Data & Knowledge Engineering xxx (2015) xxx–xxx

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Editorial

Parallel community detection on large graphs with MapReduce and GraphChi

Seunghyeon Moon [a], Jae-Gil Lee [b,*], Minseo Kang [b], Minsoo Choy [b], Jin-woo Lee [b]

[a] KAIST Institute for IT Convergence, 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea
[b] Department of Knowledge Service Engineering, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea

ARTICLE INFO

Available online xxxx

Keywords:
Clustering, classification, and association rules
Mining methods and algorithms
Community detection
Social networks
MapReduce
Vertex-centric model

ABSTRACT

Community detection from social network data gains much attention from academia and industry since it has many real-world applications. The Girvan–Newman (GN) algorithm is a divisive hierarchical clustering algorithm for community detection, which is regarded as one of the most popular algorithms. It exploits the concept of *edge betweenness* to divide a network into multiple communities. Though it is being widely used, it has limitations in supporting large-scale networks since it needs to calculate the shortest path between *every pair* of vertices in a network. In this paper, we develop two parallel versions of the GN algorithm to support large-scale networks. First, we propose a new algorithm, which we call *Shortest Path Betweenness MapReduce Algorithm* (SPB-MRA), that utilizes the MapReduce model. Second, we propose another new algorithm, which we call *Shortest Path Betweenness Vertex-Centric Algorithm* (SPB-VCA), that utilizes the vertex-centric model. An approximation technique is also developed to further speed up community detection processes. We implemented SPB-MRA using Hadoop and SPB-VCA using GraphChi, and then evaluated the performance of SPB-MRA on Amazon EC2 instances and that of SPB-VCA on a single commodity PC. The evaluation results showed that the elapsed time of SPB-MRA decreased almost linearly as the number of reducers increased, SPB-VCA outperformed SPB-MRA just on a *single* PC by 4–6 times, and the approximation technique introduced negligible errors.

1. Introduction

As social networking services (SNSs) such as Facebook and Twitter are getting more popular, analyzing social network data has become one of the most important issues in various areas [1]. Among those analysis jobs, community detection from social network data gains much attention from academia and industry since it has many real-world applications such as friend recommendation and target marketing [2,3].

Community detection is to partition the set of network vertices into multiple groups such that the vertices within a group are connected densely, but connections between groups are sparse [4]. There have been many studies regarding community detection [5]. The Girvan–Newman (GN) algorithm proposed by Girvan and Newman [6] exploits the concept of *edge betweenness*, which is a measure of the centrality and influence of an edge in a network. Though the GN algorithm is being widely used, it has limitations in supporting large-scale networks since it needs to calculate the shortest path between *every pair* of vertices. The number of vertex pairs in a large-scale network is really prohibitive.

* Corresponding author.
E-mail addresses: myth624@kaist.ac.kr (S. Moon), jaegil@kaist.ac.kr (J.-G. Lee), minseo@kaist.ac.kr (M. Kang), minsoo.choy@kaist.ac.kr (M. Choy), jinwoo.lee@kaist.ac.kr (J. Lee).

http://dx.doi.org/10.1016/j.datak.2015.05.001
0169-023X/© 2015 Elsevier B.V. All rights reserved.

In the era of Big Data, the amount of available data is growing unprecedentedly. Thus, data analysis calls for very scalable methods that can cope with huge data sets. MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. MapReduce has been widely used owing to its scalability and ease of use [7–11]. It has been the driving force behind big data analysis in recent years. In addition, as graph (or network) data become more prevalent, totally new parallel computing platforms based on the vertex-centric model are being developed especially for graph data [12–14].

In this paper, we develop two parallel versions of the GN algorithm to solve its scalability issues. First, we propose a new algorithm, which we call *Shortest Path Betweenness MapReduce Algorithm* (SPB-MRA), that utilizes the MapReduce model. Second, we propose another new algorithm, which we call *Shortest Path Betweenness Vertex-Centric Algorithm* (SPB-VCA), that utilizes the vertex-centric model [12]. In addition, we suggest an approximation technique to further speed up community detection processes. Our preliminary work [15][1] contains the former algorithm only, and this paper contains the latter algorithm as well.

Our first algorithm, SPB-MRA, consists of four major stages, and all operations are executed in parallel *on a cluster*. In the first stage, all-pair shortest paths on a network are calculated. In the second stage, the edge betweennesses of all edges in the network are calculated. In the third stage, k_{iter} edges are selected by edge betweenness, and they get removed. In the final stage, the network is updated, and this new network is provided to the next iteration. These four stages repeat until the quality of communities does not improve any more. SPB-MRA is implemented on top of Apache Hadoop [16].

Our second algorithm, SPB-VCA, consists of three major stages without the last one of SPB-MRA. That is, the update of a network after an edge removal is *not* explicitly necessary. SPB-VCA is implemented on top of GraphChi [14] and thus runs on a *single* PC.

SPB-VCA has two main advantages over SPB-MRA by virtue of the vertex-centric model. (i) Since the algorithm is represented directly on a graph, which is the underlying data structure, it looks more natural and intuitive. On the other hand, in SPB-MRA, a graph is decomposed into its constituent edges, and the algorithm manipulates the edges (i.e., key-value pairs). Accordingly, there is no need to generate and merge key-value pairs in SPB-VCA. (ii) As a result, the performance of SPB-VCA is dramatically higher than that of SPB-MRA.

The major contributions of this paper are as follows.

- We propose two algorithms—SPB-MRA and SPB-VCA—and demonstrate performance improvement. The results of performance tests showed that the elapsed time of SPB-MRA decreased almost linearly as more reducers were added to a cluster. In addition, we confirmed that the vertex-centric model, which is dedicated to graph data, allows us to achieve much higher performance than the MapReduce model. In fact, SPB-VCA was shown to outperform SPB-MRA by 4–6 times just on a *single* PC.
- We suggest an approximation technique to further speed up community detection processes. Instead of removing a *single* edge per iteration, we remove *multiple* edges that have the top-k_{iter} highest edge betweenness at once. The results of accuracy tests showed that a negligible error was introduced by the approximation.

The rest of this paper is organized as follows. Section 2 summarizes the background knowledge required for this study. Sections 3 and 4 propose SPB-MRA and SPB-VCA respectively. Section 5 presents the results of performance tests. Finally, Section 6 concludes this study.

2. Background and related work

2.1. MapReduce and Hadoop

MapReduce is a programming model for processing large-scale data in a parallel way [7]. Users can easily implement distributed, parallel processing software by writing only two functions: *map* and *reduce*. Fig. 1 shows the control flow of MapReduce. The map function processes a sub-problem for input data and emits intermediate ⟨*key*, *value*⟩ pairs. The reduce function combines the values associated with the same key and produces the final output. Apache Hadoop [16] is the most popular open-source implementation of MapReduce. However, despite its popularity for big data processing, MapReduce is known to be awkward at supporting iterative graph algorithms [17].

2.2. Vertex-centric model and GraphChi

The vertex-centric model is a programming model for iterative graph computation. It is easy to program since a programmer just needs to “think like a vertex.” Thus, this model has been adopted by parallel graph computing platforms, including Pregel [12], GraphLab [18], and GraphChi [14]. In the vertex-centric model, every vertex and edge is associated with a value, and computation is performed on a vertex by a user-specified function.

GraphChi is a disk-based system exploiting the vertex-centric model for processing graph computations on just a single machine [14]. A novel parallel sliding windows (PSW) method enables GraphChi to execute graph mining algorithms on very large graphs. The PSW method splits vertices in a graph into multiple intervals, where each interval is associated with a shard which stores all edges whose destination vertex is contained in that interval. The subgraphs divided by the PSW method are processed in three steps: 1) loading a subgraph from disk; 2) updating the values of vertices and edges; and 3) writing the updated results to disk.

Fig. 2 illustrates the operation of the PSW method on a toy graph. In this example, a toy graph with four vertices, which is split into two intervals, is used. First, the subgraph with **Vertex 1, 2** for the execution interval, **Interval 1**, is stored in a memory-shard (**Shard 1**)

[1] This work received the best paper award at BigComp 2014.

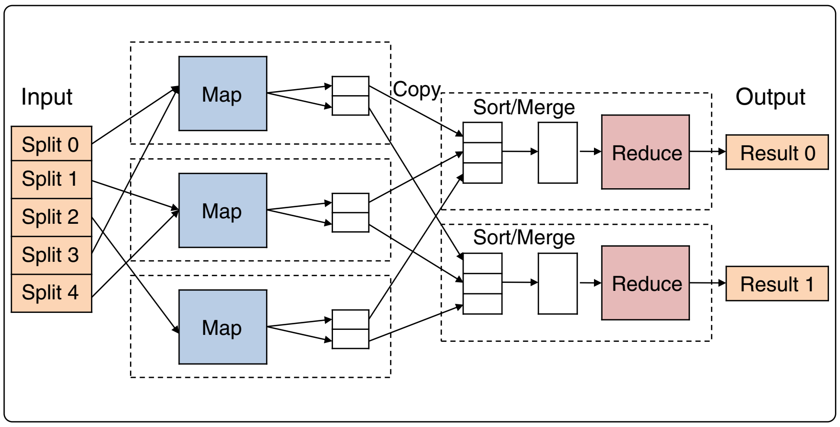


Fig. 1. The execution flow of MapReduce.

consisting of all in-edges to the interval and a sliding-shard (**Shard 2**) containing all out-edges from the interval. Second, a user-specified update function is executed for the vertices in the execution interval. Third, the updated values of the vertices and edges are written to disk. As in the right half of Fig. 2, these three steps are repeated for **Interval 2** to get full computation results for the given graph.

2.3. Girvan–Newman (GN) algorithm

The GN algorithm is a divisive hierarchical clustering algorithm exploiting the concept of edge betweenness [6]. Three methods were proposed for the calculation of edge betweenness. Among them, the shortest-path method typically shows the best results. The *edge betweenness* of an edge is informally the number of shortest paths between pairs of vertices that pass through it. Since communities are loosely connected by a few "intergroup" edges, all shortest paths between different communities must pass through one of these few edges. Then, those edges connecting communities will have high edge betweenness. Thus, the communities are detected by eliminating such edges repeatedly. However, since the shortest path should be obtained for every pair of vertices, the GN algorithm is very costly.

2.4. Parallel community detection

There have been a number of studies on parallelizing the algorithms of identifying community structures in networks. Bahmani et al. [19] proposed an algorithm of finding the densest subgraph in the streaming model and implemented it using the MapReduce model. Li et al. [20] proposed MR-LPA, which is a parallel version of the label propagation algorithm using the MapReduce model. In addition, Yang and Lonardi [21] presented a parallel implementation of the GN algorithm using the message passing interface (MPI) standard. Since their algorithm requires storing all edges and vertices of a graph in main memory of each computing node in the cluster, it has limitations in supporting large-scale graphs. On the contrary, our algorithms proposed in this paper efficiently divide a graph

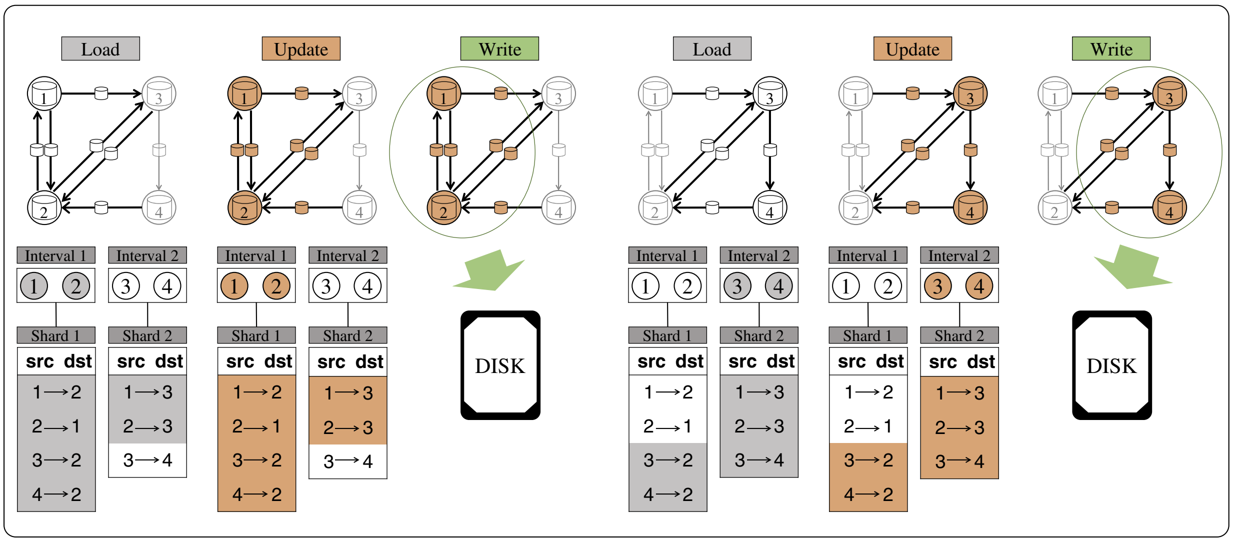


Fig. 2. The operation of PSW on a toy graph.

into subgraphs and store them in distributed file systems (in SPB-MRA) or in local file systems (in SPB-VCA) to support big data graphs. Furthermore, SPB-MRA and SPB-VCA exploit an approximation technique to further speed up community detection processes.

3. Proposed algorithm: SPB-MRA

3.1. Overview

SPB-MRA goes through *four* stages, as described in Fig. 3. The output of a stage is chained to the input of its next stage. Each stage executes its own map and reduce tasks. An iteration of these four stages produces a community detection result. In each iteration, Stage1 is executed multiple times, and the other stages are executed only once. The four stages repeat until the result quality no longer improves.

A tuple of 7 elements below is maintained for each pair of vertices in the process of community detection. It holds the network structure (i.e., an adjacency list), the shortest path obtained so far, and so on. The term “tuple” in this section refers to a tuple of these elements.

- ***targetId*** indicates the destination vertex of a shortest path and is initially set to be *sourceId*.
- ***sourceId*** indicates the source vertex of a shortest path and is initially set to be *targetId*.
- ***distance*** indicates the length of a shortest path and is initially set to be 0. The value of *distance* is updated in each iteration of Stage1.
- ***status*** indicates the status of a specific path. *a* is “active”, and *i* is “inactive” meaning that the shortest path is already detected.
- ***weight*** indicates the number of the shortest paths from *sourceId* to *targetId* and is initially set to be 1.
- ***pathInfo*** indicates the list of the vertices on a shortest path and is initially set to be *null*.
- ***adjList*** indicates the list of the vertices adjacent to *targetId*.

3.2. Stage1: finding all-pair shortest paths

In this stage, the shortest paths between every pair of vertices in the network are calculated. For this purpose, we adopted a multi-source message passing model proposed by Zeng et al. [22].

In the map stage, the frontiers are expanded from every vertex in the network to its adjacent vertices. For an input tuple, if *status* is *i*, no operation is needed; if *status* is *a*, *status* is changed to *i*, 1 is added to *distance*, and *targetId* is added to *pathInfo*. The tuple is emitted to the reduce stage. In addition, new tuples are generated by assigning each vertex in *adjList* to *targetId*. For these newly generated tuples, *status* is set to be *a*, *adjList* is set to be *null*, and the other elements are set to be the same as those of the tuple emitted just before. In this way, all shortest paths between *sourceId* and *targetId* are generated and sent to a reducer. Fig. 4 shows an example of the map stage.

In the reduce stage, among the tuples sharing *sourceId* and *targetId*, only the tuple that has the minimum value of *distance* survives. If two or more tuples have the same minimum, *weight* is changed to the number of such tuples to remember the multiplicity of the shortest path. Fig. 5 shows an example of the reduce stage. These map and reduce stages repeat until all tuples have *i* for *status*.

3.3. Stage2: calculating edge betweenness

In this stage, the edge betweennesses of all edges in the network are calculated. In the map stage, unity is divided to each edge on a shortest path according to the total number (i.e., *weight*) of the shortest paths sharing *sourceId* and *targetId*. In the reduce stage, the contribution of each shortest path is summed up for each edge. Fig. 6 shows an example of the map stage and the reduce stage.

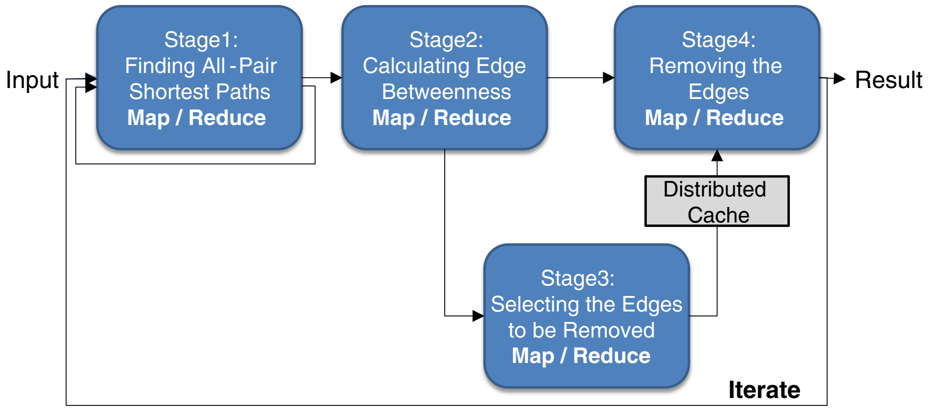

Input
Stage1: Finding All-Pair Shortest Paths Map / Reduce
Stage2: Calculating Edge Betweenness Map / Reduce
Stage4: Removing the Edges Map / Reduce
Result
Distributed Cache
Stage3: Selecting the Edges to be Removed Map / Reduce
Iterate

Fig. 3. An overview of the SPB-MRA algorithm.

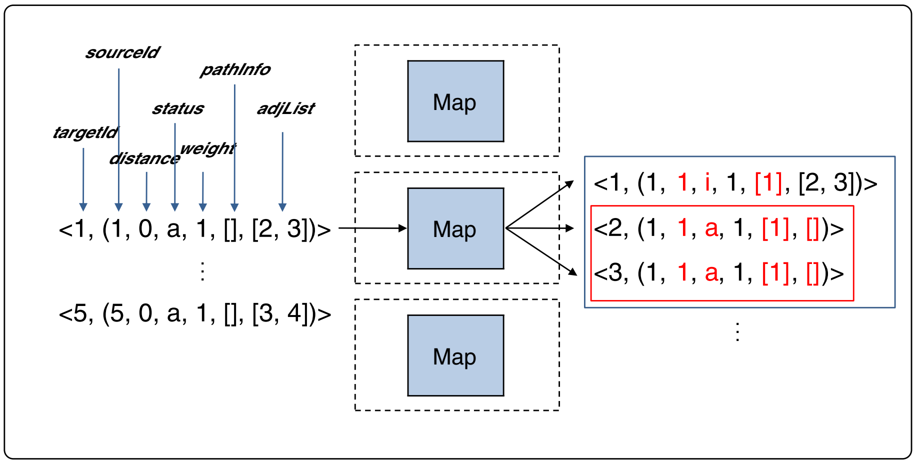

sourceId
pathInfo
status
adjList
targetId
distance
weight
<1, (1, 0, a, 1, [], [2, 3])>
<5, (5, 0, a, 1, [], [3, 4])>
Map
Map
Map
<1, (1, 1, i, 1, [1], [2, 3])>
<2, (1, 1, a, 1, [1], [])>
<3, (1, 1, a, 1, [1], [])>

Fig. 4. Map stage of Stage1.

3.4. Stage3: selecting the edges to be removed

In this stage, k_{iter} edges are selected according to edge betweenness. k_{iter} is specified by a user as a tuning parameter. In the map stage, no operation is needed. In the reduce stage, edges are sorted in the decreasing order of edge betweenness, and the top-k_{iter} edges are selected. We simply run only one reducer to get the globally sorted result. The selected edges and their value of edge betweenness are stored in the distributed cache such that all mappers of Stage4 can access the information. Fig. 7 shows an example of the map stage and the reduce stage.

Note that SPB-MRA selects multiple edges per iteration whereas the GN algorithm only one edge. That is, we remove k_{iter} edges in one iteration instead of iterating k_{iter} times. Thus, this approximation can speed up community detection by k_{iter} times. Our reasoning is that the edge with the highest value of edge betweenness in the next iteration tends to have a high value in the current iteration, too. Section 5.5 empirically proves that our reasoning is indeed correct.

3.5. Stage4: removing the edges

In this stage, the edges selected by Stage3 are removed from the network. Then, a new set of tuples are generated to reflect the removed edges since edge betweenness needs to be recalculated in the next iteration. Note that edge betweenness changes if a shortest path ran along one of the removed edges. In the map stage, if *targetId* of a tuple from Stage2 is affected by the edges selected in Stage3, its *adjList* is updated to represent a new network by removing the corresponding vertex. Then, a new tuple is created such that its key is set to *sourceId* with other elements initialized since the shortest path from that *sourceId* needs to be recalculated. In the reduce stage, duplicate tuples are first removed, and then the initial value of *adjList* is replaced with the updated value. These tuples are provided to the input of Stage1 for the next iteration. Figs. 8 and 9 show an example of the map stage and the reduce stage.

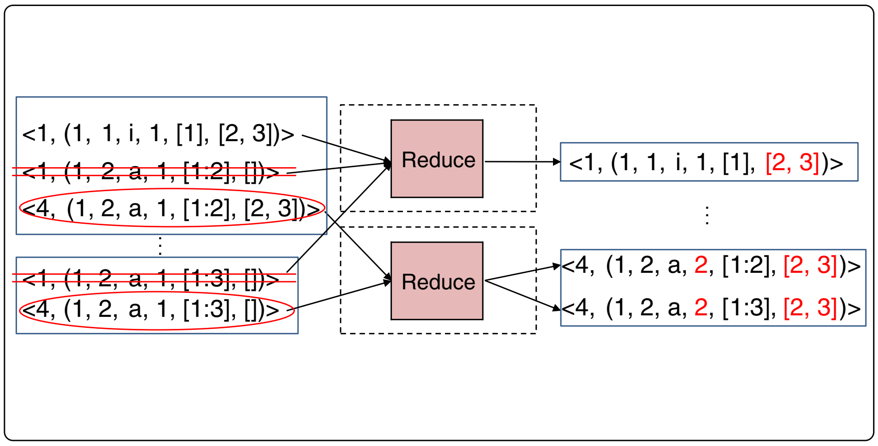
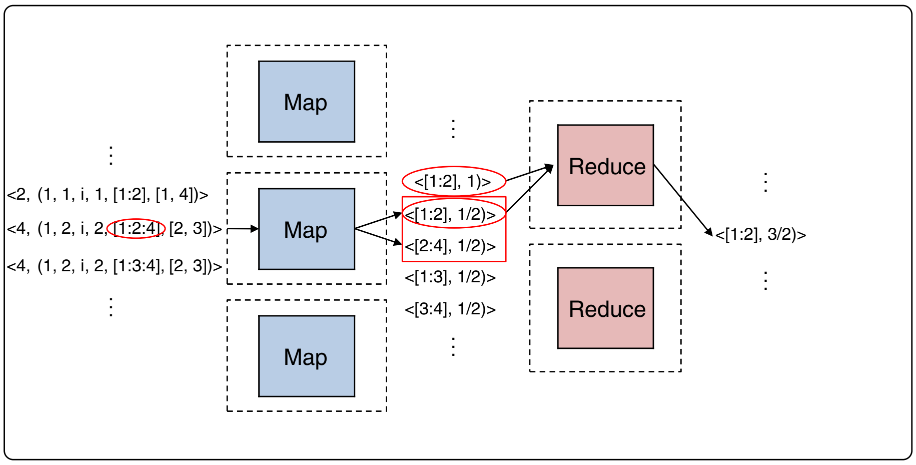

<1, (1, 1, i, 1, [1], [2, 3])>
~~<1, (1, 2, a, 1, [1:2], [])>~~
<4, (1, 2, a, 1, [1:2], [2, 3])>
~~<1, (1, 2, a, 1, [1:3], [])>~~
<4, (1, 2, a, 1, [1:3], [])>
Reduce
Reduce
<1, (1, 1, i, 1, [1], [2, 3])>
<4, (1, 2, a, 2, [1:2], [2, 3])>
<4, (1, 2, a, 2, [1:3], [2, 3])>

Fig. 5. Reduce stage of Stage1.



<2, (1, 1, i, 1, [1:2], [1, 4])>
<4, (1, 2, i, 2, [1:2:4], [2, 3])>
<4, (1, 2, i, 2, [1:3:4], [2, 3])>
Map
Map
Map
<[1:2], 1)>
<[1:2], 1/2)>
<[2:4], 1/2)>
<[1:3], 1/2)>
<[3:4], 1/2)>
Reduce
Reduce
<[1:2], 3/2)>

Fig. 6. Map stage and reduce stage of Stage2.

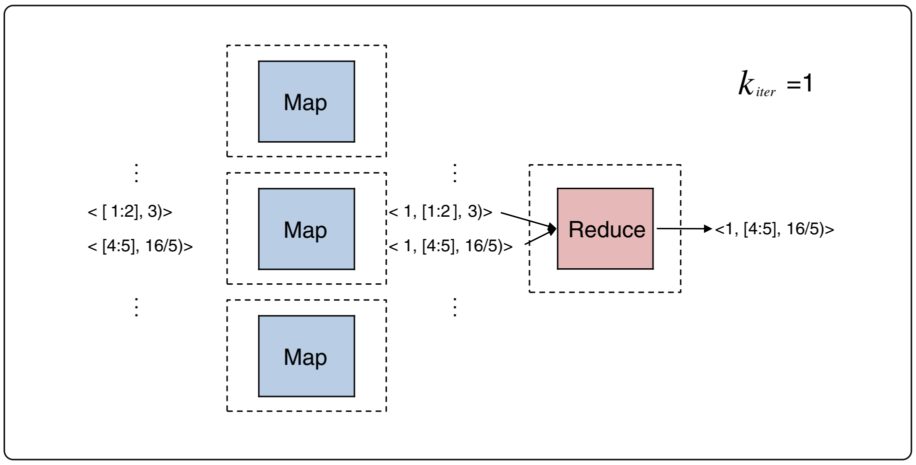
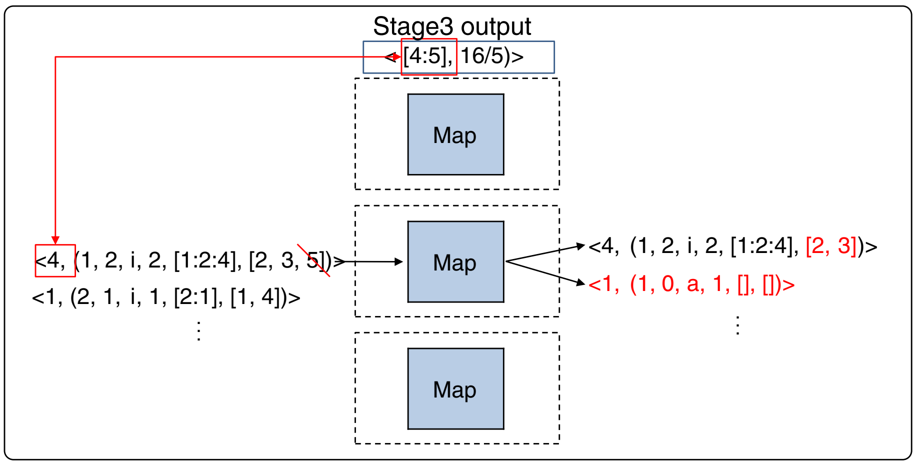


Fig. 7. Map stage and reduce stage of Stage3.



Stage3 output
<[4:5], 16/5)>
<4, (1, 2, i, 2, [1:2:4], [2, 3, 5])>
<1, (2, 1, i, 1, [2:1], [1, 4])>
Map
Map
Map
<4, (1, 2, i, 2, [1:2:4], [2, 3])>
<1, (1, 0, a, 1, [], [])>

Fig. 8. Map stage of Stage4.

3.6. Summary

Algorithm 1 describes the pseudocode of SPB-MRA, which is self-explanatory. Stage1, Stage2, Stage3, and Stage4 are executed sequentially. That is, the next stage starts to run after the current stage completes. Each stage is processed in parallel by multiple mappers and reducers.

Algorithm 1. SPB-MRA

```
INPUT: A graph of the form of key-value pairs, $k_{iter}$
OUTPUT: The top-$k_{iter}$ edges
1: repeat
2:     repeat
3:         fn Stage1-Map(key $k$, value $v$) do
4:             if $v.status$ == "a" then
5:                 $v.status \leftarrow$ "i"; $v.distance$ + = 1; $v.pathInfo.add(k)$; emit($k, v$);
6:                 for each $adj \in v.adjList$ do
7:                     $k' \leftarrow adj$; $v'.status \leftarrow$ "a"; $v'.adjList \leftarrow null$; emit($k', v'$);
8:             else
9:                 emit($k, v$);
10:        fn Stage1-Reduce(key $k$, values $v[1 \ldots n]$) do
11:            for $s \subseteq v[1 \ldots n]$ and $s$ has the same $sourceId$ do
12:                $minList \leftarrow$ the values which have minimum $distance$;
13:                for each $v_i \in minList$ do
14:                    $v_i.weight \leftarrow |minList|$; emit($k, v_i$);
15:    until $status$ == "i" for all tuples;
16:    fn Stage2-Map(key $k$, value $v$) do
17:        for each $e \in v.pathInfo$ do
18:            $k' \leftarrow e$; $v' \leftarrow 1/v.weight$; emit($k', v'$);
19:    fn Stage2-Reduce(key $k$, values $v[1 \ldots n]$) do
20:        for each $v_i \in v[1 \ldots n]$ do
21:            $sumofEbtwness$ + = $v_i$;
22:        emit($k, sumofEbtwness$);
23:    fn Stage3-Map(key $k$, value $v$) do
24:        emit($1, v$);
25:    fn Stage3-Reduce(key $k$, $v[1 \ldots n]$) do
26:        Sort the values of $v[1 \ldots n]$;
27:        Add the top-$k_{iter}$ values to DistributedCache.List;
28:    fn Stage4-Map(key $k$, value $v$) do
29:        if $k \in$ DistributedCache.List then
30:            $v.adjList.remove(k)$; emit($k, v$);
31:            $k' \leftarrow v.sourceId$; $v' \leftarrow$ a default value; emit($k', v'$);
32:        else
33:            emit($k, v$);
34:    fn Stage4-Reduce(key $k$, values $v[1 \ldots n]$) do
35:        for each $v_i \in v[1 \ldots n]$ do
36:            if $v_i$ is not a duplicate then
37:                Update $v_i.adjList$; emit($k, v_i$);
38: until no edge to cut exists;
```

4. Proposed algorithm: SPB-VCA

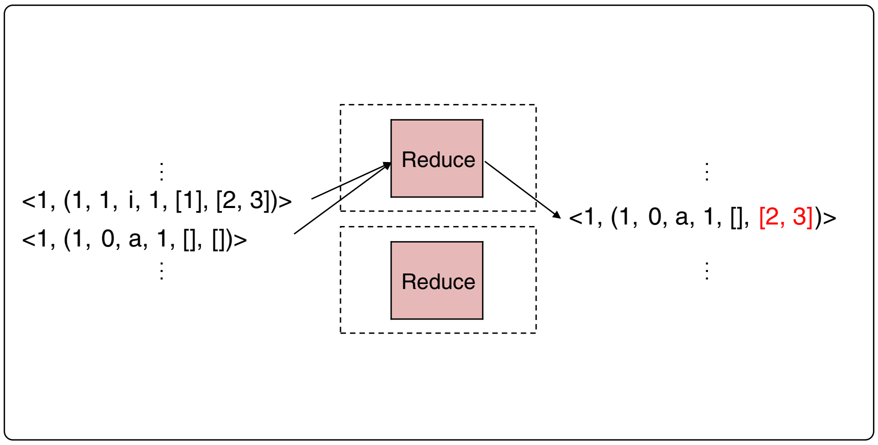
4.1. Overview

SPB-VCA goes through *three* stages, as described in Fig. 10. The output of a stage is chained to the input of its next stage. Each stage executes its own update function. An iteration of these three stages produces a community detection result. In each iteration, the pair of Stage1 and Stage2 is executed multiple times, considering each vertex as a source, whereas Stage3 is executed only once. The three stages repeat until the result quality no longer improves.

A vertex and an edge are designed to maintain a tuple since it is allowed to customize the data type associated with a vertex or an edge in GraphChi. A vertex keeps the following two elements.

- ***dist*** indicates the length of a shortest path from a source to itself. The value is initially set to be 0 if the vertex is the source and ∞ otherwise.
- ***sigma*** indicates the number of the shortest paths from a source to itself. The value is initially set to be 1 if the vertex is the source and 0 otherwise.

An edge keeps the following five elements. In GraphChi, edges typically play a role of messengers, passing a value between its adjacent vertices.



<1, (1, 1, i, 1, [1], [2, 3])>
<1, (1, 0, a, 1, [], [])>
Reduce
Reduce
<1, (1, 0, a, 1, [], [2, 3])>

Fig. 9. Reduce stage of Stage4.

- ***msg_dist*** indicates the value of *dist* passed by a vertex to its adjacent vertices.
- ***msg_sigma*** indicates the value of *sigma* passed by a vertex to its adjacent vertices. The value is the same as the number of the shortest paths passing through the edge.
- ***visited*** indicates whether the edge has been visited and is initially set to be *false*.
- ***single_eb*** indicates the value of edge betweenness originated from the current source vertex. The value is reset when a source vertex is changed.
- ***allpair_eb*** indicates the value of edge betweenness accumulated. The value is initially set to be 0 and is increased by *single_eb* when a source vertex is changed.

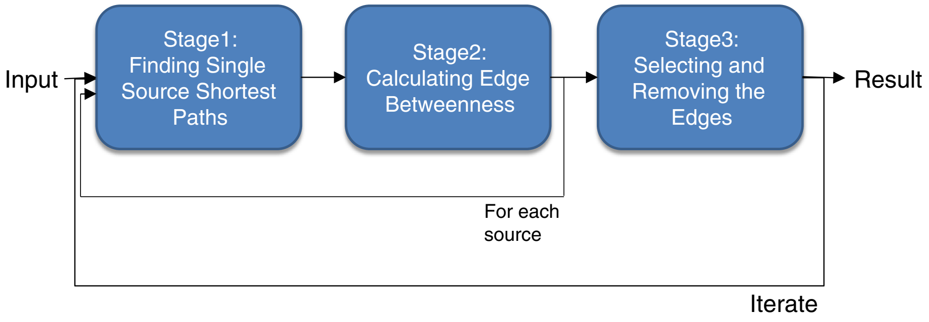
4.2. Stage1: finding single source shortest paths

In this stage, the shortest paths from a single source to every vertex in the network are calculated. For this purpose, we implemented breadth first search (BFS) in GraphChi's manner, i.e., by vertex-centric programming.

In the initialization step, the tuples associated with vertices and edges are initialized as shown in Fig. 11. This figure shows the result of the initialization step when **Vertex 0** is the source. For vertices, *dist* is set to be 0 for a source vertex and ∞ for other vertices; *sigma* is set to be 1 for a source vertex and 0 for other vertices. For edges, *msg_dist* and *msg_sigma* of an out-edge is the same as its starting vertex; *visited* is set to be *false* initially for all edges; *sigle_eb* and *allpair_eb* are set to be 0. Then, the vertices that are adjacent to the source vertex are scheduled for subsequent operations.

After the initialization step, each vertex scheduled checks whether its in-edges whose *visited* is *false*. Let's denote the vertex's current *dist* as *currDist* and each in-edge's *msg_dist* + 1 as *candDist*. For each of such in-edges, the edge is marked as *visited* (i.e., *visited* ← *true*), and the vertex compares *currDist* and *candDist* as below:

1. ***candDist*** < ***currDist*:** It means that we find a shorter path than those identified so far. Therefore, we set the vertex's *dist* to be *candDist* and *sigma* to be the corresponding in-edge's *msg_sigma*. Then, the vertex copies its new *dist* and *sigma* to its out-edges' *msg_dist* and *msg_sigma* and schedules its adjacent vertices except a source vertex.
2. ***candDist*** = ***currDist*:** It means that we find another path whose length is the same as that of the current shortest path. Therefore, the vertex's *dist* is not changed, and only the vertex's *sigma* is increased by the corresponding in-edge's *msg_sigma*. Then, the vertex copies its *dist* and *sigma* to its out-edges' *msg_dist* and *msg_sigma*.



Input
Stage1: Finding Single Source Shortest Paths
Stage2: Calculating Edge Betweenness
Stage3: Selecting and Removing the Edges
Result
For each source
Iterate

Fig. 10. An overview of the SPB-VCA algorithm.

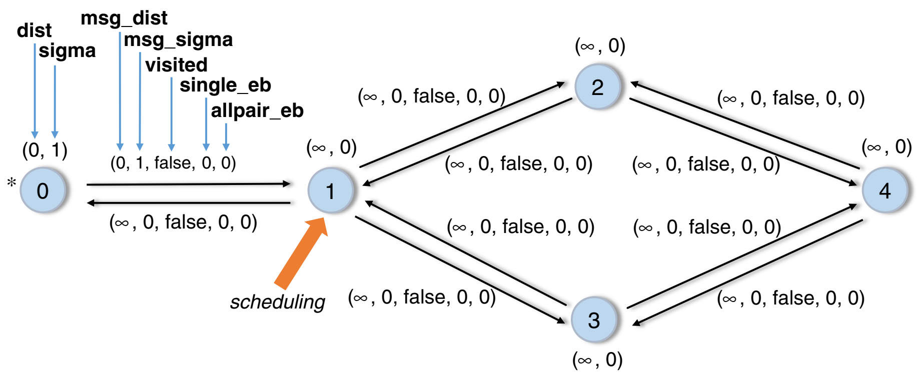


Fig. 11. Initialization of Stage1 (* indicates a source vertex).

3. ***candDist*** > ***currDist***: It means that this path is longer than the current shortest path. Thus, we do not have to do anything.

Fig. 12 shows a snapshot of the graph after executing an update function on **Vertex 1**. Here, *currDist* is ∞, and *candDist* is 1. Thus, *dist* is changed to 1, and *sigma* is changed to 1. Also, *msg_dist* and *msg_sigma* of **Edge** $\langle 1,0\rangle$, $\langle 1,2\rangle$, $\langle 1,3\rangle$ are changed to all 1's. The procedure continues until there is no more vertex to schedule, and a result of BFS is shown in Fig. 13. In addition, the *leaf vertices* of the BFS tree are also identified in this stage.

4.3. Stage2: calculating edge betweenness

In this stage, edge betweenness is calculated using the shortest paths obtained from the previous stage. The leaf vertices are first scheduled. For a vertex *v* and each in-edge *ie* on the shortest path, *single_eb* of *ie* is computed by Eq. (1). Here, $\frac{ie.msg\_sigma}{v.sigma}$ is the ratio of the number of the shortest paths passing through *ie* to that of the shortest paths passing through *v*, meaning the fraction of edge betweenness that this edge *ie* absorbs. In parenthesis, the value 1 indicates the value of edge betweenness originated from *v*; and the summation indicates the sum of edge betweennesses originated from the previous vertices.

$$ie.single\_eb = \frac{ie.msg\_sigma}{v.sigma}\left(1 + \sum_{oe \in v.outedge} oe.single\_eb\right) \quad (1)$$

Then, *single_eb* is accumulated into *allpair_eb* to calculate the final score which all vertices are considered as a source. SPB-VCA finds an edge to accumulate the computed betweenness as follows. If the identifier of an in-edge's ending vertex is greater than that of the in-edge's starting vertex, i.e., the in-edge follows the direction of BFS, *single_eb* is added to the in-edge's *allpair_eb*. Otherwise, SPB-VCA selects an edge whose direction is opposite to the in-edge and adds *single_eb* to that edge's *allpair_eb*. Fig. 14 shows a snapshot of the graph after the update function is executed on **Vertex 4**. Then, the vertices adjacent to the current vertex via its in-edges on the shortest paths are scheduled for subsequent operations.

This procedure goes on until the source vertex is scheduled. Thus, the order of scheduling vertices in Stage2 is opposite to that of scheduling vertices in Stage1. After both Stage1 and Stage2 for a specific source vertex are finished, the next vertex becomes a source, and the same procedures are carried out. Fig. 15 shows the final status of the graph when both Stage1 and Stage2 are finished for all vertices **Vertex 0–4**. The value of *allpair_eb* in each edge indicates the final edge betweenness score.

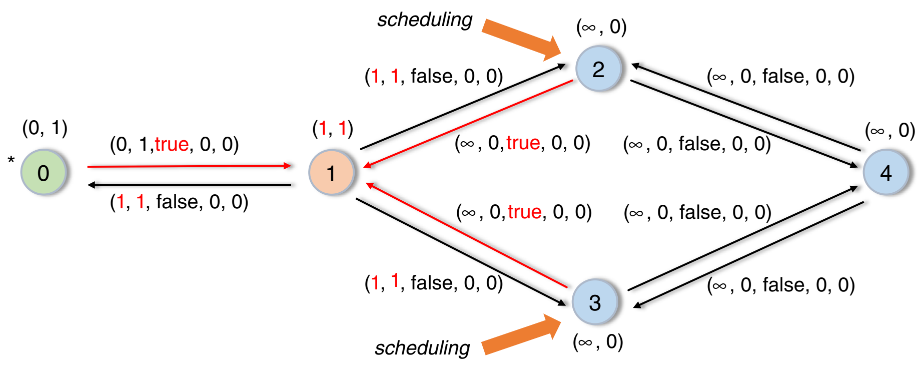


Fig. 12. The BFS step of Stage1.

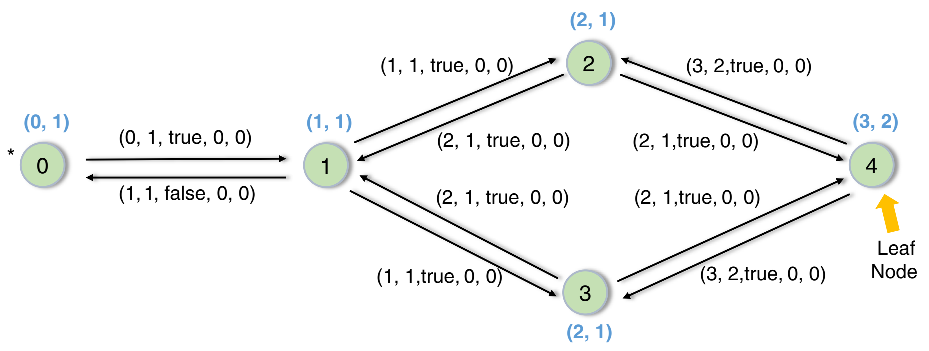


Fig. 13. The result of BFS (source: **Vertex 0**).

4.4. Stage3: selecting and removing the edges

In this stage, like SPB-MRA, k_{iter} edges are selected according to edge betweenness. To facilitate this task, SPB-VCA allocates a heap of size k_{iter} in main memory. While scanning each edge's betweenness in a graph, SPB-VCA updates the heap to maintain the top-k_{iter} edges. SPB-VCA also removes *multiple* edges per iteration, so we expect a speedup by the approximation. Since GraphChi supports an edge removal directly on a graph,[2] a new graph is implicitly generated unlike SPB-MRA.

Algorithm 2. SPB-VCA (update function for a vertex v)

```
INPUT: A graph, $k_{iter}$, a vertex $v$, a source vertex $s$
OUTPUT: The top-$k_{iter}$ edges
1:  if the current stage is STAGE1 then
2:      Initialize the data tuples of vertices and edges;
3:      for each $ie \in v.inedge$ do
4:          if $ie.visited == false$ then
5:              $ie.visited \leftarrow true$;
6:              $currDist \leftarrow v.dist$;  $candDist \leftarrow ie.msg\_dist + 1$;
7:              if $candDist < currDist$ then
8:                  $v.dist \leftarrow candDist$;  $v.sigma \leftarrow ie.msg\_sigma$;
9:                  for each $oe \in v.outedge$ do
10:                     $oe.dist \leftarrow v.dist$;  $oe.sigma \leftarrow v.sigma$;
11:                     if $oe.visited == false$ and $oe.vertex\_id \neq s$ then
12:                         addTask($oe.vertex\_id$);
13:             else if $candDist == currDist$ then
14:                 $v.sigma \leftarrow v.sigma + ie.msg\_sigma$;
15:                 for each $oe \in v.outedge$ do
16:                     $oe.dist \leftarrow v.dist$;  $oe.sigma \leftarrow v.sigma$;
17: else if the current stage is STAGE2 then
18:     $tempEB \leftarrow 0$;
19:     if $v \neq s$ then
20:         for each $oe \in v.outedge$ do
21:             $tempEB \leftarrow tempEB + oe.single\_eb$;
22:         for each $ie \in v.inedge$ and $ie$ is on a shortest path to $v$ do
23:             $EB \leftarrow (1 + tempEB) \cdot \frac{ie.msg\_sigma}{v.sigma}$;
24:             if $ie.vertex\_id < v.id$ then
25:                 $ie.allpair\_eb \leftarrow ie.allpair\_eb + EB$;
26:             else
27:                 Find the edge $e$ whose direction is opposite to that of $ie$;
28:                 $e.allpair\_eb \leftarrow e.allpair\_eb + EB$;
29: else if the current stage is STAGE3 then
30:     for each $oe \in v.outedge$ do
31:         Update the heap of size $k_{iter}$ by $oe.allpair\_eb$;
```

4.5. Summary

Algorithm 2 describes the pseudocode of SPB-VCA, which is self-explanatory. Recall that SPB-MRA needed the two functions *map* and *reduce*. On the other hand, SPB-VCA needs only one function *update*, which is more intuitive. This function specifies the task that should be performed on a vertex. Programmers can update the values of a vertex and its adjacent edges, schedule its adjacent vertices, and even modify the topology of a graph. Our update function triggers one of the three stages according to the stage being executed.

[2] This feature is supported only for the edges associated with basic data types in the latest version.

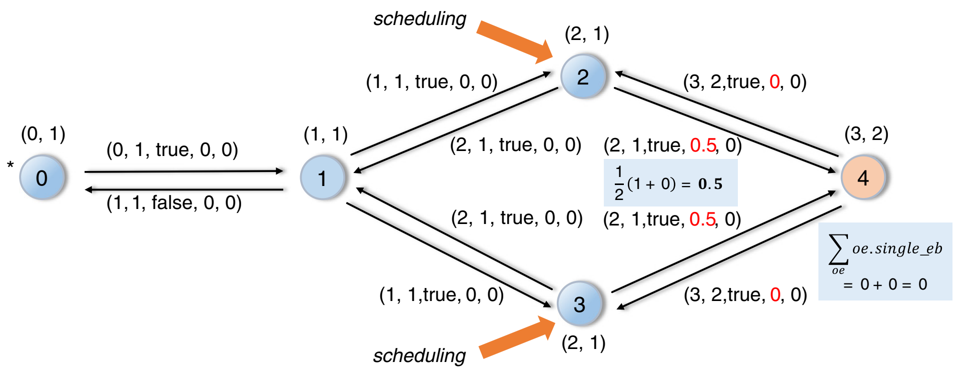


Fig. 14. A snapshot of the graph during Stage2.

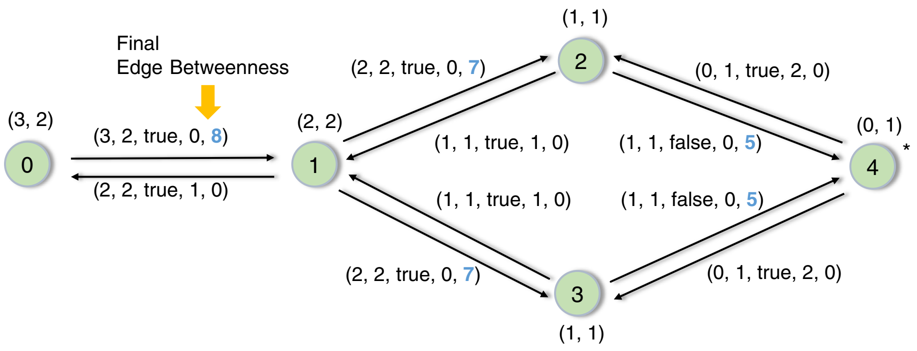


Fig. 15. The final scores of edge betweenness.

Even though one might think that SPB-VCA is simply an implementation of the GN algorithm using GraphChi, we did some clever engineering to make SPB-VCA more elegant and efficient.

- SPB-VCA shows the possibility of applying the *vertex*-centric model to an *edge*-centric task. The vertex-centric model has been mainly used for the tasks that compute values attached to vertices (e.g., PageRank and vertex centrality). However, edge betweenness is attached to an edge, not to a vertex. In SPB-VCA, the vertices are delegated to update the edge betweenness of their in-edge(s) on the shortest path.
- SPB-VCA reduces the number of times that each vertex is scheduled. SPB-VCA schedules only the *leaf vertices* when entering into Stage2. For example, in Fig. 13, **Vertex 4** is scheduled. Then, each leaf updates the edge betweennesses of its in-edges on the shortest paths (**Edge** ⟨2, 4⟩, ⟨3, 4⟩) and schedules its adjacent vertices (**Vertex 2, 3**) for subsequent operations. Here, the scheduled vertices can be processed *in parallel*. The process is repeated until a source vertex (**Vertex 0**) is scheduled. In short, Stage2 is executed in the following order: **Vertex 4** → **Vertex 2, 3** → **Vertex 1** → **Vertex 0**. Thus, every vertex is scheduled *only once*.

5. Performance tests

5.1. Data and environment

We used two sets of collaboration networks available at Stanford Large Network Dataset Collection [23]. Table 1 shows the details of the data. (i) **SPB-MRA:** We used Hadoop version 1.0.4 and Java version 1.6.0 to implement SPB-MRA. The performance tests were

Table 1
Data set description.

Data statistics	ca-GrQc	ca-HepTh
Vertices	5242	9877
Edges	28,980	51,971
Type	Undirected	Undirected
Average clustering coefficient	0.5296	0.4714
Number of triangles	48,260	28,339
Fraction of closed triangles	0.6298	0.284
Diameter	17	17
90-percentile effective diameter	7.6	7.5

Table 2
Amazon EC2 instance specifications for SPB-MRA.

Instance specification	
Instance type	m1.xlarge
Region	US-EAST (N. Virginia)
CPU	Intel(R) Xeon(R) CPU E5645 @ 2.40GHz
Memory	15 GB
Instance storage	1,690 GB
Processor architecture	64-bit
# of vCPUs	4
ECU	8
OS	Amazon Linux AMI release 2013.03
AMI	ami-05355a6c

conducted on a cluster that consists of 12 Amazon EC2 m1.xlarge instances. Table 2 shows the specifications of the Amazon EC2 instances used for SPB-MRA.

In addition, Fig. 16 shows an example of cluster configuration for running 64 tasks. One instance was dedicated to a master node, and the other instances were used for running map and reduce tasks. The map and reduce tasks were evenly distributed to the 11 instances (nodes), so the number of tasks on each instance was easily determined by the total number of tasks to invoke.

(ii) **SPB-VCA:** We used GraphChi's C++ version and g++ version 4.7.2 to implement SPB-VCA. The performance tests were conducted on a commodity PC equipped with a quad-core 3.40 GHz Intel i5 processor, 8 GB of main memory, and a 500 GB SATA HDD.

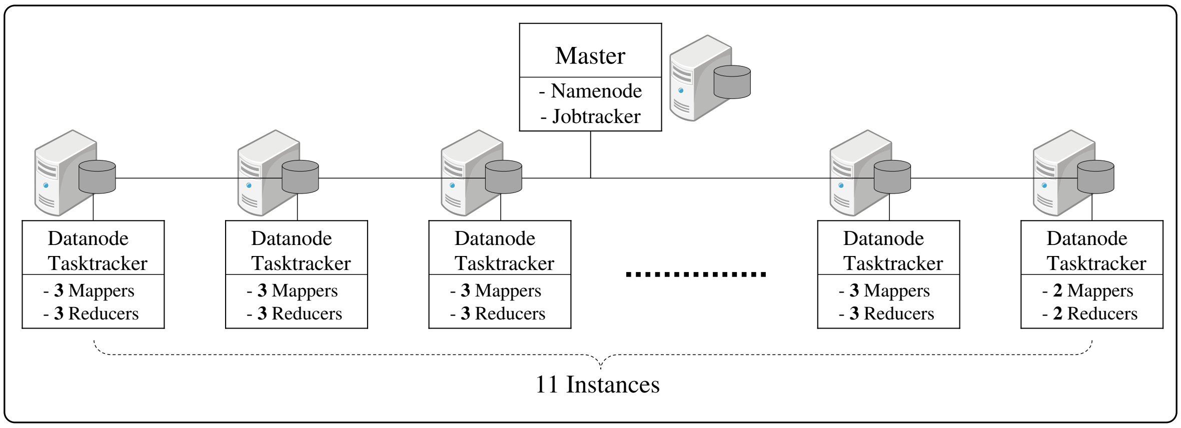


Fig. 16. A cluster configuration with 12 Amazon EC2 instances.

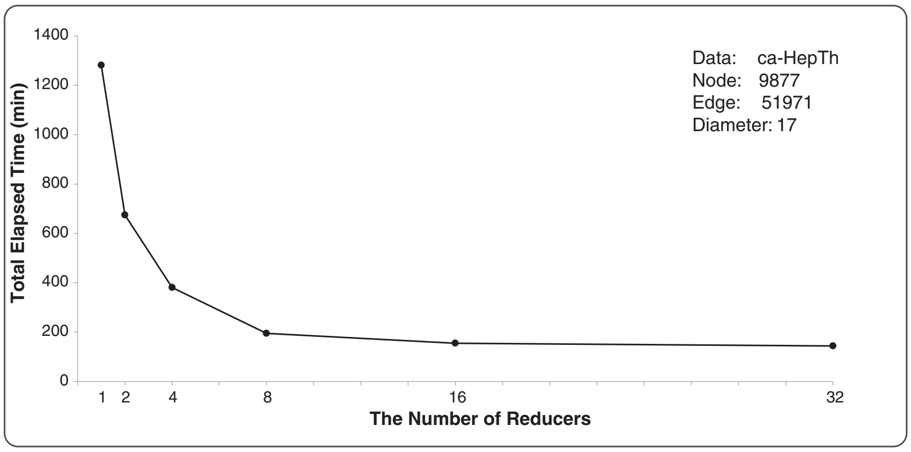


Fig. 17. Total elapsed time for the “ca-HepTh” data set.

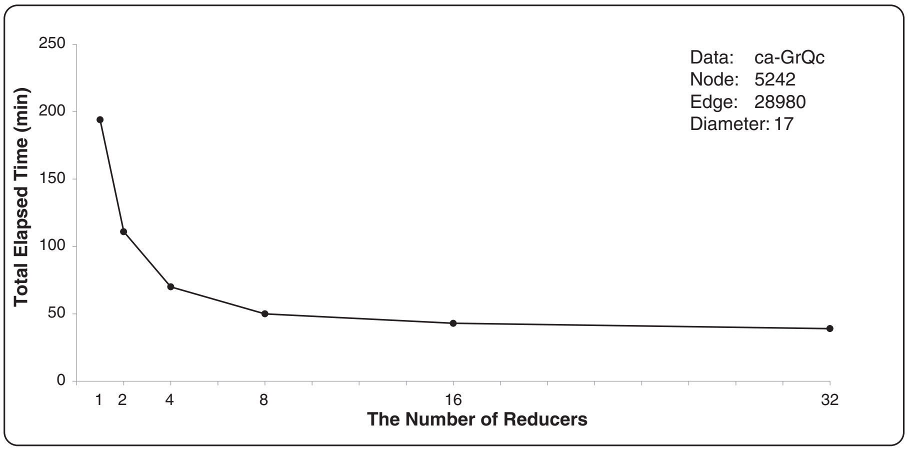


Fig. 18. Total elapsed time for the “ca-GrQc” data set.

5.2. Scalability of SPB-MRA

In order to show the scalability of SPB-MRA, we measured total elapsed time for one iteration of SPB-MRA while varying the number of reducers from 1 to 32. Figs. 17 and 18 show that elapsed time decreased by 6.6 times and 3.9 times respectively as the number of reducers increased until 8. This number of reducers was sufficient for these data sets, and adding more reducers was not effective. Since the “ca-HepTh” data set is twice larger than the “ca-GrQc” data set, the slope of the improvement became flat earlier in Fig. 18 than in Fig. 17.

5.3. Performance of SPB-VCA

We measured elapsed time for one iteration of SPB-VCA as well. Fig. 19 shows the elapsed time of SPB-MRA and SPB-VCA for the two data sets. The performance of SPB-VCA is higher than that of SPB-MRA by 4.4 times for the “ca-HepTh” data set and by 5.6 times for the “ca-GrQc” data set. It is surprising that SPB-VCA running on just a single PC (4 cores) outperformed SPB-MRA running on a cluster of twelve instances (96 cores). This result demonstrates some inefficiency of MapReduce for iterative graph computation and, at the same time, the benefits of the vertex-centric model.

To compare SPB-VCA with SPB-MRA is to compare apples with oranges. SPB-MRA suffers from the cost of disk I/O's to and from the HDFS as well as that of message passing among cluster nodes. Meanwhile, SPB-VCA introduces only local disk I/O's. Thus, if the data set in hand can fit in just a single machine, SPB-VCA is a better choice than SPB-MRA. On the other hand, if the data set is too large to hold in a single machine, it is absolutely imperative that we choose SPB-MRA.

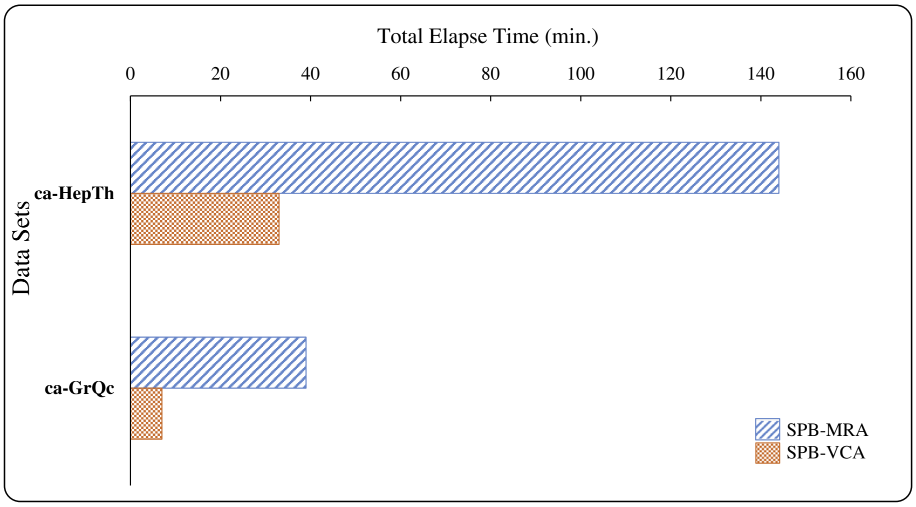


Fig. 19. Performance comparison between SPB-MRA and SPB-VCA.

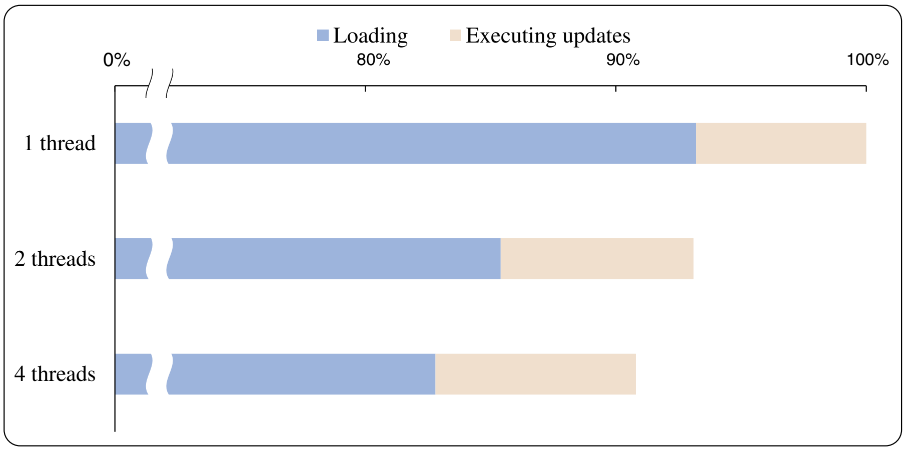


Fig. 20. Relative runtime of SPB-VCA while varying the number of threads.

Table 3
Accuracy of the approximation technique.

k_{iter}	Iteration	# of edges removed	F-score
1	40	40	1.0
2	20	40	0.875
4	10	40	0.9
5	8	40	0.875
8	5	40	0.9
10	4	40	0.8

5.4. Scalability of SPB-VCA

To magnify the effect of parallelization in SPB-VCA, the performance of SPB-VCA was tested for the LFR benchmark network data [24] generated with a high average degree 100.

Then, we measured the relative runtime of SPB-VCA for one iteration while varying the number of threads from 1 to 4, and the result is reported in Fig. 20. Unfortunately, the performance increased only by 8–10% as the number of threads increased. This phenomenon, however, almost conforms to the result reported for connected components by the GraphChi authors [14]. The main reason is that disk I/O's take up most of execution time since SPB-VCA is *not* a computation-demanding algorithm. In addition, we conjecture that GraphChi is not fully optimized yet as it is in an early stage of development.

5.5. Approximation accuracy of SPB-MRA and SPB-VCA

In order to show the approximation accuracy, we measured the F-score [25] with different k_{iter} values. k_{iter} means the number of edges to be removed for one iteration. The ground truth for the F-score is the set of 40 edges removed by selecting only one edge per iteration, as described in the original GN algorithm. In Table 3, as the value of k_{iter} increased, the error also increased because the edges not really having the highest edge betweenness could be removed more likely. However, even though we removed four edges at once, the F-score decreased only by 10%. Thus, it is shown that we can speed up by four times with only 10% error.

6. Conclusion

In this paper, we proposed two novel parallel algorithms, SPB-MRA and SPB-VCA, for discovering communities from large-scale networks. We adopted the MapReduce model and the vertex-centric model to parallelize the GN algorithm. Our two algorithms were implemented on top of Hadoop and GraphChi respectively. The results of performance evaluation demonstrated the benefits of parallel computing of edge betweenness as well as the advantages of the vertex-centric model over the MapReduce model for iterative graph computation. Our future work includes further improving the performance of SPB-VCA by adopting other parallel computing platforms.

Acknowledgment

This work was partly supported by the ICT R&D program of MSIP/IITP [B0101-15-1272, Development of Device Collaborative Giga-Level Smart Cloudlet Technology] and partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2011-0029185).

References

[1] Y. Li, D. Wu, J. Xu, B. Choi, W. Su, Spatial-aware interest group queries in location-based social networks, Data Knowl. Eng. 92 (2014) 20–38.
[2] B. Yang, J. Di, J. Liu, D. Liu, Hierarchical community detection with applications to real-world network analysis, Data Knowl. Eng. 83 (2013) 20–38.
[3] C. Shi, Y. Cai, D. Fu, Y. Dong, B. Wu, A link clustering based overlapping community detection algorithm, Data Knowl. Eng. 87 (2013) 394–404.
[4] S. Fortunato, Community detection in graphs, Phys. Rep. 486 (2010) 75–174.
[5] S. Lim, S. Ryu, S. Kwon, K. Jung, J.-G. Lee, LinkSCAN*: overlapping community detection using the link-space transformation, Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE) 2014, pp. 292–303.
[6] M.E. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69 (2) (2004) 026113.
[7] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[8] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, Scope: easy and efficient parallel processing of massive data sets, Proc. VLDB Endowment 1 (2) (2008) 1265–1276.
[9] J. Cohen, B. Dolan, M. Dunlap, J.M. Hellerstein, C. Welton, Mad skills: new analysis practices for big data, Proc. VLDB Endowment 2 (2) (2009) 1481–1492.
[10] A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a high-level dataflow system on top of Map-Reduce: the Pig experience, Proc. VLDB Endowment 2 (2) (2009) 1414–1425.
[11] H.-c. Yang, A. Dasdan, R.-L. Hsiao, D.S. Parker, Map-Reduce-Merge: simplified relational data processing on large clusters, Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data 2007, pp. 1029–1040.
[12] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data 2010, pp. 135–146.
[13] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: distributed graph-parallel computation on natural graphs, Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2012, pp. 17–30.
[14] A. Kyrola, G. Blelloch, C. Guestrin, GraphChi: large-scale graph computation on just a pc, Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2012, pp. 31–46.
[15] S. Moon, J.-G. Lee, M. Kang, Scalable community detection from networks by computing edge betweenness on mapreduce, Proceedings of the 2014 International Conference on Big Data and Smart Computing (BigComp) 2014, pp. 145–148.
[16] Apache Hadoop, http://hadoop.apache.org/ (accessed: August 1, 2013).
[17] Y. Tian, A. Balminx, S.A. Corsten, S. Tatikonda, J. McPherson, From "think like a vertex" to "think like a graph", Proc. VLDB Endowment 7 (3) (2013) 193–204.
[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, GraphLab: a new framework for parallel machine learning, Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI) 2010, pp. 340–349.
[19] B. Bahmani, R. Kumar, S. Vassilvitskii, Densest subgraph in streaming and MapReduce, Proceed. VLDB Endowment 5 (5) (2012) 454–465.
[20] Q. Li, Z. Wang, W. Wang, Y. Liu, P. Wang, T. Yu, LI-MR: a local iteration map/reduce model and its application to mine community structure in large-scale networks, Proceedings of the 2011 IEEE International Conference on Data Mining Workshops (ICDMW) 2011, pp. 174–179.
[21] Q. Yang, S. Lonardi, A parallel edge-betweenness clustering tool for protein–protein interaction networks, Int. J. Data Min. Bioinform. 1 (3) (2007) 241–247.
[22] Z.F. Zeng, B. Wu, T.T. Zhang, A multi-source message passing model to improve the parallelism efficiency of graph mining on MapReduce, Proceedings of the 2012 IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) 2012, pp. 2019–2025.
[23] Stanford large network dataset collection, http://snap.stanford.edu/data/ (accessed: August 1, 2013).
[24] A. Lancichinetti, S. Fortunato, F. Radicchi, Benchmark graphs for testing community detection algorithms, Phys. Rev. E 78 (4) (2008) 046110.
[25] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques, 3rd edition Morgan Kaufmann, 2011.