Cracker: Crumbling Large Graphs Into Connected Components

Alessandro Lulli, Laura Ricci
University of Pisa, Italy
{lulli,ricci}@di.unipi.it

Emanuele Carlini, Patrizio Dazzi, Claudio Lucchese
I.S.T.I., CNR, Pisa, Italy
{emanuele.carlini,patrizio.dazzi,claudio.lucchese}@isti.cnr.it

***Abstract*—The problem of finding connected components in a graph is common to several applications dealing with graph analytics, such as social network analysis, web graph mining and image processing. The exponentially growing size of graphs requires the definition of appropriated computational models and algorithms for their processing on high throughput distributed architectures. In this paper we present CRACKER, an *efficient* iterative algorithm to detect connected components in large graphs. The strategy of CRACKER is to iteratively grow a spanning tree for each connected component of the graph. Nodes added to such trees are discarded from the computation in the subsequent iterations. We provide an extensive experimental evaluation considering a wide variety of synthetic and real-world graphs. The experimental evaluation shows that CRACKER consistently outperforms state-of-the-art approaches both in terms of total computation time and volume of messages exchanged.**

I. Introduction

The world is becoming more and more interconnected: humans, computers, various types of devices and the information they share are going to be *always on*, namely always active and reachable. Such *web of connections* fosters both stable and ephemeral forms of intercommunications, that are changing our lives and the way in which we interact one each others, far beyond our comprehension. In fact, these interactions consist in relations among interconnected entities, that are usually represented as graphs. Graphs, whose size is huge: consider, for instance, the Web graph (Common Crawl provides 3.5 billion pages with 128 billion hyperlinks [1]), Linked Open Data datasets (LOD2 project indexes for 5.7 billion triples/edges [2]) or Facebook and Twitter social networks (1.35 billion and 284 million monthly active users). As the size of graphs increases over time, it has become infeasible to rely on classic solutions for their processing, assuming that the computation is performed by a sequential or a shared-memory parallel machine. The amount of memory required to store graphs of such size is far beyond the capabilities provided by a single, even highly-performant, machine. As a consequence, it is of paramount importance to design efficient solutions, specifically designed for distributed computing architectures. In this context, approaches based on the High Level Distributed Programming [3], [4], [5], like the MapReduce paradigm are very popular, as they allow to exploit large clusters of commodity hardware. We focus on the problem of finding Connected Components (CC) in large graphs. The discovery of CCs is a fundamental building block for graph analytics in several contexts: image clustering [6], recommendation algorithms [7], community detection [8], analysis of structure and evolution of on-line social networks [9], information diffusion [10] and many others. In this work, we propose CRACKER, a novel algorithm for the discovery of connected components in large graphs, targeting distributed computing architectures. CRACKER is a distributed iterative algorithm, designed according to the MapReduce paradigm. The strategy of CRACKER is to iteratively build a spanning tree for each connected component of the graph. Nodes being added to the spanning trees are then discarded from the computation in the subsequent iterations. Upon termination, each resulting spanning tree drives the propagation of an *id* identifying the CC among the nodes that belong to the same CC. By reducing the number of nodes involved during each iterations, CRACKER drastically reduces both the total computation time and the volume of information exchanged. The CRACKER algorithm has been implemented over the Apache Spark framework [11] as it supports fast memory-based MapReduce computations. We already used Apache Spark in the past [12], as it proved to be an effective tool for graph analytics. To conduct a fair comparison, we implemented over the same framework the most relevant state-of-the-art algorithms. We evaluated CRACKER on many synthetic and real-word datasets. Results show that CRACKER out-performs competitor algorithms on every dataset. The best competitor of CRACKER is always slower within a factor of 9% and 75%. The main contributions of this work are the following:

- we propose a new distributed algorithm for the discovery of connected components in large graphs named CRACKER; the algorithm exploits a novel node pruning strategy that allows to dramatically reduce its computational cost;
- we present an extensive experimental evaluation, conducted both on synthetic and real-world datasets, where CRACKER is compared against state-of-the-art algorithms; the analysis includes both computational and communication costs and also scalability with respect to graphs size and complexity; experimental evidence shows that CRACKER singificantly improves over the state-of-the-art;
- to make our results reproducible we made publicly available the source code[1] and the graph datasets[2] used in this

[1] https://github.com/hpclab/cracker
[2] http://www.di.unipi.it/~lulli/project/cracker.htm

978-1-4673-7194-0/15/$31.00 ©2015 IEEE

work.

The remainder of this paper is organized as the following. Section II discusses state-of-the-art approaches to CC discovery. Section III illustrates the CRACKER algorithm, and its experimental evaluation is provided in Section IV. Section V draws some final conclusions.

II. Related Work

Finding connected components is a fundamental and well studied problem in graph analysis. When the graph fits in the memory of a single machine, a visit of the graph finds the connected components in linear time [13]. In the distributed setting, earlier solutions considered the PRAM model [14], [15]. However, even if it is possible to adapt these algorithms to current distributed frameworks, they often require impractical and not efficient implementations [16]. Many proposals dedicated to the problem of finding connected components have been designed for today's distributed frameworks that support vertex-centric computations. In the following we provide a characterization of state-of-the-art approaches on the basis of their *detection strategy*, *communication pattern* and *graph simplification*.

Regarding the detection strategy, we distinguish between *labelling* and *clustering* approaches. The former requires to associate each node with the id of the CC it belongs to, which usually is the smallest id of the CC's nodes, a.k.a., the id of the *seed node*. The latter requires in addition that at least one node in each CC knows the identifiers of all the other nodes in the same CC. Usually, *Labelling* is the most effective strategy in terms of information processed, but the CCs can be efficiently reconstructed by a post-processing step.

Communication patterns specify how nodes communicate to each other. A *static* pattern means that each vertex considers the same set of receivers at every iteration, which is usually the set of neighbours in the input graph. This pattern is straightforward to implement, however is characterized by a slow convergence to solution. More effective approaches employ a *dynamic* pattern, in which the set of recipient changes over time. This approach is usually more efficient, as the algorithm adds new connections among nodes of the graph with the aim of reducing the diameter of the CC and therefore speeding up convergence.

Graph simplification allows to perform *node exclusion*, namely the ability of excluding nodes from computation. Many state-of-the-art algorithms keep iterating the same vertex-centric computation on all nodes of the graph until convergence. In this way, a large number of vertices remains involved in the computation even if they do not provide useful information toward the final goal. For instance, a small CC could quit the computation as soon as it reached convergence without affecting the discovery of other connected components.

In 2009, Cohen [17] proposed an iterative MapReduce solution (which we refer to as ZONES) that groups connected vertices around the vertex with the smallest identifier. It employs a dynamic communication pattern based on the concept of zones. Initially, the algorithm constructs one zone for each vertex. During each iteration, each edge is tested whether it connects nodes from different zones. If this is the case, the lower order zone absorbs the higher order zone. When there are no zones to be merged, each zone is known to be a connected component. One of the drawbacks of this approach is that all edges are considered for computation during each iteration (no node exclusion), which results in slow convergence.

To speed-up the convergence, Seidl et al. [18] proposed an improved version of ZONES called CCMR. The rationale of CCMR is to add *shortcut* edges, to the original graph so to reduce the amount of iterations needed to spread the information about CCs. The CCMR algorithm, modifies the input graph during each iteration, until each connected component is transformed in a star-shaped sub-graph where all nodes are connected with the one having the smallest identifier. These improvements let CCMR to reduce the completion time with respects to ZONES.

Deelman et al. proposed an algorithm for connected components within the graph mining system PEGASUS [19]. They employ a static communication pattern, in which during every iteration each node communicates the smallest node identifier known so far to its neighbours. In turn, each node updates its knowledge with the received identifiers. The algorithm labels all nodes with the seed identifier in $O(d)$ MapReduce rounds, with d the diameter of the largest connected component. Similarly, Rastogi et al. [16] proposed HASH-TO-MIN, a vertex-centric algorithm parametrized by an hashing and a merging function that determine the information travelling along the graph. The HASH-TO-MIN algorithm is based on clusterings. It iterates, as PEGASUS, by propagating the smallest node identifier, but in addition it also communicates the whole set of known nodes so to create new connections among nodes being at more than one hop distance.

In a recent paper, Kardes et al. [20] proposed a MapReduce-based algorithm named CCF. The algorithm is structured on two phases. The first one is similar to HASH-TO-MIN, but introduces some improvements that reduce the computation cost in spite of the number of MapReduce rounds. The second phase is an optimisation that reduces the amount of duplicated messages. They successfully experimented their algorithm on a graph with over 6 billion nodes. As far as we know, CCF is the only approach that employs the concept of nodes exclusions, although limited to the seed nodes. However, the impact is negligible as the number of seed nodes is usually much lower than the nodes of a graph. By comparison, CRACKER employs node exclusion in an extensive manner by processing only the relevant vertices, while discarding vertices that have no useful information to share.

III. The CRACKER Algorithm

Let $G = (V, E)$ be an undirected graph where V is a set of vertices uniquely identified by values in $\mathbb{Z}$, and $E \subseteq V \times V$ is the set of edges. A connected component (CC) in G is a maximal subgraph $S = (V^S, E^S)$ such that for any two vertices $u, v \in V^S$ there is an undirected path in S connecting

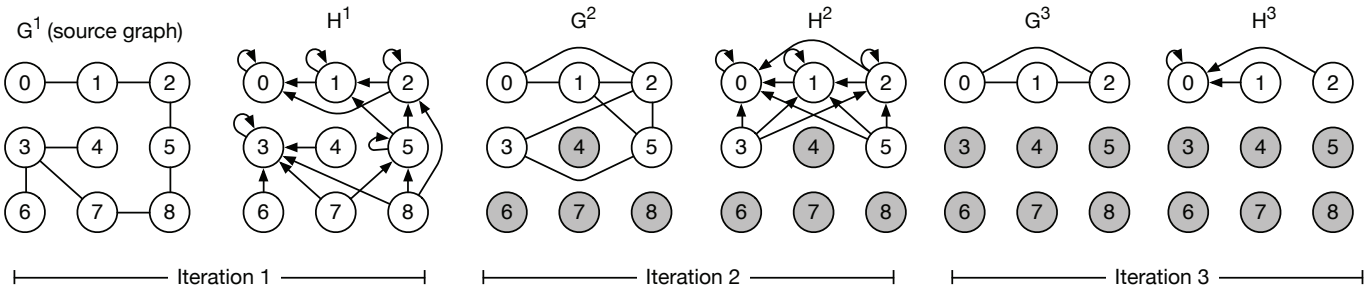


Fig. 1: CRACKER: example of seed identification. Gray nodes are excluded from the computation

them. We conform to the convention to identify each CC of the graph with the smallest vertex identifier in that component. The node having this identifier is named *seed* of the connected component.

The CRACKER algorithm achieves the identification of the CCs in two phases (outlined in Algorithm 1):

- *Seeds Identification*: CRACKER identifies the *seed* nodes of the graph, and iteratively builds a spanning tree for each CC, rooted in its *seed*; whenever a node is added to the spanning tree, it is excluded from computation in the subsequent iterations (see Alg. 1 lines 5–9).
- *Seeds Propagation*: propagates the *seed* id to all the nodes belonging to the CC by exploiting the spanning tree built in the previous phase (see Alg. 1 line 10).

In the following we describe in detail the two phases. The presentation is given adopting a vertex-centric computing metaphor: at each iteration the nodes of the input graph are processed independently and in parallel.

A. Seeds Identification

The basic idea of the seeds identification phase is to iteratively reduce the graph size by progressively pruning vertices until only one vertex for each connected components is left, i.e., its *seed*. For instance, an isolated vertex can be immediately excluded from further processing, since it does not provide useful information for the discovery of the other CCs. Similarly, as soon as a CC is identified, it can be excluded from computation since it does not impact on the other CCs in the graph. When a vertex is removed, it contributes to iteratively build a spanning tree that, at the end of the algorithm, will exist a spanning tree for each connected component.

Algorithm 1: The CRACKER algorithm

Input : an undirected graph $G = (V, E)$
Output: a graph where every node is labelled with the seed of its CC

1 u.Active = $True \;\; \forall u \in G$
2 $T \leftarrow (V, \emptyset)$
3 $t \leftarrow 1$
4 $G^t \leftarrow G$
5 **repeat**
6 $\quad H^t \leftarrow$ `Min_Selection_Step(`u`)` $\forall u \in G^t$
7 $\quad G^{t+1} \leftarrow$ `Pruning_Step(`u, T`)` $\forall u \in H^t$
8 $\quad t \leftarrow t + 1$
9 **until** $G^t = \emptyset$
10 $G^* \leftarrow$ `Seed_Propagation(`T`)`
11 **return** G^*

Algorithm 2: `Min_Selection_Step` (u)

Input : a node $u \in G$
1 $NN_{G^t}(u) = \{v : (u \leftrightarrow v) \in G^t\}$
2 $v_{min} = \min(NN_{G^t}(u) \cup \{u\})$
3 **forall the** $v \in NN_{G^t}(u) \cup \{u\}$ **do**
4 $\quad$ `AddEdge` $((v \rightarrow v_{min}), H^t)$
5 **end**

As shown in Algorithm 1, each node $u \in G$ is initially marked as *active*, meaning that at the start all nodes participate to the computation. The seed identification is an iterative algorithm made of two steps. The first step, named *Min Selection Step*, aims at defining, for each node, if it is a potential seed by at least one of its neighbours. Given the *undirected* input graph G^t, an intermediate *directed* graph H^t is generated linking nodes to potential seeds. Subsequently, the goal of the *Pruning Step* is to remove from H^t, and thereby *deactivate*, all the nodes that cannot be considered as *seeds*. A new *undirected* graph G^{t+1}, smaller that G^t, is generated and used to feed the next iteration. The nodes that are deactivated during the *Pruning Step* grow a forest of covering trees T of the CCs in the graph. Synchronisation occurs implicitly after each step, as the output graph is available only after that every node as completed its computation. Eventually, the seeds remain the only active nodes in the computation. Upon their deactivation, the *Seed Identification* terminates and the *Seed Propagation* is triggered. The *Seed Identification* phase is exemplified in Figures 1 and detailed below.

Min Selection Step. In this step, CRACKER identifies nodes that are guaranteed to not be *seed* of any connected component (see Algorithm 2). This decision is taken locally at each node, given that the only knowledge available locally to each node is its neighbourhood. A node can be considered as a *potential seed* if it is a local minimum in the neighbourhood of some node. If a node is not a local minimum in any neighbourhood then it cannot be a *seed*. To identify such nodes, the information about the local minimum in its neighbourhood, denoted

with v_{min}, is spread by each node to its neighbourhood.

Given the input graph G^t, a new directed graph H^t is built having the same nodes of G^t and new edges. The new edges are added as the following. For each node $u \in G^t$, the v_{min} is selected as the node with the minimum *id* from the set $NN_{G^t}(u) \cup \{u\}$ (see line 2 in Algorithm 2).

The node v_{min} is then notified to all the neighbours of u and to u itself. This communication is materialised as the addition of new directed edges $(v \rightarrow v_{min})$ for every $v \in \{NN_{G^t}(u) \cup u\}$ (see line 4 in Algorithm 2). After all nodes in G^t terminated the Min Selection Step, for each node $u \in H^t$ holds the following: (i) if u is not a v_{min} for any $NN_{G^t}(u)$, it has no incoming links; (ii) u has an outgoing link to its v_{min} and with the v_{min} of every node in $NN_{G^t}(u)$.

Let us consider the example in Fig. 1. The directed graph H^1, generated in the first iteration after the Min Selection Step, includes, for instance, a direct edge from node 8 to node 5, which is its v_{min} in the initial graph and from node 8 to nodes 3 and 2 which are the v_{min} of its neighbours. The knowledge of node 8 about G is improved with the knowledge of its neighbours.

Pruning Step. The main goal of this step is to exclude the nodes that are guaranteed to not be seeds and to add them to the propagation tree T. The pseudo code of the Pruning Step is presented in Algorithm 3.

At the start of the Pruning Step, each node recomputes v_{min} considering $NN_{H^t}(u)$. Then, for every node v in $NN_{H^t}(u)$ (except v_{min}), a new undirected edge linking v with v_{min} is added to the graph G^{t+1} (see line 5). This makes sure that the nodes in G^{t+1} are not disconnected in case u is deactivated and not included in the graph G^{t+1}. If a node u has not a self-link in H^t, it means that it is not considered as a local minimum by any other node, and therefore it is marked for exclusion from the computation (see line 9). The nodes marked for deactivation are inserted in the propagation tree T (see line 10). Finally, if a node is recognised as a seed, which happens if it is still active but it is the only node in its neighbourhood $NN_{G^{t+1}}(u)$, then it is marked for deactivation to complete the *Seed Identification* phase.

Algorithm 3: `Pruning_Step`(u, T)

```
   Input : a node u ∈ G and the propagation tree T
1  NN_{H^t}(u) = {v : (u → v) ∈ H^t}
2  v_min = min(NN_{H^t}(u))
3  if |NN_{H^t}(u)| > 1 then
4  |   forall the v ∈ NN_{H^t}(u) \ v_min do
5  |   |   AddEdge ((v ↔ v_min), G^{t+1})
6  |   end
7  end
8  if u ∉ NN_{H^t}(u) then
9  |   u.Active = False
10 |   AddEdge ((v_min → u), T))
11 end
12 if IsSeed (u) then
13 |   u.Active = False
14 end
```

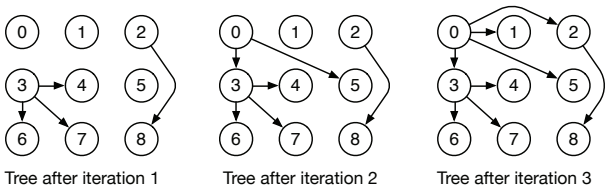


Fig. 2: CRACKER tree creation

Now, let us consider again the example in Fig. 1. Nodes 4, 6, 7 and 8 are excluded from G^2 because they have not been chosen as v_{min} of any node at the previous iteration. Those node are connected to their v_{min} in the propagation tree T. Note also that G^2 preserves the connectivity of remaining vertices, and that this holds in general for every G^t. Finally, note that node 8 generated a link between nodes 3 and 2 which are at 4 hops distance in G. Indeed the CRACKER algorithm halves the distance among any two nodes at each step.

B. Seed propagation

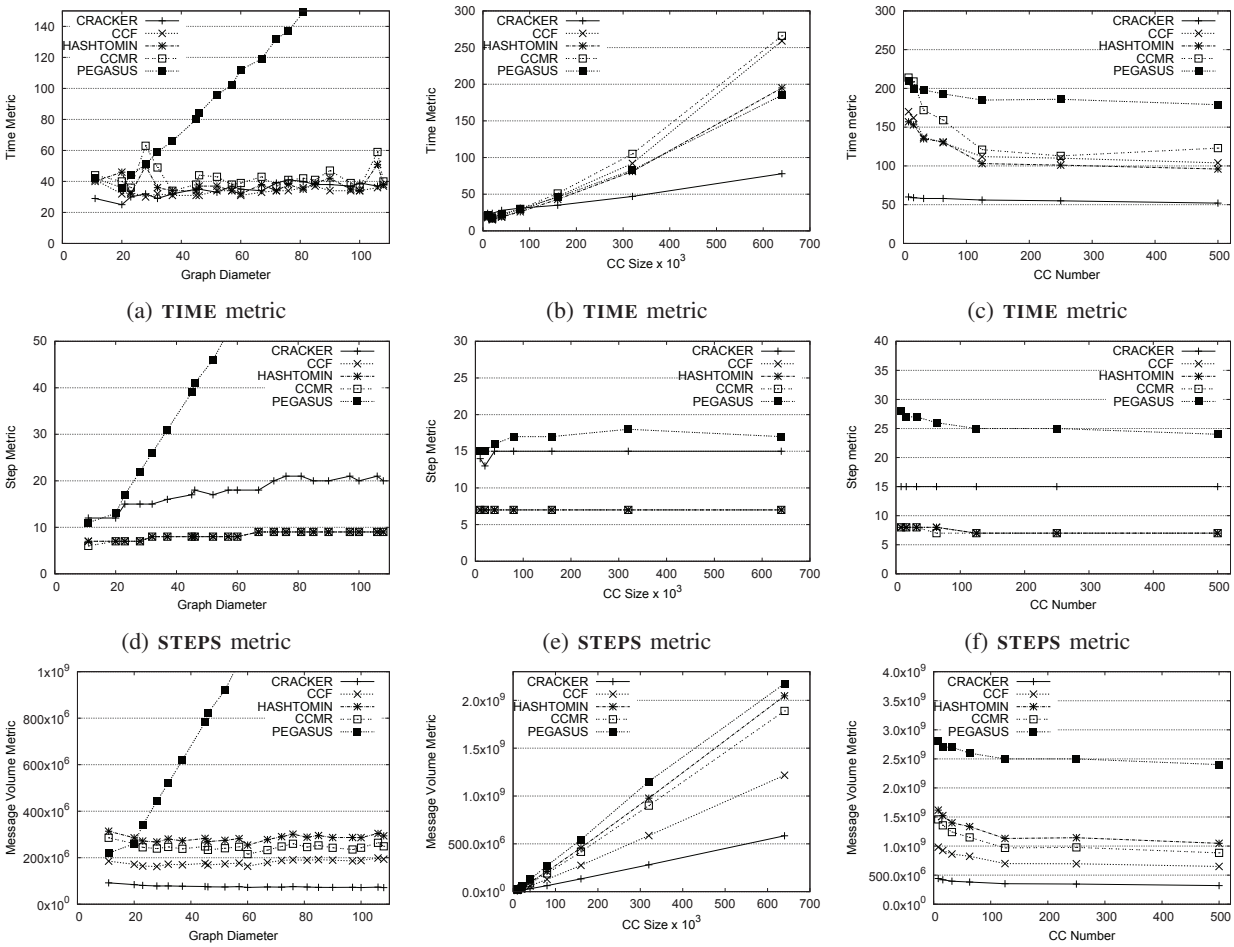
A spanning tree for each component of the graph is incrementally built during the Pruning Step of the algorithm. When a vertex v is excluded from the computation, a directed edge $(v_{min} \rightarrow v)$ is added to the tree structure T (see Line 10 in Algorithm 3). Figure 2, which refers to the example presented in Fig. 1, shows the state of the tree for each iteration of the algorithm.

When the *Seed Identification* phase of the algorithm is completed, there exists for each CC a tree rooted in its *seed* node. Such tree is then used to propagate the seed identifier to all the nodes in the tree.

IV. EXPERIMENTS

The experimental evaluation compares CRACKER against state-of-the-art solutions using both synthetic and real-world datasets. All the experiments have been conducted on a cluster running Ubuntu Linux 12.04 consisting of 5 nodes (1 master and 4 slaves), each equipped with 16 Gbytes of RAM and with a 4-core CPU supporting the Intel Hyper-threading technology. The nodes are inter-connected via a 1 Gbit ethernet network. We compared CRACKER against four other algorithms, which have been introduced in Section II: *a*) PEGASUS [19], as more efficiently implemented with the name of HASH-MIN in [16]; *b*) HASH-TO-MIN, an algorithm proposed by Rastogi et al. [16]; *c*) CCMR, a solution introduced by Seidl et al. [18]; *d*) CCF, an approach proposed by Kardes et al. [20]. We implemented CRACKER and all the other solutions using Apache Spark [11]. To conduct a fair comparison, all the algorithms have been implemented within the same Spark framework and without any particular code-level optimisation.

CRACKER has been implemented according to a MapReduce model. In the MapReduce paradigm, each job is defined by two higher-order functions: the Map and the Reduce. Generally speaking, the Map operation reads a partition of the input and generates a set of $\langle key, value \rangle$ pairs. Several Maps are executed independently and in parallel. The Reduce



(a) **TIME** metric (b) **TIME** metric (c) **TIME** metric

(d) **STEPS** metric (e) **STEPS** metric (f) **STEPS** metric

(g) **MESSAGE VOLUME** metric (h) **MESSAGE VOLUME** metric (i) **MESSAGE VOLUME** metric

Fig. 3: Performance for Varying Component Diameters, Size and Number

operation is applied to the set of generated records with the same key. Also in this case, several Reduce operations are executed in parallel to produce the final output. During the seed identification the $\langle key, value \rangle$ records are organised such that the key represents the identifier of a vertex, and the value represents a neighbour. At the start, the records are initialised according to the source graph. The represented graph evolves during the computation by means of the $addEdge$ operation. In particular, the $addEdge$ at line 4 of Algorithm 2 is implemented as the generation during the Map phase of a record $\langle v, v_{min} \rangle$; the one at line 5 of Algorithm 3 is the generation of two records: one $\langle v, v_{min} \rangle$ and the other $\langle v_{min}, v \rangle$. At the end of the Min Selection Step the set of generated records would represent what we called the H^t graph; whereas at the end of the Pruning Step it would represent the G^t graph.

The seed propagation has been implemented as a separate MapReduce job and starts once the seed identification phase terminates. In this phase, the records have the form $\langle vertex, \{seed, children\} \rangle$. The children define the structure of the trees, and they are initially empty. They are updated during the seed identification phase, with the $addEdge$ operation at line 10 of Algorithm 3. In the implementation this corresponds at the generation of a record $\langle v_{min}, \{\varnothing, u\} \rangle$ to update the parent of u.

The $seed$ information is propagated from the seed node with a few MapReduce iterations. A node whose $seed$ is set, propagates this information to all the children nodes. The number of iterations required is equal to the depth of the propagation tree.

In evaluating the performance of different algorithms, we considered some metrics.

TIME: measures the total time, expressed in seconds, from the loading of the input graph until the algorithm terminates; average over 10 runs are reported;

STEPS: measures the number of steps required by the algorithm; more specifically, we refer to the number of vertex computations, e.g., CRACKER executes two steps per iteration;

MESSAGE NUMBER: measures the total number of messages sent between the map and reduce jobs;

MESSAGE VOLUME: measures the total size of the messages sent between the map and the reduce jobs, in terms of number of vertex identifiers contained in a message.

TABLE I: Datasets description

	Name	$\|V\|$	$\|E\|$	β-index	#cc	ccMaxSize	diameter	d. AVG	d. MAX
Road Networks	Italy	19,006,129	19,939,100	0.95	153,876	14,694,405	10,534	2.09	16
	California [21]	1,965,206	5,533,214	0.36	2,638	1,957,027	849	2.81	12
Social Networks	Flickr [22]	1,715,255	22,613,981	0.08	20,318	1,624,992	13	18.14	27,236
	YouTube [22]	3,223,585	9,375,374	0.34	2,168	3,216,075	31	5.82	91,751
Web graphs	Skitter [23]	1,696,415	11,095,298	0.15	756	1,694,616	25	13.08	35,455
Protein-to-Protein	PPI-All	4,670,194	664,471,350	$<$0.01	16,018	36,255	4	142.28	8,561

TABLE II: Results on real-world datasets. The results marked with star are relative to not terminated executions due to errors.

	Italy				California				PPI-All			
Algorithm	Time	Steps	Msg.s	Vol.	Time	Steps	Msg.s	Vol.	Time	Steps	Msg.s	Vol.
CRACKER	461	33	726	1724	37	24	54	132	1389	12	885	2134
CCF	807	18	1214	4744	60	14	100	408	2182	6	239	3733
HASH-TO-MIN	1681	18	1774	6456	87	14	147	539	1777	6	1103	4360
CCMR	$>$3957*	$>$15*	–	–	88	14	197	333	2235	6	3099	5174

	Flickr				YouTube				Skitter			
Algorithm	Time	Steps	Msg.s	Vol.	Time	Steps	Msg.s	Vol.	Time	Steps	Msg.s	Vol.
CRACKER	45	12	60	147	43	15	68	164	35	14	50	126
CCF	54	7	40	239	49	7	65	267	38	7	39	193
HASH-TO-MIN	82	7	90	338	113	7	109	394	77	7	82	305
CCMR	107	6	170	287	211	7	140	239	127	7	151	255
PEGASUS	121	14	459	1354	140	19	417	1191	136	21	501	1469

A. Performance on synthetic datasets

We generated several artificial datasets by varying their basic features in order to analyse the behaviour of the algorithms in different scenarios. We discuss CRACKER performance varying the diameter of CCs, the graph scale, and the number of CC in the graph.

1) Varying diameters of CC*s:* We evaluated how CRACKER is affected by the diameter of CCs in the graph. To this end, we generated 21 datasets, characterised by different average diameter of its components. Each dataset was generated as a concatenation of Erdos-Renyi random graphs generated with the Snap library [21]. By concatenating them we obtained graphs with arbitrary diameter. Each dataset consists of 1,000,000 vertices and 10 connected components, with (approximately) the same diameter (in the range $[11, 108]$) and composed of approximately 100,000 vertices each.

Figure 3a shows that the diameter of the CCs in the graph has only a marginal effect w.r.t. the **TIME** metric in all algorithms but PEGASUS, whose performance degrades linearly. All the other algorithms have similar running time, with CRACKER being more efficient in graphs with small diameter. The **STEPS** metric has a similar behaviour as depicted in Figure 3d. In this case, the number of steps required by CRACKER is larger than the best competitors. In spite of this, CRACKER is the approach that requires the least amount of data to be exchanged, as shown in Figure 3g. This is due to the effective pruning strategy which removes nodes and any related communication cost.

2) Varying graph scale: We obtain graph of different *scales* by varying the overall size of the artificial graph but keeping fixed the ratio of each component size to graph size. Each random graph is made of 10 connected components, each one approximately of the same size. We built 7 different datasets having connected components in the size range $[10 \cdot 10^3, 640 \cdot 10^3]$. The overall graphs have a total number of vertices in the range $[100 \cdot 10^3, 6.4 \cdot 10^6]$. These datasets have been generated with an approximately constant CCs diameters (in the range $[20, 24]$). Figure 3b shows the results we obtained with respect to the **TIME** metric. When *scale* is small CRACKER performs slightly worse than the other approaches. Conversely, when the size of the graph reaches 6.4M vertices, CRACKER requires less than one third of the time spent by PEGASUS and HASH-TO-MIN, and less than half of the time required by CCF and CCMR. In fact, the slope of the CRACKER **TIME** curve is lower than other algorithms, suggesting better scalability. From the viewpoint of the **STEPS** metric, the algorithms are either not affected or only slightly affected by changes in the *scale* (see Figure 3e). Note that also in this case we can see significant differences in terms of **MESSAGE VOLUME**, as shown in Figure 3h.

3) Varying the number of CC*s:* To evaluate the performance of CRACKER with a variable number of connected components we built 7 different random graphs, all having approximately 5,000,000 vertices, but with a 8, 16, 32, 63, 125, 250 and 500 CCs respectively. The graphs have similar diameter in the range $[27, 33]$. CRACKER is the less affected algorithm when varying the number of CCs. CRACKER is about twice as fast as any other algorithms, as shown in Figure 3c. This gap is even larger when the number of CC is small, and therefore their size increases. Figure 3f reports similar results to the previous experiment. Finally, Figure 3i confirms the efficiency of CRACKER in terms of **MESSAGE VOLUME** metrics.

In conclusion, CRACKER showed to be the most efficient

algorithm in most of the experiments with synthetic graphs. This gap increases when scalability to large graphs is a relevant issue, i.e., when increasing the number and size of CCs. It requires more steps than other algorithms, however, this does not affect the overall running time of the algorithm. Indeed, CRACKER is capable to significantly reduce the graph size at each iteration, thus reducing the number and volume of messaged exchanged.

B. Performance on real-world datasets

All the datasets used in our experiments are publicly available[3]. Table I reports the main features of such datasets. The datasets have been properly chosen to build a comprehensive scenario to generalise as much as possible the empirical evaluation of CRACKER. In the following, we differentiate the datasets considering two features: the diameter of the graph and its β-index, i.e. the ratio of the number of vertices to the number of edges.

1) Road Networks: The first class of datasets contains the road system of two countries: Italy and California. The dataset of Italy has been generated from the data harvested by Geofabrik [4] which collects data from the Open Street Map project [24]. Italy is characterised by a very large connected component covering the 75% of the entire graph and a large number of smaller CC. The California dataset has been downloaded from Snap [21].

Road networks graphs are characterised by large β-index and diameter. Table II presents in a comparative manner the results achieved by CRACKER, CCF, HASH-TO-MIN, CCMR. PEGASUS was not reported because of excessive running time. Note that the results of CCMR for Italy refer to partial, not terminated, executions, due to out-of-memory errors happened during the computation. CRACKER clearly outperforms other algorithms: it runs between the 28% and the 75% faster than CCF and more than 75% faster than HASH-TO-MIN. Also concerning the **MESSAGE NUMBER** and **MESSAGE VOLUME** metrics, CRACKER exhibits the best performance, as it generates at least 20% less messages and 50% less message volume than any other algorithm on all datasets.

2) Social Networks: We considered the Flickr [22] and YouTube [22] datasets available from the Snap repository. Social network datasets are characterised by a small diameter, large node degree, and a large number of edges, with $\beta < 0.35$.

Also on these graphs CRACKER is the best performing algorithm as reported in Table II. CCF and HASH-TO-MIN are, respectively, at least 14% and 82% slower on every datasets. CCMR and PEGASUS show considerably worse performance. CCF is the best algorithm in terms of **MESSAGE NUMBER** metric, but CRACKER always generates a significantly smaller communication volume, with a reduction of at least 35%.

3) Web Graphs: We conducted experiments on the Skitter [23] dataset, representing the structure of the Internet topology graph taken by running daily the *traceroute*s command in 2005. As reported in Table II, the performance of CRACKER are similar to that observed on social network datasets. CRACKER is the fastest algorithm and it also generates the smallest traffic volume.

4) Biological dataset: In the PPI-All dataset, the vertices correspond to protein and edges correspond to interactions between them thus forming a protein network. Among those considered, PPI-All is the dataset with the largest number edges (665 millions), but it has a small diameter (4). As shown in Table II, HASH-TO-MIN is the best competitor to CRACKER, which is still the best performing. In this case HASH-TO-MIN is the nearer competitor due to the characteristic of the dataset: a small diameter and the largest CC of just 36K nodes approximately.

C. Scalability

This section evaluates how CRACKER scales, in terms of the **TIME** metric when increasing the number of computational resources. Scalability is an important feature for CRACKER, since it has been conceived, designed and implemented to target distributed computing environments.

We run the scalability experiments on all the real world dataset presented in Table I. All the dataset shown the same trend. Due to space constraints, here we show the result with Flickr (Figure 5) as representative. PEGASUS is the slowest algorithm, but it has the best improvement factor of 3.45x with quadruple computational power. CRACKER exhibits similar behaviour to other algorithms with a 3x improvement, but it is always the fastest algorithm.

D. Node pruning

The ability of prune nodes from the computation is one of the key features of CRACKER. Indeed, CRACKER selectively exploits only the vertices that are required for finding the connected components in the input graph. We evaluate node pruning by considering the *simplification degree*, defined as the percentage of active nodes during the various steps of computation. Figure 4 shows that the Pruning Step significantly reduces the number of active vertices on every dataset. It interesting to note how after the third step all the graphs are below their 50% size, and most of them below the 10%. Dealing with a graph an order of size less with respect to the original size is what allows CRACKER to achieve lower completion time when compared to the other approaches.

E. Discussion

From the empirical evidence presented, we can conclude that CRACKER resulted to be the fastest algorithm of its class both when used on real-world and synthetic datasets. The wide spectrum of datasets analysed reveals that the pruning mechanism is effective and let CRACKER to outperform the existing state-of-the-art solutions, especially with respect to the **TIME** and **MESSAGE VOLUME** metrics. CRACKER revealed to be a very flexible solution, resulting the most efficient solution both when the largest CC is near the dimension of the entire graph and when it is of smaller size. In addition,

[3] http://www.di.unipi.it/~lulli/project/cracker.htm

[4] http://download.geofabrik.de/

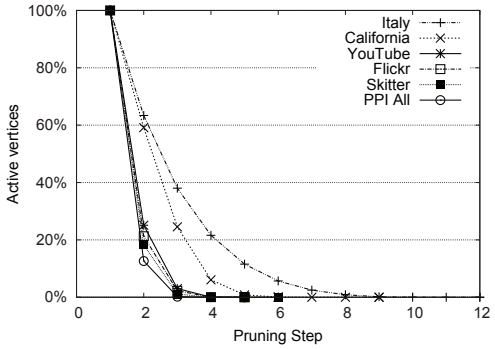


Fig. 4: Simplification Degree

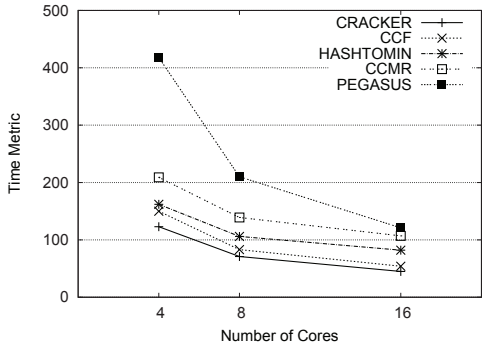


Fig. 5: Scalability with Flickr dataset

the performance of CRACKER are marginally affected by the diameter of the graph. In datasets with a high average degree, CCF achieves the better results in term of **MESSAGE NUMBER**. However, even in those cases, CRACKER confirmed to be the fastest algorithm.

V. Conclusion

In this paper we presented CRACKER, a distributed algorithm for finding connected components in large graphs targeting large distributed computing platforms. CRACKER is organised in two main phases. The first phase consists in an iterative process devoted to CCs identification and graph pruning. The second phase labels each node with the *id* of the CC it belongs to. We implemented CRACKER and other state-of-the-art approaches to evaluate their performance by means of a comprehensive set of experiments. The experiments have been conducted on a wide spectrum of both synthetic and real-world data. In all the experiments CRACKER proved to be a very effective and fast solution for finding CCs in large graphs. In terms of time, CRACKER outperforms its competitors in every dataset used, with the best competitor being from 9% to 75% slower. In addition, CRACKER obtained the least message volume among all the competitors. As a future work we plan to conduct a comprehensive analysis on the factors (e.g., diameter, denseness, etc.) that impact both on simplification mechanism and completion time as well. In this paper we give a preliminary evaluation of the scalability of our approach varying the amount of computing resources from 1 to 4 machines (4 to 16 cores). We plan to further test the scalability of CRACKER when using hundreds of machines.

References

[1] "Common crawl," http://commoncrawl.org/.
[2] "Linked open data datasets (lod2)," http://stats.lod2.eu/, Nov. 2014.
[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.
[4] M. Danelutto, M. Pasin, M. Vanneschi, P. Dazzi, D. Laforenza, and L. Presti, "Pal: exploiting java annotations for parallelism," in *Achievements in European Research on Grid Systems*. Springer US, 2008.
[5] M. Danelutto and P. Dazzi, "A java/jini framework supporting stream parallel computations," in *Proceedings of the International Conference ParCo 2005*, G. R. J. et al., Ed., 2005.
[6] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S. M. Seitz, and R. Szeliski, "Building rome in a day," *Communications of the ACM*, vol. 54, no. 10, pp. 105–112, 2011.
[7] B. J. Mirza, B. J. Keller, and N. Ramakrishnan, "Studying recommendation algorithms by graph analysis," *Journal of Intelligent Information Systems*, vol. 20, no. 2, pp. 131–160, 2003.
[8] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.
[9] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *Link mining: models, algorithms, and applications*. Springer, 2010, pp. 337–357.
[10] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Trans. Web*, vol. 1, no. 1, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1232722.1232727
[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.
[12] E. Carlini, P. Dazzi, A. Esposito, A. Lulli, and L. Ricci, "Balanced graph partitioning with apache spark," in *Euro-Par 2014: Parallel Processing Workshops*. Springer, 2014, pp. 129–140.
[13] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973. [Online]. Available: http://doi.acm.org/10.1145/362248.362272
[14] D. R. Karger, N. Nisan, and M. Parnas, "Fast connected components algorithms for the erew pram," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1992.
[15] D. B. Johnson and P. Metaxas, "Connected components in o(log(3/2 n)) parallel time for the crew pram," *journal of computer and system sciences*, vol. 54, no. 2, pp. 227–242, 1997.
[16] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma, "Finding connected components in map-reduce in logarithmic rounds," in *29th IEEE International Conference on Data Engineering*. IEEE, 2013.
[17] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.
[18] T. Seidl, B. Boden, and S. Fries, "Cc-mr–finding connected components in huge graphs with mapreduce," in *Machine Learning and Knowledge Discovery in Databases*. Springer, 2012, pp. 458–473.
[19] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.
[20] H. Kardes, S. Agrawal, X. Wang, and A. Sun, "Ccf: Fast and scalable connected component computation in mapreduce," in *Computing, Networking and Communications (ICNC), 2014 International Conference on*. IEEE, 2014, pp. 994–998.
[21] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, 2009.
[22] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 29–42.
[23] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 177–187.
[24] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *Pervasive Computing, IEEE*, vol. 7, no. 4, pp. 12–18, 2008.