Scalability Analysis of Cluster-based Betweenness Computation in Large Weighted Graphs

Andrea Castiello, Gianmarco Fucci
University of Sannio
Benevento, Italy
Email: {name.surname}@unisannio.it

Angelo Furno
University of Lyon, ENTPE-IFSTTAR, LICIT
Lyon, France
Email: angelo.furno@ifsttar.fr

Eugenio Zimeo
University of Sannio
Benevento, Italy
Email: zimeo@unisannio.it

***Abstract*—Computation of node betweenness centrality (BC) of weighted and directed graphs is a time-consuming task that could limit the application of such a metric for monitoring large, dynamic networks. As widely demonstrated in previous work, approximated approaches represent a solution to reduce computation time when ranking nodes according to their BC values is sufficient with respect to knowing their exact BC values. According to this observation, we have proposed a fast algorithm for computing approximated BC values for large weighted and directed graphs. It is based on the identification of pivot nodes that equally contribute to BC values of the other nodes of the network discovered via a cluster-based approach.**

In this paper, we focus on the performance and scalability analysis of the proposed algorithm in order to characterize its behavior with different sets of computing resources and to identify room for further improvements. To this end, we exploit a real dataset related to a transportation network. The results show that the proposed algorithm exhibits significantly lower execution times if compared with the Brandes's solution for computing exact BC values, especially when the number of available computing resources is limited. However, the speedup is not negligible even when the number of resources grows, where improvements are possible, as shown by our analysis.

***Index Terms*—Betweenness Centrality, Weighted Big Graphs, Performance Evaluation, Scalability Analysis, Transportation Networks**

I. INTRODUCTION

Graph models are used to study topological bottlenecks of many kinds of networks. For these models, *Betweenness centrality* (BC) [1] is a very popular metric to characterize nodes that are most traversed by shortest paths connecting pairs of other nodes of a graph. It has been widely adopted in many application domains but its high computation time limits the adoption for real-time monitoring of very large networks.

The fastest algorithm to compute the exact value of betweenness centrality has been proposed by Brandes [2]. Given a graph $G(V,E)$, it exhibits $O(n+m)$ space and $O(nm)$ time complexities for unweighted graphs and $O(nm+n^2log(n))$ for weighted ones, where $n=|V|$ is the number of nodes and $m=|E|$ the number of edges. To achieve this performance, Brandes adopts a single-source shortest-paths (SSSP) algorithm based on breadth-first graph search or on Dijkstra algorithm for unweighted and weighted graphs, respectively. Each exploration is computed with a complexity $O(m)$ and $O(m+n\cdot log(n))$, for unweighted and weighted graphs respectively, where $m \ll n^2$ is the number of edges. This is good for sparse graphs but insufficient for real-time monitoring of very large networks.

A faster approach, useful for some kinds of applications, allows achieving lower computation time by calculating approximated BC values. These strategies try to penalize some shortest paths or exploit heuristics (possible based on topological properties) to identify only $k \ll n$ pivots as sources for the computation of single-source shortest-paths (SSSP).

While several attempts exist for computing approximated values of betweenness centrality for unweighted graphs, directed and weighted graphs have been scarcely addressed in the literature. These kinds of graphs are becoming more and more important in different application domains, such as social networking (where the edge weights can indicate the strength of the interaction between people) and transportation (where weights can convey information on travel times, vehicle flows, etc.) among many others. Moreover, the growing size of the graphs considered in such application domains poses significant challenges from a scalability point of view, demanding more and more computing power when it comes to compute network metrics on top of them.

In this paper, we discuss accuracy, performance and scalability (with reference to the number of computing resources) of our algorithm introduced in [3]. Similarly to the ones proposed in [4] and [5], the algorithm exploits topological characteristics of graphs to classify nodes for their selection as pivots, thus allowing for rapidly computing approximated values of BC on very large weighted and directed graphs. Differently from the papers above, it exploits clustering to identify reference nodes (clusters' border nodes) to perform a topological analysis. The solution can calculate an almost exact value of betweenness centrality for several nodes, i.e., the most critical ones, while keeping a good approximation for the others, with an execution time that strictly depends on the number of retained pivots.

The evaluation, which is the main focus of this paper, is organized in two sections. In the first part, we evaluate the accuracy of our algorithm on a very-large graph, corresponding to the transportation road network covering the whole Rhone-Alpes region, France. The original graph is directed and unweighted, but we enrich it with dynamic weights derived from Global Positioning System (GPS) taxi traces, thus generating a dynamic, directed and weighted graph. Edge weights represent the hourly median travel time on the corresponding

road segments, as estimated from the taxi trips. Computing betweenness centrality on such a graph (e.g., every hour) can provide indications about re-distributions of the traffic flow (or potential congestion), if we make the assumption that people prefer the shortest (fastest) paths to reach their destinations.

In the second part, we present an in-depth discussion on the performance behavior exhibited by the algorithm, with a fixed dataset and by varying the set of available resources. A full more general analysis with multiple dataset has been addressed in [6] for unweighted graphs, and it will be extended to the weighted version of the algorithm in future work. The evaluation shows that our approximated solution is able to significantly reduce the number of sources nodes (pivots) for SSSP computation thus, consequently, lowering the sequential computing time. Additionally, we show that our solution preserve a good level of parallelism, thus permitting to exploit multiprocessor and distributed computers to further reduce computation time when such hardware is available or when higher accuracy is needed.

The rest of the paper is organized as follows. In Sec. II, we present related work. In Sec. III, we describe our algorithm for fast BC calculation, while Sec. IV presents the evaluation of our approach in terms of accuracy on a large-scale transportation dataset. Sec. V reports on an in-depth performance and scalability analysis of the proposed algorithm. Finally, Sec. VI concludes the work by also discussing future directions.

II. Related Work

Betweenness centrality, originally proposed in [7] for undirected graphs was extended to directed graph in [8]. Brandes in [2] proposed a faster algorithm which also works for weighted networks. The idea was the adoption of single-source shortest-paths algorithm via breadth-first graph search and Dijkstra algorithm for unweighted and weighted graphs, respectively.

Several approaches, aiming to evaluate exact or approximated solutions, have been developed to further reduce the computation time. The proposed solutions can be classified according to three main categories: *(i)* exploiting and increasing parallelism, *(ii)* estimating BC values through a partial exploration of graphs, *(iii)* calculating BC values of fraction of nodes in dynamically evolving graphs.

Parallel approaches: In [9], the first parallel implementation for computing betweenness centrality is presented, which handles weighted graphs as well. It is based on a fine-grained multi-level parallelism, in which the neighbors of a given node are traversed concurrently on a shared data structure with granular locking. The algorithm was successively improved [10] by removing the need for locking in the dependency accumulation stage of Brandes' algorithm through the adoption of a successor list instead of a predecessor list for each node.

Incremental approaches: A different set of approaches (stream-based) tries to avoid recomputing the BC values of all the nodes of a graph $G' \equiv G + \Delta G$ when they are known for a previous configuration G. For example, in [11], researchers proposed an efficient approach that reduces the search space by focusing only on the vertices whose betweenness centralities get changed as a consequence of an update in the graph. Similarly, in [12], computation time is reduced by using the hypergraph sketch data structure, i.e., a weighted hyper-edge representation of shortest paths. Based on sampling based techniques [13], Bergamini et al. first proposed a semi-dynamic [14] and later a fully dynamic approach [15] for dynamic networks (both weighted and unweighted), capable of performing in-memory computation with millions of edges. Recently, an efficient algorithm for incremental BC computation [16] has been proposed. The algorithm has good performance when the graph changes for a very limited number of nodes. Conversely, the high speedup drastically reduces when the graph is subject to significant changes of its topology.

Approximated approaches: The third research trend focuses on achieving low computation time by calculating approximated BC values. These strategies penalize some shortest paths, whose computation is the most expensive task in the whole process. For example, in [17], the authors only consider paths up to fixed length k. Brandes and Pich [4] also proposed an approximated algorithm for faster BC calculation, by choosing only $k \ll n$ pivots as sources for the SSSP algorithm through different strategies, but they overestimate the BC of unimportant nodes that are near a pivot. To overcome this problem, various solutions have been proposed. A generalization framework for betweenness approximation [5] proposes to scale BC values in order to reduce them with reference to nodes close to pivots. In another paper, a solution to reduce the pivots for nodes with high centrality is proposed via adaptive sampling techniques [18]. A recent work [19] based on approximation shows large fluctuations of accuracy over the top-100 nodes on a scale-free graph. A random, shortest path based [20] approximation approach was presented in [13].

For directed and unweighted networks an approach is presented in [21], where similarly to [22], authors pre-compute reachable vertices for all the graph nodes. However, at the best of our knowledge, there is a scarcity of contributions focused on both weighted and directed networks. A proposal in this direction is presented in our recent work [3]. Here, we briefly describe the approach to present its main internals and support the discussion on performance and scalability of the algorithm, which is the main contribution of this paper.

III. Fast BC computation of weighted and directed graphs

In this section, we present the *W2C-Fast-BC* algorithm [3], the weighted and directed version of the one previously proposed in [6], [23]. It allows reducing BC computation time of weighted and directed graphs in a parametric way, i.e., by acting on the accuracy of BC values. The algorithm is based on the Brandes' one for weighted graphs and on the heuristic proposed in [24] for identifying graph pivots.

As in our previous work [6], [23], we exploit a fast clustering algorithm based on modularity for identifying graph communities and their related border nodes. Specifically, we used a distributed implementation [25] of Louvain method for weighted and directed graphs [26].

In the following subsections, we briefly introduce the adopted notation, the Brandes' algorithm and discuss modularity for weighted graphs.

A. Notation

We assume the following definition throughout the paper. Let $G(V, E, T, W, f(E, T))$ be a dynamic, weighted, directed graph, where V denotes the set of nodes and $E \subseteq V \times V$ the set of edges. $N = |V|$ denotes the number of nodes in the graph. W represents the set of weights and T the set of time units. For instance, for very large networks, T may represent hours of the day. We highlight that the length of the considered time unit (e.g., 1 hour) represents the period of observations before a new computation of BC is launched, and translates therefore into a time constraint for computing betweenness centrality. The function $f : E \times T -> W$ maps each edge $e_{ij} \in E$ at time slot $t \in T$ to a weight $w \in W$. We denote as $\hat{G}(V, E, \hat{W})$ a directed and weighted instance of the dynamic graph G related to a specific time slot $\hat{t}$ and therefore associated to a subset of weights $\hat{W} \subseteq W$. The algorithms reported in the following are related to a specific instance $\hat{G}$ of the dynamic graph G, i.e., BC computation is iteratively performed (in a quasi-real time fashion) at the beginning of time slot $\hat{t}+1$ on the instance of the dynamic graph related to time slot $\hat{t}$.

A path $p(v_i, v_j)$, between two nodes v_i and v_j of $\hat{G}$, consists of a set of nodes and edges that connect these two nodes.

The length of a path between any two nodes v_i and v_j, represented by $len(p(v_i, v_j))$, is the sum of the weights of the edges (or hops) to reach v_j from v_i. If nodes v_i and v_j are directly connected, then the path length is the weight of the link, or 1 for unweighted graphs. A shortest path between any two nodes v_i and v_j, denoted as $sp(v_i, v_j)$, is a path with the minimum length, among all the paths connecting the two nodes. Multiple shortest paths may exist between the same pair of nodes, i.e., multiple paths having the same length. The distance $d(v_i, v_j) = len(sp(v_i, v_j))$ is the length of the shortest path between nodes v_i and v_j. We denote $\sigma_{v_i v_j}$ as the number of shortest paths between v_i and v_j, while $\sigma_{v_i v_j}(v_k)$ represents the number of shortest paths from v_i to v_j that cross node v_k.

B. Brandes' algorithm

Given a graph $\hat{G}$, the *pair-dependency* of a *source* node s on an another node v for a *destination* t of the graph is defined as $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The betweenness centrality of any node v can be expressed in terms of *dependency score* $\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v)$, obtained by summing the pair-dependencies of each pair of nodes on v that has s as source node. To compute this score, Brandes' algorithm exploits a recursive relation that is motivated by this observation: let $R = \{w : v \in P_s(w)\}$ be the set of nodes w such that v is a predecessor of w along a shortest path that starts from node s, and $P_s(w) = \{v \in V : \{v, w\} \in E, d(s, w) = d(s, v) + d(v, w)\}$ the set of direct predecessors of a generic node w in the shortest paths from the source node s to w; then, v is a predecessor also in any other shortest path starting from s and passing through a different $w \in R$ [2]. Consequently, we have:

$$\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)), \quad (1)$$

Finally, the *BC* of node v is obtained as:

$$BC(v) = \sum_{s \in V} \delta_{s\bullet}(v). \quad (2)$$

For scaling purpose, BC values are often normalized by dividing them by $(n-1) \cdot (n-2)/2$ for undirected graphs and by $(n-1) \cdot (n-2)$ for directed ones.

Conceptually, Brandes' algorithm runs in two phases. During the first phase, it performs a search on the whole graph to find all the shortest paths starting from every node s, considered as source of the breadth-first exploration of the whole graph. In the second phase, it performs dependency accumulation by backtracking along the discovered shortest paths. During these two phases, the algorithm maintains four data structures for each node found on the way: a predecessor set $P_s(v)$, the distance $d_s(v)$ from the source, the number of shortest paths from the source $\sigma_{st}(v)$ and the dependency accumulation when backtracking at the end of the search.

C. Weighted modularity and Louvain method

Modularity is a metric, defined as a value between -1 and 1, to measure how tightly the nodes are attached to each other in the network. It was introduced to identify communities in undirected and unweighted (or weighted) networks [27]. Given a graph $\hat{G}$, partitioned into a set of communities $C = \{c_1, c_2, ..., c_D\}$, formally, modularity [28] of graph $\hat{G}$ is defined as follows:

$$Q = \frac{1}{m} \sum_{i,j \in V} \left[A_{ij} - \frac{k_i^{in} k_j^{out}}{m} \right] \delta(c_i, c_j) \quad (3)$$

where δ is 1 if $c_i = c_j$ (nodes i and j belong to the same community) or 0 otherwise, m is the sum of all of the edge weights in the graph, k_i^{in}, k_j^{out} are the sum of the weights of the edges entering node i and the edges exiting node j, respectively; A_{ij} is 0 if nodes n_i and n_j are not connected. In case the nodes n_i and n_j are connected then A_{ij} is $w_{i,j}$, that is the weight of the edge connecting nodes i and j.

We exploit modularity for clustering weighted directed graphs with the Louvain method [29], [30]. The algorithm initially searches for *small* communities. Then, it creates a new graph whose nodes are the communities identified in the previous step. These two steps are iteratively run until there is no modularity gain derived by aggregating clusters in larger communities. In our implementation, the weights used to compute weighted modularity are assumed as in the notion of closeness (nodes are tighter if they have lower distance or travel time), i.e. “smaller is tighter”. This choice is motivated by the fact that we want to reduce the number of border nodes for each cluster. Therefore, we generate communities whose nodes are highly locally inter-connected with short (or fast to travel) local paths. Conversely, when

computing shortest paths in SSSPs, weights are assumed as in the notion of length (or travel time), i.e., "higher is farther". We use a distributed variant of the Louvain algorithm for weighted and directed graphs [25], [26]: all vertices select a new community simultaneously, updating the local view of the graph after each change. Even though some choices will not maximize modularity, after multiple iterations, communities will typically converge thus producing a final result relatively close to the sequential version of the algorithm.

D. W2C-Fast-BC

Given a graph $\hat{G}$, we split it into a set of clusters (i.e., $\mathbf{C}$) by using the Louvain (Alg. 1, line 3) method for weighted graphs [25][1].

Algorithm 1 Pseudo code of the W2C-Fast-BC Algorithm

```
1:  function W2C-FAST-BC(Ĝ, C, kFrac)
2:                                                        ▷ Phase1
3:     C ← weightedLouvainClustering(Ĝ)
4:                                                        ▷ Phase2
5:     map i ← 1,|C| do
6:        bordernodes_i ← findBorderNodes(Ĝ, C_i)
7:     end map
8:     map i ← 1,|V| do
9:        localδ_i ← computeLocalδ(i, C, bordernodes)
10:    end map
11:    reduce i ← (1,|V|), localδ_s, localδ_z, i = j do
12:       localBC_i ← localδ_s(i) + localδ_z(j)
13:    end reduce
14:    map i ← 1,|C| do
15:       superClasses_i ← WkMeansClustering(C_i, classes_i, kFrac)
16:    end map
17:    map i ← 1,|superClasses| do
18:       P_i ← selectPivotOf(superClasses_i, localBC)
19:    end map
20:                                                       ▷ Phase3
21:    map i ← 1,|P| do
22:       δ_i ← computeδFrom(P_i)
23:       δ_i ← (δ_i − localBC) · |superClasses_i|
24:    end map
25:    reduce i ← (1,|V|), δ_s, δ_z, i = j do
26:       BC_i ← δ_s(i) + δ_z(j)
27:    end reduce
28:    for i ← 1,|V| do
29:       BC_i ← BC_i + localBC_i
30:    end for
31:    return BC
32: end function
```

The main result of clustering is the identification of *border nodes* (an array for each cluster). A border node is a node having at least one neighbor node in a different cluster (line 6). Then, a parallel execution of Brandes' algorithm (based on Dijkstra[2]) is performed inside each cluster to compute the *local BC* (lines 8-13). This computation generates the partial inner-cluster contribution to the BC of each node and additional information, such as the number of weighted shortest paths and distances from a node of a cluster towards each border node of the same cluster. The information above is used to identify the nodes inside each cluster that equally contribute to the dependency score of each node of the graph (equivalence class, see [6], [24] for more details). Taking into account that nodes belonging to the same class produce the same dependency score on each node of the graph, one representative node should be identified as a source node for applying Dijkstra's algorithm (line 18). This node is called class *pivot*.

The partial dependency score calculated for the pivot is then multiplied by the cardinality of the pivot class (line 23). This method avoids re-applying Dijkstra' algorithm to another node of the same class, thus ensuring fast calculation of BC if $P \ll N$, where P represents the set of pivots selected and N represents the number of nodes of the graph. The final value of BC is obtained for each node by summing up all partial contributions (produced by the reduce operation, lines 25:27) with local BC values (lines 28:30).

To further reduce the computation time, we extended the concept of *class* by introducing *super classes* through an additional clustering operation inside each initial Louvain-derived cluster (line 15). A super class is a group of classes, belonging to the same Louvain cluster and obtained by clustering (via K-means) the vectors generated by considering, for each node, the normalized distances from the Louvain cluster's bordernodes and the amount of shortest paths towards them.

To perform class grouping, we exploit a parallel K-means algorithm by using a different K for each initial Louvain cluster. K is defined as a fraction (*K-Fraction*) of the initial number of classes belonging to each Louvain cluster. For example, by considering a fraction equals to 0.4, the algorithm adopts a 0.4 fraction of the number of classes in each Louvain cluster. By this approach, we are able to drive the behavior of the algorithm towards the desired computation time. However, when the computation time decreases the approximation worsens, as deeply illustrated in our previous work [6], [23], [24].

IV. Analysis of a Dynamically Weighted Real-World Road Network

To evaluate our W2C-Fast-BC algorithm, we leveraged a large-scale transportation graph, namely **Rhone-ROADS**, corresponding to the entire road network of the Rhone department, France [31]. The graph includes the agglomeration of Lyon and its surroundings and has a geographical extent of approximately 3,300 Km^2. The network is directed and unweighted, with 117,605 nodes and 248,337 edges.

We transformed the Rhone-ROADS graph into a dynamic weighted graph by relying on an additional dataset, namely **Rhone-TAXIS**, which reports on anonymized GPS traces of taxis active in the Rhone department. Rhone-TAXIS has been collected by the French operator Radio Taxi via a fleet of approximately 400 taxis during 2011-2012. Geo-referenced taxi trips are collected according to a variable sampling interval (between 10 and 60 seconds), with a global average of 800,000 measurements per day.

The generic sample of the Rhone-TAXIS dataset, i.e., an *elementary taxi trip*, includes the time-stamped start and

[1]The implementation leverages a Scala parallel solution partially based on the Distributed Graph Analytics (DGA) by Sotera: https://github.com/Sotera/distributed-graph-analytics.

[2]The adoption of Dijkstra algorithm instead of breadth first search represents a main variant of our FastBC algorithm proposed in this paper.

arrival GPS positions of a small segment traveled by the associated taxi identifier. These measures permit to roughly estimate the traveled distance and the instant speed of the taxi moving along a given road segment.

To obtain the final dynamic weighted graph from the above-mentioned datasets the following procedure was applied:

1) As a preliminary cleansing step, we filtered out elementary trips with unrealistic speeds ($\geq$ 130 Km/h).
2) By considering only the retained elementary trips, we map-matched all taxi trips to the edges of the Rhone-ROADS graph. This step allows identifying the association between elementary trips and the directed edges of the Rhone-ROADS graph. To that purpose we used the Python Mapillary map-matching open source library[3].
3) From the map-matched elementary trips, we computed the hourly median speeds (referred to a typical day) of each Rhone-ROADS edge. More precisely, we created a 24-sized vector (one entry for each hour of the day) for each edge e, where the entry associated to the generic time slot (i.e., hour) t is computed considering the median of all the speed values of the elementary trips associated to edge e and related to time slot t from possibly different days.
4) Edges with no elementary trip associated during time slot t are filtered out and not considered in the following phases.
5) We calculated the median hourly travel time vector for each edge e from the previous vectors. The estimated travel time to cross an edge at the generic time slot t is obtained as the ratio of the length of the corresponding road link (a static information known from the Rhone-ROADS graph) to the hourly median speed estimated on that link (the entry at time slot t from the speed vector).
6) We reconstructed the final dynamic graph as a graph composed of 24 different weighted instances. The snapshot instance $\hat{G}$ of the dynamic graph for a specific time slot $\hat{t}$ is obtained by considering the weights associated to each edge e as from the entry $\hat{t}$ of the travel time vector above.

The W2C-Fast-BC and Brandes algorithms are applied iteratively to each hourly instance of the dynamic graph (i.e., a snapshot $\hat{G}$). This is conceptually equivalent to an on-line operational situation, where the graph naturally emerges from sensor-collected data used to continuously compute up-to-date traffic information on each edge with hourly periodicity.

It is worth noting that, given the relatively small size of the observed taxi fleet and the circadian rhythm characterizing human mobility, snapshots of the dynamic graph related to rush hours (e.g., 7:00-09:00 and 17:00-19:00) have a much smaller size with respect to the original Rhone-ROADS graph, i.e., approximately 30,000 nodes and 60,000 edges (see the framed portion of the graph in Fig. 1). Such size further reduces for graph snapshots related to non-rush hours. Indeed, most of the observed elementary trips are condensed within the city center of Lyon, with only few observations recorded in the outskirts and within rural areas as well as during night time. However, since the goal of the paper is to prove the efficiency of our solutions with respect to very-large scale weighted networks, we have decided to increase the scale of the dynamic graph by means of a spatial interpolation technique.

[3]https://github.com/mapillary/map_matching

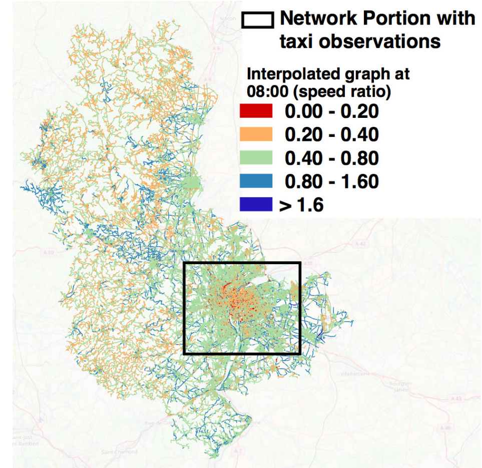

Fig. 1. KNR-interpolated graph at 08:00

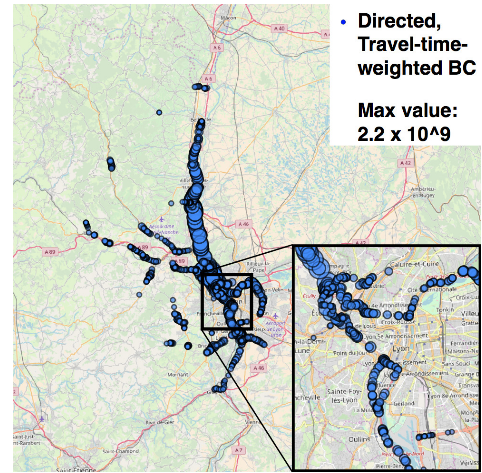

Fig. 2. Top-1000 nodes' BC values at 08:00

To obtain a dynamic, realistic, weighted network, larger than the one directly observed from taxi trips, we leveraged an interpolation technique that we call *KNR-interpolation*[4]. The technique allows estimating the hourly value of median speed (and thus the median travel-time weight) for those edges of the original Rhone-ROADS network with no available observation from taxi trips at time slot t. The KNR-interpolation is applied in place of *step 4)* of the previously described graph weighing procedure. Fig. 1 graphically shows the KNR-interpolated snapshot at 08:00 of a typical working day. Fig. 1 also presents speed-ratios (i.e., median speed divided by road speed limit) either estimated via taxi traces (for the framed portion of the graph) or via the KNR interpolation technique. Red and orange

[4]KNR-interpolation is based on K-nearest-neighbor regression [32], a non-parametric supervised machine-learning technique. Each edge is modeled as a data point with multiple topological features. The median speed at time slot t, available for some edges (labeled instances) and missing for other ones (unlabeled instances), represents the target interpolated feature.

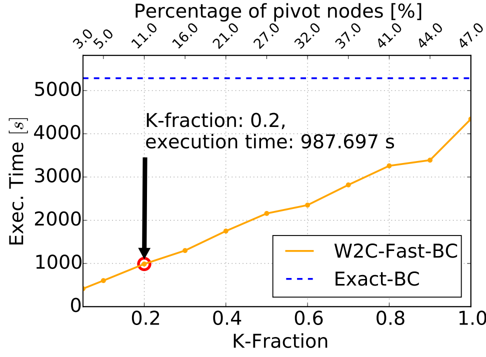

Fig. 3. Execution time of W2C-Fast-BC vs Brandes-BC at 08:00 (with 10 cores)

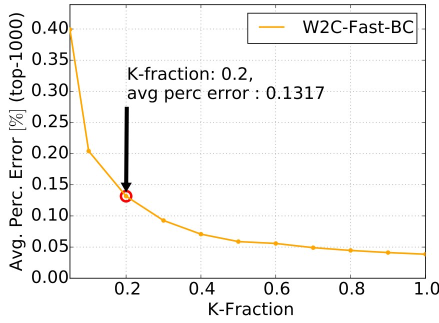

Fig. 4. W2C-Fast-BC average perc. error (top 1,000 nodes) at 08:00

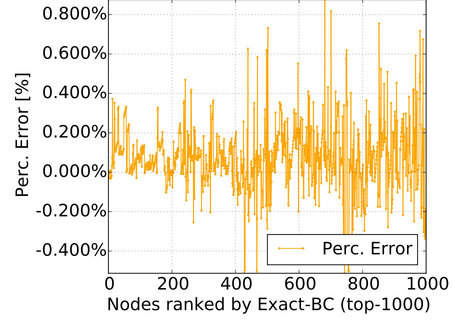

Fig. 5. Percentage errors on the top-1000 nodes with *K-fraction* = 0.2

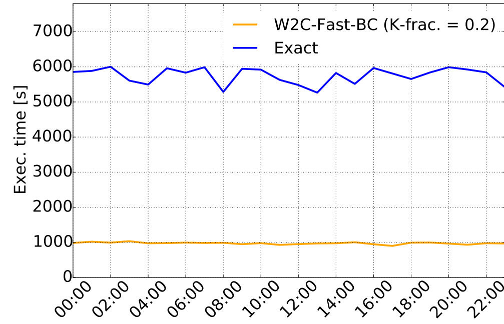

Fig. 6. Dynamic Graph: execution time

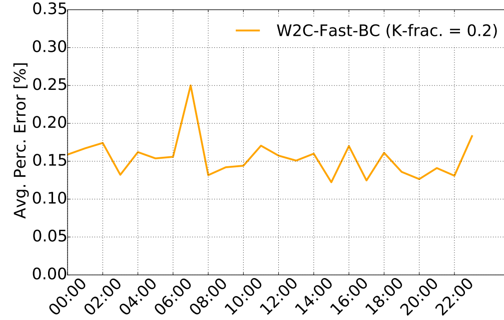

Fig. 7. Dynamic Graph: average percentage error (top-1000)

colors indicate highly-congested edges, i.e., lower values of the speed ratio, while greens and blues indicate a smooth, non-congested situation at time t. The resulting graph has approximately the same size of the Rhone-ROADS network.

The values of BC for the top-1000 nodes are reported in Fig. 2 (nodes with larger circles have higher BC) for the snapshot related to 08:00. Fig. 3 shows that the exact algorithm for computing BC on the weighted graph requires a computation time of more than one hour (when using 10 cores for parallel execution of Brandes' algorithm), therefore being unable to complete within the duration of the time slot.

Remarkably, our W2C-Fast-BC computes in only 987 seconds (i.e., approximately 15 minutes) when using a *K-fraction* equal to 0.2 (always using 10 cores for parallelism, as with Brandes' exact algorithm). Moreover, W2C-Fast-BC shows an average percentage error of only 0.13%, as it can be seen in Fig. 4, related to different values of *K-fraction*, and a maximum percentage error of 0.8% over the top-1000 BC nodes (as reported in Fig. 5, which describes the percentage errors associated to each of the top-1000 nodes of the graph, with *K-fraction* equal to 0.2).

Similar results have been observed over the whole dynamic graph (i.e., the 24 hourly time slots, as reported in Fig. 6 and Fig. 7), thus proving the adequacy of our solution for quasi real-time monitoring of dynamic, weighted road-networks.

V. PERFORMANCE AND SCALABILITY ANALYSIS

In order to analyze the performance of the W2C-Fast-BC algorithm when a different number of computing resources is used, in this section we report and dicuss several test results. We performed the tests by using a Scala-based implementation of both W2C-Fast-BC and exact algorithms, by leveraging multi-core processing for parallel execution through Apache Spark. This framework was configured to work in the standalone cluster mode on two Intel Xeon E5 2640 2.4 GHz multi-core machines, each equipped with 56 virtual cores and 128 GB of DDR4 RAM.

We considered different execution configurations by varying the number of cores used in the cluster and the *K-fraction* parameter exploited by the K-means clustering algorithm. The analysis involved different numbers of cores (i.e., 1, 2, 4, 8, 16, 32, 56) and different values of the *K-fraction* parameter

(i.e., 0.01, 0.2, 0.4, 0.6, 0.8, 1).

The first tests we conducted aimed to compare the execution time of the algorithm by evaluating the elapsed time between the end time and the start time for each configuration. Subsequently, we calculated the efficiency by evaluating how the different configurations of the algorithm exploit the parallelism permitted by the available computing resources of the cluster. Then we evaluated the performance of the proposed W2C-Fast-BC algorithm by comparing it with the exact solution proposed by Brandes. This comparison, reported as exact-fast ratio, allowed us to evaluate the speed increase of the different behaviors (by varying *K-fraction*) of the W2C-Fast-BC algorithm compared to the exact solution.

In order to unveil the reasons behind a decreasing speedup with a growing number of cores, we conducted additional intensive tests to analyze the performance of each main part of the proposed algorithm. For this breakdown analysis, we identified three phases of the algorithm (see Alg. 1): *i)* parallel *Louvain clustering* execution, *ii) local BC* evaluation with *identification of classes/super-classes*, *iii) pivot-based computation* and *final reduce* operations to sum up the different contributions of BC.

A. Elapsed Time

The first test we performed was referred to the measure of the elapsed time observed with a different number of cores and different values of the *K-fraction* parameter. In particular, the plot in Fig. 8 shows on the X-axis the number of cores in a base-2 logarithmic scale and on the Y-axis the elapsed time in a base-10 logarithmic scale. The figure shows a different curve for each tested scenario: one curve (labelled as *exact*) represents the elapsed time of the execution based on the exact (Brandes) algorithm while the other ones (each labelled with the value of the *K-fraction* parameter) show the elapsed times obtained with W2C-Fast BC algorithm, when using one of the different values of *K-fraction* from the interval reported above.

The execution carried out with the exact algorithm shows, as expected, an elapsed time greater than all the executions with W2C-Fast-BC. In all cases, by increasing the number of cores, the execution time decreases in a non-linear way. Comparing the different curves of the W2C-Fast-BC algorithm, we observe that with a fixed number of cores the execution time strongly decreases when the value of *K-fraction* decreases, even though one should remind these configurations introduce not negligible errors on the BC values. However, the gap between the different curves reduces when considering values of *K-fraction* greater than or equal to 0.4. Moreover, when *K-fraction* is lower than the values above, a higher number of cores does not contribute to significantly improve performance and in some cases the elapsed time remains almost unchanged or even increases as in case of W2C-Fast-BC executed with *K-fraction* $= 0.01$. This behavior suggested a more in depth analysis that we performed by firstly comparing the algorithms with reference to efficiency and exact-fast ratio and then by exploring the behavior of each phase of the algorithm.

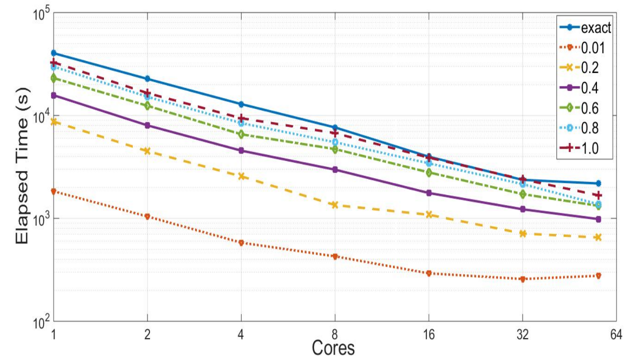

Fig. 8. Elapsed time

B. Efficiency

We analyzed the efficiency of the exact and W2C-Fast-BC algorithms with different values of *K-fraction*, by increasing the number of cores exploited for the execution.

The plot in Fig. 9 shows on the X-axis the number of cores in a base-2 logarithmic scale and on the Y-axis the value of the efficiency. The efficiency can be calculated as:

$$E = \frac{t_s}{(t_p \cdot \#core)} \quad (4)$$

where, for each configuration, t_s represents the elapsed time with one core while t_p is the value of the elapsed time of the multi-core execution. The figure shows that by increasing the number of cores efficiency reduces. This is true for all configurations and also the exact algorithm follows a similar behavior. However, we can clearly observe that W2C-Fast-BC behaves better when *K-fraction* is high. With *K-fraction*$= 0.01$ the efficiency reduction is more evident when the number of cores increases. This result suggests that our approach (in the current implementation) is very effective when the available resources are limited. To better appreciate the speedup boost of the proposed algorithm, an additional analysis was performed in order to compare the computation time of W2C-Fast-BC with reference to the exact solution proposed by Brandes.

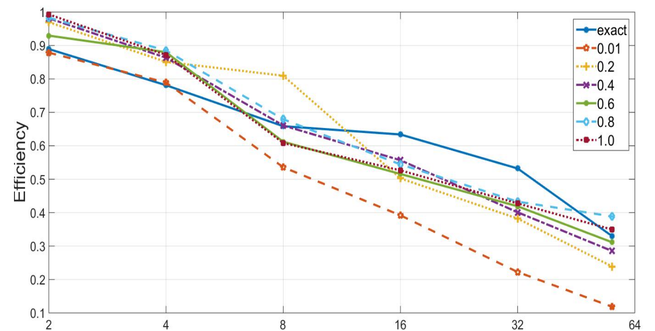

Fig. 9. Efficiency

C. Exact-Fast ratio

We analyzed the ratio, by varying the number of cores, between the execution time of the exact algorithm and that of the W2C-Fast-BC algorithm for different values of *K-fraction*.

The plot in Fig. 10 shows on the X-axis the number of cores in a base-2 logarithmic scale and on the Y-axis the value of the ratio. The figure clearly shows that when *K-fraction* is very low (i.e. 0.01), our approach exhibits an important performance boost with a speedup equal to 22 when the number of cores is two or four. However when the number of cores increases, the observed speedup linearly decreases, reaching seven with 56 cores. With higher values of *K-fraction*, the ratio with the exact version appears to be almost constant by increasing the number of cores. This result is very interesting. With a value of *K-fraction* that ensures tolerable errors, computing time can be reduced by increasing the number of cores as for the Brandes' algorithm. When a small number of resources is available, very fast executions can be still performed by using a small value of *K-fraction* even though higher errors will be observed. Therefore a small number of resources can be used for performing coarse-grained monitoring under normal traffic conditions whereas more accurate monitoring (e.g. for emergency handling) could be performed by allocating a larger number of computing resources.

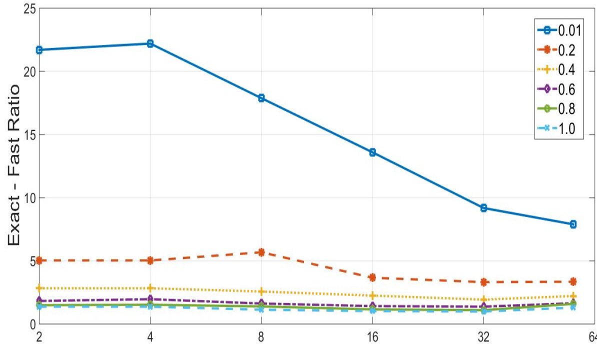

Fig. 10. Exact-Fast ratio

D. Performance breakdown

To understand the reasons of different behaviors when *K-fraction* varies, we performed further tests to identify the parts of the W2C-Fast-BC algorithm that negatively impact on efficiency. To this end, the W2C-Fast-BC algorithm was divided in phases to perform a breakdown analysis and to discover opportunities for possible improvements.

The phases selected for the performance breakdown are:

- **Louvain Time**: time for executing the first level of clustering based on the Louvain algorithm.
- **Local BC + Class Detection time**: time needed to (a) compute the exact value of BC inside each cluster (local BC), obtained from the previous phase, (b) identify the equivalence classes inside each cluster and (c) perform the additional clustering based on K-means to aggregate small classes in wider super-classes and to select the pivots for the last phase of the algorithm.
- **Pivot based BC + final BC reduction time**: time needed for the execution of the Brandes algorithm for each pivot (global contribution) including the one needed to compute the sum of the local and global contributions of BC for each node of the graph.

The following figures report the processing time of each phase on the Y-axis, in a logarithmic scale.

1) 2 Cores: By analyzing the breakdown with 2 cores of the W2C-Fast-BC algorithm (Fig. 11), we can observe that for almost all the values of *K-fraction* (except for *K-fraction* equal to 0.01) the phase that most affects the elapsed time is the one concerning the calculation of pivot-based BC and the successive reduction operation. The largest time is observed when using a *K-fraction*=1 and corresponds to more than 10,000 s. In fact, as expected, this phase requires more time as *K-fraction* increases since a larger number of pivots must be considered for the SSSP exploration of the whole graph. The other two phases take an almost constant processing time when *K-fraction* varies. It is worth to note that even if the Louvain phase is independent from *K-fraction*, a small variability could be observed due to different runs that may adopt different initial communities configurations.

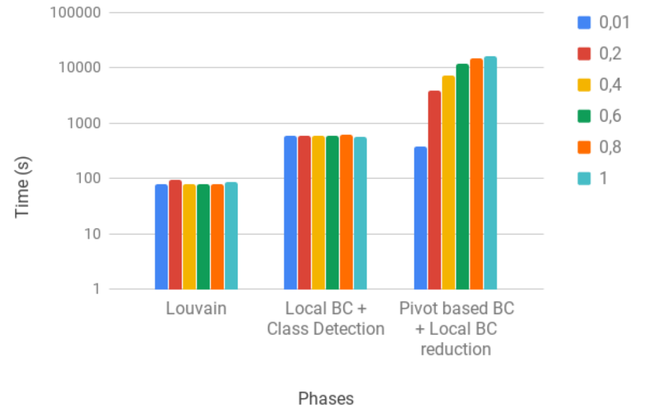

Fig. 11. Performance breakdown with 2 cores

2) 4 cores: The behavior of the three phases of the algorithm with 4 cores (Fig. 12) is similar to the one observed with 2 cores but with an improvement of performances. In fact, the processing time of each phase (varying *K-fraction*) decreases according to the available amount of additional resources (even if the log scale does not contribute to appreciate the almost linear improvement).

As with the previous configuration, the calculation of the BC through the pivots and the reduction operations represent the most significant contribution to the overall time. Both local BC with class detection and Louvain phases benefit from the increment of the available cores, even if the improvement for running the Louvain algorithm is less appreciable.

3) 8 cores: With 8 cores (Fig. 13), the time for running each phase further decreases, following the behavior already observed with 4 cores.

4) 16 cores: When the number of exploited cores is 16, the trend previously observed for the performance breakdown exhibits an important change (Fig. 14). In fact, not all the

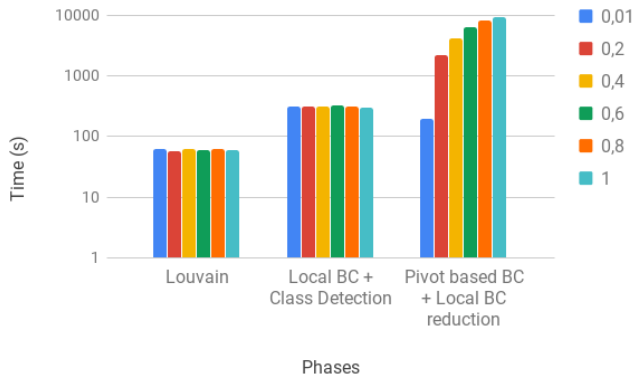

Fig. 12. Performance breakdown with 4 cores

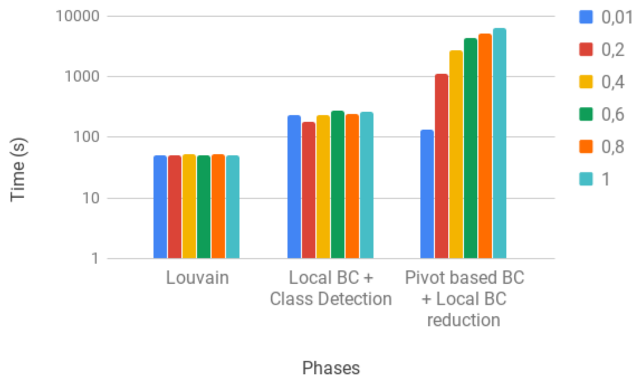

Fig. 13. Performance breakdown with 8 cores

phases continue to benefit from the growing number of cores, since the execution of the Louvain algorithm exhibits almost the same time as in the previous case of 8 cores. This behavior is motivated by the particular implementation of the parallel version of Louvain that needs to exchange the most modular configuration of communities obtained after each iteration. Therefore, the advantage due to the selection of a better configuration for each step is overwhelmed by the need for additional communication and synchronization.

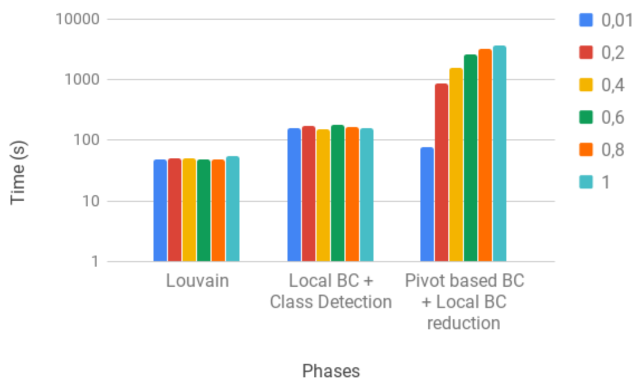

Fig. 14. Performance breakdown with 16 cores

5) 32 cores: With 32 cores, the processing time of the first phase slightly grows (Fig. 15). The second phase takes approximately the same time for each *K-fraction* value, which is around 100s.

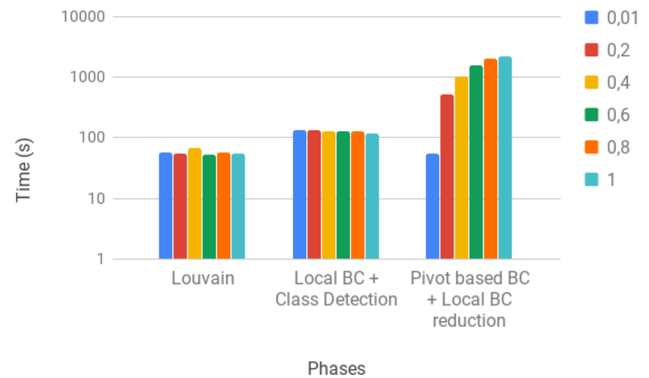

Fig. 15. Performance breakdown with 32 cores

6) 56 cores: When the number of exploited cores reaches 56 (Fig. 16), which is the maximum available number of cores in our hardware configuration, the Louvain phase is executed in much more time than the one spent with 32 cores, since it is now almost the same obtained with only 2 cores.

As concerning the other two phases, while the computation of pivot-based SSSP explorations and reduction continues to benefit from available computing resources, the second phase exhibits a sort of saturation for some values of the *K-fraction* parameter. We observe also that the time needed to perform SSSP explorations from a pivot in case of *K-fraction*$= 0.01$ is lower than the time needed for computing Louvain and class detection algorithms. This behavior motivates the small values of efficiency with very low values of *K-fraction*.

Finally, the figure shows that the third phase strongly benefits from parallelism as in case of exact algorithm, since with this hardware configuration the time needed for computing BC is about 1,000s in the worst case of *K-fraction*=1 compared to about 12,000s with 2 cores.

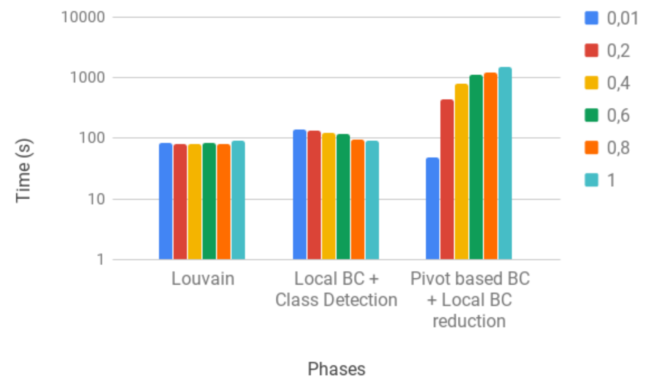

Fig. 16. Performance breakdown with 56 cores

VI. Conclusion

The paper analyzed the performance and the scalability issues of a fast approximated algorithm for computing node betweenness centrality of weighted graphs. The analysis has been performed by using a real transportation network enriched with dynamic weights derived from GPS-taxi traces. The results show that the algorithm is able to find the most critical nodes of a directed and weighted graph with a significant speedup and a negligible error if compared to the exact Brandes' algorithm. The speedup appears to be higher when the amount of computing resources is low, even though it remains relevant when a higher number of cores is used. The in-depth analysis revealed that the efficiency of our approach is comparable to the one exhibited by the exact solution when the values of the *K-fraction* parameter is not too low, since with low values the boost of performance for SSSP explorations is overwhelmed by a higher computation time of the Louvain method executed with a high number of resources.

In the future, we will focus on Louvain clustering optimization to improve the first phase of the algorithm. Additionally, we aim to exploit dynamic resource allocation to avoid useless parallelism that slows-down the performance of the whole algorithm. Moreover, since the performance of the algorithm could depend on the kind of graph, we plan to extend the scalability analysis to graphs with different sizes and topology (e.g., small-world, random networks), related to different application domains, by comparing the results with other approximated approaches.

Acknowledgment

This work has been supported by the French research project PROMENADE (grant number ANR-18-CE22-0008) and by the GAUSS project (MIUR, PRIN 2015, Contract 2015KWREMX)

References

[1] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 25, pp. 35–41, 1997.

[2] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 163, 2001.

[3] A. Furno, N.-E. El Faouzi, R. Sharma, and E. Zimeo, "Fast approximated betweenness centrality of directed and weighted graphs," in *International Conference on Complex Networks and their Applications*. Springer, 2018 to appear.

[4] U. Brandes and C. Pich, "Centrality estimation in large networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, pp. 2303–2318, 2007.

[5] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2008, pp. 90–100.

[6] A. Furno, N.-E. El Faouzi, R. Sharma, and E. Zimeo, "Reducing pivots of approximated betweenness computation by hierarchically clustering complex networks," in *International Conference on Complex Networks and their Applications*. Springer, 2017, pp. 65–77.

[7] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215 – 239, 1978.

[8] D. R. White and S. P. Borgatti, "Betweenness centrality measures for directed graphs," *Social Networks*, vol. 16, no. 4, pp. 335 – 346, 1994.

[9] D. A. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 2006, pp. 539–550.

[10] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[11] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung, "Qube: A quick algorithm for updating betweenness centrality," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 351–360.

[12] T. Hayashi, T. Akiba, and Y. Yoshida, "Fully dynamic betweenness centrality maintenance on massive networks," *Proc. VLDB Endow.*, vol. 9, no. 2, pp. 48–59, Oct. 2015.

[13] M. Riondato and E. M. Kornaropoulos, "Fast approximation of betweenness centrality through sampling," *Data Mining and Knowledge Discovery*, vol. 30, no. 2, pp. 438–475, 2016.

[14] E. Bergamini, H. Meyerhenke, and C. L. Staudt, "Approximating betweenness centrality in large evolving networks," in *17th Workshop on Algorithm Engineering & Experiments*, ser. ALENEX '15. Philadelphia, PA, USA: SIAM, 2015, pp. 133–146.

[15] E. Bergamini and H. Meyerhenke, "Approximating betweenness centrality in fully dynamic networks," *Internet Mathematics*, vol. 12, no. 5, pp. 281–314, 2016.

[16] N. Kourtellis, G. D. F. Morales, and F. Bonchi, "Scalable online betweenness centrality in evolving graphs," *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 1580–1581, 2016.

[17] S. White and P. Smyth, "Algorithms for estimating relative importance in networks," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 266–275.

[18] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*, ser. WAW'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 124–137.

[19] K. Ohara, K. Saito, M. Kimura, and H. Motoda, *Accelerating Computation of Distance Based Centrality Measures for Spatial Networks*.

[20] M. J. Newman, "A measure of betweenness centrality based on random walks," *Social Networks*, vol. 27, no. 1, pp. 39 – 54, 2005.

[21] M. Borassi and E. Natale, "KADABRA is an adaptive algorithm for betweenness via random approximation," in *ESA*, ser. LIPIcs, vol. 57. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 20:1–20:18.

[22] M. H. Chehreghani, A. Bifet, and T. Abdessalem, "Efficient exact and approximate algorithms for computing betweenness centrality in directed graphs," *arXiv:1708.08739*, 2017.

[23] A. Furno, N.-E. El Faouzi, R. Sharma, and E. Zimeo, "Two-level clustering fast betweenness centrality computation for requirement-driven approximation," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1289–1294.

[24] P. Suppa and E. Zimeo, "A clustered approach for fast computation of betweenness centrality in social networks," in *2015 IEEE International Congress on Big Data*, June 2015, pp. 47–54.

[25] Sotera, "dga-graphx: Graphx algorithms," *online*. [Online]. Available: https://github.com/Sotera/spark-distributed-louvain-modularity

[26] M. E. Newman, "Analysis of weighted networks," *Physical Review E*, vol. 70, no. 5, p. 056131, 2004.

[27] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, p. 026113, Feb 2004.

[28] E. A. Leicht and M. E. J. Newman, "Community Structure in Directed Networks," *Physical Review Letters*, vol. 100, no. 11, pp. 118 703+, Mar. 2008.

[29] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[30] N. Dugu‘e and A. Perez., "Directed louvain: maximizing modularity in directed networks," in *PhD Thesis, gUniversite dOrlans*, 2015.

[31] A. Furno, N.-E. El Faouzi, R. Sharma, E. Zimeo *et al.*, "Fast computation of betweenness centrality to locate vulnerabilities in very large road networks," in *Transportation Research Board 97th Annual Meeting*, 2018.

[32] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.