LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-751522

Computing Exact Vertex Eccentricity on Massive-Scale Distributed Graphs

K. Iwabuchi, G. Sanders, K. Henderson, R. Pearce

May 18, 2018

IEEE Cluster 2018
Belfast, United Kingdom
September 10, 2018 through September 13, 2018

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Computing Exact Vertex Eccentricity on Massive-Scale Distributed Graphs

Keita Iwabuchi, Geoffrey Sanders, Keith Henderson, and Roger Pearce
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Email: {**iwabuchi1**, sanders29, henderson43, rpearce}@llnl.gov

***Abstract*—The eccentricity of a vertex is defined as the length of the longest shortest path to any other vertex. While eccentricity is an important measure of vertex centrality, directly computing exact eccentricity for all vertices on large-scale graphs is prohibitively costly. Takes and Kosters proposed an iterative algorithm that uses multiple runs of single-source shortest path (SSSP) to compute lower and upper bounds on eccentricity at every vertex [1]. Their technique converges to exact eccentricity by performing SSSP from only a small percentage of vertices, when sources are efficiently selected. However, their source selection strategies do not always yield rapid convergence.**

We propose a *pincer movement* source selection algorithm that efficiently selects source vertices based on analysis of the lower and upper bounds produced by SSSP. We also leverage k-BFS, which runs breadth-first search (BFS) from multiple sources concurrently on HavoqGT, a high performance vertex-centric message-passing graph processing framework [2], [3], to achieve an additional significant performance improvement on distributed-memory systems.

We demonstrate that our novel source vertex selection strategy has better performance on various real-world graph datasets compared with the previous strategy [1]. In addition, we compute exact eccentricity for graphs with more than 1000$\times$ more edges (112B undirected edges) than graphs in the previous literature [1], [4].

I. Introduction

The eccentricity $\epsilon(i)$ of a vertex i is defined as the longest shortest path from i to any other vertex and can be computed directly by running a single-source shortest-path (SSSP) algorithm from i. Vertex eccentricity has been considered as an important metric of vertex centrality and is well-studied [5] — its applications include tissue characterization and classification in biology [6] and analyzing chemical structures [7], [8]. The exact eccentricity of all vertices in an unweighted graph can be computed by performing breadth-first search (BFS) from all vertices in a graph. However, the computation time of such a naïve approach is not acceptable for large-scale graphs (direct computation is $O(|V|*|E|)$ on unweighted graphs). Takes and Kosters proposed an iterative algorithm that bounds $\epsilon(i)$ to avoid executing BFS from every vertex in a graph [1]. The algorithm utilizes values achieved after running each BFS to update all vertices' lower and upper bounds, and the heuristic for BFS source selection is critical for rapid convergence to $\epsilon(i)$. The current source selection strategies are non-optimal for some datasets (see Section V-C) and we develop theory that informs our source selection algorithm in Appendix A.

We propose a novel source selection strategy which is designed for selecting efficient source vertices based on theory and detailed observations of the behavior of the bounding algorithm. The intuitive idea of our *pincer movement*[1] (PM) source selection strategy is that it attempts to select non-redundant source vertices in the graph *periphery*, and as well those near the graph *core*, in order to efficiently accomplish correct lower and upper bounds of many vertices by *attacking* from all sides. In Section IV, we first illustrate the properties of important vertices that would deliver the correct lower or upper bounds for large sets of vertices. Additionally, we describe details about the strategies to select those vertices efficiently by utilizing heuristic scores without causing any notable overhead.

In addition, we leverage k-BFS (also called multi-source BFS), which conducts BFS from multiple k sources simultaneously [4], [9]–[12]. When processing massive-scale graphs, using distributed-memory systems is common since it needs large amount of main memory capacity and computing power. The vertex-centric message-passing is a popular models for distributed memory graph processing [13], [14], and more mentioned in [15]. We utilize the idea of k-BFS in a vertex-centric framework to reduce the number of total messages for additional performance improvements. We implement k-BFS in HavoqGT, a high performance vertex-centric and message-passing graph processing framework [2], [3].

We demonstrate that our PM strategy outperforms the original source selection strategy combined with k-BFS (TK-k) by up to 3$\times$ speed up and 1.66$\times$ on average on various real-world graph datasets. We also found that there is a real-graph dataset for which TK-k does not efficiently select source vertices but PM does. Using our efficient source selection strategy and k-BFS, we were able to compute exact eccentricity in graphs with up to 112 billion edges; to our knowledge, this is more than three orders of magnitude larger than graphs in previous studies.

II. Preliminaries

We denote a graph by $G(V, E)$, where V is the set of n vertices and E is the set of m edges, pair-wise relationships of the form (i, j) for $i, j \in V$. In this work, we assume G is *undirected*, $(i, j) \in E$ iff $(j, i) \in E$, G has no *self-loop edges*,

[1]https://en.wikipedia.org/wiki/Pincer_movement

$(i,i) \notin E$ for any $i \in V$, and G is *unweighted*. To simplify the rest of description, we only consider *connected graphs* where all vertices in a graph are reachable from other vertices by a sequence of connected edges. The distance $d(v,w)$ between two vertices $v,w \in V$ is defined as the length of a shortest path between v and w.

A. Eccentricity

The eccentricity $\epsilon(v)$ of a vertex $v \in V$ is

$$\epsilon(v) := \max_{w \in V}\{d(v,w)\}. \quad (1)$$

The diameter $diam(G)$ of a graph G is given by $diam(G) = \max_{v \in V} \epsilon(v)$.

We can compute $\epsilon(v)$ by running a single source shortest path (SSSP) algorithm with v as the source. A naïve approach to computing exact eccentricity for all vertices is performing SSSP from all vertices (known as all-pairs shortest path). Given the assumption G is unweighted, we can use faster algorithms such as breadth-first search (BFS). Performing BFS from all vertices takes $O(nm)$ time as the time complexity of a single BFS is $O(m)$. While significant prior research has enabled large-scale BFS [16]–[20], driven in large part by Graph500 list [21], it is not realistic to perform BFS from all vertices on large-scale graphs.

B. Eccentricity Bounds

In order to avoid computing SSSP from every vertex, an algorithm that produces lower/upper bounds on eccentricity was proposed by Takes and Kosters [1]. If we run SSSP from a vertex s, then we know $\epsilon(s)$ and $d(s,w)$ for any other vertex $w \in V$. Then, we have the following lower/upper bounds

$$\epsilon(w) \quad \geq \quad L_w^{(s)} := \max\{\,\epsilon(s) - d(s,w),\, d(s,w)\,\}, \quad (2)$$
$$\epsilon(w) \quad \leq \quad U_w^{(s)} := \epsilon(s) + d(s,w). \quad (3)$$

Note that for any one source s, $L_w^{(s)} < U_w^{(s)}$ for any $w \neq s$. The basic approach is to run SSSP from a subset of vertices, $\mathcal{S} = \{s_1, s_2, ...\}$, and iteratively update the best lower and upper bounds

$$L_w \leftarrow \max\left\{\, L_w,\, L_w^{(s_k)} \right\}, \quad U_w \leftarrow \min\left\{\, U_w,\, U_w^{(s_k)} \right\}.$$

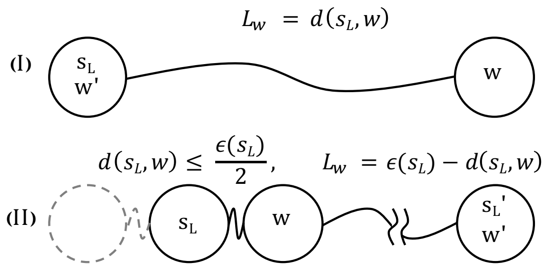
When $L_w = U_w$, we say vertex w is *solved*.

We observe that for a non-source vertex w to be solved there must be a pair of source vertices s_L, s_U such that $L_w^{(s_L)} = U_w^{(s_U)}$. We detail the implications of this observation in Appendix A, but summarize the main conclusions here. Figure 1 is an intuitive illustration of the configurations where Algorithm 1 solves a vertex w. A furthest vertex from w is denoted with a prime (i.e. w').

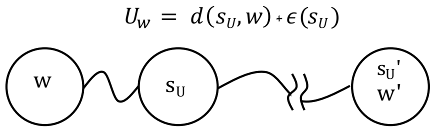
(a) **Lower Bound Is Correct.** There are two patterns where the correct lower bound of vertex w is obtained by Algorithm 1. First pattern (I) is that SSSP is performed from a furthest vertex of w, that is, $s_L = w'$ and $L_w = d(s_L, w)$. Another pattern (II) is that SSSP is performed from a vertex which shares a common furthest vertex with w and a shortest path from the source to the furthest vertex that goes through w exists, i.e., $L_w = \epsilon(s_L) - d(s_L, w)$. To meet with this pattern, $d(s_L, w)$ must be equal to or less than $\frac{\epsilon(s_L)}{2}$; otherwise s_L and w can not have a common furthest vertex.

Note that this situation also demonstrates where redundant sources may occur: there could be other vertices on the left side of s_L for which a shortest path to w' goes through w. Thus, when SSSP is performed from those vertices, the correct lower bound of w is also delivered.

(b) **Upper Bound Is Correct.** There is only one pattern where the correct upper bound of vertex w is obtained. The correct upper bound of w is delivered when SSSP is performed from a vertex which shares a common furthest vertex with w and is in between vertex w and the furthest vertex. The correct upper bound U_w of w is equal to $d(s_U, w) + \epsilon(s_U)$


$L_w = d(s_L, w)$
(I)
s_L w'
w
$d(s_L, w) \leq \frac{\epsilon(s_L)}{2}$, $L_w = \epsilon(s_L) - d(s_L, w)$
(II)
s_L
w
s_L' w'

(a) Cases Where Correct Lower Bound Is Delivered



(b) Case Where Correct Upper Bound Is Delivered

Fig. 1. Intuitive Ideas of Eccentricity Lower and Upper Bounds

Pseudocode for a serial algorithm that computes eccentricity via lower and upper bounds is given in Algorithm 1. We first initialize L and U, arrays for eccentricity lower and upper bounds of all vertices, with 0 and ∞, respectively (lines 1 – 4). Next a SSSP source vertex s is selected from the set of unsolved vertices W (line 7) and SSSP is performed to achieve distances to the other vertices in $G(V,E)$ from s (line 8). The eccentricity of vertex s is achieved and s is removed from the set of unsolved vertices W (line 9). Then, we apply Equations (2) and (3) to all unsolved vertices (lines 11 – 12). If a vertex is solved, it is removed from the set of unsolved vertices W (line 14). Repeat lines 6 – 17 until the set of unsolved vertices W becomes empty.

C. Interchanging Eccentricity Bounds Source Selection Strategy

By utilizing the eccentricity of a SSSP source vertex to solve other vertices' eccentricity, Algorithm 1 may drastically reduce the number of SSSPs to achieve the exact eccentricity of all vertices in a graph. However, selecting proper source vertices is critical for optimality.

```
Input: $G(V,E)$
Output: $\epsilon$ {Array of all eccentricity of $v \in V$}
1: for all $v \in V$ do
2:     $L[v] \leftarrow 0$ {L is an array for lower bounds}
3:     $U[v] \leftarrow \infty$ {U is an array for upper bounds}
4: end for
5: $W \leftarrow V$ {$W$ is a set of unsolved vertices}
6: while $W \neq \emptyset$ do
7:     $s \leftarrow$ SELECTSOURCE($W$)
8:     $d_s \leftarrow$ SSSP($G$, $s$) {Get distances from $s$ to all other vertices}
9:     $\epsilon[s] \leftarrow$ MAX($d_s$), $W \leftarrow W \setminus \{s\}$
10:    for all $w \in W$ do
11:        $L[w] \leftarrow$ MAX($L[w]$, $\epsilon[s] - d_s[w], d_s[w]$)
12:        $U[w] \leftarrow$ MIN($U[w]$, $\epsilon[s] + d_s[w]$)
13:        if $L[w] = U[w]$ then
14:            $W \leftarrow W \setminus \{w\}$
15:        end if
16:    end for
17: end while
```

Algorithm 1: ECCENTRICITY BOUNDS ALGORITHM [1]

Takes and Kosters proposed to select SSSP sources using the lower bound, upper bound, and degree of vertices based on their other work regarding graph diameter [22]. Specifically, the algorithm alternately selects source vertices with the smallest lower bound and the highest upper bound at each iteration; when two vertices have the same lower or upper bound, degree is used to break ties.

D. Optimization Technique for Single-Degree Vertices

To reduce the number of iterations required in Algorithm 1, Takes and Kosters also proposed an optimization technique for single degree vertices [1]. Let v be a vertex which has only one edge, and let w be the *parent*, the single vertex that is connected to v. As paths from vertex v go through vertex w to access other vertices in the graph, the eccentricity of vertex v is automatically determined as $\epsilon(v) = \epsilon(w) + 1$. After the eccentricity of a source vertex is achieved by performing SSSP, this technique is applied for all single degree vertices the source vertex has (this routine can be inserted between line 9 and 10 in Algorithm 1).

III. K-SOURCES BREADTH-FIRST SEARCH (k-BFS) ON DISTRIBUTED-MEMORY

Along with using the eccentricity bound algorithm, in this section, we describe another key technique, k-BFS, to accelerate computing eccentricity.

Due to the high demands for performing analysis on massive real-world graphs, there has been much recent work on graph processing frameworks in distributed computing systems. To leverage those studies, we design our eccentricity computing algorithm targeting vertex-centric message-passing communication model, which is a popular graph processing computation and communication model widely used in many frameworks such as Pregel [13], Giraph [14], and HavoqGT [3]. Vertex-centric graph processing is represented as a message exchange between adjacent vertices — each vertex sends/receives messages to/from its neighbor(s) as the core algorithmic operation. In such frameworks, reducing the number of messages is important to accelerate graph processing workloads.

Aiming at earning another significant performance improvement for BFS on such graph processing frameworks, we leverage k-BFS (also known as multi-source BFS), which conducts BFS from k sources simultaneously. When a vertex is visited by multiple different source vertices at the same level (the distance from a source vertex), messages that will be sent from the vertex to its neighbors can be aggregated. Specifically, each message holds the *visit information* of multiple source vertices. The *visit information* consists of flags denoting which sources have visited the vertex, represented as a bitmap for memory and message compactness.

A. Pseudocode

In order to fully utilize the advantage of k-BFS, we use level-synchronous BFS, where all vertices in level l are visited before visiting any vertices in level $l + 1$. A pseudocode of a level-synchronous k-BFS algorithm with a vertex-centric message-passing communication framework is described in Algorithm 2.

As the pseudocode is designed for a distributed-memory framework, we assume that $G(V,E)$ is distributed across multiple processes; to simplify the pseudocode, we assume that each vertex is assigned to only one process, and all edges of a vertex are stored in the same process where the vertex is assigned. The pseudocode takes $G(V,E)$ and k BFS source vertices. A function *LOCAL(input_set)* returns a set of vertices assigned to the process from the *input_set* argument.

At the beginning, variables *visited* and *distance* are initialized (line 1 – 6) to ∞. Next, variables *visited* and *distance* for the source vertices assigned to each process are initialized (line 7 – 10): each source vertex is marked as "visited" by itself, and the distance from itself is set as 0. At line 11, a set of frontier vertices *frontier* is initialized with the source vertices assigned to each process. Line 13 is the beginning of the main loop; *global_bfs_termination_check(frontier)* returns *true* when variables *frontier* in all process are empty, that is, there are no vertices to be visited. Line 14 – 18 are a scatter step; vertices in *frontier* send messages to their neighbors to visit. Each message is initialized with its destination vertex n and visited information *visited[v]* of vertex v which the message is sent from. Line 19 – 28 are for a visit step where processes receive messages from its neighbors. At line 22, it checks whether destination vertex dv has visited by source vertex s. If dv has not yet visited by s, we update the visit information of dv and put it into the next frontier (line 23 – 25). At line 31, we synchronize all processes before moving to the next level.

```
Input: $G(V, E)$
Input: sources {BFS k source vertices}
Output: distance {2D array of distances from source vertices}
 1: for all v ∈ LOCAL(V) do
 2:   visited[v] ← ∅
 3:   for all s ∈ sources do
 4:     distance[v][s] ← ∞
 5:   end for
 6: end for
 7: for all s ∈ LOCAL(sources) do
 8:   visited[s] ← visited[s] ∪ s
 9:   distance[s][s] ← 0
10: end for
11: frontier ← LOCAL(sources)
12: level ← 1
13: while not global_bfs_termination_check(frontier) do
14:   for all v ∈ frontier do
15:     for all n ∈ neighbor[v] do
16:       SEND(msg_queue, n, visited[v])
17:     end for
18:   end for
19:   for all msg ∈ RECEIVE(msg_queue) do
20:     dv ← msg.destination_vertex
21:     for all s ∈ msg.visited do
22:       if s ∉ visited[dv] then
23:         visited[dv] ← visited[dv] ∪ s
24:         distance[dv][s] ← level
25:         next ∪ {dv}
26:       end if
27:     end for
28:   end for
29:   frontier ← next
30:   level ← level +1
31:   global_sync()
32: end while
```

Algorithm 2: Pseudocode of Level-synchronous k-BFS on a Vertex-centric Message-passing Communication Framework

B. Space Complexity

Compared with a single source (conventional) BFS, k-BFS requires more memory for the following two data structures:

- **Message**. To propagate by which source vertices the vertex was visited at the level, k bits of additional space is used.
- **Distance array**. To store the distances from each source vertex to other vertices, the length of the array is $|V|k$ (10 – 16 bits would be enough for each element in many real-world graphs).

IV. Pincer Movement Source Selection Algorithm

In this section, we propose the *pincer movement* (PM) source selection strategy which is designed for selecting efficient source vertices based on the theory in Appendix A and the detailed observations of the behavior of the bounding algorithm described in Section II-B. First we illustrate the properties of important vertices that would deliver the correct lower or upper bounds for large sets of vertices. We describe the details about our strategies to select those important vertices efficiently.

A. Important Vertices To Achieve Correct Lower and Upper Bounds

1) For Lower Bound: For a vertex w whose lower bound is not yet correct, there are two types of source vertices that deliver the correct lower bound of w as illustrated in Figure 1: (I) furthest vertices of w; (II) vertices that shares a common furthest vertex with w and have a shortest path to the furthest vertex that goes through w. However, due to the following two reasons, we consider furthest vertices (type I) as the most important vertices and select them with the highest priority.

- There are many graph structures that make it obvious that furthest vertices should be selected as source vertices. For example, to achieve the correct lower bound of a single degree vertex, SSSP has to be started from either the vertex itself or one of its furthest vertices. Given the well known observation that many single degree vertices are found in real-world graphs, selecting furthest vertices is highly beneficial since it is common that multiple unsolved vertices share a common furthest vertex.
- As it has been reported in many studies (for example in [23]), it is likely that real world graphs have long tails and/or *skewed* structures; accordingly, we believe that the number of those furthest vertices is considerably smaller than the number of total vertices in the graph. Moreover, if this expectation is true, we can find furthest vertices without any additional heavy work by using the results of previous BFS.

It is less straight-forward to directly and efficiently choose source vertices of type II for a large set of unsolved vertices.

2) For Upper Bound: The correct upper bound of vertex w is delivered when SSSP is performed from a vertex situated in between w and a common furthest vertex with w (see the illustration at the bottom of Figure 1). Let W be a set of vertices that share a common furthest vertex w'; to maximize the number of vertices that achieve correct upper bounds, the priority of each vertex $w \in W$ must be the inverse of the distance to w'.

B. Source Selection Strategies

Here we describe details about the source selection strategies we designed to efficiently select such important vertices by utilizing heuristic scores without causing any notable overhead.

1) Selecting Non-Redundant Furthest Vertices: After running k-BFS, we can easily get the list of furthest vertices of the source vertices. However, we need to take care of *redundant* vertices that share the exact or almost equal shortest paths for other further vertices. Figure 2 shows an example of redundant vertices. Let the three vertices on the far right be furthest vertices of vertex w. In this example, two of the three vertices are considered as redundant vertices. This is a common phenomena we have observed in many large, real-world graphs.

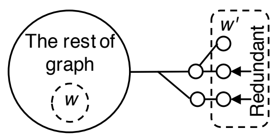


Fig. 2. Example of Redundant Furthest Vertices

However, it is not easy to determine whether multiple furthest vertices are sharing similar shortest paths; therefore, we take a heuristic approach to remove such redundant vertices so as to not cause any notable overhead. We select only one furthest vertex for each source vertex by comparing hash values of the furthest vertices. In addition, we remove the furthest vertices which have selected before — note that even if the exact eccentricity of a vertex is already solved, it is still one of candidates of source vertices. By applying this filter, we will achieve less than or equal to k source vertices for the next k-BFS.

2) Selecting Central Vertices: When the number of selected furthest vertices is less than k, we select the remaining source vertices as follows. We combine the following 3 sub-strategies to select vertices that are expected to deliver the correct upper bounds efficiently. Specifically, we use the three sub-strategies in the nested order of *max*$|U$ - $L|$, *min(L)*, *Degree*; when multiple vertices have the same priority next sub-strategy is used to break a tie.

(a) **min(L).** Takes et al. proposed *min(L)* as one of source selection strategies inspired by traditional branch-and-bound algorithms [1]. We also use this source selection criteria and provide some additional insight to why it is useful. Here, let s be a source vertex, t be a furthest vertex of s, and i be a vertex sharing the same furthest vertices with s (thus, i is on a shortest path between s and t). As we move down a shortest path from source s towards t, $d(s,i)$ is increased and $\epsilon(s) - d(s,i)$ is decreased. Consequently, the vertex at the center of a shortest path has the minimum lower bound. This shows that fairly central vertices tend to be selected by this criteria, particularly in the case where s was a furthest vertex in a previous run. Given this property, we select vertices that have the smallest lower bounds.

(b) **max$|$U - L$|$.** If we just use only *min(L)*, there is a possibility that our source selection strategy selects sources in some small area. To deal with the concern, we use this score along with *min(L)* to select vertices in an *unexplored area*, relatively far from previous sources. Specifically, we select vertices which have the largest |U - L| to find unexplored areas since the purpose of Algorithm 1 is decreasing the gap between upper and lower bound of each vertex.

(a) **Degree.** When two vertices have the same *min(L)* and $|$U - L$|$, we use the degrees of the vertices. Although the degree of a vertex does not directly determine the position of the vertex, we deem it is a useful score because all other vertex properties being equivalent, higher degree vertices may be on more shortest paths than lower degree vertices. As for the source selection for the first iteration (BFS), all vertices are compared by this strategy since other information like lower and upper bounds are same. Takes et al. also use this to break ties.

When 3 sub-strategies return the same values, a hash value of each vertex's ID is used as the final tie breaker. Note that all vertices that are marked as the furthest vertices of past source vertices, including redundant furthest vertices, are removed from the candidate vertices for this strategy in the beginning. When all unsolved vertices are redundant furthest vertices, we select sources by only using *Degree*.

V. Performance Evaluation

In this section, we evaluate the performance of our PM algorithm on multiple real-world graph datasets. In addition, at the end of this section, we show eccentricity distributions of some large-scale real-world graphs.

A. Experimental Setup

1) Dataset: To perform the series of experiments, we used 9 real-world graph datasets listed in Table I. We use 6 datasets (yt, wk_t, rn, or, pt, fr) from Stanford Large Network Dataset Collection (SNAP Datasets) [24] and 3 other real-world graph datasets (wk_h, tw, wb).

Wikipedia hyperlink graph (wk_h) is a new graph dataset that we have curated. We constructed the dataset by extracting hyperlinks between all pages in English Wikipedia dump as of July 1st, 2017. The dump data contains the entire edit history of the pages in English Wikipedia from January 15, 2001, which is the date English Wikipedia was founded. The graph contains hyperlinks not only between article pages but also other types of pages such author (user) pages and Category pages.

Since our exact eccentricity algorithm assumes that graphs are *undirected*, have no *self-loop edges*, and are *connected*, we removed all duplicated edges and self-loop edges; we ran a connected component algorithm and extracted the largest component from each graph. The numbers of vertices and edges shown in Table I are after this preprocessing. To our knowledge, this is the first effort which computed all exact eccentricity on graphs larger than 40 million edges. We have processed graphs 1 – 3 order(s) of magnitude larger than prior work.

2) Implementation: For performance comparison, we implemented the source selection algorithm and single degree vertex optimization technique proposed by Takes et al. — details are described in Section II-C and Section II-D. To conduct a fair performance comparison with our PM algorithm, we implemented their algorithm with k-BFS, and we refer to this implementation as *TK-k*.

We implemented TK-k and our PM algorithm on HavoqGT [2], [3]. HavoqGT is a high performance vertex-centric and

TABLE I
GRAPH DATASETS

Graph Name	Type	# Vertices in the Largest CC	# Unique Edges
com-Youtube (yt) [25]	Social network	1,134,890	2,987,624
California road network (rn) [26]	Road network	1,957,027	2,760,388
wiki-Talk (wk_t) [27], [28]	Communication network	2,388,953	4,656,682
com-Orkut (or) [25]	Social network	3,072,441	117,185,083
cit-Patents (pt) [29]	Citation network	3,764,117	16,511,740
Wikipedia hyperlink (wk_h)	Hyperlink graph	40,311,467	851,853,231
Twitter (tw) [30]	Social network	41,652,230	1,202,513,046
com-Friendster (fr) [25]	Social network	65,608,366	1,806,067,135
Webgraph (wb) [31]	Hyperlink graph	3,355,386,234	111,635,885,335

message-passing graph processing framework; it is written in C++ and uses MPI for interprocess communication. HavoqGT constructs graph data into files so that it can utilize node-local storage devices; however, we store the graph data (files) into tmpfs space in this evaluation.

3) Machine: Our experiments were run on the Quartz cluster at LLNL. Each compute node has two Intel Xeon E5-2695 CPUs (18 physical cores per socket) with 128 GB DRAM (available tmpfs size is 64GB); compute nodes are connected with Intel Omni-Path. We used up to 128 compute nodes with 36 MPI ranks per node. The actual number of compute nodes used for each graph is listed in Table II).

B. k-BFS Performance

To demonstrate the efficiency gains with k-BFS, we show the performance of k-BFS varying k, number of concurrent BFSs, in Figure3. We used the Twitter (tw) graph on 64 compute nodes with BFS source vertices selected randomly. The y-axis denotes the execution time to perform 128 BFSs in seconds, that is, 128 independent BFSs are required with $k = 1$ while only a single k-BFS is required with $k = 128$. The total execution time decreases as k increases from 1 to 128. It required 76.8 seconds to finish with $k = 1$ while it took only 8.9 seconds with $k = 128$; thus, we were able to achieve 8.7× speed up by employing k-BFS.

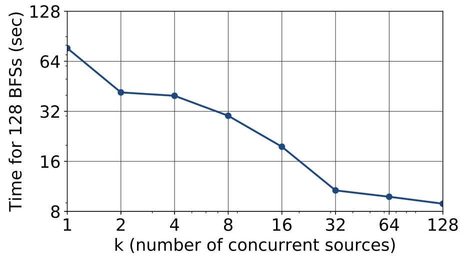


Fig. 3. Execution Time of k-BFS computing 128 BFS traversals and varying k on the Twitter (tw) graph.

C. Performance Comparison between TK-k and PM

We evaluated the performance of *PM* against TK-k in terms of the number of iterations and execution time. Both algorithms use k-BFS, we used $k = 64$ for com-Youtube (yt) and wiki-Talk (wk_t) and $k = 128$ for the rest of datasets except Webgraph (wb). For Webgraph (wb), it turned out that $k = 4$ was enough to finish the whole eccentricity computing in spite of the size of the graph. We used 1 compute node for com-Youtube (yt) and wiki-Talk (wk_t); 8 compute nodes for California road network (rn), com-Orkut (or), and cit-Patents (pt); 64 compute nodes for Twitter (tw), com-Friendster (fr) and Webgraph (wb); 128 compute nodes for Wikipedia hyperlink (wk_h).

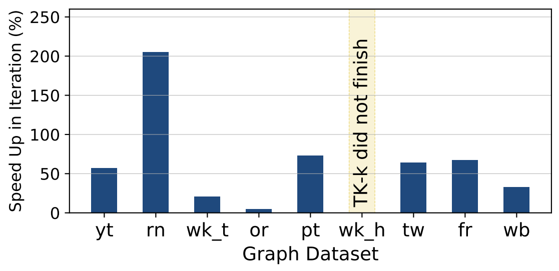
Results of the performance comparison experiment are presented in Figure 4. Table II has the actual number of iterations and execution time (sec.) TK-k and PM took, as well as the number of k-BFS sources (k) and compute nodes used.

For Wikipedia hyperlink (wk_h), we stopped the experiment with TK-k before it finished but after a reasonable time. It turned out that TK-k did not efficiently select source vertices even after running more than twice as long as PM, resulting in running k-BFS from many vertices without receiving benefits from eccentricity lower and upper bounds algorithm. In Section V-D1, we describe more details regarding the progress of eccentricity computing — the number of unsolved vertices over time — on Wikipedia hyperlink (wk_h) with TK-k.

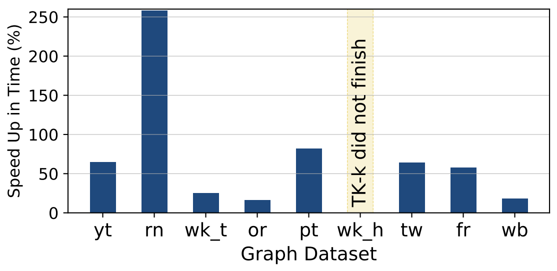
1) Performance Improvement in Number of Iterations: Figure 4a shows the performance improvements of PM over TK-k in terms of the number of iterations to finish eccentricity computing. The values shown in the figure are calculated by $\frac{I_{TK\text{-}k}}{I_{PM}} - 1$, where I is a number of iterations. Overall, PM exhibits better performance than TK-k on all graph datasets — PM achieved its best result with 205.4% speed up on California road network (rn) and 65.6% speed up on average on all datasets. As for large-scale graphs, PM outperforms TK-k by 63.8% on Twitter (tw), 66.7% on com-Friendster (fr), 33.3% on Webgraph (wb).

It is important to note that the number of source vertices required to compute the eccentricity of all vertices are surprisingly small on com-Friendster (fr) and Webgraph (wb), even though the two datasets have 65 million and 3.3 billion vertices respectively. For the Webgraph (wb) with our PM algorithm, only 3 iterations with $k = 4$, that is, 12 source vertices, are needed for solving all vertices' eccentricity.

2) Performance Improvement in Execution Time: Next, Figure 4b shows the performance improvements of PM over TK-k in total execution time. The reported total execution time does not include the graph construction or pre-processing time, while it includes the other steps such as source selec-



(a) Performance Improvement in Number of Iterations



(b) Performance Improvement in Execution Time

Fig. 4. Performance Improvement of PM Over TK-k (PM outperforms TK-k in all cases; speed up % is calculated by $\frac{TK\text{-}k}{PM} - 1$)

TABLE II
NUMBER OF ITERATIONS AND EXECUTION TIME PM TOOK

Graph Name	(k)	# Iterations		Time in sec.		(# compute nodes)
		TK-k	**PM**	TK-k	**PM**	
com-Youtube (yt)	64	58	**37**	28.19	**17.09**	1
California road network (rn)	128	2086	**683**	9609.23	**2682.41**	8
wiki-Talk (wk_t)	64	222	**183**	144.96	**115.67**	1
com-Orkut (or)	128	4994	**4746**	12084.22	**10389.95**	8
cit-Patents (pt)	128	2681	**1554**	2383.96	**1308.54**	8
Wikipedia hyperlink (wk_h)	128	(N/A)	**4326**	(N/A)	**20568.47**	128
Twitter (tw)	128	1864	**1138**	11842.59	**7229.32**	64
com-Friendster (fr)	128	20	**12**	156.71	**99.34**	64
Webgraph (wb)	4	4	**3**	2636.7	**2238.04**	64

tion, eccentricity bound algorithm, and single-degree vertices optimization time in TK-k. However, it resulted in that k-BFS accounted for most of execution time and other steps took negligible time — this result indicates that there is no notable overhead in PM algorithm for selecting sources. The performance comparison metric used in Figure 4b is defined as $\frac{T_{TK\text{-}k}}{T_{PM}} - 1$, where T is a total execution time.

Same as observed in Figure 4a, PM outperforms TK-k on all graphs. On average PM achieved 73.3% of speed up over TK-k. As for graphs with over 10 million vertices, PM is 63.8%, 57.8%, and 17.8% faster than TK-k on Twitter (tw), com-Friendster (fr), and Webgraph (wb) respectively.

D. Detailed Analysis

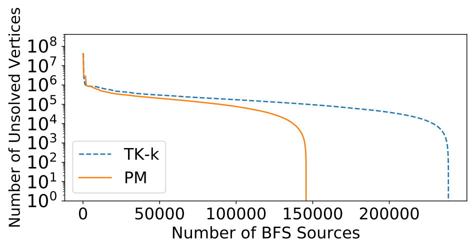
1) Progress of Eccentricity Computing: The figures in Figure 5 show the progress of eccentricity computing on the four largest graph datasets in our study: Wikipedia hyperlink (wk_h), Twitter (tw), com-Friendster (fr), and Webgraph (wb). For each figure, the x-axis denotes the number of BFS sources; y-axis (log scale) denotes the number of unsolved vertices remaining.

First, for Twitter (tw), TK-k and PM were able to solve 96% and 93% of vertices with the first 1024 BFS sources, respectively. After around 100K BFS sources the gap between the two lines started increasing gradually; TK-k finished with 239K sources while PM required only 146K sources. Second, for Wikipedia hyperlink (wk_h), TK-k was not able to efficiently select source vertices; as a result, only 2.3M vertices were solved with 963K BFS sources. However, PM was able to select efficient sources and solved 97% of vertices (39M vertices) with only 1024 BFS sources, and used 554K to solve all vertices (40M vertices). Finally, for com-Friendster (fr) and Webgraph (wb), TK-k and PM were able to compute all eccentricity in the graphs with remarkably small number of sources regardless of the large size of the two graphs.

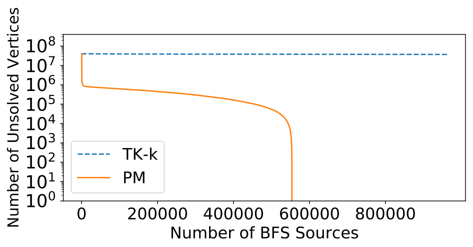
2) Breakdown of Selected Source Types: Another detailed analysis for PM algorithm is a breakdown of the number of selected sources by source types. In Figure 6, *Non-Redundant Furthest* and *Central* corresponds to the two types of source vertices PM selects, non-redundant furthest vertices and central vertices, as described in Section IV. *Redundant Furthest* corresponds to the furthest vertices marked as "redundant" and were not selected by the two source selection strategies; thus, after performing k-BFS from all vertices belong *Non-Redundant Furthest* and *Central*, we had to perform k-BFS also from *Redundant Furthest* vertices if they remained. The y-axis denotes the number of selected vertices by source type. Note that vertices selected at the first step and the last step with a case where the number of left unsolved vertices is equal to or less than k are not included in the reported numbers.

As we expected, significantly small number of non-redundant furthest vertices were selected in overall. Specifically, 2, 7, 4, and 2 non-redundant furthest vertices are selected on Wikipedia hyperlink (wk_h), Twitter (tw), com-Friendster (fr), and Webgraph (wb), respectively. On the other hand, no redundant vertices were selected on all graphs except com-Orkut (or).

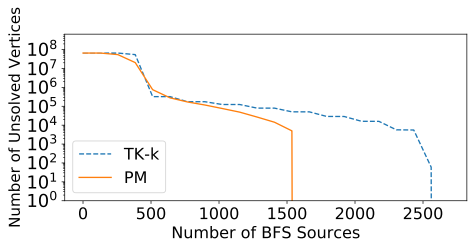
The ratio of the total selected sources over the total number of vertices for each graph by PM is 0.2%, 4.5%, 0.5%, 19.8%, 5.3%, 1.4%, 0.3%, 2.3E-05%, and 3.6E-09%, respectively



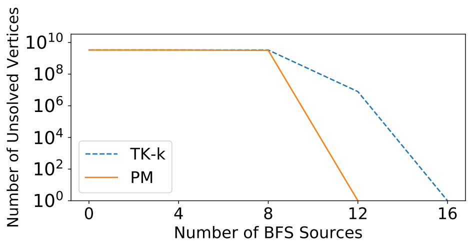
(a) Twitter (tw)



(b) Wikipedia Hyperlink (wk_h)



(c) com-Friendster (fr)



(d) Webgraph (wb)

Fig. 5. Progress of Number of Unsolved Vertices

(from yt to wb in Figure6).

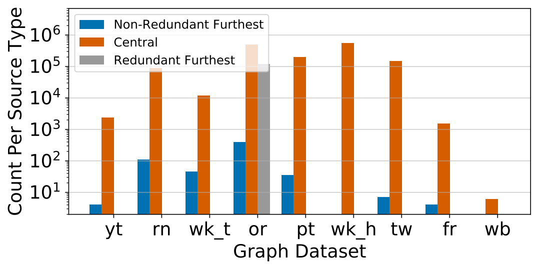


Fig. 6. Selected Source Type in PM Algorithm

E. Scaling Study

We performed scaling study of PM on Twitter (tw), com-Friendster (fr) and Webgraph (wb). We used 32, 64 and, 128 compute nodes on Twitter (tw) and com-Friendster (fr); 64 and 128 nodes on Webgraph (wb) because it did not run with 32 nodes due to out-of-memory error during the graph construction step — the size of constructed graph were around 2 TB which is close to the total available tmpfs size on the compute nodes. The execution time of PM varying the number of compute nodes is shown in Table III. PM achieved 1.7× and 3.0× speed up by increasing the number of compute nodes from 32 to 128 on Twitter (tw) and com-Friendster (fr), respectively; 1.7 speed up from 64 to 128 compute nodes on Webgraph (wb).

F. Eccentricity Computing with Error ≤ 1

While exact eccentricity is used as a very important metric, it would be interesting if our PM algorithm also works well on computing other important metrics. Thanks to strong error tolerant properties of machine learning algorithms, there are

TABLE III
EXECUTION TIME (SEC.) OF PM (STRONG SCALING)

	# Compute Nodes		
	32	64	128
Twitter	10432.2	7229.3	6187.2
com-Friendster	163.0	99.3	54.3
Webgraph	(N/A)	2238.0	1285.5

some cases where those algorithms can accept data with a certain error. We found that PM also can be beneficial for computing eccentricity with a certain error. Figure 7 shows how many BFS sources TK-k and PM needed until all vertices $w \in V$ met with $|U_w - L_w| \leq 1$, i.e., the gap of the lower and upper bounds of every vertex becomes less than or equal to 1.

Regarding small and middle size graphs (5 graphs on the left side), PM shows up to 5.4× speed up against TK-k. Same as the previous results, TK-k did not finish on Wikipedia hyperlink (wk_h) within a reasonable time while PM needed only 640 BFS sources. As for other large-scale graphs (3 graphs on the right side) TK-k and PM both showed similar results: TK-k required 1536, 512, and 16 sources while PM required 1792, 512, 12 sources, respectively.

G. Eccentricity Distribution

Finally, we show the exact eccentricity distributions of some large-scale graphs. Leveraged by our PM algorithm, we successfully achieved the exact eccentricity distributions for the largest real-world graphs studied in the literature with reasonable execution time. We show the eccentricity distributions in Figure 8. First, only one peak and very skewed distribution were observed in each graph. For our Wikipedia Hyperlink Graph (wk_h), its diameter is 67; the eccentricity of 99.9% of vertices are between 43 and 45; there are 3 vertices

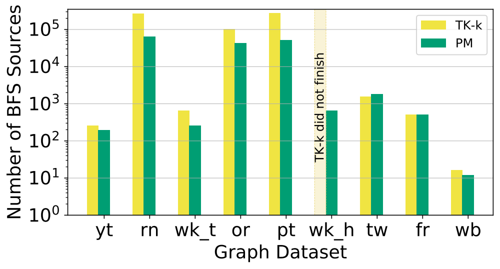
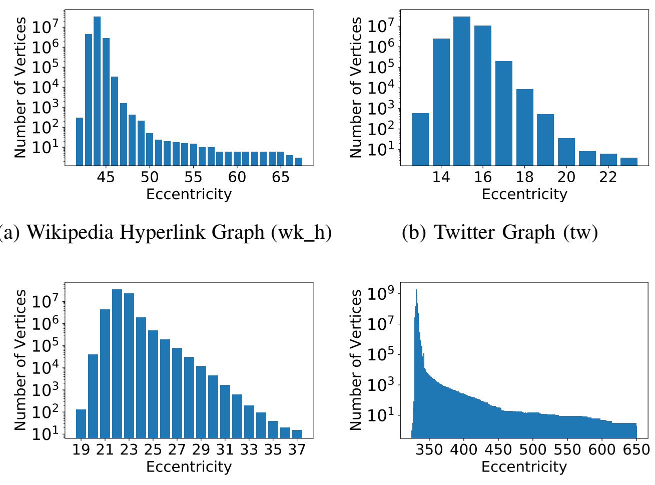


Fig. 7. Number of BFS Sources To Compute All Eccentricity with Error ≤ 1

which have the largest eccentricity (67). For Webgraph (wb), its peak is at eccentricity 331 and contain 57% of vertices; the eccentricity of 86% of vertices are between 330 and 333; 2 vertices have the largest eccentricity (650).



(a) Wikipedia Hyperlink Graph (wk_h) (b) Twitter Graph (tw)

(c) com-Friendster (fr) (d) Webgraph (wb)

Fig. 8. Eccentricity Distribution of Large Graphs

VI. Related Work

In addition to Takes and Kosters [1], Shun et al. [4] also conducted a study on large-scale graph eccentricity. However, this work mainly focuses on eccentricity estimation on a shared-memory machine. Taking our distributed system approach, we could compute exact eccentricity on up to more than three orders of magnitude larger graphs than these previous studies. Even compared with the estimation of eccentricity done by [4], we were able to compute exact eccentricity on an approximately 17 times larger graph.

Sariyüce et al. proposed graph manipulation techniques for fast centrality computation [32]. They described some vertex types can be useful to compress and split graphs in order to reduce the amount of work to compute centrality. Although, finding some types of vertices, such as articulation/bridge vertices and *side* vertices, involves high overhead without developing efficient techniques, their proposed techniques could bring an additional speed up to eccentricity computing. We address detailed evaluation and developing new techniques for finding such featured vertices as future work.

There are some studies about k-BFS: targeting shared-memory machines [9], [10], including GPU and Intel Xeon Phi [11]; GPU and distributed-memory systems [12]. Shun et al. [4] also applied k-BFS to compute exact eccentricity, yet on a shared memory machine. Our work is different in that we demonstrate the impact of k-BFS on a distributed message-passing communication framework. Some proposed techniques by those studies, such as efficient multiple source selection strategies in terms of the performance of k-BFS, could be applied to the distributed setting.

As for speeding up BFS performance, direction-optimizing (DO) BFS is a well known algorithm [18]. Compared with the performance improvement by DO BFS in a shared- and distributed-memory systems [18], [19], our k-BFS was able to achieve similar speed up on the same Twitter graph (see Figure3 in SectionV-B) against the conventional (top-down) BFS algorithm. Moreover, Pan et al. reported that HavoqGT's delegation technique (splitting high-degree vertices) and DO BFS fitted together well [33]. Combining k-BFS and DO BFS and evaluate its performance would be one of our future work.

VII. Conclusion

We proposed the pincer movement (PM) source selection strategy to accelerate computing eccentricity with the lower and upper bounds algorithm from [1]. We proved several theoretical results that were used to improve heuristics in the design of the PM source selection strategy. Furthermore, we employed k-BFS on a vertex-centric message-passing graph processing framework to achieve an additional significant performance improvement on distributed-memory systems. Compared with the original source vertex selection strategy from [1] combined with k-BFS (TK-k), our PM algorithm was able to achieve up to 3× speed up (1.66× on average) on various real-world graph datasets. These advances allowed us to compute exact eccentricity in graphs with up to 112 billion edges; to our knowledge, this is more than three orders of magnitude larger than graphs in previous studies.

Appendix A
Theoretical Results

Graph distance satisfies the *triangle inequality*,

$$d(i,k) \leq d(i,j) + d(j,k). \quad (4)$$

We use this, properties of shortest paths, and the definition of eccentricity in Equation (1) to derive several conditions on solved vertices that we use to justify the source selection strategies we present in Section IV. First, we present an important lemma that we leverage throughout this section.

Lem. 1. (On a Shortest Path) Let $i, j, k \in V$. If $d(i,k) = d(i,j) + d(j,k)$, then there exists a shortest path from i to k through j.

Proof. Let $\mathcal{P}_1^*$ be any shortest path from i to j and $\mathcal{P}_1^*$ be any shortest path from j to k. Concatenating $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ gives a walk containing j. To show paths $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ share no other vertex than j, we form a contradiction. If $q \neq j$ is in both $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$, then $d(i,q) < d(i,j)$ and $d(q,k) < d(j,k)$ and

$$d(i,k) \leq d(i,q) + d(q,k) < d(i,j) + d(j,k) = d(i,k),$$

a clear contradiction. This implies that the concatenated walk is a path from i to k. Because

$$|\mathcal{P}_1^*| + |\mathcal{P}_2^*| = d(i,j) + d(j,k) = d(i,k),$$

it is also shortest path. Therefore j is on a shortest path by construction. □

Def. 1. (FURTHEST VERTICES) The set of all *furthest vertices* of a vertex $i \in V$ is $\mathcal{F}(i)$. For any $j \in \mathcal{F}(i)$, $d(i,j) = \epsilon(i)$.

From Equation (2), we see that there are two possibilities for the best lower bound from a given source. This yields two possible situations for the associated lower bound to be correct.

Lem. 2. (LOWER BOUND IS CORRECT) Given a source s and another vertex i such that $\epsilon(i) = L_i^{(s)}$, we have either

(a) $s \in \mathcal{F}(i)$, or
(b) There exists $t \in \mathcal{F}(s) \cap \mathcal{F}(i)$ such that i is on a shortest path between s and t.

Proof. If $\epsilon(i) = d(i,s)$, then clearly $s \in \mathcal{F}(i)$ and case (a) is realized. On the other hand, if $\epsilon(i) = L_i^{(s)} = \epsilon(s) - d(i,s)$, then let $t \in \mathcal{F}(s)$. Rearranging the equality, we have $\epsilon(i) + d(i,s) = d(s,t) = \epsilon(s)$. Using triangle inequality, we see

$$d(s,i) + d(i,t) \geq d(s,t) = \epsilon(s) = \epsilon(i) + d(i,s),$$

or $\epsilon(i) \leq d(t,i)$. If this inequality is strict, then $\epsilon(i) < d(t,i)$, which is a contradiction, so $\epsilon(i) = d(t,i)$. This means that $t \in \mathcal{F}(i)$ as well. Lastly, we apply Lem. 1 to

$$\begin{aligned} d(i,s) + \epsilon(i) &= \epsilon(s) \\ d(i,s) + d(i,t) &= d(s,t) \end{aligned}$$

to see that i must be on a shortest path between s and t. □

Similarly, from Equation (3) we get the conditions on the upper bound being correct.

Lem. 3. (UPPER BOUND IS CORRECT) Given a source s and another vertex i such that $\epsilon(i) = U_i^{(s)}$, we have a vertex $t \in \mathcal{F}(s) \cap \mathcal{F}(i)$, exists such that s is on a shortest path between i and t.

Proof. Pick any $t \in \mathcal{F}(i)$. Using triangle inequality,

$$d(t,s) + d(s,i) \geq d(i,t) = \epsilon(i) = U_i^{(s)} = \epsilon(s) + d(i,s),$$

or $\epsilon(s) \leq d(t,s)$. If this inequality is strict, then $\epsilon(s) < d(t,s)$, which is a contradiction, so $\epsilon(s) = d(t,s)$. This means that $t \in \mathcal{F}(s)$ as well. Lastly, we apply Lem. 1 to

$$\begin{aligned} d(i,s) + E_s &= E_i \\ d(i,s) + d(s,t) &= d(i,t) \end{aligned}$$

to see that s must be on a shortest path between i and t. □

For the final result we observe that two sources s_1, s_2 are required for a non-source vertex to have $\epsilon(i)$ solved (and no information is gained for i from a third source). We combine this and the previous results to determine the necessary relationships between a solved vertex and any solving pair of sources.

Thm. 4. (SOLVED VERTEX) If $i \in V$ is solved but was never a source, then there exist two sources s_1, s_2, such that either

(a) $s_1 \in \mathcal{F}(i) \cap \mathcal{F}(s_2)$ and s_2 is on a shortest path between i and s_1, or
(b) there exists $t_1 \in \mathcal{F}(i) \cap \mathcal{F}(s_1) \cap \mathcal{F}(s_2)$ and a shortest path between s_1 and t_1 containing both i and s_2.

Proof. First assume $\epsilon(i) = d(s_1,i) = \epsilon(s_2) + d(s_2,i)$. In this case, $\epsilon(i) = d(s_1,i)$, so $s_1 \in \mathcal{F}(i)$. We have,

$$d(s_1,s_2) + d(s_2,i) \geq d(s_1,i) = \epsilon(s_2) + d(s_2,i),$$

implying $d(s_1,s_2) \geq \epsilon(s_2)$, for which equality must hold. Thus, $s_1 \in \mathcal{F}(s_2)$ as well. Then we apply Lem. 1 to

$$\begin{aligned} d(i,s_2) + \epsilon(s_2) &= d(i,s_1) \\ d(i,s_2) + d(s_2,s_1) &= d(i,s_1) \end{aligned}$$

to see that s_2 must be on a shortest path between i and s_1.

In the case $\epsilon(i) = \epsilon(s_1) - d(s_1,i) = \epsilon(s_2) + d(s_2,i)$, we apply Lem. 2(b) to see that there exists a $t_1 \in \mathcal{F}(s_1) \cap \mathcal{F}(i)$. Then,

$$\begin{aligned} d(t_1,s_2) + d(s_2,i) + d(i,s_1) &\geq \\ d(t_1,s_1) = \epsilon(s_1) &= \\ \epsilon(s_2) + d(s_2,i) + d(i,s_1), \end{aligned}$$

implying $d(t_1,s_2) \geq \epsilon(s_2)$, where equality must hold. Thus, $t_1 \in \mathcal{F}(s_2)$ as well. Then we apply Lem. 1 to

$$\begin{aligned} d(i,s_2) + \epsilon(s_2) &= \epsilon(i) \\ d(i,s_2) + d(s_2,t_1) &= d(i,t_1) \end{aligned}$$

to see that s_2 must be on a shortest path between i and t_1, and to

$$\begin{aligned} d(i,s_1) + \epsilon(i) &= \epsilon(s_1) \\ d(i,s_1) + d(i,t_1) &= d(s_1,t_1) \end{aligned}$$

to see that i must be on a shortest path between s_1 and t_1. Combining these two facts we get the result. □

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-751522). Experiments were performed at the Livermore Computing facility.

References

[1] F. W. Takes and W. A. Kosters, “Computing the eccentricity distribution of large graphs,” *Algorithms*, vol. 6, no. 1, pp. 100–118, 2013.

[2] R. Pearce, M. Gokhale, and N. M. Amato, “Scaling techniques for massive scale-free graphs in distributed (external) memory,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 825–836.

[3] ——, “Faster parallel traversal of scale free graphs at extreme scale with vertex delegates,” *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.50

[4] J. Shun, “An evaluation of parallel eccentricity estimation algorithms on undirected real-world graphs,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’15. New York, NY, USA: ACM, 2015, pp. 1095–1104. [Online]. Available: http://doi.acm.org/10.1145/2783258.2783333

[5] P. Hage and F. Harary, “Eccentricity and centrality in networks,” *Social networks*, vol. 17, no. 1, pp. 57–63, 1995.

[6] C. Bilgin, C. Demir, C. Nagi, and B. Yener, “Cell-graph mining for breast tissue modeling and classification,” in *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, Aug 2007, pp. 5311–5314.

[7] D. Bonchev, “The concept for the centre of a chemical structure and its applications,” *Journal of Molecular Structure: THEOCHEM*, vol. 185, pp. 155 – 168, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0166128089850110

[8] S. Gupta, M. Singh, and A. K. Madan, “Connective eccentricity index: a novel topological descriptor for predicting biological activity,” *Journal of Molecular Graphics and Modelling*, vol. 18, no. 1, pp. 18 – 25, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1093326300000279

[9] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, “The more the merrier: Efficient multi-source graph traversal,” *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 449–460, Dec. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735496.2735507

[10] M. Kaufmann, M. Then, A. Kemper, and T. Neumann, “Parallel array-based single-and multi-source breadth first searches on large dense graphs.” in *EDBT*, 2017, pp. 1–12.

[11] A. E. Saryüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Regularizing graph centrality computations,” *Journal of Parallel and Distributed Computing*, vol. 76, pp. 106 – 119, 2015, special Issue on Architecture and Algorithms for Irregular Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001282

[12] H. Liu, H. H. Huang, and Y. Hu, “ibfs: Concurrent breadth-first search on gpus,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16. New York, NY, USA: ACM, 2016, pp. 403–416. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882959

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[14] C. Avery, “Giraph: Large-scale graph processing infrastructure on hadoop,” in *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, no. 3, 2011, pp. 5–9.

[15] R. R. McCune, T. Weninger, and G. Madey, “Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing,” *ACM Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2818185

[16] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, “A scalable distributed parallel breadth-first search algorithm on bluegene/l,” in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov 2005, pp. 25–25.

[17] A. Buluç and K. Madduri, “Parallel breadth-first search on distributed memory systems,” *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11*, 2011. [Online]. Available: http://dx.doi.org/10.1145/2063384.2063471

[18] S. Beamer, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389013

[19] S. Beamer, A. Buluç, K. Asanovic, and D. Patterson, “Distributed memory breadth-first search revisited: Enabling bottom-up search,” in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, May 2013, pp. 1618–1627.

[20] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka, “Extreme scale breadth-first search on supercomputers,” in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 1040–1047.

[21] “The graph 500 list.” [Online]. Available: https://graph500.org/

[22] F. W. Takes and W. A. Kosters, “Determining the diameter of small world networks,” in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM ’11. New York, NY, USA: ACM, 2011, pp. 1191–1196. [Online]. Available: http://doi.acm.org/10.1145/2063576.2063748

[23] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, “Graph structure in the web,” *Comput. Netw.*, vol. 33, no. 1-6, pp. 309–320, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(00)00083-9

[24] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, Jun. 2014.

[25] J. Yang and J. Leskovec, “Defining and evaluating network communities based on ground-truth,” in *2012 IEEE 12th International Conference on Data Mining*, Dec 2012, pp. 745–754.

[26] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, “Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters,” *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009. [Online]. Available: https://doi.org/10.1080/15427951.2009.10129177

[27] J. Leskovec, D. Huttenlocher, and J. Kleinberg, “Signed networks in social media,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’10. New York, NY, USA: ACM, 2010, pp. 1361–1370. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753532

[28] ——, “Predicting positive and negative links in online social networks,” in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW ’10. New York, NY, USA: ACM, 2010, pp. 641–650. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772756

[29] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: Densification laws, shrinking diameters and possible explanations,” in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD ’05. New York, NY, USA: ACM, 2005, pp. 177–187. [Online]. Available: http://doi.acm.org/10.1145/1081870.1081893

[30] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?” in *WWW ’10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[31] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, “The graph structure in the web: Analyzed on different aggregation levels,” *The Journal of Web Science*, vol. 1, no. 1, pp. 33–47, 2015.

[32] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, “Graph manipulations for fast centrality computation,” *ACM Trans. Knowl. Discov. Data*, vol. 11, no. 3, pp. 26:1–26:25, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3022668

[33] P. Yuechao, R. Pearce, and J. Owens, “Scalable breadth-first search on a gpu cluster,” in *Proceedings of the 2018 32nd IEEE International Parallel and Distributed Processing Symposium*, May 2018, pp. 1090–1001.