Distributed k-Core Decomposition

Alberto Montresor, *Member, IEEE*, Francesco De Pellegrini, *Member, IEEE*, Daniele Miorandi, *Member, IEEE*

Abstract—Several novel metrics have been proposed in recent literature in order to study the relative importance of nodes in complex networks. Among those, k-coreness has found a number of applications in areas as diverse as sociology, proteinomics, graph visualization, and distributed system analysis and design. This paper proposes new distributed algorithms for the computation of the k-coreness of a network, a process also known as k–core decomposition. This technique (i) allows the decomposition, over a set of connected machines, of very large graphs, when size does not allow storing and processing them on a single host, and (ii) enables the run-time computation of k-cores in "live" distributed systems. Lower bounds on the algorithms complexity are given, and an exhaustive experimental analysis on real-world datasets is provided.

Index Terms—k-core decomposition, Graph analysis, Bulk Synchronous Parallel

1 Introduction

In the last few years, a number of metrics and methods have been introduced for studying the relative "importance" of nodes within complex network structures. Examples include betweenness, eigenvector and closeness centrality indexes [2], [3]. Such studies have been applied in a variety of settings, including real networks like the Internet topology, social networks like co-authorships graphs, protein networks in bio-informatics, and so on.

Among these metrics, k-coreness is a well-established method for identifying a special family of maximal induced subgraphs of a graph called *k-cores*, or *k-shells* [4]. Informally, a k-core is obtained by recursively removing all nodes of degree smaller than k, until the degree of all remaining vertices is larger than or equal to k. This is also the intuition of the standard Batagelj–Zaveršnik algorithm for k-coreness calculation [5]. The process of computing the k-coreness is also called k-core decomposition: nodes are said to have coreness k (or, equivalently, to belong to the k-shell) if they belong to the k-core but not to the $(k+1)$-core. As an example of k-core decomposition for a sample graph, consider Figure 1. Note that by definition cores are nested, meaning that nodes belonging to the 3-core belong to the 2-core and 1-core, as well. Larger values of "coreness", though, clearly correspond to nodes with a more central position in the network structure.

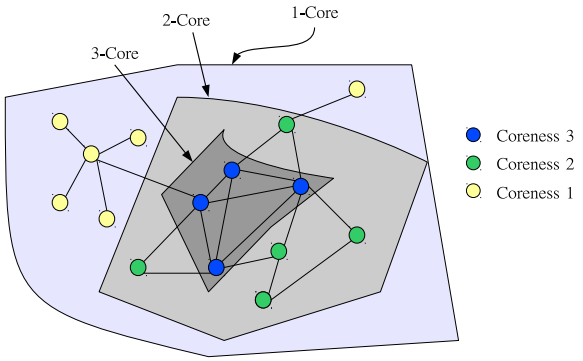

Fig. 1: k–core decomposition for a sample graph.

Motivation: k-core decomposition has found a number of applications; for example, it has been used to characterize social networks [4], to help in the visualization of complex graphs [6], to determine the role of proteins in complex proteinomic networks [7], to analyze the static structure of large-scale software systems [8], to describe the architecture of randomly damaged uncorrelated networks [9]. and finally to identify good "spreaders" in epidemiological studies [10].

Centralized algorithms for the k-core decomposition already exist [5]. The distributed version of this problem is motivated by two possible scenarios: the graph could be so large to not fit into a single host, due to memory restrictions; or its description could be inherently distributed over a collection of hosts, making it inconvenient to move each portion to a central site. As examples of the former scenario, consider the Facebook social graph, with 800 million users (nodes) and more than 100 billion friend connections (edges) in January 2012; or the web crawls of Google and Yahoo, which stopped to announce the size of their indexes in 2005, when they both surpassed the 10 billion pages (nodes) milestone. As an example

A preliminary version of this work appeared as brief announcement in the Proceedings of ACM PODC, San Jose (CA), Jun. 2011 [1].
Alberto Montresor is with University of Trento, Italy. Email: `alberto.montresor@unitn.it`. *Francesco De Pellegrini and Daniele Miorandi are with CREATE-NET, Trento, Italy. Email:* `name.surname@create-net.org`

of the latter, consider "live" distributed systems, such as P2P overlays, that self-inspect their topologies (peers are nodes in the graph, and their network connections are edges): given that cores with larger k are known to be good spreaders [10], this information can be used at runtime to optimize the diffusion of messages in epidemic protocols [11].

Contribution: We consider two computational models:

- *One-to-one*, in which *one* computational unit is associated with *one* node in the graph and communication occurs only through direct messages between nodes connected through an edge. This is the model underlying Pregel [12], a distributed framework proposed by Google for processing large-scale graphs; but it can also be applied to the live distributed systems described above.
- *One-to-many*, in which *one* host stores *many* nodes together with their local and remote edges, while communication occurs through messages between hosts. This model is not supported by any existing distributed framework, but it well fits a cloud and/or grid computing scenario where the graph is already distributed over a collection of storage units, and the computation can be efficiently performed as close as possible to them.

The main contribution of this paper is a novel algorithm for distributed k-core distributed decomposition that can be applied to both models. Section 4 first proposes a version that can be used in the one-to-one scenario, and then shows how to migrate it to the one-to-many scenario, by efficiently placing a collection of nodes under the responsibility of a single host. We prove that the resulting algorithm completes the k-core decomposition in $O(N)$ rounds, with N being the number of nodes; more precisely, Section 5 shows an upper bound equal to $N-K+1$, with K being the number of nodes with minimal degree, and describes a class of graphs that approach the bound by requiring $N-K$ rounds. While such upper bounds are rather high, it turns out that real world graphs —such as the Slashdot comment network, the citation graph of Arxiv or the Gnutella overlay network— require a surprisingly low number of rounds, as demonstrated in the experiments described in Section 6.

2 Related Work

In this section we revise the relevant state-of-art in the field. We consider two research areas, models for distributed and parallel graph computation and centralized algorithms for computing the k-core decomposition.

2.1 Distributed Graph Computation Models

A number of related works for the distributed and/or parallel processing of graph structures has been presented in the literature. This field attracted considerable attention in the last few years, when the need of effectively performing computations over large-scale graphs became very relevant for a number of Web-related applications. An overview of the problems related to the processing of massively large graphs can be found in [13].

One popular framework for massively parallelizing computational tasks is MapReduce [14], introduced by Google in 2004 for the parallel processing of large data-sets. MapReduce is meant to allow developers to quickly and easily write applications that can process vast amounts of data on large clusters built using commodity hardware. MapReduce builds around a recursive structure, with the data-sets being split into independent chunks that can be processed in parallel. While Map-Reduce can be used for processing graphs, its structure is not optimized for such tasks. This is the reason that led Google researchers to develop another framework, called Pregel [12], optimized for mining graphs data.

Pregel is largely inspired by the Bulk Synchronous Parallel model (BSP) [15], [16], [17]. The computation in Pregel consists of a sequence of iterations, called supersteps, during which the framework runs a user-defined function on each vertex. In this function, a node receives messages from neighbor nodes sent during the previous superstep, modifies its local state and sends messages to its neighbor nodes, to be received in the next superstep. Barrier synchronization is used, so that each superstep is separated from the next one. Individual nodes may leave the computation when they have reached the convergence to their final state. Although they can run in an asynchronous environment, the algorithms described in this paper can be directly translated in the Pregel model.

In [18], the authors propose DisNet, a master-worker framework for parallel computation over large graphs. In the paper, the authors present the computation of betweenness centrality as a use case. Similarly to Pregel, DisNet is built around a vertex-centric approach to parallelize the computational process. The main difference with Pregel lies in the way communication with workers take place. The solution proposed by DisNet is able to achieve higher performance at the cost of lower flexibility.

In [19] a programming model (called 'Signal/Collect') for synchronous and asynchronous graph algorithms is presented. The work is motivated by some issues arising in the Semantic Web domain, relative to the processing of large graphs of RDF triplets. Signal/Collect is also a vertex-centric parallel model. The synchronous case is similar to computations in Pregel. A use case based on the computation of PageRank is considered.

A distributed framework for large graphs processing is presented in [20]. Distributed computations are organized into a hierarchy and coordinated by appropriate synchronizers. This framework is vertex-centric as well; yet, coordination is achieved by means of asynchronous

messages.

In [21] deterministic parallel algorithms for solving a number of graph computation problems (list ranking, Euler tour, connected components, spanning forests, etc.) over BSP and coarse grained multicomputer are presented.

In [22] the authors evaluate the performance of three platform models (relational, data-parallel and special-purpose in-memory) for computing a number of metrics for very large-scale graphs (including, PageRank, Strongly Connected Components and Approximate Shortest Paths). Their results show that, for metrics like PageRank, data-parallel models present very good performance levels.

2.2 k-Core Decomposition

The de facto standard algorithm for computing k-core decomposition is the one originally proposed by Batagelj and Zaveršnik (BZ) [5]. Their algorithm is based on the recursive deletion of vertexes (and edges incident to them) of degree less than k. The algorithm makes use of bin-sort, and can run in $O(\max(m, n))$, which equals $O(m)$ for connected networks.

BZ algorithm requires random access to the whole graph, which should therefore be kept in the main memory for the sake of performance. In [23] the authors address the case in which the graph cannot – due to size constraints – be kept in the main memory, but has to be accessed through a (slow) external memory. They propose a modification of BZ algorithm that requires $O(k_{max})$ scans of the graph, where k_{max} is the largest coreness index of the graph.

The BZ algorithm does not lend itself to a distributed implementation. When dealing with extremely large-scale graphs, which can neither be held in the main memory nor stored on a single external memory, novel approaches are necessary.

3 Notation and System Model

Let G be an undirected graph $G = (V, E)$ with $N = |V|$ nodes and $M = |E|$ edges. We denote $d_G(u)$ the degree of u in G, whereas $G(C) = (C, E|C)$ is the subgraph of G induced by subset of nodes C, where $E|C = \{(u, v) \in E : u, v \in C\}$. The concept of *$k$–core decomposition* [5] is condensed in the following two definitions:

Definition 1. *A subgraph $G(C)$ induced by the set $C \subseteq V$ is a* k-core *if and only if $\forall u \in C : d_{G(C)}(u) \geq k$, and $G(C)$ is maximal, i.e., for $\overline{C} \supset C$, there exists $v \in \overline{C}$ such that $d_{G(\overline{C})}(v) < k$.*

Maximality of k-cores guarantees uniqueness, i.e., there exists at most one k-core in G for every $k = 1, 2, \ldots$.

Definition 2. *A node in G is said to have* coreness k *if and only if it belongs to the k-core but not the $(k + 1)$-core.*

Let $k_G(u)$ denote the coreness of u in G. In what follows, G will be dropped by the notation when it is clear from the context which subgraph of G we are referring to.

The distributed system is composed by a collection of hosts H, whose overall goal is to compute the k-core decomposition of G. Each node u is associated to exactly one host $h(u) \in H$, that is responsible for computing the coreness of u. Each host x is thus responsible for a collection of nodes $V(x)$, defined as follows:

$$V(x) = \{u : h(u) = x\}.$$

Each host x has access to two functions, $\textit{neighbor}_V()$ and $\textit{neighbor}_H()$, that return a set of *neighbor nodes* and *neighbor hosts*, respectively. Host x may apply these functions to either itself or to the nodes under its responsibility; it cannot obtain information about neighbors of other hosts or nodes under the responsibility of other nodes. Formally, for each $u \in V$ and each $x \in H$, the functions are defined as follows:

$$\textit{neighbor}_V(u) = \{v : (u, v) \in E\}$$
$$\textit{neighbor}_V(x) = \{v : (u, v) \in E \wedge u \in V(x)\}$$
$$\textit{neighbor}_H(x) = \{y : (u, v) \in E \wedge u \in V(x) \wedge v \in V(y)\}$$

A special case occurs when the graph to be analyzed coincides with the distributed system, i.e. $H = V$. When this happens, the label u will be used to denote both the node and the host, and in general we will use the terms node and host interchangeably. Also, note that in this case $\textit{neighbor}_V(u) = \textit{neighbor}_H(u)$.

Hosts communicate through reliable channels. For the duration of the computation, we assume that hosts do not crash.

4 Algorithm

Our distributed algorithm is based on the property of *locality* of the k-core decomposition: due to the maximality of cores, the coreness of node u is the largest value k such that u has at least k neighbors that belong to a k-core or a larger core. The locality property writes:

Theorem 1 (Locality)**.** *$\forall u \in V : k(u) = k$ if and only if*

(i) *there exist a subset $V_k \subseteq \textit{neighbor}_V(u)$ such that $|V_k| = k$ and $\forall v \in V_k : k(v) \geq k$ and*

(ii) *there is no subset $V_{k+1} \subseteq \textit{neighbor}_V(u)$ such that $|V_{k+1}| = k + 1$ and $\forall v \in V_{k+1}$: $k(v) \geq k + 1$.*

Proof.

$\Rightarrow$) Since $k(u) = k$, $W_k \subseteq V$ exists such that $u \in W_k$ and $G(W_k)$ is a k-core, and there is no set $W_{k+1} \subseteq V$ such that $u \in W_{k+1}$ and $G(W_{k+1})$ is a $(k+1)$-core. Part (i) follows since $d_{G(W_k)}(u) \geq k$, so that at least k neighbors of u belong to k-core $G(W_k)$.
Part (ii) follows by contradiction: assume that $v_1, \ldots, v_{k+1}$ are $k + 1$ neighbors of u with coreness $k + 1$ or more. For $i = 1, \ldots, k + 1$, denote

$W_i \subseteq V$ such that $v_i \in W_i$ and $G(W_i)$ is a $(k+1)$-core. Consider set $U = \{u\} \cup \bigcup_{i=1}^{k+1} W_i$. Indeed, if $v \in U$, $d_{G(U)}(v) \geq k+1$: if $v = u$, indeed $d(u) \geq k+1$ by construction, whereas if $u \neq v \in W_i$, $d_{G(U)}(v) \geq d_{G(W_i)}(v) \geq k+1$. Hence, a $(k+1)$-core exists ($G(U)$ may well not be maximal) and it is the $(k+1)$-core for u. Contradiction.

$\Leftarrow$) For each node $v_i \in V_k$, $1 \leq i \leq k$, $k(v_i) \geq k$ there exists $W_i \subseteq V$ such that $G(W_i)$ is a k-core and $v_i \in W_i$. Consider set $U = \{u\} \cup \bigcup_{i=1}^{k} W_i$. With the same argument used above we see that for each $v \in U$, $d_{G(U)}(v) \geq k$: if $v = u$, indeed $d(u) \geq k$ by construction, whereas if $u \neq v \in W_i$, $d_{G(U)}(v) \geq d_{G(W_i)}(v) \geq k$. Again, this proves that $k(u) \geq k$.

Suppose now that $k(u) = k' \geq k+1$, by contradiction. This means that there is subset $W \subseteq V$ such that $G(W)$ is a k'-core containing u, i.e. u has at least $k' \geq k+1$ neighbors in $d_{G(W)}(u)$, each of them thus with coreness $k' \geq k+1$; but this contradicts our hypothesis (ii). We can conclude that $k(u) = k$. □

The locality property tells us that the information about the coreness of the neighbors of a node is sufficient to compute its own coreness. Based on this idea, our algorithm works as follows: each node produces an *estimate* of its own coreness and communicates it to its neighbors; at the same time, it receives estimates from its neighbors and use them to recompute its own estimate; in the case of a change, the new value is sent to the neighbors and the process goes on until convergence.

This outline must be formalized in a real algorithm; we do it twice, for both the one-to-one and the one-to-many scenarios. We conclude the section with a few ideas about termination detection, that are valid for both versions.

4.1 One host, one node

Each node u maintains the following variables:

- *core* is an integer that represents the local estimate of the coreness of u; it is initialized with the local degree.
- *est*[ ] is an integer array containing one element for each neighbor; *est*$[v]$ represents the most up-to-date estimate of the coreness of v known by u. In the absence of more precise information, all its entries are initialized to $+\infty$.
- *changed* is a Boolean flag set to true if *core* has been recently modified; initially set to false.

The protocol is described in Algorithm 1. Each node u starts by broadcasting a message $\langle u, d(u)\rangle$ containing its identifier and degree to all its neighbors. Whenever u receives a message $\langle v, k\rangle$ such that $k <$ *est*$[v]$, the entry *est*$[v]$ is updated with the new value. A new temporary estimate t is computed by function computeIndex() in Algorithm 2. If t is smaller than the previously known value of *core*, *core* is modified and the *changed* flag is set to true. Function computeIndex() returns the largest value i such that there are at least i entries equal or larger than i in *est*, computed as follows: the first three loops compute how many nodes have estimate i or more, $1 \leq i \leq k$, and store this value in array *count*. The **while** loop searches the largest value i such that *count*$[i] \geq i$, starting from k and going down to 1.

The protocol execution is divided in periodic rounds: every δ time units, variable *changed* is checked; if the local estimate has been modified, the new value is sent to all the neighbors and *changed* is set back to false. This periodic behavior is used to avoid flooding the system with a flow of estimate messages that are immediately superseded by the following ones.

It is worth remarking that during the execution, variable *core* at node u (i) is always larger or equal than the real coreness value of u, and (ii) cannot increase upon the receipt of an update message. Informally, these two observations are the basis of the correctness proof contained in Section 5.

Algorithm 1: Distributed algorithm to compute the k-core decomposition; routine executed by node u.

```
on initialization do
    changed ← false;
    core ← d(u);
    foreach v ∈ neighbor_V(u) do est[v] ← ∞;
    send ⟨u, core⟩ to neighbor_V(u);

on receive ⟨v, k⟩ do
    if k < est[v] then
        est[v] ← k;
        t ← computeIndex(est, u, core);
        if t < core then
            core ← t;
            changed ← true;

repeat every δ time units (round duration)
    if changed then
        send ⟨u, core⟩ to neighbor_V(u);
        changed ← false;
```

Algorithm 2: **int** computeIndex(**int**[ ] *est*, **int** u, k)

```
for i = 1 to k do count[i] ← 0;
foreach v ∈ neighbor_V(u) do
    j ← min(k, est[v]);
    count[j] = count[j] + 1;
for i = k downto 2 do
    count[i − 1] ← count[i − 1] + count[i];
i ← k;
while i > 1 and count[i] < i do
    i ← i − 1;
return i;
```

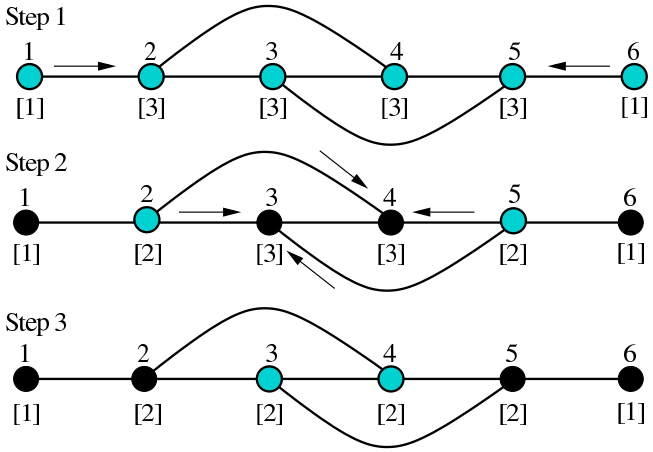

Fig. 2: A simple example describes the run of the algorithm: *core*$[u]$ value at node u is reported in squared brackets, a node v marked blue is active, i.e., *changed*$[v]$ is true, whereas arrows along edges denote where *est*$[v]$ triggers *core*$[u]$ to change at the receiving node u.

The pseudo-code algorithm 1 can be easily adopted to existing frameworks like Map-Reduce or Pregel. In Pregel, for example, all the new estimates received at the beginning of a superstep are stored in *est*, and a single invocation to computeIndex is performed. If the local *k* variable is decreased, its value is sent to all its neighbors; otherwise, the node leaves the computation, to be awakened again when new estimates are received.

4.1.1 Example

We describe here a run of the algorithm on the sample graph reported in Fig. 2. At the first round, all nodes v have *core* $= d(v)$; nodes $2, 3, 4$ and 5 send the same value *core* $= 3$ to their neighbors: these messages do not cause any change in the estimates of the coreness of receiving nodes. However, in the same round, nodes 1 and 6 notify their *core* $= 1$ value to nodes 2 and 5, respectively: as a consequence, node 2 and 5 update their estimates to *core* $= 2$. Thus, in the second round another message exchange occurs, since nodes 2 and 5 notify their neighbors that their local estimate changed, i.e., they send *core* $= 2$ to nodes $1, 3, 4$ and $3, 4, 6$, respectively. This causes an update *core* $= 2$ at nodes 3 and 4, which have to send out another update *core* $= 2$ to nodes 2 and 4 and nodes 3 and 5, respectively, in the third round. However, no local estimate changes from now on, which in turns means that the algorithm converged. Finally, *core* $= 2$ for $v = 2, 3, 4, 5$ and *core* $= 1$ for $v = 1, 6$.

4.1.2 Optimization

Depending on the communication medium available, some optimizations are possible. For example, if a broadcast medium is used (like in a wireless network) and the neighbors are all in the broadcast range, the **send to** primitive can be actually implemented through a broadcast. If the **send to** primitive is implemented through point-to-point send operations, a simple optimization is the following: message updates $\langle u, \mathit{core}\rangle$ are sent to a node v if and only if *core* $<$ *est*$[v]$; in other words, it is sent only if a node u knows that the new local estimate *core* can potentially lower the coreness estimation at node u; otherwise, it is skipped. In our experiments, described in Section 6, this optimization has shown to be able to reduce the number of exchanged messages by approximately 50%.

4.2 One host, multiple nodes

The algorithm described in the previous section can be easily generalized to the case where a host x is responsible for a collection of nodes $V(x)$: x runs the algorithm on behalf of its nodes, storing the estimates for all of them and sending messages to the hosts that are responsible for their neighbors. Described in this way, the new version of the algorithm looks trivial; an interesting optimization is possible, though. Whenever a host receives a message for a node $u \in V(x)$, it "internally emulates" the protocol: the estimates received from outside can generate new estimates for some of the nodes in $V(x)$; in turn, these can generate other estimates, again in $V(x)$; and so on, until no new internal estimate is generated and the nodes in $V(x)$ become quiescent. At that point, all the new estimates that have been produced by this process are sent to the neighboring hosts, where they can ignite these cascading changes all over again.

Each node x maintains the following variables:

- *est*$[\,]$ is an integer array containing one element for each node in $V(x) \cup \mathit{neighbor}_V(x)$; *est*$[v]$ represents the most up-to-date estimate of the coreness of v known by x. Given that elements of $\mathit{neighbor}_V(x)$ could belong to $V(x)$ (i.e. some of the neighbors nodes of nodes in $V(x)$ could be under the responsibility of x), we store all their estimates in *est*$[\,]$ instead of having a separate array *core*$[\,]$ for just the nodes in $V(x)$.
- *changed*$[\,]$ is a Boolean array containing one element for each node in $V(x)$; *changed*$[v]$ is **true** if and only if the estimate of v has changed since the last broadcast.

The protocol is described in Algorithm 3. At the beginning, all nodes $v \in V(X)$ are initialized to *est*$[v] = d(v)$; in the absence of more precise information, all other entries are initialized to $+\infty$. Function improveEstimate() is run to compute the best estimates x can obtain with the local information; then, all the current estimates for the nodes in $V(x)$ are sent to all nodes.

Whenever a message is received, the array *est* is updated based on the content of the message; function improveEstimate() is called to take into account the new information that x may have received.

Algorithm 3: Distributed algorithm to compute the k-core decomposition, executed by host x.

```
on initialization do
    foreach v ∈ neighbor_V(x) do est[v] ← +∞;
    foreach u ∈ V(x) do est[u] ← d(u);
    improveEstimate(est);
    S ← {(u, est[u]) : u ∈ V(x)};
    send ⟨S⟩ to neighbor_H(x);

on receive ⟨S⟩ do
    foreach (v, k) ∈ S do
        if k < est[v] then est[v] ← k;
    improveEstimate(est);

repeat every δ time units (round duration)
    S ← ∅;
    foreach u ∈ V(x) do
        if changed[u] then
            S ← S ∪ {(u, est[u])};
            changed[u] ← false;
    if S ≠ ∅ then
        send ⟨S⟩ to neighbor_H(x);
```

Periodically, node x computes the set S of all pairs $(v, \mathit{est}[v])$ such that (i) x is responsible for v and (ii) $\mathit{est}[v]$ has changed since the last broadcast. If S is not empty, it is sent to all nodes in the system. Alternatively, barrier synchronization could be adopted, starting a new round whenever all the messages sent during the previous round from all hosts have been received.

Function improveEstimate() (Algorithm 4) performs the local emulation of our algorithm. In the body of the **while** loop, x tries and improve the estimates by calling computeIndex() on each of the nodes it is responsible for. If any of the estimates is changed, variable *again* is set to **true** and the loop is executed another time, because a variation in the estimate of some node may lead to changes in the estimate of other nodes.

4.2.1 Communication policy

There are two policies for disseminating the estimate updates. The above version of the algorithm assumes that a broadcast medium is available. This means that a single message containing all the updates received since the last round could be created and sent to all.

Alternatively, we could adopt a communication system based on point-to-point send operations. In this case, it does not make sense to send all updates to all nodes, because each update is interesting only for a subset of nodes. So, for each host $y \in H$, we create a message containing only those updates that could be interesting for y. The modification to be applied to Algorithm 3 are contained in Algorithm 5.

4.2.2 Node-hosts assignment policy

The graph to be analyzed could be "naturally" split among different hosts, or nodes could be assigned to hosts based on a well-defined policy. It is difficult to identify efficient heuristics to perform the assignment in the general case. In this paper, we adopt a very simple policy: assuming that nodes identifiers are integers in the range $[0 \ldots n-1]$ and hosts identifiers are integers in the range $[0 \ldots |H|-1]$, each node u is assigned to host $(u \bmod |H|)$.

Algorithm 4: improveEstimate(**int**[] *est*)

```
again ← true;
while again do
    again ← false;
    foreach u ∈ V(x) do
        k ← computeIndex(est, u, est[u]);
        if k < est[u] then
            est[u] ← k;
            changed[u] ← true;
            again ← true;
```

Algorithm 5: Code to be substituted in Algorithm 3

```
repeat every δ time units (round duration)
    foreach y ∈ neighbor_H(x) do
        S ← { (u, est[u]) : u ∈ V(x) ∧
                (u, v) ∈ E ∧ v ∈ V(y) };
        if S ≠ ∅ then
            send ⟨S⟩ to y;
    foreach u ∈ V(x) do
        changed[u] ← false;
```

4.3 Termination

To complete both algorithms, we need to discuss a mechanism to detect when convergence to the correct coreness values has been reached. There are plentiful of alternatives:

- *Centralized approach*: each host may inform a centralized server whenever no new estimate is generated during a round; when all hosts are in this state, messages stop flowing and the protocol can be terminated. This is particularly suited for the "one node, multiple hosts" scenario, where it corresponds to a master-slaves approach.
- *Decentralized approach*: epidemic protocols for aggregation [24] enable the decentralized computation of global properties in $O(\log |H|)$ rounds. These protocols could be used to compute the last round in which any of the hosts has generated a new estimate (namely, the execution time): when this value has not been updated for a while, hosts may detect the termination of the protocol and start using the computed coreness.
- *Barrier synchronization*: if barrier synchronization is adopted in the one-to-many version, nodes may

decide to stop when none of them has produced any new estimate update during the previous round.
- *Fixed number of rounds*: as shown in Section 6, most of real-world graphs can be completed in a very small number of rounds (few tens); furthermore, after very few rounds the estimation error is extremely low. Results show that if one tolerates a small error, i.e., on the order of a few units, an approximate k-core decomposition can be generated running the protocol for a fixed number of rounds, i.e., 15–20 rounds.

5 CORRECTNESS PROOFS

We now prove that our algorithms are correct and eventually terminate. While we focus on the one-to-one scenario, the results can be easily extended to the one-to-many case.

5.1 Safety and liveness

Theorem 2 (Safety)**.** *During the execution, variable core at each node u is always larger or equal than $k(u)$.*

Proof. By contradiction, assume there exists node u_1 such that $core(u_1) < k(u_1)$. By Theorem 1, there is a set $V_1 \subseteq V$ such that $|V_1| = k(u_1)$ and for each $v \in V_1$: $k(v) \geq k(u_1)$. In order to set $core(u_1)$ smaller than $k(u_1)$, u_1 must have received a message containing an estimate smaller than $k(u_1)$ from at least one of the nodes in V_1. Notice this cannot happen at time $t = 0$, since initialization forces $d_G(v) = core(v) \geq k(v)$, $\forall v \in V$. Thus, let us consider any subsequent round at time $t = 1, 2, \ldots$ Formally, u_1 must have received a message $\langle u_2, core(u_2)\rangle$ from u_2 at time t_2, such that $u_2 \in V_1$ and $core(u_2) < k(u_1)$. Given that $k(u_1) \leq k(u_2)$ (because $u_2 \in V_1$), we conclude that $core(u_2) < k(u_2)$: in other words, we found another node whose estimate is smaller than its coreness. By applying Theorem 1 again, we derive that u_2 received a message $\langle u_2, core(u_3)\rangle$ from u_3 at time $t_3 < t_2$, such that $core(u_3) < k(u_2) \leq k(u_3)$. This reasoning leads to an infinite sequence of nodes $u_1, u_2, u_3, \ldots$ such that $core(u_i) < k(u_i)$ and u_i received a message from u_{i+1} at time t_i, with $t_i > t_{i+1}$. Given the finite number of nodes, this sequence contains a cycle $u_i, u_{i+1}, \ldots, u_j = u_i$; but this means $t_i > t_{i+1} > \ldots > t_j = t_i$, a contradiction. □

Theorem 3 (Liveness)**.** *There is a time after which the variable core at each node u is always equal to $k(u)$.*

Proof. By Theorem 2 variable $core(u)$ cannot be smaller than $k(u)$; by construction, variable *core* cannot grow. So, if we prove that the estimate will eventually become equal to the actual coreness, we have proven the theorem. The proof is by induction on the coreness $k(u)$ of node $u \in V$.

- $k(u) = 0$: in this case, u is isolated. Its degree, used to initialize $core(u)$, is equal to its coreness and the protocol terminates at the very beginning for node u.
- $k(u) = 1$: assume, by contradiction, that $k(u) = 1$ but $core(v)$ is always at least 2. Let D be the set of all nodes v such that $core(v)$ is always at least 2; clearly, $u \in D$.E By Algorithm 1 and 2, each node $v \in D$ must have two neighbors belonging to D. Therefore, for all $v \in D$, $d_{G(D)}(v) \geq 2$ and D is contained in a 2-core. Given that $u \in D$, u belongs to a 2-core, a contradiction.
- *Induction step*: by contradiction, suppose there is a node u_1 such that $k(u_1) = k > 1$ and $core(u_1) > k$ forever. By Theorem 1, there are $f \geq k$ neighbors of u with coreness greater than or equal to k, and $d(u) - f$ neighbors of u whose coreness is smaller than k.
  If $f = k$, by the inductive assumption, u_1 will eventually receive $d(u_1) - k$ estimates smaller than k, while the other k estimates will always be larger or equal to k by Theorem 2. So, u_1 eventually sets $core(u_1)$ equal to k, a contradiction.
  If $f \geq k + 1$, let C be the set of nodes v such that $k(v) = k$ but $core(v) \geq k + 1$ forever. Because $f > k$, we know that every $v \in C$ has at least $f + 1$ neighbors, such that for each of these neighbors u, either $k(u) \geq k+1$ or $k(u) = k$ but $core(u) \geq k+1$ forever. Also, let D be the set of all nodes v such that $core(v)$ is at least $k+1$ forever and note that $c \subseteq D$. By Algorithm 1 and 2, for any $v \in D$, v must have at least $k + 1$ neighbors belonging to D. Therefore, for all $v \in D$, $d_{G(D)}(v) \geq k + 1$ and D is contained to a $k + 1$-core. Given that $C \subseteq D$, this means that nodes in C belong to a $k+1$-core, which contradicts the definition of C. □

5.2 Time complexity

We proved that our algorithms eventually converge to the correct coreness; we now discuss upper bounds on the *execution time*, defined as the total number of rounds during which at least one node broadcasts its new estimate (when no new estimates are produced, the algorithm stops and the correct values have been obtained).

For this purpose, we assume that rounds are synchronous; during one round, each node receives all messages addressed to it in the previous round (if any), computes a new coreness estimate and broadcasts a message to all its neighbors if the estimate has changed with respect to the previous round. At round 1, each node broadcasts its current estimate (equal to its degree) to all its neighbors. To simplify the analysis, no further optimizations are applied. In the final round, messages are sent but they do not cause any variation in the estimates, so the protocol terminates.

The first observation is that after the first round, in any subsequent round before the final one at least one node must change its own estimate, reducing it by at least 1. This brings to the following theorem:

Theorem 4. *Given a graph $G = (V, E)$, the execution time is bounded by* $1 + \sum_{u \in V} [d(u) - k(u)]$.

Proof. The quantity $[d(u) - k(u)]$ represents the "initial error" at node u, i.e. the difference between the initial estimate (the degree) and the actual coreness of u. In the worst case, at most one message is broadcast per round, and each broadcast reduces the error by one unit, apart from the last one which has no effect. Thus the execution time is bounded by the sum of all initial errors plus one. □

While the previous bound is based on the knowledge of the actual coreness index of nodes, we can define a bound on the execution time that depends only on the graph size:

Theorem 5. *The execution time is not larger than N.*

Proof.

Given a run of the algorithm, denote

$$A(r) = \{u \in V | \, \textit{core}(u) = k(u) \text{ at round } r \, \}.$$

We make the following observations:

i) $A(1) \neq \emptyset$: each node u with minimum degree δ is included in $A(1)$. In fact, u is such that $k(u) = \delta$, otherwise there would be a node $v \in \textit{neighbor}_V(u)$ with a degree less than δ, which is impossible. Given that $\textit{core}(u) = \delta$ at round 1 by initialization, u belongs to $A(1)$.

ii) If $u \in A(r)$, then u does not send any message for all remaining rounds $r + 2, r + 3, \ldots$.

iii) $A(r) \subseteq A(r + 1)$ $\forall r$.

It is easy to see that the statement is true for $N \leq 2$, so that we will consider only $N \geq 3$ in the rest of the proof.

We denote by T the smallest round index at which $A(T) = V$. By definition, the execution time equals $T+1$.[1]

Denote $m(r) = \min\{k(u) : u \notin A(r)\}$, i.e., the minimal coreness of a node that did not yet attain the correct value at round r. Also, denote $M(r) = \{v : k(v) = m(r), v \notin A(r)\}$, the set of all such nodes.

Assume $A(r) \neq V$ so that $M(r) \neq \emptyset$: at round $r + 1$ there must exist $v \in M(r)$ such that $v \in A(r + 1)$, i.e., v attains the correct value at round $r + 1$ and thus notifies such value at round $r + 2$ to its neighbors. To see this, assume by contradiction that no node u in $M(r)$ attains the correct value at round $r + 1$, i.e., $M(r) = M(r + 1)$: this means that all nodes in $M(r)$ have $m(r)+1$ neighbors, $v_1, \ldots, v_{m(r)+1}$, such that $\textit{core}(v_i) \geq m(r) + 1$ at round $r + 1$ for $i = 1, \ldots, m(r) + 1$. We claim that for all such nodes $\textit{core}(v_i) \geq m(r)+1$ at successive rounds $r+2, r+3, \ldots$.

There are three possible cases for the v_is:

1) $k(v_i) \geq m(r)+1$: due to Theorem 2 such nodes can only notify $\textit{core}(v_i) \geq m(r) + 1$ to node u;
2) $k(v_i) < m(r)$: from the definition of $m(r)$, $v_i \in A(r)$, and they never notify such value at rounds $r + 2, r + 3, \ldots$ due to ii);
3) $k(v) = m(r)$: either v_is belong to $A(r)$, and they will never notify such value at $r + 2$ due to ii), or they belong to $M(r) = M(r + 1)$, so that if any value is notified by such nodes, it will be $\textit{core}(v_i) \geq m(r) + 1$ at round $r + 2$.

Hence, all nodes in $v_1, \ldots, v_{m(r)+1}$ will still have $\textit{core}(v_i) \geq m(r) + 1$ at round $r + 2$, so that $M(r + 2) = M(r + 1) = M(r)$. We can iterate the same argument at rounds $r + 3, r + 4, \ldots$, so that for all nodes in $M(r)$, the correct estimate $m(r)$ will never be attained, contradicting Theorem 3.

We hence proved that $D(r) = A(r) \setminus A(r - 1) \neq \emptyset$ for $r = 1, \ldots, T$, where we let $A(0) = \emptyset$ for the sake of notation and $A(1) \neq \emptyset$ because of i). Also, it is easy to see that $V = A(T) = \cup_{r=1}^{T} D(r)$ and $D(r) \cap D(s) = \emptyset$ for $r \neq s$. Thus,

$$N = |\bigcup_{r=1}^{T} D(r)| = \sum_{r=1}^{T} |D(r)| \geq T$$

The tighter bound $T \leq N - 1$ is obtained by contradiction. Consider round $N - 2$ and assume $T > N - 1$. Using the same arguments as above, $|A(N-2)| \geq N-2$.

Case $|A(N - 2)| = N - 1$: consider v such that $\{v\} = M(N - 2)$; due to ii) all neighbors of v would notify their true coreness at round $N - 1$ at the latest. Hence, v could calculate $\textit{core}(v) = k(v)$ at round $N - 1$, i.e., $v \in A(N-1)$. Finally, $A(N-1) = V$, so that $T = N-1$ against our assumption.

Case $|A(N - 2)| = N - 2$: denote v_1 and v_2 such that $v_1, v_2 \notin A(N - 2)$. Indeed, v_1 and v_2 must be neighbors: otherwise from ii) all their neighbors would notify their true core value by $N-1$ so that v_1 and v_2 would compute their own correct core value by $N - 1$. Observe that both v_1 and v_2 have neighbors in the set $A(N - 2)$, otherwise one of them would have degree 1, which is not possible since it would belong to $A(1)$.

Consider node v_1 for simplicity: at round $N - 1$, v_1 estimates $\textit{core}(v_1) \geq k(v_1)+1$. However, since only v_2 has a wrong estimation, from Thm. 1 there need to be $k(v_1)$ nodes $v_2 \neq u_i$, $i = 1, \ldots, u_{k(v_1)}$ such that $\textit{core}(u_i) = k(u_i) \geq k(v_1) + 1$. But,

- $\textit{core}(v_2) \geq k(v_1)+1$ because v_1 estimates $\textit{core}(v_1) \geq k(v_1) + 1$
- $\textit{core}(v_1) = k(v_1) + 1$ for only v_2 has a wrong estimation.

The same reasoning applies to v_2: $\textit{core}(v_1) \geq k(v_2) + 1$ and $\textit{core}(v_2) = k(v_2) + 1$ Thus, $\textit{core}(v_1) = k(v_1) + 1 \geq k(v_2) + 1$ and also $\textit{core}(v_2) = k(v_2) + 1 \geq k(v_1) + 1$ so that $\textit{core}(v_1) = \textit{core}(v_2)$. However, nodes in $A(N - 2)$ will not notify again their correct estimate from round N on and nodes v_1 and v_2 will perform the same estimate

1. This is due to the fact that, by our definition, the execution time includes also the last round, in which updates are sent but they have no further effect on the computed coreness.

they had at round $N-1$, i.e., $k(v_2)+1 = core(v_1) = core(v_2) = k(v_1)+1$. Thus, no message can be exchanged from round N on, while $core(v_i) \neq k(v_i)$ $i = 1, 2$. But, this contradicts the liveness property so that it must be $T \leq N-1$.

□

From the proof, we observe that the nodes of minimal degree attain the correct coreness at the first round. We can slightly refine the bound as:

Corollary 1. *Let K be the number of nodes with minimal degree in G. Then the execution time on G is not larger than $N-K+1$ rounds.*

We observe that the bound provided by Theorem 5 is tighter than that provided by Theorem 4 if and only if the initial average estimation error $\frac{1}{N}\sum_{u\in V}\big(d(u)-k(u)\big)$ is larger than $1-\frac{1}{N}$.

Some important questions are (i) how tight is the bound of Theorem 5, and (ii) is there any graph that actually requires N rounds to complete? Experimental results with real-life graphs show that the bound is far from being tight (graphs with millions of nodes converge in less than one hundred rounds, see Sec. 6). However, we managed to identify a class of graphs close to the bound, i.e., with execution time equal to $N-1$ rounds for $N \geq 5$. Assuming that nodes are numbered from 1 to N, the rules to build such graphs are:

- node N is connected to all nodes apart from node $N-3$;
- each node $i = 1 \dots N-2$ is connected with its successor $i+1$;
- node $N-3$ is also connected with node $N-1$.

Figure 3 shows the graph obtained by this scheme for $N = 12$. Graphically, it is convenient to represent node N as the *hub* of a polygon, where nodes are located at the corners. All nodes have degree 3, apart from the hub which has degree $N-2$ and node 1 which has degree 2. When starting our algorithm, node 1 acts as a *trigger*: it has the smallest degree and its broadcast causes node 2 to change its estimate to 2, which in turn will cause node 3 to change its estimate to 2, and so on until the estimate of node $N-4$ changes to 2. Note that node N has changed its estimate from $N-2$ to 3 after the first round, and has maintained this estimate so far. In the next next round, nodes $N-3$ and N change their estimate to 2; in the last round, node $N-1$ and $N-2$ change their estimate to 2 as well and the algorithm terminates. Given that during each round apart from the last two, at most one node has changed its estimate, the total number of rounds is exactly $N-1$ ($N-2$ plus the last round).

It is worth remarking that other simple structures one may think of as potential worst cases offer lower execution time. As an example, a linear chain of size N requires $\lceil N/2 \rceil$ rounds to converge.

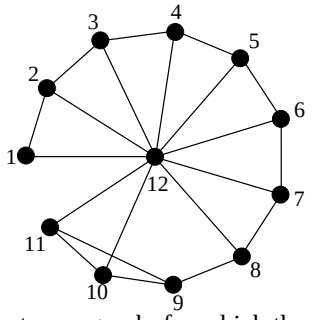

Fig. 3: The worst-case graph, for which the execution time is exactly $N-1$ rounds, $N = 12$.

One would expect that there should be a relation between diameter and execution time. This is true for example for linear chains of N, where $\lceil N/2 \rceil$ rounds are required. The smaller the diameter, the shorter should be the execution time. However, despite we noticed a beneficial effect of small diameters, this does not hold in general: in fact, the example of Figure 3 provides a case when the convergence time increases linearly with N but the diameter is 3, i.e., a constant regardless of N.

5.3 Message complexity

The maximum number of exchanged messages can be computed using a double counting argument: during the run of the algorithm, each node u can at most receive $d(v)-k(v)$ updates from each neighbor $v \in neighbor_V(u)$. If a node is connected (and thus exchange messages), its coreness is at least 1; thus, there are at most $d(u)+d(v)-2$ messages that can be exchanged over the undirected edge (u, v).

Theorem 6. *Given a graph $G = (V, E)$, the message complexity is bounded by $\left[\sum_{v\in V(G)} d^2(v)\right] - 2M$.*

Proof: It is simple to note that each node contributes $d(v)$ times a value of $d(v)-1$ to the sum; summing over all links,

$$\sum_{(u,v)\in E} [d(u)+d(v)-2] = \left[\sum_{v\in V(G)} d^2(v)\right] - 2\cdot M$$

□

Denoting the maximum degree in the graph with Δ, an upper bound to the above sum is $2M(\Delta-1)$, which is $O(\Delta \cdot M)$.

5.4 Special graphs

We can observe the convergence properties and the message complexity of the algorithm in some particular cases:

- The algorithm converges in exactly 1 round and $2M$ messages for every graph of given constant degree, or, more in general, for all graphs such that $d_v = k(v)$;

- The algorithm converges in $1+L$ rounds and with $3M$ messages for every tree with L levels; observe that the calculation of the k-shell of a tree is equivalent to the recursive removal of edges, so that the minimal required number of rounds coincides with the number of levels; in this case the algorithm convergence time scales linearly with L;
- For all grid-type of topologies, e.g., meshes, triangular grids or where all nodes have same degree apart from border nodes, the convergence time is dictated by the maximum distance of an inner node from the border. For instance, in the case of a mesh with p^2 nodes, the number of rounds is p for p even and $p+1$ for p odd: this is also the number of hops that are required for the information on the degree of corner nodes to reach the opposite side of the grid. In this example the convergence time scales linearly with the diameter, i.e., $2(p-1)$.

6 EXPERIMENTAL EVALUATION

This section reports experimental results, both through a simulator and a real implementation. Simulations have been performed using Peersim [25], on both the one-to-one and the one-to-many versions of the algorithm, over a selection of graphs contained in the Stanford Large Network Dataset collection [2]. Undirected graphs have been transformed in directed graphs by considering both directions (i.e., two edges) for each link present in the original one. We deployed a real implementation of the one-to-many version of the algorithm in Amazon EC2 and compared it with the state-of-the-art BZ algorithm [5], using a social network model called Nearest Neighbor [26] to generate graphs of increasing size.

Unless otherwise stated, the results show the average over 50 experiments. Experiments differ in the (non-deterministic) order with which operations are performed at different nodes.

6.1 One-to-one version

For this version, the main results are summarized in Table 1, which is divided in two parts. On the left, the main features of each graph considered are reported: name, number of nodes, number of edges, diameter, maximum degree, to conclude with maximum and average coreness.

On the right, the table reports information about the performance of the one-to-one protocol, based on two figures of merit: execution time (measured in rounds, i.e., fixed-size time intervals during which each node has the opportunity to send one update message to all its neighbors) and total number of messages exchanged. In particular, t_{avg}, t_{min} and t_{max} represent the average, minimum and maximum execution time measured over 50 experiments. m_{avg} and m_{max} represent the average and maximum number of messages per node.

2. `http://snap.stanford.edu/data/`

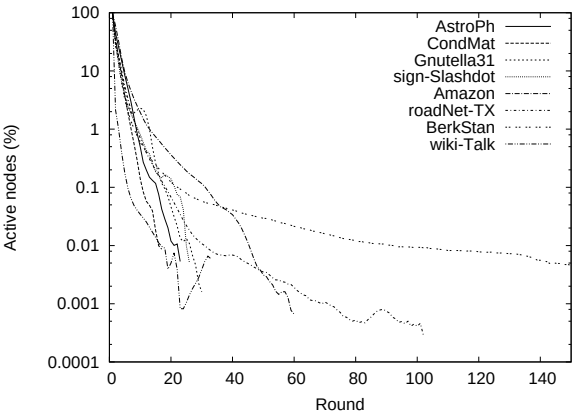

Fig. 4: One-to-one distributed k-core decomposition: percentage of active nodes versus time.

A few observations are in order. First of all, the execution time is of the order of few tens of rounds for most of the graphs, with only a couple of them requiring few hundreds of rounds (web-Berkstan, the web graph of Berkeley and Stanford, and RoadNet-TX, the road network of Texas). Compared with our theoretical upper bounds (number of nodes and total initial error), this suggests that our algorithm can be efficiently used in real-world settings.

The average and maximum number of messages per node is, in general, comparable to the average and maximum degree of nodes. Clearly, nodes with several thousands neighbors will be more overloaded than others.

In order to understand why web-Berkstan requires so many rounds to complete, we performed an in-depth analysis of the dynamic behavior of the proposed algorithms. In particular, we considered, for each core, the time taken for all nodes within it to reach the correct coreness value. Results are reported in Table 2. The first two columns report the problematic cores and their cardinality, respectively. The remaining columns represent the percentage of nodes whose estimate is still erroneous at round $t = 25, 50, \ldots, 300$; an empty column corresponds to 0%, i.e., the core computation has been completed. At first look, the 55-core seems particularly problematic, given that more than one half of it is still incorrect at round 25. But the 55-core completes before round 225, well before the 1-core that terminates after round 300. Delays in computing the 1-core may be associated to the high diameter of this particular graph, with "deep" pages very far away from the highest cores.

Additional information about the temporal behavior of the protocol can be obtained by analyzing the percentage of active nodes over time, shown in Figure 4 – where a node is defined active at a given round if it has sent at least one message during that round. It is possible to see that by

Name	$\|V\|$	$\|E\|$	$\oslash$	d_{max}	k_{max}	k_{avg}	t_{avg}	t_{min}	t_{max}	m_{avg}	m_{max}
1) CA-AstroPh	18 772	198 110	14	504	56	12.62	19.55	18	21	47.21	807.05
2) CA-CondMat	23 133	93 497	15	280	25	4.90	15.65	14	17	13.97	410.25
3) p2p-Gnutella31	62 590	147 895	11	95	6	2.52	27.45	25	30	9.30	131.25
4) soc-sign-Slashdot090221	82 145	500 485	11	2 553	54	6.22	25.10	24	26	29.32	3 192.40
5) soc-Slashdot0902	82 173	582 537	12	2 548	56	7.22	21.15	20	22	31.35	3 319.95
6) Amazon0601	403 399	2 443 412	21	2 752	10	7.22	55.65	53	59	24.91	2 900.30
7) web-BerkStan	685 235	6 649 474	669	84 230	201	11.11	306.15	294	322	29.04	86 293.20
8) roadNet-TX	1 379 922	1 921 664	1049	12	3	1.79	98.60	94	103	4.45	19.30
9) wiki-Talk	2 394 390	4 659 569	9	100 029	131	1.96	31.60	30	33	5.89	103 895.35

TABLE 1: Results obtained for one-to-one distributed k-core decomposition. Name of the data set, number of nodes, number of edges, diameter, maximum degree, maximum coreness, average coreness, average-minimum-maximum number of cycles to complete, average/maximum number of messages sent per node.

k	#	25	50	75	100	125	150	175	200	225	250	275	300
1	55 776	14.12%	10.26%	7.36%	4.97%	2.99%	1.65%	0.92%	0.56%	0.21%	0.13%	0.08%	0.02%
2	83 109	3.81%	1.35%	0.55%	0.27%	0.14%	0.06%						
3	67 910	1.42%	0.23%										
4	44 548	0.95%	0.07%										
5	68 728	0.46%	0.05%										
6	35 985	3.48%	1.01%	0.01%									
8	32 412	1.21%	0.46%	0.10%									
9	28 042	0.18%											
10	22 322	1.96%	0.64%										
15	6 842	0.99%											
55	2 548	50.78%	43.84%	36.77%	29.71%	22.76%	15.46%	8.40%	1.73%				

TABLE 2: Information about nodes that are delaying the completion of the protocol in the web-Berkstan graph. The first column k represents a coreness value; the second column # represents the size of the k-core, i.e., the number of nodes whose coreness is k; the column labeled $t = 25, 50, \ldots, 300$ represents the percentage of nodes in the given core that do not know the correct coreness value after t rounds. Empty cells corresponds to 0%. All other coreness are correctly computed at round 25.

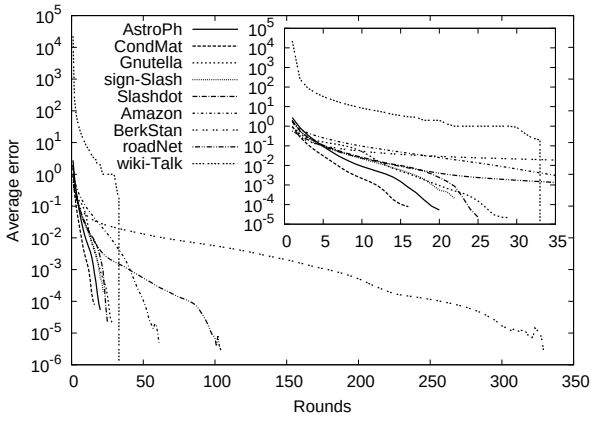

Fig. 5: One-to-one distributed k-core decomposition: evolution of evaluation error (averaged over all nodes and all repetitions) versus time. The smaller graph shows the details of the first rounds of the computation.

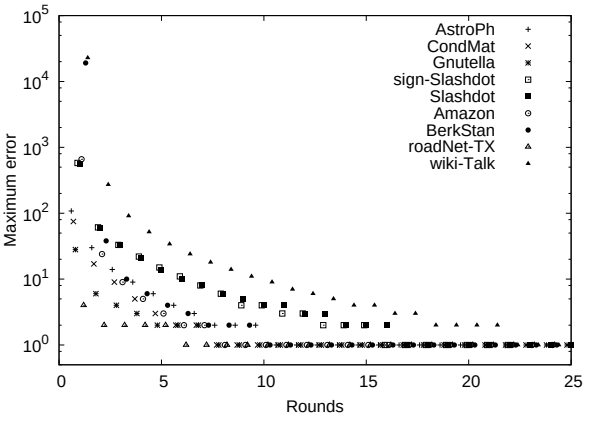

Fig. 6: One-to-one distributed k-core decomposition: evolution of maximum evaluation error (over all nodes and all repetitions) versus time.

round 20, less than 1% of the nodes are still active, with BerkStan and wiki-Talk showing residual activity even after one hundred rounds.

Another figure of merit is the temporal evolution of *error*, measured as the difference – at each node – between the current estimate of the coreness and its correct value. Figure 5 shows the average error for our experimental graphs. When the line stops, it means that the algorithm has reached the correct coreness estimate, so the error is zero. The "subfigure" zooms over the first rounds, to provide a closer look to the test cases that converge quickly. Figure 6 shows the maximum error (computed over all nodes, and over 50 experiments) for all our

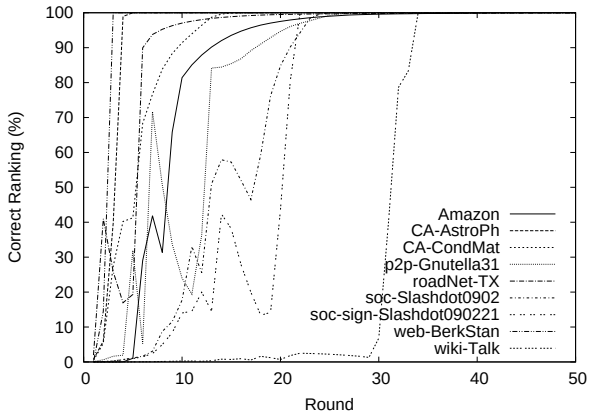

Fig. 7: One-to-one distributed k-core decomposition: percentage of correctly identified nodes in the top 1% ranking.

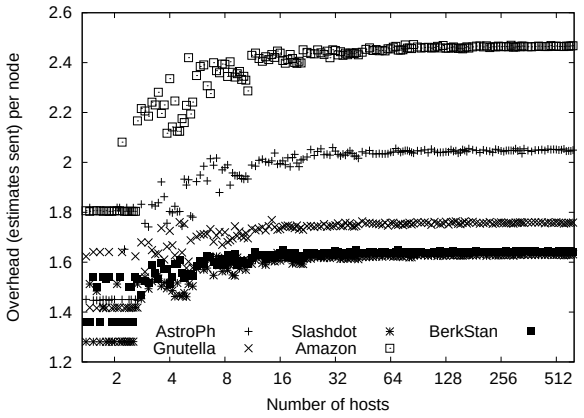

Fig. 8: One-to-many distributed k–core decomposition: overhead per node – with broadcast medium.

graphs (points have been slightly translated to improve visualization). As it can be seen, in all our experimental data sets, the maximum error is at most equal to 1 by cycle 22.

Error can be measured in yet another way – the capability of the protocol to identify the inner cores of the graph. This is relevant in applications where the goal is to identify the most influential nodes within a complex networked structure instead of actually computing the exact k–coreness of all nodes. To measure this, we ranked the nodes by decreasing shell index, and we selected the first 1% of nodes; we then counted, after each round, the number of nodes whose estimate is correctly included in the top 1%. This metric tells us that although index estimates may be approximate, the composition of the inner cores (the more important ones) has been correctly discovered. The results, averaged over 50 rounds, are shown in Figure 7. The outcome is consistent with our previous evaluations, with the top 1% nodes being correctly identified in few rounds. Two facts are worth being mentioned: in all our datasets, the correct identification of the inner cores happens suddenly; and wiki-Talk is again the most problematic one, with none of the members of the inner cores identified by round 30, to sudden reach 100% by round 32.

These error figures tell us that if the exact computation of coreness is not required (for example if coreness is used to optimize gossip protocols in a social network), the k-core decomposition algorithms proposed may be stopped after a predefined number of rounds, knowing that both the average and the maximum errors would be extremely low.

6.2 One-to-many version

The main reason for running the one-to-many version of the protocol is to compute the k-core decomposition over large graphs, that cannot fit into the memory of a single machine. Experimental results showed that the number of rounds needed to complete the protocol was equivalent to that of the one-to-one version. One of the key performance figures to be considered for the one-to-many version is the communication overhead generated by update messages exchanged among hosts. The overhead is computed as the average number of times a node generates a new estimate that has to be sent to another host.

Figures 8 and 9 show the overhead per node with a variable number of hosts, with and without a medium broadcast available, respectively. For visualization reasons, only some of the original data sets have been considered; but the results are similar for all of them. Twenty experiments were considered for this case. In the graph, the outcome of each experiment was represented as a point (slightly translated for the sake of visualization clarity).

When a broadcast medium is not available and point-to-point communication is used, the overhead increases with the number of hosts available, tending to stabilize to the levels of the one-to-one protocol (see the m_{avg} column of Table 1 – values are slightly higher given that the optimization of Section 4.1.2 cannot be applied in this case). When a broadcast medium is available, on the other hand, the efficiency is much higher. In this case, a single message is sent at each round, containing all the estimates that have changed since the previous one. Most of the nodes reach the correct estimate after few rounds and very few estimates are sent on their behalf after the first rounds; the effect is that the average number of estimates sent per node is extremely low, always smaller than 3, making the one-to-many algorithm particularly well-suited for clusters connected through fast local area networks.

6.3 Realistic deployment

To test our protocol in a real deployment, we implemented the one-to-many version with barrier synchronization and executed it using machines rented from Amazon EC2.

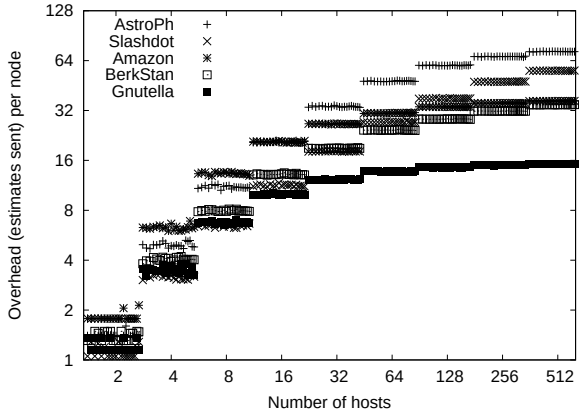

Fig. 9: One-to-many distributed k–core decomposition: overhead per node – without broadcast medium.

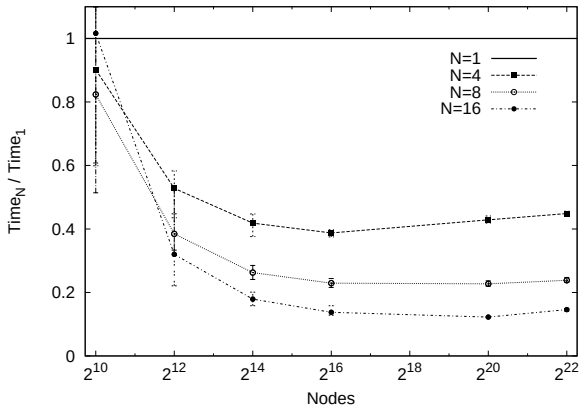

Fig. 10: One-to-many distributed k–core decomposition: computation time with respect to the centralized version.

Furthermore, we compared it with the state-of-the art BZ algorithm executed in the same environment.

The implementation of the BZ algorithm is based on the following assumptions: (i) it is possible to store in main memory a small, constant-size amount of information associated to each of the nodes in the graph; (ii) the entire graph can be randomly accessed through an external storage, but storing it entirely in memory is not possible. While these assumption are somehow realistic (the "High-Memory Quadruple Extra Large Instance" rented by Amazon has 68GB of main memory and 1.7TB of storage, which are sufficient for a few billions of nodes in memory and hundreds of billions of edges on storage), still they show that computing large-scale graphs cannot be done on commodity hardware.

We rented up to 16 "small instances" from Amazon and run the BZ algorithm on one of them, and then the one-to-many version on 4,8,16 machines.[3] We created a series of graphs using the Nearest Neighbor model [26], with size varying between 2^{10} and 2^{22} nodes, to illustrate the scalability of our approach. Figure 10 contains the results. Instead of showing the actual execution time, which behaves linearly in both the centralized and distributed versions, we decided to show the ratio between the execution time of the distributed implementation and the centralized one. As the reader can see, this ratio tends to converge when size increases. To avoid an unfair comparison between a centralized algorithm and a distributed one, we also computed the efficiency of our distributed implementation (measured as the ratio between speedup and number of processes), which is around 50% for all the configurations.

7 Conclusions

To the best of our knowledge, this paper is the first to propose distributed algorithms for the k-core decomposition of online and/or large graphs. While theoretical analysis provided us with fairly large upper bounds on the number of rounds required to complete the algorithm, which are strict for specific worst-case examples, experimental results have shown that for realistic graphs, our algorithms efficiently converge in few rounds.

We provided a distributed implementation of the one-to-many version and deployed on a fairly large network composed by Amazon EC2 nodes. These results are far from being definitive (many optimizations could be still be applied to both the centralized and distributed versions, e.g. exploring the use of threads in modern multi-threaded processors); yet, the results suggest that a distribute approach could enable the analysis of larger graphs in less time. The next logical step will be the implementation of these algorithms in Pregel-like frameworks [27], [28], [29]; unfortunately, at the time of writing, none of them demonstrated sufficient stability to perform extensive testing.

3. Amazon EC2 limits to 20 the number of machines that can be rented under the same account.

Acknowledgements

Alberto Montresor is supported by the project "Autonomous security", sponsored by the Italian Ministry of Research under the PRIN 2008 Programme, and by the EIT Activity "Europa" n. 12115.

References

[1] A. Montresor, F. De Pellegrini, and D. Miorandi, "Brief announcement: distributed k–core decomposition," in *Proc. of ACM PODC*, San Jose, Jun. 2011, pp. 207–208.

[2] L. Freeman, "Centrality in social networks: Conceptual clarification," *Social Networks*, no. 1, pp. 215–239, 1978.

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

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

[5] V. Batagelj and M. Zaveršnik, "Fast algorithms for determining (generalized) core groups in social networks," *Advances in Data Analysis and Classification*, vol. 5, pp. 129–145, 2011.

[6] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Advances in Neural Information Processing Systems 18*. MIT Press, 2006, pp. 41–50.
[7] G. Bader and C. Hogue, "Analyzing yeast protein–protein interaction data obtained from different sources," *Nature biotechnology*, vol. 20, no. 10, pp. 991–997, 2002.
[8] 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," *The Journal of Supercomputing*, vol. 53, pp. 352–369, 2010.
[9] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "k-core organization of complex networks," *Phys.Rev.Lett.*, vol. 96, 2006.
[10] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. Stanley, and H. Makse, "Identification of influential spreaders in complex networks," *Nature Physics*, vol. 6, pp. 888–893, Nov. 2010.
[11] J. A. Patel, I. Gupta, and N. Contractor, "JetStream: Achieving predictable gossip dissemination by leveraging social network principles," in *Proc. of the Int. Symposium on Network Computing and Applications (NCA'06)*. Cambridge, MA: IEEE, 2006, pp. 32–39.
[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of the 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*. New York, NY, USA: ACM, 2009.
[13] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, pp. 5–20, 2007.
[14] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.
[15] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, August 1990.
[16] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *J. Parallel Distrib. Comput.*, vol. 22, no. 2, pp. 251–267, Aug. 1994.
[17] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, "Bulk synchronous parallel computing–a paradigm for transportable software," in *Proc. of the 28th Hawaii Int. Conf. on System Sciences*, vol. 2, Jan. 1995, pp. 268–275 vol.2.
[18] R. Lichtenwalter and N. Chawla, "DisNet: A framework for distributed graph computation," in *Proc. of the Int. Conf. on Advances in Social Networks Analysis and Mining (ASONAM)*, Jul. 2011, pp. 263–270.
[19] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: Graph algorithms for the (semantic) web," in *The Semantic Web – ISWC 2010*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6496, pp. 764–780.
[20] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal, "A high-level framework for distributed processing of large-scale graphs," in *Distributed Computing and Networking*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6522, pp. 155–166.
[21] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. Song, "Efficient parallel graph algorithms for coarse grained multicomputers and bsp," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds. Springer, 1997, vol. 1256, pp. 390–400.
[22] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, and S. Gollapudi, "Of hammers and nails: An empirical comparison of three paradigms for processing large graphs," in *Proc. of ACM Int. Conf. on Web Search and Data Mining (WSDM)*, 2012.
[23] J. Cheng, Y. Ke, S. Chu, and M. Ozsu, "Efficient core decomposition in massive networks," in *Proc. of the 27th Int. Conf. on Data Engineering (ICDE'11)*, Apr. 2011, pp. 51–62.
[24] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 1, pp. 219–252, Aug. 2005.
[25] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conf. on Peer-to-Peer (P2P'09)*, Seattle, WA, Sep. 2009, pp. 99–100.
[26] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao, "Measurement-calibrated graph models for social network experiments," in *Proc. of the 19th Int. Conf. on the World wide web (WWW'10)*. Raleigh, NC, USA: ACM, 2010, pp. 861–870.
[27] S. Seo, E. J. Yoon, J.-H. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the mapreduce framework," in *Proc. of the 2nd Int. Conf. on Cloud Computing (CloudCom'10)*, Indianapolis, Indiana, USA, Nov., pp. 721–726.
[28] "GoldenOrb Web Site," 2012. [Online]. Available: http://www.goldenorbos.org/
[29] "Giraph Web Site," 2012. [Online]. Available: http://incubator.apache.org/giraph/

Alberto Montresor is Associate Professor at the University of Trento since 2005. Before that, he has been on the faculty of the University of Bologna (2002-2005). He has authored more than 60 papers on large-scale distributed systems, cloud computing and P2P networks. His main goal is to develop distributed protocols and systems that "survive": to large scale, to dynamism, to failures, to adversarial environments. He is Steering Committee Chair of the IEEE Conference on P2P Computing, Associate Editor of Springer Computing and has served as General Chair and Program Chair in several conferences (DOA, DAIS, SASO, P2P).

Francesco De Pellegrini received the Laurea degree in 2000 and the Ph.D. degree in 2004, both in Telecommunication Engineering, from the University of Padova. During year 2001/2002 he spent one year at Boston University as a visiting scholar. He is currently senior researcher at CREATE-NET. His research interests are location detection, multirate systems, routing, wireless mesh networks, VoIP, Ad Hoc and Delay Tolerant Networks. F. De Pellegrini has been TPC member of IEEE Infocom, WiOpt, SDN and served several other international networking conferences and journals as reviewer. Francesco serves in the Steering Programm Commitee of ICST Mobiquitous and Complex. Francesco was the Vice-chair for the first edition of ICST Robocomm.

Daniele Miorandi received a Ph.D. in Communications Engineering from University of Padova in 2005, and a Laurea degree (summa cum laude) in Communications Engineering from University of Padova in 2001. He joined CREATE-NET in Jan. 2005, where he is leading the iNSPIRE group (Networking and Security Solutions for Pervasive Computing Systems: Research & Experimentation). His research interests include bio-inspired approaches to networking and service provisioning in large-scale computing systems, modeling and performance evaluation of wireless networks, prototyping of wireless mesh solutions. Dr. Miorandi has co-authored more than 100 papers. He serves on the Steering Committee of various international events (WiOpt, Autonomics, ValueTools), for some of which he was a co-founder (Autonomics and ValueTools). He also serves on the TPC of leading conferences in the networking field, including, e.g., IEEE INFOCOM, IEEE ICC, IEEE Globecom. He is a member of IEEE, ACM and ICST.