World Wide Web
https://doi.org/10.1007/s11280-018-0605-y

Parallel computation of hierarchical closeness centrality and applications

Hai Jin[1] · Chen Qian[1] · Dongxiao Yu[1] · Qiang-Sheng Hua[1] · Xuanhua Shi[1] · Xia Xie[1]

Received: 15 December 2017 / Revised: 4 April 2018 / Accepted: 25 May 2018

Abstract It has long been an area of interest to identify important vertices in social networks. Closeness centrality is one of the most popular measures of centrality of vertices. Generally speaking, it measures how a node is close to all other nodes on average. However, closeness centrality measures the centrality from a global view. Consequently, in real-world networks that is normally composed by some communities connected, using closeness centrality may suffer from the flaw that local central vertices within communities are neglected. To resolve this issue, we propose a new centrality measure, Hierarchical Closeness Centrality (HCC), to depict the local centrality of vertices. Experiments show that comparing with closeness centrality, HCC is a better index in finding most influential vertices and community detection. Furthermore, we present a parallel algorithm for HCC computation,

This article belongs to the Topical Collection: *Special Issue on Social Computing and Big Data Applications*
Guest Editors: Xiaoming Fu, Hong Huang, Gareth Tyson, Lu Zheng, and Gang Wang

✉ Dongxiao Yu
dxyu@hust.edu.cn

Hai Jin
hjin@hust.edu.cn

Chen Qian
M201572720@hust.edu.cn

Qiang-Sheng Hua
qshua@hust.edu.cn

Xuanhua Shi
xhshi@hust.edu.cn

Xia Xie
shelicy@hust.edu.cn

[1] Services Computing Technology and System Lab, Big Data Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan 430074, People's Republic of China

Published online: 19 June 2018

by well analyzing the independence between vertices in the computation procedure. Extensive experiments on real-world datesets demonstrate that the parallel algorithm can greatly reduce the computation time compared to trivial algorithms.

Keywords Data mining · Closeness centrality · Parallel algorithm

1 Introduction

In the social network, a crucial problem is to identify nodes that are most central. The most central nodes can be for example those that are traversed by a large fraction of shortest paths [4], those that can quickly reach the rest of the graph or the ones that recur more often in random walks [17]. Many centrality indices have been proposed, such as Betweeness Centrality [10], Closeness Centrality [15], and PageRank [7]. Among these indices, closeness centrality is a widely-used centrality measure. Formally, given a graph $G = (V, E)$ to represent the network and a distance-decay (monotone decreasing) $f : \mathbb{N} \rightarrow \mathbb{R}^+$, the Generalized Closeness Centrality (GCC) [15] of a vertex u, denoted as $C_c(u)$, is defined as

$$C_c(u) = \sum_{v \in V \setminus \{u\}} f(d(u, v)) \tag{1}$$

Intuitively, GCC measures how the node is close on average to all the other nodes in the network. Hence, nodes with higher GCC means they are more important regarding centrality.

But in social networks, the central vertex is not unique. In most instances, a social network is composed of some clusters (also called communities) connected. Hence, in many applications, it is more vital to identify locally central nodes in each community. Unfortunately, this might not be accomplished using GCC, since GCC measures centrality globally, and local centrality cannot be reflected by this index. Take the network in Figure 1 as a simple example. The most central nodes identified by GCC is the green one and the red one, but the blue one will be ignored, even if it is a central node in its community.

In this work, we propose a centrality measure, called Hierarchical Closeness Centrality(HCC), which can reflect the local nature of communities. In principle, by removing the most central nodes in each layer, HCC measures the connectivity of a node with others that have equal or smaller closeness centrality. Our contributions are summarized as follows.

- We propose a local centrality measure HCC, to complement the default that GCC cannot reflect local centrality within communities. Experiments on real datasets show that compared with GCC, HCC can be used to select more influential vertices and better detect communities.

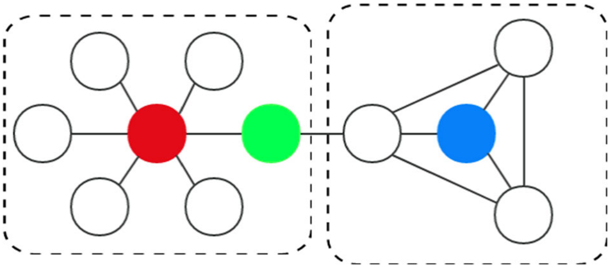


Figure 1 The central nodes in a network

- By analyzing sufficient conditions for independently reconnecting vertices in a BFS tree after deleting some vertices, we give a parallel algorithm for computing HCC. Extensive experiments on real datasets show that our parallel algorithm can greatly reduce the computation time of HCC, compared with the natural one obtained by the definition of HCC.

2 Related work

Centrality is a fundamental tool in the study of social networks. Many centrality measures, such as closeness centrality [4], betweenness centrality [10] and Pagerank [7], have been proposed. Among these measures, closeness centrality is a widely adopted one.

Closeness centrality can be computed by solving the APSP problem. For this problem, there is not known solution that is always better than running a BFS from each vertex. But these BFS-based algorithms take $O(n(m+n))$ time, which may be unacceptable for large networks. So there have been many efforts on deriving approximate solutions. Eppstein and Wang [9] proposed an approximate algorithm using sampling. With that the time complexity is reduced to $\Theta(m + n * \log n)$ and an error bound is provided by applying Hoeffding's inequality. Brandes and Pich [6] conducted an experimental evaluation of this approximation algorithm, considering different ways of sampling the source vertices. Okamoto [18] also proposed a similar approximate algorithm. A more refined approximation algorithm with better practical performance has recently been given in [15]. Although approximation algorithms can often offer solutions that are close to the real ones, they may fail in preserving the ranking, in particular for vertices with similar closeness.

Another line of research is on exactly computing the ranking of the top-k vertices with the highest closeness centrality, exactly with high probability guarantee [19] or heuristic [2, 5, 20]. Although these algorithms can actually save time compared to the exhaustive computation of closeness centrality for all vertices, it was shown that in many instances their running time is very close to that of APSP.

Recently, there are increasingly many works scaling centrality calculations by distributing the computation using MapReduce. For example, Kang [16] developed a parallel graph mining tool to estimate single node centrality on Hadoop. Oktay [8] presented a method to estimate pair-wise nodes shortest distance using MapReduce. Sariyuce [22] presented a distributed framework for calculating closeness centrality incrementally over dynamic graphs.

3 Hierarchical closeness centrality

We are given a undirected graph $G = (V, E)$, where V and E denote the vertex and edge sets respectively. Without loss of generality, graph G is assumed to be connected. For each pair of vertices $u, v \in V$, let $d(u, v)$ denote the distance between them, i.e., the length of the shortest path between u, v. Let $N(v)$ be the set of neighbors of v in G and $N[v] = N(v) \cup \{v\}$.

The hierarchical closeness centrality of vertices is defined hieraichically. Basically, the HCC of vertices are defined as follows:

- In each step, the vertices in the current graph that have the largest closeness centrality are identified, denoted as V^*;

- The HCC of vertices in V^* is defined as their closeness centrality;
- Delete V^* and edges connected with vertices in V^*, and repeat the above process, until the HCC of all vertices are computed.

Clearly, from the above computation process, we can see that HCC measures the centrality of a vertex in the network that are constituted by vertices with similar or poorer centralities.

We next define HCC formally. In particular, for $L \geq 0$, denote by G_L the graph obtained after deleting $L-1$ layers of vertices with large HCC, and let V_L^* be the set of vertices that have the largest closeness centrality in G_L. Furthermore, we use E_L^* to denote the set of edges connected with vertices in V_L^*. $G_L = (V_L, E_L)$ can be formally defined as follows

$$G_L = \begin{cases} G & L = 0, \\ G_{L-1} \setminus (V_{L-1}^*, E_{L-1}^*) & L \geq 1. \end{cases}$$

For two vertices u, v in G_L, denote by $d_L(u, v)$ the distance in G_L. From above definitions, $\{V_L^*\}$ constitute a division of vertices in V. Then we define the HCC of each vertex.

Definition 1 (Hierarchical Closeness Centrality, HCC) Given a connected graph $G = (V, E)$, the hierarchical closeness centrality of a vertex $u \in V_L$ for $L \geq 0$ is defined as

$$C_H(u) = \sum_{v \in V_L \setminus \{u\}} f(d_L(v, u))$$

4 Applications: HCC VS. GCC

Using hierarchical closeness centrality, one can gain a deeper understanding of the hierarchical structure of a data graph. We next show that the HCC can replace or supplement the generalized closeness centrality in some sense, by illustrating the comparison of these two indexes in two applications: maximum influence vertices identification and community detection.

4.1 Maximum influence vertices identification

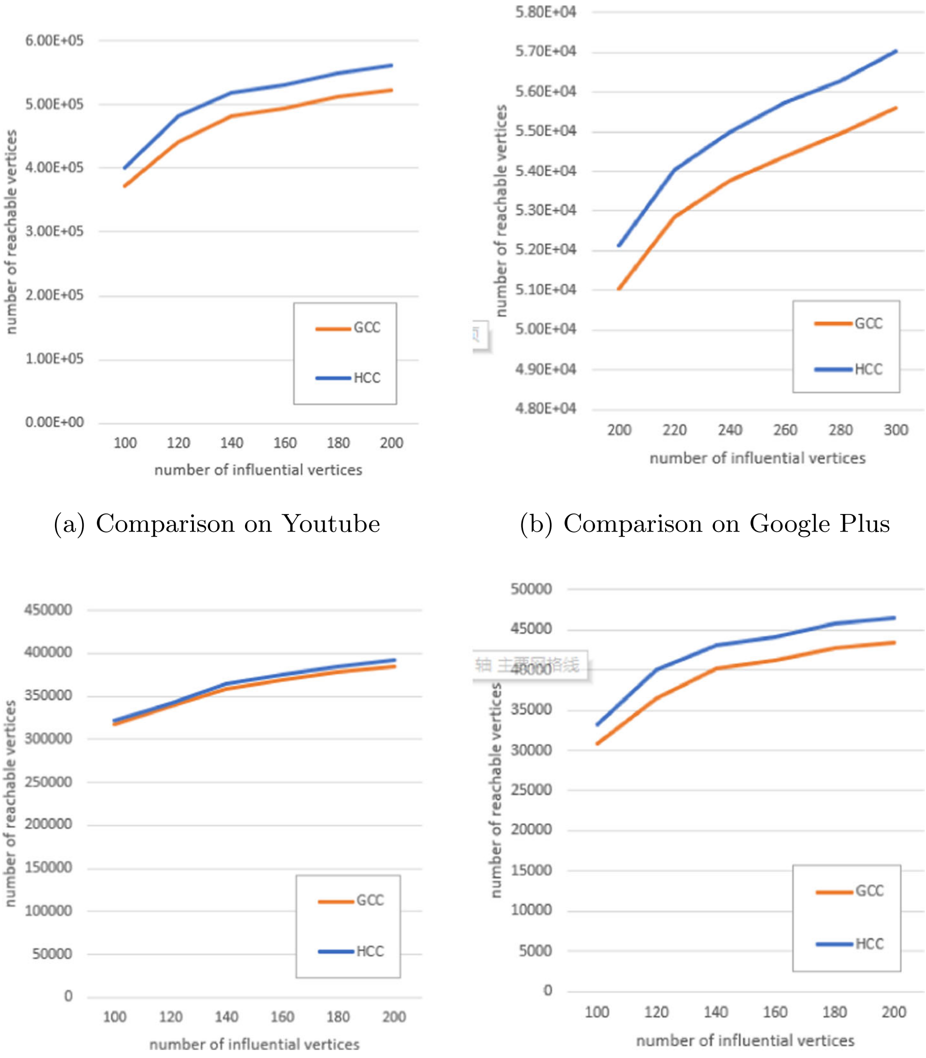
The maximum influential vertices identification has been extensively studied, due to its wide applications with approaches developed in network analysis [13, 23]. A natural measure for influence of a vertex is the number of vertices that it can reach. The closeness centrality is a commonly used index to find the most influential vertices.

To compare the effect of GCC and HCC on finding the maximum influence vertices, we fix the number of the most influential vertices that are needed to find, and compare the influence of these vertices by the number of vertices they can reach. The experiments are implemented on four datasets YT (Youtube), GP (Google Plus) DB(DBLP) and Sl (soc-Slashdot) given in Table 1.

The comparison results on the four datasets are shown in Figure 2. In the figures, the x-axis represents the specified number of most influential vertices, and the y-axis represents the number of reachable vertices from selected influential vertices. From the figures, we can

Table 1 DataSets

Datasets	$n = \|V\|$	$m = \|E\|$
DB(DBLP)	0.31M	1.01M
YT(Youtube)	1.13M	5.97M
Sl(soc-Slashdot)	82.1K	500.5K
GP(Google Plus)	82.1K	500.5K



(a) Comparison on Youtube

(b) Comparison on Google Plus

(c) Comparison on DBLP

(d) Comparison on soc-Slashdo

Figure 2 Comparisons of HCC and GCC on finding most influential vertices

see that comparing with GCC, with the same specified number of influential vertices, the influential vertices found using HCC can reach more vertices. Hence, using HCC can find vertices that are more influential.

4.2 Community detection

Community detection is one of the pivotal tools for understanding the underlying structure of complex networks and extracting useful information from them. It has been widely used in fields such as biology [21], economics [3], human mobility [14], communications [11], and scientific collaborations [12].

We compare effects of GCC and HCC as indexes in communicty detection. In particular, we use the modularity to measure the effect of community detection. The modularity is a widely used index for measuring the strength of division of a network into communities. High modularity means in the division, the nodes have dense connections within communities and sparse connections with nodes in different communities.

The comparison results are given in Table 2. Comparing the results in Table 2, in all of datasets using HCC in community detection gets higher modularity than using GCC.

5 Natural algorithm for HCC computation

By the definition, we can get a natural algorithm for computing HCC of vertices, by iteratively computing the closeness centrality of vertices in the graph obtained by deleting vertices with larger HCC. The algorithm is given in Algorithm 1, and it is easy to see that the algorithm can correctly compute the HCC of each vertex. However, Algorithm 1 incurs a high complexity in running time. In the subsequent section, by investigating the parallelism in maintaining the BFS tree during the process of hierarchical computation of HCC, we propose a more efficient parallel algorithm for HCC computation.

Algorithm 1 Natural algorithm for HCC computation

```
Input: $G(V, E)$
Output: $C_H(v), v \in V$
$L = 0, V_L = V, E_L = E$;
while $V_L \neq \emptyset$ do
    foreach vertex $v \in V_L$ do
        foreach vertex $u \in V_L, u \neq v$ do
            compute $d_L(u, v)$;
            $C_c(v) += f(d(u, v))$;
    $C_c = \max\{C_c(v) : v \in V_L\}$;
    $V_L^* = \{v : v \in V_L \wedge C_c(v) = C_c\}$;
    foreach vertex $s \in V_L^*$ do
        $C_H(s) = C_c(s)$;
    $V_{L+1} = V_L \setminus V_L^*; E_{L+1} = E_L \setminus E_L^*$;
    $G_{L+1} = (V_{L+1}, E_{L+1})$;
    $L += 1$;
```

Table 2 Modularity of community detection using GCC and HCC

Datasets	$n = \|V\|$	$m = \|E\|$	GCC	HCC
DB(DBLP)	0.31M	1.01M	0.406563	0.417814
YT(Youtube)	1.13M	5.97M	0.519832	0.540152
Sl(soc-Slashdot)	82.1K	500.5K	0.915008	0.943071
GP(Google Plus)	82.1K	500.5K	0.528459	0.551347

6 Parallel algorithm for HCC computation

In this section, based on some key observations, we propose a parallel algorithm to accelerate the computation of HCC.

The most costly part of the computation of HCC is to calculate the distance of a vertice with others. Our basic idea is to construct BFS trees for each vertex, and update the BFS trees after the deletion of vertices in each layer. With some key observations on the independence of reconnecting vertices on a BFS tree, a parallel approach for updating each BFS tree will be proposed.

In subsequence, we first discuss the independence of vertices, i.e., the reconnection on a BFS tree can be processed independently. In this case, the reconnection of these vertices on a BFS can be processed in parallel. Based on some key observations on independence, we propose our parallel algorithm.

6.1 Theoretical basis

We next present some theoretical results that constitute the basis of our parallel algorithm. We will first discuss the impact on the BFS tree reconnection when a vertex is deleted, and then define the independence of reconnecting of two vertices in a BFS after some vertices are deleted. Based on these discussions, we will prove some sufficient conditions for independence of reconnecting vertices in a BFS tree.

We assume that BFS trees are constructed for each vertex. Speifically, the BFS tree rooted at a vertex r is denoted as $T(r)$. The root r is at level 0 in $T(r)$, and vertices with distance i from r are at level i. The following property is a straightforward result of BFS tree.

Property 1 In a BFS tree $T(r)$ and for each vertex v at level $i > 0$, it may only be possible to be adjancent to vertices at levels $i - 1$, i and $i + 1$.

Based on the above property, in a BFS tree $T(r)$, we divide the neighbors of v at level $i > 0$ into three classes $\alpha(v)$, $\beta(v)$ and $\gamma(v)$. Speicifically, these three sets denote the sets of neighbors of v that are at level $i - 1$, i and $i + 1$ respectively. Furthermore, we use $des_r(v)$ to denote the set of vertices in the subtree of $T(r)$ that is rooted at v, and let $dom_r(v) = des_r(u) \cup \{u\}$.

We next consider the reconnection of vertices on a BSF tree $T(r)$ after some vertices are deleted. Clearly, when a vertex u is deleted, only vertices in $des_r(u)$ that are not deleted need to reconnect to the BFS tree during the update procedure. So in the following, for a BFS tree $T(r)$, after a set of vertices V^* are deleted, we consider the reconnection of vertices in $des_r(V^*) = \cup_{u \in V^*} des_r(u)$. In subsequence, all notations are defined on the left vertex set after the vertex deletion.

To analyze parallelly processing the reconnections of vertices, we define independence of vertices and subtrees as follows.

Definition 2 (Independence of Vertices) In the maintenance procedure of a BFS tree $T(r)$, if the reconnections of a vertex u and a vertex v can be processed in parallel, i.e., the reconnections of these two vertices do not interact with each other, then we say these two vertices are independent.

Definition 3 (Independence of Subtrees) In the maintenance procedure of a BFS tree $T(r)$, if the reconnections of vertices in two subtrees can be processed in parallel, i.e., the reconnections of vertices in these two subtrees do not interact with each other, then we say these two subtrees are independent.

In the following, we analyze the independence of subtrees, such that we can get sufficient conditions for processing reconnections of vertices on subtrees in parallel. In subsequence, for a vertex v at level i, we use $\alpha'(v)$ to denote the set of v's neighbors u satisfying: (i) $u \in \alpha(v)$; and (ii) all vertices in the path on $T(r)$ between the root r and u are not deleted.

Speicifically, there are two cases for a vertex $v \in des_r(V^*)$: $\alpha'(v) \neq \emptyset$ and $\alpha'(v) = \emptyset$. At first, we consider the case that $\alpha'(v) \neq \emptyset$. In this case, v can be reconnected to another vertex except its deleted parent, such that the length of the shortest path between r and each vertex in the subtree rooted at v does not change. Then we have the following result.

Lemma 1 *In a given graph $G(E, V)$ and a BFS tree $T(r)$, after a vertex u was deleted from G, if for a vertex $v \in des_r(u)$, $\alpha'(v) \neq \emptyset$, the update of reconnection of vertices in the subtree rooted at v can be accomplished by just connecting v to a neighbor in $\alpha'(v)$ and keeping the subtree rooted at v stable.*

Proof Assume that v is at level i. By the BFS construction process, we know that in $T(r)$ the path between r and any vertex belong to $T(r)$ is a shortest path between them. So the path between r and each vertex in $\alpha'(v)$ is still the shortest path. Since the ditance between r and any other vertex that is not deleted cannot decrease after the vetex deletion. Hence, each vertex in the subtree rooted at v cannot decrease the levels they locate in the new BFS tree. Clearly, by connecting v to a vertex in $\alpha'(v)$, the levels of vertices in $dom_r(v)$ keeps the same with their levels at $T(r)$. This means that the update is correct, which will not break the structure of the BFS tree. □

By the definition of independence, it can easily to see that if a vertex v with $\alpha'(v) \neq \emptyset$, the subtree rooted at v is independent with other subtrees, which is formally given in the following Lemma.

Lemma 2 *If a vertex v has $\alpha'(v) \neq \emptyset$, the subtree rooted at v is independent with other subtrees.*

But in the case of $\alpha'(v) = \emptyset$, the reconnection becomes much more complicated, as its and its neighbors' reconnections may interact each other. We next present two sufficient conditions that can guarantee independence of reconnection. For a vertex u, let $D(u) = \cup_{w \in dom_r(u)} N[v]$.

Lemma 3 *Consider the update of a BFS tree $T(r)$ on a graph G. For two vertices u and v, if $D(u) \cap dom_r(v) = \emptyset$ and $D(v) \cap dom_r(u) = \emptyset$, then the subtrees rooted at u and v are independent.*

Proof For the reconnection of each vertex w in the subtree rooted at u, it can be connected only to its neighbors. By the condition, it means that w's neighbor set is disjoint with $dom_r(v)$. Hence, the reconnection of vertices in the subtree rooted at u do not rely on the reconnection of vertices in the subtree rooted at v. Similarly, it can be shown that the reconnection of vertices in the subtree rooted at v also do not rely on the reconnection of vertices in the subtree rooted at u. Hence, the subtrees rooted at u and v are independent. □

For a vertex u, the largest level of vertices in $dom_r(u)$ locate in the tree $T(r)$ is called the depth of $dom_r(u)$, denoted as dep_u, and the level of u is called the peak of $dom_r(u)$, denoted as $peak_u$. We consider two vertices u and v. Without loss of generality, we assume that the level of u is larger than that of v.

Lemma 4 *Consider a graph G and a BFS tree $T(r)$ rooted at vertex r. For two vertices u and v, if $peak_u - dep_v > 2$, then the subtrees rooted at u and v are independent.*

Proof By Lemma 1, for a vertex w at level i in $T(r)$, its neighbors locate at levels $i-1$, i and $i+1$. Hence, if $peak_u - dep_v > 2$, this means that $D(u) \cap dom_r(v) = \emptyset$ and $D(v) \cap dom_r(u) = \emptyset$. Then by Lemma 3, the result is proved. □

6.2 Parallel algorithm for BFS tree maintenance

Based on above observations on the independence of vertices and subtrees in the reconnection procedure of a BFS tree, we present a parallel algorithm for computing HCC. The basic idea is as follows. BFS trees rooted at each vertex are used to compute the distance between each node and others in each layer. After deleting vertices whose HCC have been computed in each layer, each BFS tree is updated by reconnecting independent vertices and subtrees in parallel. Notice that we here give a parallel algorithm for updating a particular BFS tree, not updating different BFS trees in parallel. Clearly, parallelly updating different BFS trees is trivial.

The algorithm for computing HCC is given in Algorithm 2. Initially, BFS trees rooted at each vertex are constructed. The layer number L is set as 0 and the set of survival vertices is set as $V' = V$. The states of all vertices in each BFS tree is set as $certain$.

At the first step, the GCC of each vertex is computed by the definition. The HCC of vertices that have the largest GCC is set as the value of GCC. Then those vertices and connected edges are deleted from the graph. In each of the subsequent iterations, the procedure is repeated, until every vertex gets its HCC. The main costly part of the algorithm is the maintenance of the BFS trees. We introduce a parallel algorithm to speed up the procedure, as given in Algorithm 3.

Algorithm 2 Parallel HCC computation

```
Input: $G(V, E)$
Output: $C(v), v \in V$
$V' = V, E' = E$;
foreach $v, v \in V$ do
    construct $T(v)$;
    foreach $u, u \in V, u \neq v$ do
        calculate the distance $d(v, u)$;
        if $d(v, u) \neq \infty$ then
            $C(v) += f(d(v, u))$;
search these vertices to get $V^*$;
get $E^*$ in $G(V, E)$;
$V' -= V^*$,$E' -= E^*$;
foreach $v \in V^*$ do
    $C_h(v) = C(v)$;
while $V' = \emptyset$ do
    foreach $v', v' \in V'$ do
        $C(v) = 0$;
        maintain($V^*$,$T(v')$,$G'(V', E')$);
        foreach $u', u' \in V', u' \neq v'$ do
            if $d(v', u') \neq \infty$ then
                $C(v) += f(d(v', u'))$;
        search these vertices to get $V^*$;
        get $E^*$ in $G'(V', E')$;
        $V' -= V^*$,$E' -= E^*$;
```

In Algorithm 3, the basic idea is as follows: after deleting some vertices after a layer, all vertices whose parents are deleted and their descendants are set as *uncertain*; then independent vertices and subtrees among uncertain vertices are first found using sufficient conditions given before; finally the reconnection of these vertices and subtrees are assigned to different threads such that they can be processed in parallel.

Specifically, we first delete vertices that are unreachable from the root r. Then vertices are divided into two categories by the condition of $\alpha'(v) \neq \emptyset$ or not. By Lemma 2, these subtrees rooted at vertices with $\alpha'(v) \neq \emptyset$ are independent with others. The determination on whether a vertex v satisfies the condition $\alpha'(v) \neq \emptyset$ is given in Algorithm 4.

For vertices with $\alpha(v) = \emptyset$, we need to use Lemma 4 and Lemma 3 to determine the independence between these vertices and subtrees rooted at these vertices. The detailed algorithms are given in Algorithms 6 and 7. In particular, in Algorithm 6, by computing the depth of each subtree, we can obtain the independence of subtrees that can be determined by Lemma 4. And in Algorithm 7, by computing the neighborhoods of vertices in subtrees, independent subtrees are found using the condition given in Lemma 3.

After all independent vertices and subtrees are found, these independent vertices and subtrees are assigned to different threads. In each thread, the assigned vertices or subtree are reconnected to the BFS tree, using Algorithm 8, in which the vertices in each subtree are traversed and each vertex is reconnected to one of its neighboring vertices that are in the state of *certain* and at the smallest level.

Algorithm 3 Maintain(V^*,$T(v')$,$G'(V', E')$)

```
Input: $V^*$,$T(r)$,$G'(V', E')$
create queue $q$,$Fall$;
$q.push(r)$;
while $q \neq \emptyset$ do
    $n = q.front()$;
    $q.pop()$;
    if $n \in V^*$ then
        foreach $l, l \in n.child$ do
            $l.parent = NULL$;
            $cert(l) = false$;
    else if $cert(n.parent) = false$ then
        $cert(n) = flase$;
        $Fall.push(n)$;
    foreach $m$,$m$ is child of $n$ do
        $q.push(m)$;
$id = 0$;
$char(Fall, T(r))$;
$alink(Fall, T(r), G'(V', E'))$;
$dep - ind(Fall, T(r), id)$;
if $completed = false$ then
    $scope - ind(Fall)$;
```

Algorithm 4 $alink(Fall, T(r), G'(V', E'))$

```
Input: $V^*$,$T(r)$,$G'(V', E')$
Output: $T(r)$ after the deletion of $V^*$
$Tra = Fall$;
while $Tra \neq \emptyset$ do
    $a = Tra.front()$;
    $Tra.pop()$;
    if $\exists$ $b$,$b \in adj(a)$,$d(r, b) < d(r, a)$, $cert(b) = true$ then
        $a.parent = b$;
        insert $a$ into $b.child$;
        create queue $p$;
        $p.push(a)$;
        while $p \neq \emptyset$ do
            $n = p.front()$;
            $p.pop()$;
            $cert(n) = true$;
            remove $n$ from $Fall$;
            foreach $m$,$m \in n.child$ do
                $p.push(m)$;
```

Algorithm 5 $char(Fall, T(r))$

```
Input: $Fall, T(r), G'(V', E')$
$Tra = Fall$;
while $Tra \neq \emptyset$ do
    $a = Tra.back()$;
    $Tra.drop()$;
    $dom(a) = adj(a) + a$;
    $a.dep = d(r, a)$;
    if $a.child = \emptyset$ then
        $c = a$;
        while $c.parent \neq NULL$ do
            $b = c.parent$;
            if $c.dep > d(r, b)$ then
                $b.dep = c.dep$;
            $dom(b) = adj(b) + dom(c)$;
            $c = b$;
```

Algorithm 6 $dep - ind(Fall, T(r), id)$

```
Input: $Fall, T(r), G'(V', E'), id$
$Tra = Fall$;
while $Tra \neq \emptyset$ do
    $a = Tra.front()$;
    $Tra.pop()$;
    creat queue $ind_{id}$;
    $cert(ind_{id}) = true$;
    $ind_{id}.push(a)$;
    $b = Tra.frout()$;
    while $d(r, b) <= a.dep + 2$ do
        $cert(ind_n) = flase$;
        $ind_{id}.push(b)$;
        $Tra.pop()$;
    $id = id + 1$;
foreach $ind_p, p <= id$ do
    if $certain(ind_p) = true$ then
        create a thread;
        execute relink($ind_p$);
    else
        $completed = flase$;
```

Algorithm 7 $scope - ind(Fall)$

```
Input: $V^*$,$T(r)$,$G'(V', E')$
Output: $T(r)$ after the deletion of $V^*$
$p = 0$;
while $p <= id$ do
    if $cert(ind_p) = false$ then
        $a = ind_p.front(), exist = false$;
        foreach $b, b \in ind_p, b \neq a$ do
            if $dom(a) \cap dom(b) = \emptyset$ then
                $ind(b) = exist = true$;
            else if $\nexists\ v, cert(ind_{id}) = false, v \in dom(a) \cap dom(b)$ then
                $ind(b) = exist = true$;
        if $exist = true$ then
            $id = id + 1$;
            creat queue $ind_{id}$;
            $n = 0$;
            foreach $b, b \in ind_p, b \neq a$ do
                if $ind(b) = true$ then
                    $ind_{id}.push(b)$;
                    $n = n + 1$;
            if $n > 1$ then
                $cert(ind_i d) = false$;
            else
                $cert(ind_i d) = true$;
        $cert(ind_p) = true$;
    $p = p + 1$;
foreach $ind_s, s <= id$ do
    create a thread to execute $relink(ind_s)$;
```

Table 3 Graph datasets

Datasets	$n = \|V\|$	$m = \|E\|$
AP(ca-Astropg)	18.7K	198.1K
Sl(soc-Slashdot)	82.1K	500.5K
DB(DBLP)	0.31M	1.01M
YT(Youtube)	1.13M	5.97M
WT(wiki-Talk)	2.4M	9.3M
BS(web-BerkStan)	0.68M	13.3M
LJ(live-Journal)	4.0M	34.7M

Algorithm 8 $relink(ind_s)$

```
Input: $ind_x\{des(v_0), des(v_1), des(v_2)\ldots\}$
Output: $T(r)$ relink with $v, des(v) \in ind_x\{\}$
while $ind_x \neq \emptyset$ do
    $Tra = ind_x$;
    while $Tra \neq \emptyset$ do
        $v = Tra.front()$;
        $Tra.pop()$;
        if $\exists a, a \in adj(v), cert(a) = true$ then
            $v.parent = a$,$a.child = a.child + v$;
            $cert(v) = true$;
            $Tree.push(v)$;
            while $Tra \neq \emptyset$ do
                $b = Tree.front()$;
                $Tree.pop()$;
                foreach $c, c \in Adj(b), certain(c) = flase$ do
                    $c.parent = b$,$a.child = a.child + v$;
                    $cert(c) = true$;
                    $Tree.push(c)$;
                    remove $c$ from $ind_x$;
```

7 Experimental studies

In this section, we conduct empirical studies to evaluate the performances of our proposed algorithms. Throughout all of the experiments, the distance-decay function $f(d(u, v))$ was set to $1/d(u, v)$.

We perform evaluations as follows.

- We first evaluate the efficiency of our parallel algorithms on real-world graphs, by comparing with the natural algorithm. The datasets used are six real-world graphs

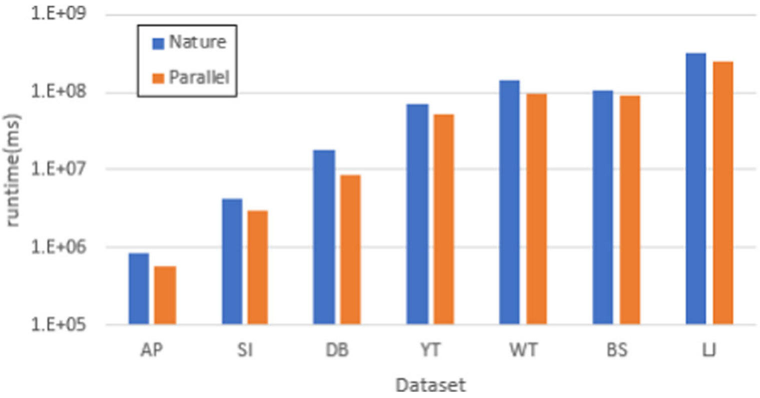


Figure 3 Runtime of our parallel algorithm on different datasets

Table 4 Runtime of natural algorithm and parallel algorithm

Datasets	$n = \|V\|$	$m = \|E\|$	Natural	Parallel
AP(ca-Astropg)	18.7K	198.1K	855847	584853
Sl(soc-Slashdot)	82.1K	500.5K	4327148	2966559
DB(DBLP)	0.31M	1.01M	18124961	8797673
YT(Youtube)	1.13M	5.97M	70485762	52287903
WT(wiki-Talk)	2.4M	9.3M	142898453	96183489
BS(web-BerkStan)	0.68M	13.3M	106283712	90605892
LJ(live-Journal)	4.0M	34.7M	324578615	246296702

of different sizes, including social network graphs (Youtube, soc-Slashdot LiveJournal), collaboration network graphs (DBLP, ca-astroph), communication network graphs (WikiTalk) and Web graphs (web-BerkStan). Table 3 summarizes basic information of these six graphs. Then we evaluate our parallel algorithm using synthetic graphs, by changing the sizes of synthetic graphs.

- We also evaluate the scalability of our parallel algorithm by changing the number of threads used.
- We finally evaluate the key factors that may impact the performance of the parallel algorithm.

All experiments are conducted on a Linux machine with Intel Xeon CPU E5-4655@3.20 GHz and 32 GB main memory, implemented in C++ and compiled by g++ compiler. If without specified, the number of threads used in our experiments is 16.

7.1 Performance evaluation

We compare the performances of the parallel algorithm and the natural one on real-world graphs. The evaluation results are shown in Figure 3 and Table 4. It can be seen that

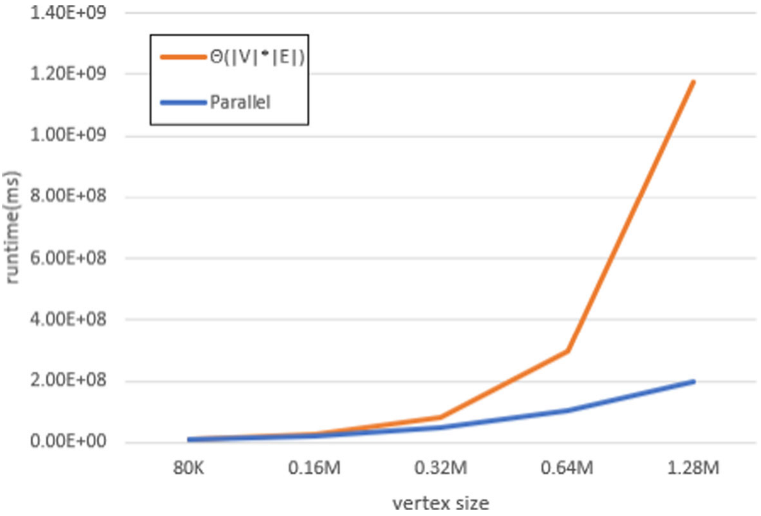
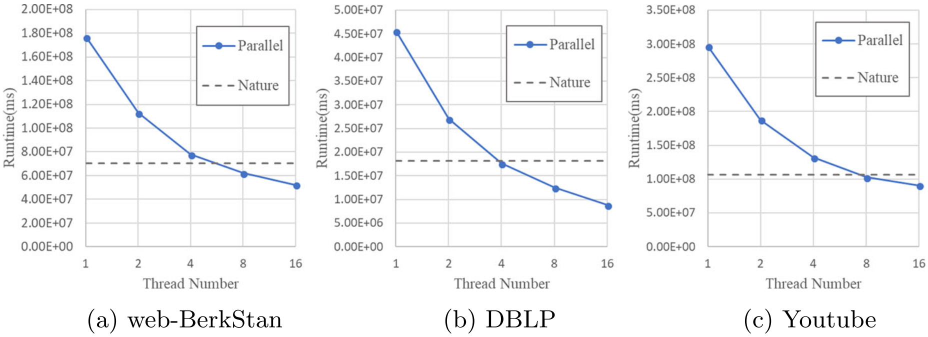


Figure 4 Runtime of our parallel algorithm on synthetic graphs



(a) web-BerkStan (b) DBLP (c) Youtube

Figure 5 Runtime of our parallel algorithm on different datasets with different threads

the parallel algorithm can greatly reduce the computation time of HCC, by 10% to 50%, comparing the natural algorithm.

We then evaluate the performance of the parallel algorithm on synthetic graphs generated using Chung-Lu model [1]. This model is specified by a collection of weights $w = (w_1, \ldots, w_n)$ that represent the expected degree sequence. The probability of an edge between i to j is $\frac{w_i * w_j}{\sum_k w_k}$.They allow loops from i to i so that the expected degree at i is

$$\sum_j \frac{w_i * w_j}{\sum_k w_k} = w_i \tag{2}$$

The evaluation results are shown in Figure 4. The x-axis represents the number of vertices, and the y-axis represents the runtime. From the figure, we can see that the increasing speed of the runtime of the parallel algorithm is much slower than $\Theta(|V| * |E|)$, which is the runtime of the natural algorithm.

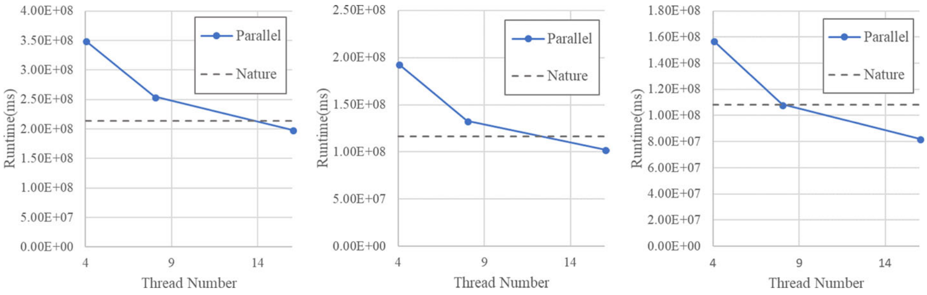
7.2 Scalability evaluation

We select Youtube, web-BerkStan and DBLP as test datasets to evaluate the scalability of the parallel algorithm. For the evaluation, HCC is computed for each dataset using our parallel algorithm that runs on 1, 2, 4, 8 and 16 threads respectively. The natural algorithm is used as baseline.

The evaluation results are shown in Figure 5. From the figures, it can be seen that as the number of thread increases, the computation time decreases nearly linearly. Furthermore, the figures also show that when the number of threads is small, such as less than 8 for the case of web-BerkStan, the runtime of the parallel algorithm is larger than that of the natural algorithm. This is because, comparing with the natural algorithm, the parallel one needs to perform some preprocessing computation, to determine the independence of vertices and subtrees in a BFS tree's reconnection. But as the number of threads increases, the runtime of the parallel algorithm is reduced to be smaller than that of the natural one.

Table 5 Generated graphs data

Generated graphs no.	$n = \|V\|$	$m = \|E\|$
1	0.64M	13.51M
2	1.28M	27.14M
3	0.64M	7.21M



(a) Generated Graph 1 (b) Generated Graph 2 (c) Generated Graph 3

Figure 6 Runtime of our parallel algorithm in different generated graph datasets with different threads

7.3 Impact factors

We evaluate the key factors that may impact the speedup of the parallel algorithm over the natural one. Two factors are considered, the network size and the average degree. To evaluate the impact of these two factors, we perform our parallel algorithm on 3 generated graphs given in Table 5 using 4, 8 and 16 threads respectively. The evaluation results are shown in Figure 6. By Figure 6a and b, it can be seen that the number of vertices does not heavily affect the speedup of the parallel algorithm over the natural one. Figure 6a and c show that the average degree impact the speedup significantly. The larger the average degree is, the smaller the speedup is. This is because when the average degree gets larger, nodes will have more neighbors, and hence independence of vertices and subtrees in the reconnection of a BFS tree becomes poorer.

7.4 Summary of experiment results

The experiment results show that comparing with the natural algorithm, the parallel algorithm can greatly reduce the computation time of HCC, when multiple threads are used. The parallel algorithm exhibits good scalabilty and parallelism.

8 Conclusion

We proposed a new centrality measure, Hierarchical Closeness Centrality, to reflect the local centrality of vertices. Experiments on real-world graphs show that HCC is better than the global measure GCC in some applications, such as maximum influence vertices identification and community detection. We also gave an efficient parallel algorithm for computing HCC, by investigating the independence of vertices and subtrees during the reconnection procedure of a BFS tree. Extensive experiments demonstrate that our parallel algorithm is much more efficient than the natural algorithm. It is also shown that our algorithm has good scalability and parallelism.

HCC has exhibited its usefulness in some crucial applications in social networks. So it deserves to futher understanding this centrality measure, and applying it in more domains. Furthermore, implementing our algorithm on real parallel platform is also our future work.

Acknowledgements This work is supported by the National Key Research and Development Program of China No.2017YFC0803700, the National Natural Science Foundation of China Grants 61602195, 61572216, 61433019 and U1435217, Natural Science Foundation of Hubei Province 2017CFB301, the Outstanding Youth Foundation of Hubei Province 2016CFA032, and Fundamental Research Funds for HUST.

References

1. Aiello, W., Chung, F., Lu, L.: A random graph model for power law graphs. Exp. Math. **10**(1), 53–66 (2001)
2. Al-Baghdadi, A.: Computing Top-K Closeness Centrality in Unweighted Undirected Graphs Revisited. Ph.D. thesis (2017)
3. Bargigli, L., Gallegati, M.: Finding communities in credit networks. Economics **7**(17), 1 (2013)
4. Bavelas, A.: Communication Patterns in Task-Oriented Groups. J. Acoust. Soc. Am. **22**(6), 725–730 (1950)
5. Bergamini, E., Borassi, M., Crescenzi, P., Marino, A., Meyerhenke, H.: Computing top-k closeness centrality faster in unweighted graphs. In: Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX), vol. V, pp. 68–80 (2016)
6. Brandes, U., Pich, C.: Centrality estimation in large networks. Other **17**(7), 2303–2318 (2007)
7. Cutts, M.: How does Google collect and rank results? http://www.google.com/librariancenter/articles/0601_03.html (2010)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the of the OSDI, pp. 137–149 (2004)
9. Eppstein, D., Wang, J.: Fast Approximation of Centrality. J. Graph Algorithms Appl. **8**(1), 2 (2000)
10. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry **40**(1), 35 (1977)
11. Fu, L., Gao, L., Ma, X.: A centrality measure based on spectral optimization of modularity density. Sci. China Inf. Sci. **53**(9), 1727–1737 (2010)
12. Girvan, M., Newman, M.E.: Community structure in social and biological networks. Proc. Natl. Acad. Sci. **99**(12), 7821–7826 (2002)
13. Gu, Q., Xiong, S., Chen, D.: Correlations between characteristics of maximum influence and degree distributions in software networks. Sci. China Inf. Sci. **57**(7), 1–12 (2014)
14. Hossmann, T., Spyropoulos, T., Legendre, F.: A complex network analysis of human mobility. In: 2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 876–881 (2011)
15. Inariba, W.: Random-radius ball method for estimating closeness centrality. In: Proceedings of the 31th Conference on Artificial Intelligence (AAAI 2017), pp. 125–131 (2017)
16. Kang, U.: Centralities in large networks: algorithms and observations. In: SIAM International Conference on Data, pp. 119–130 (2011)
17. Mora, R., Gutierrez, C.: Random-walk closeness centrality satisfies boldi-vigna axioms. In: CEUR Workshop Proceedings, vol. 1378, pp. 110–120 (2015)
18. Okamoto, K., Chen, W., Li, X.Y.: Ranking of closness centrality for large-scale social networks. In: Proceedings of the 2nd Annual International Workshop on Frontiers in Algorithmics, pp. 186–195 (2008)
19. Olsen, P.W., Labouseur, A.G., Hwang, J.H.: Efficient top-k closeness centrality search. In: Proceedings of ICDE, pp. 196–207 (2014)
20. Olsen, P.W. Jr., Labouseur, A.G., Hwang, J.H.: Efficient top-k closeness centrality search 196–207 (2014)
21. Sah, P., Singh, L.O., Clauset, A., Bansal, S.: Exploring community structure in biological networks with random graphs. BMC Bioinf. **15**(1), 220 (2014)
22. Sariyüce, A.E., Saule, E., Kaya, K., Çatalyürek, Ü.V.: Incremental closeness centrality in distributed memory. Parallel Comput. **47**, 3–18 (2015)
23. Zhuang, H., Sun, Y., Tang, J., Zhang, J., Sun, X.: Influence maximization in dynamic social networks. In: 2013 IEEE 13th International Conference on Data Mining (ICDM). IEEE, pp. 1313–1318 (2013)