia-growth` (`wikipedia`) from the hyperlink network of the English Wikipedia. In the experiments, we transform these datasets into undirected temporal graphs by inserting one edge (v, u, t) into the graphs for every edge (u, v, t).

Table I lists the number of vertices and edges in G and G^-, and the average degree in G (denoted by $d_{avg}(u, G)$) and in G^- (denoted by $d_{avg}(u, G^-)$). The table also shows that the value of π varies significantly for different datasets, indicating different levels of temporal activity between vertices. We also give the number of distinct time instances in G, denoted by $|T_G|$, which shows that G can span over a large time interval. For example, if we break G into snapshots such that all edges with the same starting time belong to the same snapshot, then the `edit` graph consists of 134,074,906 snapshots.

A. Block-Centric vs. Vertex-Centric

We first show the effects of the block-centric computing model and the vertex-centric computing model on distributed

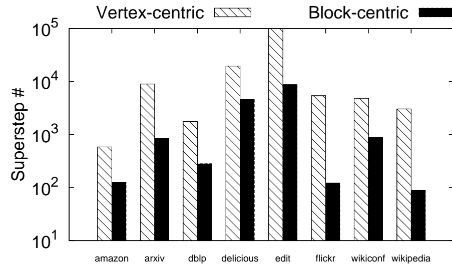


Figure 2. Number of supersteps

Table II
COMPUTATION TIME IN SECONDS

Dataset	*VGiraph*	*MonGiraph*	*VGraphlab*	*MonGraphlab*	*Block*
amazon	383.55	102.89	170.85	78.50	12.05
arxiv	5,540.89	492.18	2,639.19	374.44	21.75
dblp	1,099.57	208.86	596.56	146.30	21.62
delicious	17,385.96	7,078.84	20,060.00	9,720.81	1,412.68
edit	-	14,975.74	-	57,321.40	1,751.42
flickr	3,450.12	115.72	1453.84	91.70	56.84
wikiconf	2,852.86	549.33	1463.21	466.04	22.54
wikipedia	1,981.79	89.73	791.08	76.11	32.14

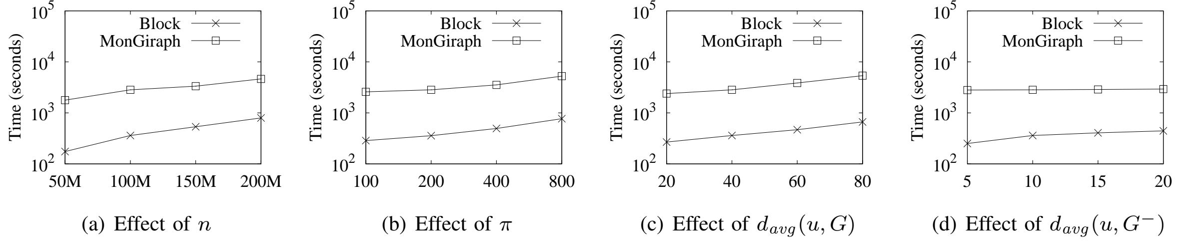
core decomposition. We implemented our vertex-centric algorithm, i.e., Algorithm 2, in Giraph [4] and Graphlab [8], denoted by **VGiraph** and **VGraphlab**, since Giraph and Graphlab are the two most popularly used vertex-centric graph computing systems. We implemented our block-centric algorithm, i.e., Algorithm 5, in a block-centric graph computing system called Blogel [23], denoted by **Block**.

Figure 2 shows the number of supersteps needed by the vertex-centric algorithm and the block-centric algorithm, for running on different datasets. Note that the number of supersteps needed by VGiraph and VGraphlab is the same, while both VGiraph and VGraphlab cannot finish in 24 hours on the `edit` dataset. The figure shows that the vertex-centric algorithm uses 5 times to 40 times more supersteps than the block-centric algorithm. As we discussed in Section IV-A, the performance bottleneck of vertex-centric algorithm lies with the large number of supersteps. Thus, this result demonstrates that the block-centric algorithm can effectively address the performance bottleneck of the vertex-centric algorithm.

B. Performance Comparison

To evaluate the performance of our algorithms, we modified existing distributed k-core algorithms to compute $\Phi(v)$ for every vertex v in a temporal graph. Specifically, we adopt the state-of-the-art, vertex-centric distributed k-core algorithm proposed by Montresor et al. [15]. We also implemented their algorithm in both Giraph and Graphlab, denoted by **MonGiraph** and **MonGraphlab**.

Tables II reports the running time of the algorithms. The sign '-' indicates that the corresponding algorithm cannot finish in 24 hours. The results show that our block-centric algorithm Block is significantly faster than all the other algorithms. MonGiraph and MonGraphlab are faster than our



(a) Effect of n (b) Effect of π (c) Effect of $d_{avg}(u, G)$ (d) Effect of $d_{avg}(u, G^-)$

Figure 3. Performance on synthetic graphs with different n, π, $d_{avg}(u, G)$, and $d_{avg}(u, G^-)$

vertex-centric algorithms VGiraph and VGraphlab, but they are from a few times to over an order of magnitude slower than our block-centric algorithm. This result demonstrates the efficiency of our block-centric algorithm.

We also observe that the performance of VGraphlab is faster than VGiraph in graphs with a small π value, but slower in graphs with a large π value. This is because that Graphlab does not support graph mutation, while Giraph supports the removal of vertices and edges from the graph. When the value of π is large, there are many G_i. As i becomes larger, more vertices and edges are removed, and G_i becomes smaller. Thus, when i is larger, VGiraph is run on a small graph G_i, while VGraphlab is still run on the original large input graph since vertices and edges are not really removed but only marked as "removed". Similar results are also observed for MonGraphlab and MonGiraph.

C. Scalability Study

We now test the scalability of our block-centric algorithm. We generate synthetic temporal graphs with different number of vertices n, different π value, different average degree in G (denoted by $d_{avg}(u, G)$) and in G^- (denoted by $d_{avg}(u, G^-)$). We generate synthetic temporal graphs using the idea similar to random graph generator [16]. As default values, we set $n = 100M$ ($M = 10^6$), $\pi = 200$, $d_{avg}(u, G^-) = 10$, $d_{avg}(u, G) = 40$, and $|T_G| = 100,000$. Then, we vary the value of n, π, $d_{avg}(u, G)$ and $d_{avg}(u, G^-)$, respectively, while fixing other parameters as their default values.

As Figure 3(a) shows, when we vary the number of vertices from 50M to 200M, the running time of Block increases approximately linearly with the number of vertices. As Figure 3(b) shows, the running time of Block increases sub-linearly as π increases from 100 to 800. Similarly, Figures 3(c) and 3(d) show that the running time of Block increases sub-linearly when $d_{avg}(u, G)$ and $d_{avg}(u, G^-)$ increase. Thus, the results verify the scalability of our block-centric algorithm with respect to the increases in n, π, $d_{avg}(u, G)$, and $d_{avg}(u, G^-)$.

As a comparison, we show the running time of MonGiraph, since it is the fastest among all the four vertex-centric algorithms for processing these synthetic datasets. We can see from the figures that, though they show a similar trend in the change of performance as the values of different parameters change, Block is approximately an order of magnitude faster than MonGiraph in almost all cases.

D. Analysis of Temporal Cores

In the following set of experiments, we analyze the properties of temporal cores, by comparing with the cores of the corresponding de-temporal graphs. Our objective is to show that temporal cores are more useful and carrying more accurate information for analyzing temporal graphs than the cores of their de-temporal graphs.

Vertex Ranking in Temporal Graphs:

As shown in [11], the core number of a vertex is a direct indicator of the expected spread of a vertex. The larger the core number of a vertex is, the higher the probability that the vertex is an influential spreader. Thus, the core number of a vertex indicates its importance in the graph. Let $\phi_i(v)$ be the core number of a vertex v in G_i, we use the value of $\sum_i \phi_i(v)$ to rank the vertices in a temporal graph. Note that although $\sum_i \phi_i(v)$ is a simple summation, edges that appear in more G_i's essentially contribute to more times in the summation since they appear in the cores of more G_i's, which implies that higher intensity of communication between two vertices is given higher weight in the summation.

We then select the top-k vertices according to the above-described ranking. As a comparison, we also rank the vertices according to their core number in the de-temporal graphs. To show why the core number computed in the temporal graph should be used for ranking, we report the following measures. Let $T_s(G)$ and $T_s(G^-)$ be the top-s vertices ranked by their core number in the temporal graph and in the de-temporal graph, respectively.

We first compute the *normalized discounted cumulative gain* (*NDCG*) of $T_s(G)$ with reference to $T_s(G^-)$. Note that here we only use NDCG to show the difference between the two rankings rather than to measure the quality of $T_s(G)$. We report the result in Table III, where a value closer to 1 indicates that $T_s(G)$ and $T_s(G^-)$ are more similar, and a value closer to 0 indicates dissimilarity. The result shows that $T_s(G)$ and $T_s(G^-)$, where $100 \leq s \leq 500$, are significantly different from each other except for the

wikipedia dataset. Especially when s is smaller, the difference is obvious; for example, for the dblp and edit datasets, the top-100 vertices in $T_s(G)$ and $T_s(G^-)$ are totally disjoint.

Next, we show that the vertices in $T_s(G)$ are more useful. We compute (1)the number of vertices that each vertex $v \in T_s(G)$ or $v \in T_s(G^-)$ can reach, and (2)the average minimum duration from v to other reachable vertices [22]. Reachability and duration of a path in a temporal graph G are defined as follows. A *temporal path* P in G is a sequence of vertices $P = \langle v_1, v_2, \ldots, v_p, v_{p+1}\rangle$, such that $(v_i, v_{i+1}, t_i) \in E$ is the i-th temporal edge on P for $1 \leq i \leq p$, and $t_i \leq t_{i+1}$ for $1 \leq i < p$. The duration of P is given by $(t_p - t_1)$. The *minimum duration* from u to v is simply the duration of the temporal path from u to v whose duration is the minimum among all temporal paths from u to v. We say u can reach v if there exists a temporal path from u to v.

Let $R(v)$ denote the number of vertices v can reach in G. The larger value $R(v)$ is, the more likely v is important. Let $\overline{f(v)}$ be the average minimum duration from v to other reachable vertices in G. The value $\overline{f(v)}$ indicates how long it takes to spread information from v to other vertices on average. The smaller value $\overline{f(v)}$ is, the higher probability that v is an important vertex. We report the average value of $R(v)$ and $\overline{f(v)}$ for top-s vertices in $T_s(G)$ and $T_s(G^-)$ for the arxiv dataset in Figures 4 and 5, respectively.

From Figure 4, the average number of vertices that can be reached by vertices in $T_s(G)$ is always much larger than that can be reached by vertices in $T_s(G^-)$. The difference is especially significant for small values of s as there are fewer common vertices in $T_s(G)$ and $T_s(G^-)$. Figure 5 shows that the average value of $\overline{f(v)}$ of vertices in $T_s(G^-)$ is always smaller than that of vertices in $T_s(G^-)$, especially for small values of s. Similar conclusions can also be drawn in most other datasets and the results are omitted due to space limitation. The results in these two figures demonstrate that the top-s vertices ranked by their core number in the temporal graph are more important, at least with respect to the scope to which information can be spread from them and the amount of time they need to spread information, than those ranked by their core number in the de-temporal graph. Thus, it is beneficial to analyze temporal graphs using temporal cores.

Table III
NDCG VALUES OF $T_s(G)$ AND $T_s(G^-)$

s	100	200	300	400	500
amazon	0.1589	0.2180	0.3007	0.3259	0.3636
arxiv	0.3462	0.4033	0.4006	0.3928	0.3949
dblp	0.0000	0.0773	0.2068	0.4210	0.5585
delicious	0.3027	0.3672	0.3822	0.3856	0.3930
edit	0.0000	0.0047	0.0274	0.0523	0.0685
flickr	0.4383	0.4325	0.4406	0.4388	0.4305
wikiconf	0.3559	0.3558	0.3787	0.4138	0.4371
wikipedia	0.6823	0.7228	0.7319	0.8360	0.9146

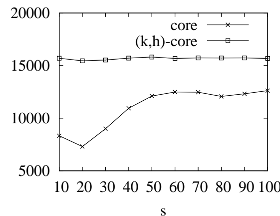


Figure 4. $R(v)$ of arxiv

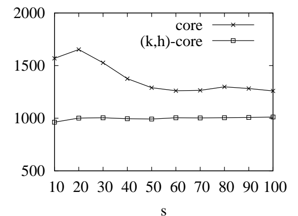


Figure 5. $\overline{f(v)}$ of arxiv

VI. Related Work

In this section, we discuss related work on distributed graph computing and core decomposition.

A. Distributed Graph Computing

Vertex-centric model. Malewicz et al. proposed the vertex-centric computing model for Pregel [14]. In this model, each vertex is an independent computational unit. A program in Pregel implements a user-defined *compute*() function and proceeds in iterations, called *supersteps*, based on the bulk synchronous parallel (BSP) model. In each superstep, the program runs *compute*() for each active vertex, v, in which v receives incoming messages from its neighbors sent in the previous superstep, modifies its value, sends messages to its neighbors (to be received in the next superstep), and votes to halt. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Giraph [4] is a open source of Pregel, while GraphLab [8] adopts the edge-centric model to eliminate unbalanced workload caused by skewed vertex degree. Both Giraph and GraphLab provide an implementation of the k-core algorithm as an application, the idea of which is similar to Algorithm 1. There are also other Pregel-like systems (e.g., Pregel+ [24]) proposed in recent years, and we refer readers to a recent performance study on these systems [13].

Block-centric model. Yan et al. proposed the block-centric graph computing system, Blogel [23]. Blogel partitions an input graph into a set of subgraphs called blocks. In the block-centric computing model, every block is a computational unit (in contrast to the vertex-centric model in which every vertex is a computational unit). Blogel works in a similar way as Pregel, but allows both blocks and vertices to exchange messages with each other. The block-centric model was designed to reduce the number of iterations and the number of messages in the vertex-centric model. This paper proposes the first block-centric distributed algorithm for core decomposition.

B. Core Decomposition

The most efficient in-memory algorithm for core decomposition was proposed in [5]. When main memory is not sufficient, an I/O-efficient algorithm was proposed in [7]. These sequential algorithms are not efficient for processing

massive graphs; thus, Montresor et al. [15] proposed the first non-trivial distributed algorithm for k-core decomposition by iteratively refining the upper bound on the core number of a vertex based on those of its neighbors. Their algorithm is essentially a vertex-centric algorithm but they did not implement it in any Pregel-like systems.

Li et al. [12] presented an efficient incremental algorithm to maintain the core number of every vertex when the graph is updated with single edge insertion/deletion. The algorithm first identifies the set of nodes that may change their core number after inserting/deleting one edge. Then a recoloring algorithm is used to determine the set of vertices whose core number needs to be updated. However, their incremental algorithm is not linear. In [18], the authors proposed linear time algorithms to handle single edge insertion/deletion. The algorithms also first identify a subgraph consisting of a small subset of vertices that may change their core number in the case of edge insertion/deletion. Then, the algorithms work on the subgraph to update the core number in linear time. However, all these algorithms are not practical when the input graph is large.

VII. CONCLUSIONS

We studied the problem of core decomposition in a temporal graph and devised efficient distributed algorithms to compute (k,h)-cores. In particular, our block-centric distributed algorithm is significantly faster than the vertex-centric counterparts. We also showed that (k,h)-cores can be used for vertex ranking in a temporal graph.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. This work was supported by the SHIAE Grant (No. 8115048), the Hong Kong Research Grants Council GRF Project No. CUHK 2150851, and the CUHK Direct Grant No. 4055043.

REFERENCES

[1] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*, 2005.

[2] 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.

[3] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *WAW*, pages 25–37, 2009.

[4] Apache Giraph. http://giraph.apache.org/.

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

[6] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. Medusa - new model of internet topology using k-shell decomposition. *PNAS*, 104:11150–11154, 2007.

[7] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[9] P. Holme and J. Saramäki. Temporal networks. *CoRR*, abs/1108.1780, 2011.

[10] S. Huang, J. Cheng, and H. Wu. Temporal graph traversals. *CoRR*, abs/1401.1919, 2014.

[11] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, 2010.

[12] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE TKDE*, 26(10):2453–2465, 2014.

[13] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[14] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[15] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE TPDS*, 24(2):288–300, 2013.

[16] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT*, pages 331–342, 2011.

[17] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[18] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.

[19] S. B. Seidman. Network structure and minimum degre. *Social Networks*, 5:269–287, 1983.

[20] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[21] H. Wu and J. Cheng. Core decomposition in large temporal graphs. *CUHK Technical Report*, 2015. (http://www.cse.cuhk.edu.hk/~jcheng/tcore.pdf).

[22] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.

[23] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[24] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[25] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.