CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3354

Estimating graph distance and centrality on shared nothing architectures

Atilla Soner Balkir[1,*,†], Huseyin Oktay[2] and Ian Foster[1,3]

[1]*The University of Chicago, 1100 East 58th Street, Chicago, IL 60637, USA*
[2]*University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA 01003, USA*
[3]*Argonne National Laboratory, 9700 S. Cass Avenue, Lemont, IL 60439, USA*

SUMMARY

We present a parallel toolkit for pairwise distance computation in massive networks. Computing the exact shortest paths between a large number of vertices is a costly operation, and serial algorithms are not practical for billion-scale graphs. We first describe an efficient parallel method to solve the single source shortest path problem on commodity hardware with no shared memory. Using it as a building block, we introduce a new parallel algorithm to estimate the shortest paths between arbitrary pairs of vertices. Our method exploits data locality, produces highly accurate results, and allows batch computation of shortest paths with 7% average error in graphs that contain billions of edges. The proposed algorithm is up to two orders of magnitude faster than previously suggested algorithms and does not require large amounts of memory or expensive high-end servers. We further leverage this method to estimate the closeness and betweenness centrality metrics, which involve systems challenges dealing with indexing, joining, and comparing large datasets efficiently. In one experiment, we mined a real-world Web graph with 700 million nodes and 12 billion edges to identify the most central vertices and calculated more than 63 billion shortest paths in 6 h on a 20-node commodity cluster. Copyright © 2014 John Wiley & Sons, Ltd.

Received 2 July 2014; Revised 3 August 2014; Accepted 2 July 2014

KEY WORDS: graph mining; shortest paths; graph centrality; MapReduce

1. INTRODUCTION

Computing the shortest path between arbitrary pairs of nodes is a prominent problem, which has been studied extensively. Typical solutions to the single source shortest path problem (SSSP) are breadth-first search (BFS) for unweighted graphs and Dijkstra's algorithm [1] for weighted graphs with non-negative edge weights. Given a graph G with n vertices and m edges, the BFS traversal runs in $O(n + m)$ time, whereas efficient implementations of Dijkstra's algorithm have $O(m + n \log(n))$ time complexity. In large graphs, computing exact answers becomes prohibitively expensive because of computational complexity and storage space requirements. For example, the all pairs shortest path (APSP) problem can be solved using Floyd–Warshall [2] algorithm in $O(n^3)$ time and $O(n^2)$ space complexity. However, in large networks with millions of nodes and billions of edges, computing the exact shortest paths between all pairs of nodes is not practical. To address this problem, several fast estimation algorithms have been developed.

The first step toward estimating the shortest paths in large graphs is to perform some pre-computation to index and summarize the link structure of the graph. In landmark-based methods [3–8], this involves selecting a set of nodes called `landmarks` and computing the exact shortest paths from the landmarks to the rest of the graph. Using the shortest path trees rooted at the

*Correspondence to: Atilla Soner Balkir, The University of Chicago, 1100 East 58th Street, Chicago, IL 60637, USA.
†E-mail: soner@cs.uchicago.edu

landmarks, the distance between an arbitrary pair of vertices can be computed in almost constant time. The main focus of landmark-based methods has been on providing fast estimations to shortest path queries. Performance gain is achieved by the assumption that the pre-computed trees from the landmarks are small enough to fit in memory. This information is typically stored in a high-end multi-core server for random memory access. The shared memory central server approach has inherent scalability limitations. The computation time increases as the graph gets denser. Also, the performance of these methods under heavy load with many distance estimation requests arriving in parallel has not been evaluated. Recent studies show that applications running on multi-core architectures tend to be limited by off-chip bandwidth because of shared resources in the memory hierarchy [9, 10].

Centrality is an area of graph mining where computing the shortest paths is frequently encountered. In this study, we concentrate on two common definitions of centrality: `closeness` and `betweenness`. Exact computation of these measures involves solving the APSP problem. Existing parallel algorithms for estimating closeness and betweenness centrality are designed for high-end shared memory symmetric multiprocessor architectures [11, 12]. Graph size is still a limiting factor as these algorithms are memory intensive, and they do not exhibit high degrees of cache and memory locality [13]. There is also the hardware cost and availability issue. Most network scientists do not have direct access to super computers or high-end shared memory multiprocessor architectures. On the other hand, commodity clusters and shared nothing architectures with independent compute nodes are becoming widely available. They are also offered through cloud computing vendors in the form of infrastructure as a service.

MapReduce is a parallel programming abstraction widely used in processing large datasets using commodity clusters [14]. MapReduce applications are not limited by the total cluster memory. Local disk is actively used for storing intermediate data during the computation. High throughput is achieved by carrying out the same computation on different parts of the data in large batches. These features of MapReduce allow scaling to arbitrarily large datasets at the cost of local disk access and extra computation time for grouping and managing the data in the cluster.

In this paper, we present Hoba, an open source library for large scale distance estimation that uses MapReduce as the underlying platform. Hoba is built on top of Hadoop, a widely used MapReduce framework. It is compatible with the Amazon Elastic MapReduce cluster, so researchers without immediate access to compute clusters can benefit from it. Our main contributions are summarized as follows:

- We present a parallel algorithm to solve the SSSP in MapReduce. It produces exact answers, runs nearly in linear time for unweighted graphs, and involves several optimization techniques to minimize the amount of data sent over the network. Experimental evaluation shows that it is up to seven times faster than naïve implementations.
- We introduce a novel parallel algorithm to estimate the pairwise shortest paths in large batches. In particular, we eliminate the random access to shared memory requirement in the previously suggested landmark-based methods and leverage more pre-computation data to increase the accuracy of the results. We achieve average error rates between 0.02% and 7% in several real-world graphs ranging from a few thousand nodes to more than 700 million nodes in size. In a single compute node, the proposed method can return multiple estimated shortest paths between a pair of vertices under 8 ms on average, 100 times faster than previously reported results on comparable graphs [6]. In addition, our method can run in parallel in a cluster of nodes where each compute node can process a different subset of pairs independently.
- We use the parallel distance estimation algorithm to approximate the closeness and betweenness centrality metrics in large graphs. We present alternative implementations of these methods in distributed environments and discuss their efficiency using a communication cost model. Experimental evaluation on medium-sized networks reveals that we can identify vertices with top centrality scores within [75–96%] accuracy range.

2. BACKGROUND

For the rest of this paper, we assume that the graphs are unweighted for simplicity, although all algorithms described in this paper are applicable to weighted graphs as well. We explain how to cover the weighted setting when we describe each algorithm in the corresponding subsection.

2.1. *Definitions*

Let G be a graph with $|V| = n$ vertices, and $|E| = m$ edges. Given two arbitrary vertices v_0 and v_k, a path $\pi(v_0, v_k)$ of length $|\pi(v_0, v_k)| = k$ is an ordered sequence of vertices $\{v_0, v_1, \ldots, v_k\}$ such that

$$\begin{aligned} v_i \in V, &\quad 0 \leqslant i \leqslant k \quad \text{and} \\ (v_i, v_{i+1}) \in E, &\quad 0 \leqslant i < i+1 \leqslant k, \quad i,k \in \mathbb{N} \end{aligned}$$

A graph is `connected` if and only if there exists a path between any pair of vertices. A shortest path between two vertices u and v is denoted by $\pi^*(u,v)$. Their distance $d(u,v)$ is the length of the shortest path, that is, $d(u,v) = |\pi^*(u,v)|$ or ∞ if v is not reachable from u. The term `geodesic path` is also used to represent the shortest paths. The `diameter` D of a graph is the longest shortest path between any two vertices. Note that $D < \infty$ for connected graphs. The effective diameter D' is the minimum number of hops in which 90% of all vertex pairs can be reached [15].

Given two paths $\pi(u,v)$ and $\pi(v,s)$, there exists another path $\pi(u,s)$ constructed by concatenating the two as

$$\pi(u,s) = \pi(u,v) \,.\, \pi(v,s) \tag{1}$$

where . is the path concatenation operator. Observe that $\pi(u,s)$ is not necessarily the shortest path between u and s, and it may contain cycles. In general, the following triangle inequality

$$d(u,s) \leqslant d(u,v) + d(v,s) \tag{2}$$

holds for any three vertices $\{u,v,s\} \in V$.

2.2. *Distance estimation in large networks*

The main focus of previous work on distance estimation in large networks has been on providing fast distance estimations to the shortest path queries.

2.2.1. Simple scalar methods. Potamias *et al.* [3] described an algorithm to estimate the distance between a pair of vertices using a landmark (sketch) based method. They sample a set of landmarks $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_k\}$, where $\mathcal{L} \subset V$, and solve the SSSP for each $\ell_i \in \mathcal{L}$. This is achieved by doing a BFS traversal of the graph for unweighted graphs or running Dijkstra's algorithm for the weighted case. The actual paths from the landmarks are ignored, and only pairwise distances are stored in memory. Using the triangle inequality in Equation (2) as an upper bound, the distance between a pair of vertices s and t is estimated by

$$d'(s,t) = \min_{\ell_i \in \mathcal{L}} \{d(s,\ell_i) + d(\ell_i,t)\} \tag{3}$$

This method requires $O(k)$ time to estimate the distance between a pair of vertices and $O(nk)$ space for the pre-computation data.

2.2.2. Path concatenation. The `Sketch` algorithm [7] extends the scalar landmark-based methods via path concatenation. In addition to distance, the actual shortest paths $\pi^*(\ell, v)\ \forall \ell \in \mathcal{L}, v \in V$

are stored as part of the pre-computation data. Consider the simplest case with a single landmark ℓ in a connected graph G. Given two vertices s and t, there exists at least two shortest paths from ℓ, namely $\pi^*(\ell, s)$ and $\pi^*(\ell, t)$. Using these, a path from s to t can be constructed as follows:

$$\pi(s,t) = \pi^*(s,\ell) \,.\, \pi^*(\ell,t) \tag{4}$$

Because there are multiple landmarks in practice, the shortest path obtained by using any of the landmarks is returned as the final estimation result.

$$\pi'(s,t) = \min_{\ell_i \in \mathcal{L}} \{\pi^*(s,\ell_i) \,.\, \pi^*(\ell_i,t)\} \tag{5}$$

Note that $\pi^*(s,\ell)$ is obtained by reversing $\pi^*(\ell,s)$ if G is undirected. Otherwise, the pre-computation step is also run on G', the graph built by reversing all edges of G. Further optimization techniques are applied to improve the length of the initially estimated path. These include eliminating cycles and observing the neighbors of vertices to discover new edges that form a shortcut. These operations are briefly summarized in Figure 1 on a sample graph. The worst-case space complexity of the `Sketch` algorithm is $O(nkD)$ because the shortest path between a landmark and arbitrary vertex cannot be longer than D.

The `Landmark-BFS` [6] algorithm improves the accuracy of the results by combining the shortest paths together to create an induced sub-graph rather than using each landmark independently. Given all the shortest paths between $s, t \in V$ and $\forall \ell_i \in \mathcal{L}$, an induced sub-graph $G_{\mathcal{L}}(s,t)$ is constructed by taking a union of all vertices and edges. That is,

$$G_{\mathcal{L}}(s,t) = \bigcup_{\ell_i \in \mathcal{L}} \pi^*(s,\ell_i) \,.\, \pi^*(\ell_i,t) \tag{6}$$

An estimated shortest path $\pi'(s,t)$ is obtained by running BFS (or Dijkstra's Algorithm if G is weighted) on $G_{\mathcal{L}}(s,t)$. The size of the induced graph is $O(kD)$, and the estimation algorithm runs in at most $O(k^2D^2)$ time for a pair of vertices. Observe that the main advantage of `Landmark-BFS` over the `Sketch` algorithm is that it leverages more data from multiple landmarks at the same time, and this yields results with higher accuracy.

The landmark-based algorithms summarized here assume the size of the pre-computation data, and the graph structure is small enough to fit in memory for random access. In Section 3.5, we present an extension to the `Landmark-BFS` algorithm that leverages more pre-computation data while eliminating the need to store everything in main memory. This allows massive parallelism as multiple compute nodes can process different subset of vertices independently without relying on a central server.

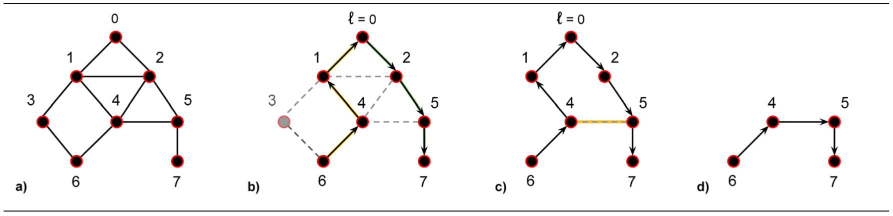


Figure 1. (a) Sample undirected graph with no edge weights. Zero is chosen as the landmark. (b) A shortest path tree rooted at 0 is computed using breadth-first search. $\pi^*(6,0)$ and $\pi^*(0,7)$ are concatenated to construct an estimated shortest path from 6 to 7. The initially estimated path length is six. (c) Short-cutting optimization is applied to the initial path. This is done by expanding the neighbors of each vertex on the estimated path to find a short-cutting edge. When neighbors of four are expanded, the edge (4,5) is observed to improve the estimation quality. (d) Final result after removing the extra edges from the initial path. Note that the short-cutting method requires random access to the entire graph structure and is not feasible for arbitrarily large graphs.

2.3. Graph centrality

The goal of graph centrality is to measure the structural prominence of a vertex. Freeman [16] describes closeness centrality of a vertex as the inverse of its average distance to the rest of the graph. More formally,

$$c(v) = \frac{|V| - 1}{\sum_{u \in V} d(u, v)} \tag{7}$$

Betweenness centrality is the number of times a vertex occurs in any shortest path between any pair of vertices in the graph [17]. A more formal expression can be given by defining σ_{st} as the number of geodesic paths from s to t and σ_{svt} as the number of such paths that v lies on. The betweenness centrality of v is then defined as

$$b(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{svt}}{\sigma_{st}} \tag{8}$$

In general, calculating the exact closeness and betweenness centrality metrics involves solving the APSP problem. For unweighted graphs, Brandes' algorithm [18] can compute betweenness centrality in $O(nm)$ time. Parallel estimation algorithms for closeness and betweenness centrality leverage high-end shared memory symmetric multiprocessor and multi-threaded architectures [11, 12]. Kang describes alternative centrality metrics that can be computed efficiently using MapReduce [19]. However, the proposed metrics are not direct replacements to closeness and betweenness centrality, and their scalability is only studied for artificial graphs. With Hoba, we were able to handle real-world graphs that are 10 times bigger using only 20 compute nodes.

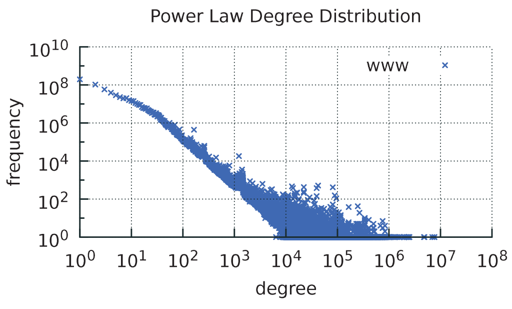
2.4. Statistical patterns in large networks

Real-world graphs are called `scale-free` when they exhibit skewed degree distributions where the fraction $p(k)$ of vertices with degree k asymptotically follows a power law given by

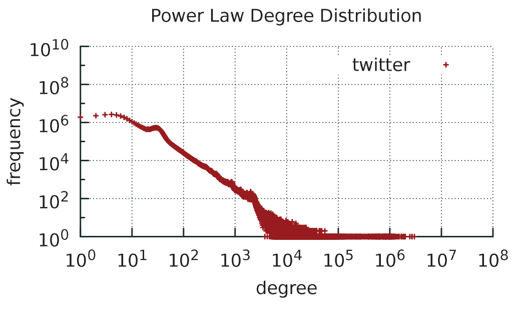
$$p(k) = Ak^{-\gamma}, \quad \gamma > 1 \quad \text{and} \quad k \geqslant k_{min} \tag{9}$$

Table I. Summary of datasets.

Name	Nodes	Edges
Facebook	4K	176K
Wikipedia	7K	214K
Enron	36K	423K
Twitter	41M	2.4B
WWW-2002	720M	12.8B



(a) WWW Degree Distribution



(b) Twitter Degree Distribution

Figure 2. Power law degree distribution of the large graphs used in the experiments.

Examples of scale-free networks include the World Wide Web graph [20] and the citation network between scientific papers [21]. A common characteristic often observed in scale-free networks is the `small-world` phenomena (also known as six degrees of separation), which states that the diameter of large graphs is usually not too long. In addition, previous studies show that the graph diameter tends to get smaller as the graph grows over time [22, 23]. This is known as the `shrinking diameter` phenomenon in scale-free dynamic networks.

2.4.1. Datasets. The characteristics of the datasets used in our experiments are summarized in Table I. The first three graphs are from the Stanford Large Network Dataset Collection and used to compare the true and estimated values for distance and centrality. The Facebook dataset contains friendship relations between network members. The Wikipedia graph is a voting network between contributors. There is an edge from user *i* to *j* if one of them voted on the other. Enron email communication network is composed of individuals who exchanged at least one email in a company network. We intentionally chose medium-sized networks to be able to compute the true distance and centrality values by running the NetworkX Python library on a single desktop computer.

The last two graphs are used for large scale experiments. The Twitter dataset is a subset of the network from a snapshot in 2008 with over 41 million nodes. The WWW graph contains billions of hyperlinks from a Web crawl in 2002 by the Yahoo! Altavista search engine. Figure 2 shows the long-tail degree distribution of the WWW and Twitter datasets plotted on logarithmic scale. Notice that as the degree becomes larger, the frequency gets smaller following a power law distribution. In Section 3, we present novel optimization techniques that exploit these characteristics to speed-up the computations. We considered all graphs as undirected to increase the data size and work on one strongly connected component, which otherwise would not be possible.

3. LARGE SCALE ALGORITHMS

This section contains detailed descriptions of the graph algorithms and their distributed communication cost analysis. Figure 3 shows a roadmap of the methods we present and the relationships between them. We start with an exact algorithm to compute the single source shortest paths in large graphs. Using this as the basis, we continue with a parallel algorithm to estimate the distance between arbitrary pairs of vertices in parallel. Finally, we present efficient techniques to estimate closeness and betweenness centrality in large graphs, which use the parallel distance estimation algorithm as the main building block.

3.1. Parallel-SSSP

We begin with an efficient MapReduce algorithm for solving the SSSP, which provides the basis for the landmark-based distance estimation algorithms. Given ℓ as the source (landmark), the traditional definition of this problem asks for finding the shortest path $\pi^*(\ell, v)$ for all $v \in V$. Although there are often multiple shortest paths between a pair of vertices, practical applications typically return the first one as the answer and neglect the rest. We slightly modify the problem definition to discover $\Pi^*(\ell, v)$, a set of shortest paths for each (ℓ, v) pair. More formally,

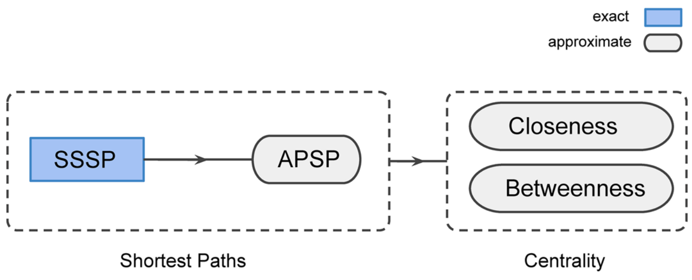


Figure 3. Roadmap of the large scale graph algorithms.

$$\Pi^*(\ell, v) = \{\pi(\ell, v) : |\pi(\ell, v)| = d(\ell, v)\} \tag{10}$$

Instead of a shortest path tree rooted at the landmark, we are interested in generating a shortest path sub-graph where there are many ways to reach a vertex. Note that the output of this computation is larger, but it is stored on the distributed file system (DFS) rather than memory. A sample graph is given in Figure 4.

For consistency, we use the same binary graph file format for all algorithms described in this paper. Figure 5 shows the logical representation of a vertex record. In the beginning, the `distance` and `active` fields of ℓ are set to `0` and `true`, respectively. All other vertices have their `distance` fields equal to ∞ and `active` fields set to `false`. When the algorithm terminates, the `sub-graph` field of a vertex v contains one or more shortest paths from ℓ to v combined in a sub-graph as shown in Figure 4, and the `distance` field is set to $d(\ell, v)$.

All vertex ID's are represented by variable length long integers to reduce data size. We also use delta encoding [24] to compress neighbor lists. For each vertex, the neighbors are sorted in ascending order, and only the difference between each consecutive pair is stored. Sample encoded and decoded representations of a neighbor list are given in Figure 6. The same compression scheme is also applied to the `sub-graph` field. Each `sub-graph` is converted to the adjacency list representation, and for each vertex, the neighbors attached to it are compressed separately using delta encoding.

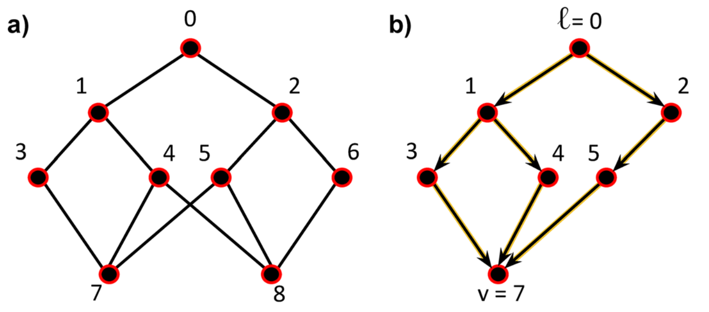


Figure 4. (a) Sample undirected graph with no edge weights and nine vertices. (b) The sub-graph for vertex 7, which contains the output of single source shortest path problem for the pair (0,7). Three shortest paths are highlighted with arrows: {0,1,3,7}, {0,1,4,7}, and {0,2,5,7}.

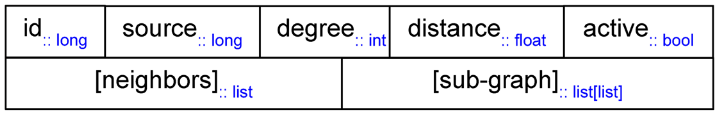
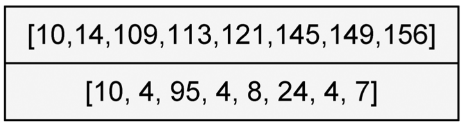


Figure 5. Logical representation of a vertex record. Each field is associated with a name and type. The `neighbors` and `sub-graph` fields can be arbitrarily long, and they are compressed using delta encoding when serialized on disk.



[10,14,109,113,121,145,149,156]
[10, 4, 95, 4, 8, 24, 4, 7]

Figure 6. Delta encoding of a neighbor list. Neighbors are sorted in ascending order, and only the difference between two consecutive vertices is stored on disk except for the first one. To re-construct, a sequential addition operation is applied from the first to the last element of the compressed list. Note that the differences are stored using variable length integers so smaller values occupy less physical space.

Algorithm 1 Parallel-SSSP

```
 1: Input: Graph G=(V,E) with $\ell \in \mathcal{L}$ specified as the source.
 2: function MAP(v.id, v)
 3:     if v.active then
 4:         d ← v.distance+1
 5:         msg ← (d, v.paths)
 6:         for all u ∈ v.out-neighbors do
 7:             if ¬ DISCOVERED(u.id) then
 8:                 EMIT(u.id, msg)                                   ▷ distance
 9:             end if
10:         end for
11:         v.active ← False
12:         DFS.WRITE(v)                                          ▷ final output
13:     else
14:         msg ← v
15:         EMIT(v.id, msg)                                      ▷ vertex record
16:     end if
17: end function
18: function REDUCE(v.id, [$msg_1$,...,$msg_k$])
19:     $d_{min} \leftarrow \infty$
20:     $\Pi_{min} \leftarrow \emptyset$                          ▷ set of min length paths
21:     for all m ∈ [$msg_1$,...,$msg_k$] do
22:         if VERTEXRECORD(m) then
23:             v ← m
24:         else
25:             if m.distance < $d_{min}$ then
26:                 $d_{min}$ ← m.distance
27:                 $\Pi_{min}$ ← m.paths
28:             else if m.distance = $d_{min}$ then
29:                 $\Pi_{min} \leftarrow \Pi_{min} \cup$ m.paths
30:             end if
31:         end if
32:     end for
33:     if $d_{min}$ < v.distance then                            ▷ update record
34:         v.distance ← $d_{min}$
35:         APPEND($\Pi_{min}$, v)
36:         v.paths ← $\Pi_{min}$
37:         SAMPLE(v.paths, MAX_PATHS)
38:         v.active ← True
39:         DISCOVERED(v.id) ← True
40:     end if
41:     if v.degree < $\delta$ then
42:         EMIT(v.id, v)                                     ▷ pass to next MAP
43:     else
44:         DFS.WRITE(v)                                ▷ send to fat vertex pool
45:     end if
46: end function
```

The parallel-SSSP algorithm is a BFS. The idea is to propagate the minimum distance from the landmark to the rest of the graph. Let $d(v, \ell)^{(k)}$ be the distance between v and ℓ at iteration

iteration	input graph	final output on DFS
0	**0**,1,2,3,4,5,6,7,8	**0**
1	**1**,**2**,3,4,5,6,7,8	0,**1**,**2**
2	**3**,**4**,**5**,**6**,7,8	0,1,2,**3**,**4**,**5**,**6**
3	**7**,**8**	0,1,2,3,4,5,6,**7**,**8**

Figure 7. IGS: Discovered vertices at a given iteration are highlighted in bold. They are serialized on distributed file system to construct the final output incrementally. The input graph for each iteration is smaller than the preceding one.

k. Initially, $d(\ell,\ell)^{(0)} = 0$ and all other distances are set to ∞. In an unweighted graph, we can compute $d(v,\ell)^{(k)}$ iteratively as follows:

$$d(v,\ell)^{(k)} = \min\left\{d(v,\ell)^{(k-1)}, \min_{u\,\in\,\mathcal{N}(v)}\left\{d(u,\ell)^{(k-1)}\right\} + 1\right\} \tag{11}$$

where $\mathcal{N}(v)$ contains the incoming neighbors of v. This computation can be expressed as a sequence of MapReduce jobs. At the map step, an active vertex adds 1 to its distance, sends it to its out-neighbors, and becomes inactive. At the reduce step, a vertex iterates over the distance messages received from its in-neighbors and finds the minimum. If the minimum is smaller than the current distance, the current distance is updated, and the vertex is marked as active so that it can propagate the new minimum to its out-neighbors when the next iteration starts. This cycle continues until all vertices become inactive, which indicates that the algorithm converged. In a connected graph, all vertices contain a distance value that is less than ∞ after convergence.

Algorithm 1 shows the pseudo-code for the MapReduce implementation. The `MAP` function sends two different types of output to the reducers. Inactive vertices only send the vertex record `(lines 14-15)`. Messages sent from an active vertex v to its out-neighbors contain the shortest paths along with the distance `(lines 4-5)`. After an active vertex sends its distance and paths, it is marked as `inactive` and written to the final output location on DFS, rather than being sent to reducers `(line 12)`. This is a special IO optimization for unweighted graphs. In an unweighted setting, all shortest paths from ℓ to v are discovered during the i'th iteration where $d(\ell,v) = i$. There is no need to process this vertex again in the subsequent iterations because any additional path found from ℓ to v will be sub-optimal. Thus, v can be removed from the input for the rest of the computation, which is achieved by writing it to the final output location on DFS. We call this scheme `Incremental Graph Serialization (IGS)`. Figure 7 shows the operation of `IGS` for the sample graph when $\ell = 0$.

The `REDUCE` function saves the vertex record in `line 23`. It also maintains the minimum length paths and the minimum distance received so far `(lines 25-30)`. Between `lines 33` and `40`, the record is updated if a smaller distance is found. Note that for unweighted graphs, this happens at most once for each vertex. The first time a vertex v is discovered (i.e., $d(\ell,v)$ is set to i for some $i < \infty$), all shortest paths from ℓ to v are found.

In billion-scale graphs, there may be thousands to millions of shortest paths between a `(landmark,vertex)` pair. Storing and using millions of paths for estimating pairwise distances is impractical and should be avoided for performance reasons. In `line 37`, paths are sampled randomly, and at most, `MAX_PATHS` of them are stored as the pre-computation data. Hoba allows users to control `MAX_PATHS` according to the size of the graph. We set `MAX_PATHS` to 250 for all graphs in our experiments.

3.2. *Selective push using bit vectors*

Most MapReduce adaptation of existing graph algorithms exhibits sub-optimal performance because of redundant data being sent from mappers to reducers multiple times. A fundamental reason for this inefficiency is the `vertex-centric` approach that lacks a global application state. A message is `pushed` from a vertex to its neighbors regardless of whether the neighbors already have the same information or not.

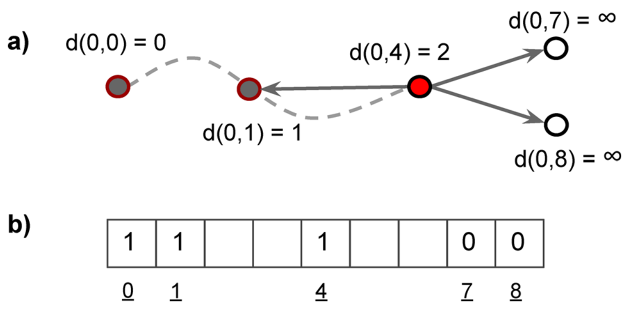


Figure 8. (a) Vertex 4 when it is active, along with its neighbors. Observe that 7 and 8 are not discovered yet, so a distance message should be sent to both. However, 1 was discovered previously because it has a direct link to 0. A distance message from 4 to 1 will have no contribution to the final results and should be avoided. (b) The corresponding bit vector for the sample graph. Only those fields that correspond to the vertices earlier are highlighted.

We partially address this problem by labeling vertices as `active` or `inactive`. A vertex only becomes `active` when it receives a smaller distance value from its in-neighbors. As soon as the new information is propagated to the out-neighbors, it switches back to `inactive` state. This optimization avoids sending the same distance redundantly at each iteration.

The second redundancy occurs when a vertex sends a message to one of its neighbors, which already has a smaller distance value. Consider vertex `4` from the sample graph in Figure 4. The distance values of its neighbors when vertex `4` is active are given in Figure 8. Notice that, because `d(0,1) < d(0,4)`, there is no need to send a distance message to vertex `1`.

Such redundant messages can be suppressed using a global bit vector that is accessible from all compute nodes. The bit vector technique is reminiscent of coloring vertices as white, gray, or black in serial implementations of BFS or Dijkstra's algorithm. Let δ be an n-bit vector where $n = |V|$. In unweighted graphs, `Parallel-SSSP` maintains the following condition for δ:

$$\delta(i) = \begin{cases} 1, \text{ if } d(\ell, i) < \infty, \\ 0, \text{ otherwise} \end{cases}$$

Initially, only $\delta(\ell) = 1$, and the rest of the bit vector is set to `0`. At the beginning of each iteration, compute the nodes that read the latest bit vector from DFS and store it in local memory. Before sending a message to an out-neighbor, an active vertex first checks whether it was previously discovered by querying its local copy of the bit vector (`line 7`). When a new vertex is discovered, the corresponding location of the local bit vector is set to `1` (`line 39`). At the end of each iteration, all bit vectors are merged into a single one by a bitwise-OR operation to maintain the global state for δ, which is then stored on DFS. It is worth noting that the bit vector optimization does not require any modifications to the MapReduce programming model, and it can be implemented in common MapReduce frameworks such as Hadoop. The coordination is achieved by merging the local copy of all bit vectors at the end of each iteration and re-distributing the newest copy to all compute nodes through the DFS.

An unweighted graph with 100 million vertices can be represented using a 12.5 MB bit vector, which easily fits in the memory of a single compute node. For weighted graphs, it is necessary to store the actual distance information inside the vector so a slight modification is required. Depending on the size of the weights, a few bits or multiple bytes may be reserved for each vertex.

3.3. Load balancing high-degree vertices

The work done inside a map task is dominated by the total number of distance messages generated (`lines 6-10`). The more edges processed inside a map task, the longer it takes to finish. The skewed degree distribution of large real-world graphs results in disproportionate map task run times. While most map tasks finish within minutes, a small percentage of those that process vertices with very high degrees can take over an hour to complete. This is an undesired behaviour in MapReduce because the `REDUCE` phase cannot start until all map tasks are finish. The cluster utilization drops significantly toward the end of the `MAP` step as the majority of the compute nodes stay idle, waiting for a small fraction of the map tasks to complete.

We address this problem by processing high-degree vertices simultaneously in all map tasks rather than assigning each vertex to a random map task as usual. A vertex is identified as a `fat vertex` if its degree is greater than a threshold value δ. For the large graph datasets considered in this paper, we set δ to 50,000. If a fat vertex is identified during the `REDUCE` step, it is sent to a special directory called the `fat vertex pool` on DFS `(lines 41-45)`. Each reduce task creates a separate file under the `fat vertex pool` for serializing high-degree vertices it identifies.

Before the next `MAP` step begins, all map tasks execute `PROCESS-FAT-VERTICES` during the task setup. Algorithm 2 shows the pseudo-code for this function. Input is the path to the `fat vertex pool` on DFS and the number of map tasks spawned for the job. Each task pulls all `fat vertices` discovered in the `REDUCE` step of the previous iteration from the DFS in randomized order `(lines 4-10)`. The randomization prevents file system swamping. If all tasks read the `fat vertices` in the same order, data nodes serving the initial DFS requests may not be able to keep up with the large number of queries, and the file system may become unstable. Randomizing the file list helps balancing the load on DFS and results in higher IO throughput.

The `PROCESS-ONE` function is structurally similar to the `MAP` operation described earlier. In addition, it involves a range partitioning mechanism for load balancing. When a `fat vertex` is processed, its out-neighbors are divided into disjoint subsets of almost equal size using a partitioning function. Assume there are m map tasks for the job and let $\mathcal{N}(v)$ be the out-neighbor set of v.

Algorithm 2 PROCESS-FAT-VERTICES

```
1:  Input: Pool: DFS directory that contains fat vertex files
2:  M: Number of map tasks
3:  function PROCESS-ALL(Pool, M)
4:      file_list ← DFS.READ(Pool)
5:      RANDOMIZE(file_list)
6:      for all file ∈ file_list do
7:          for all v in file do
8:              PROCESS-ONE(v)
9:          end for
10:     end for
11: end function
12: function PROCESS-ONE(v, M)
13:     task_id ← RUNTIME.TASKID                      ▷ get current task id
14:     d ← v.distance+1
15:     msg ← (d, v.paths)
16:     for all u ∈ v.out-neighbors do
17:         if u.id mod M = task_id then              ▷ range partitioning
18:             if ¬ DISCOVERED(u.id) then
19:                 EMIT(u.id, msg)                   ▷ distance
20:             end if
21:         end if
22:     end for
23:     if v.id mod M = task_id then
24:         msg ← v
25:         EMIT(v.id, msg)                           ▷ vertex record
26:     end if
27: end function
```

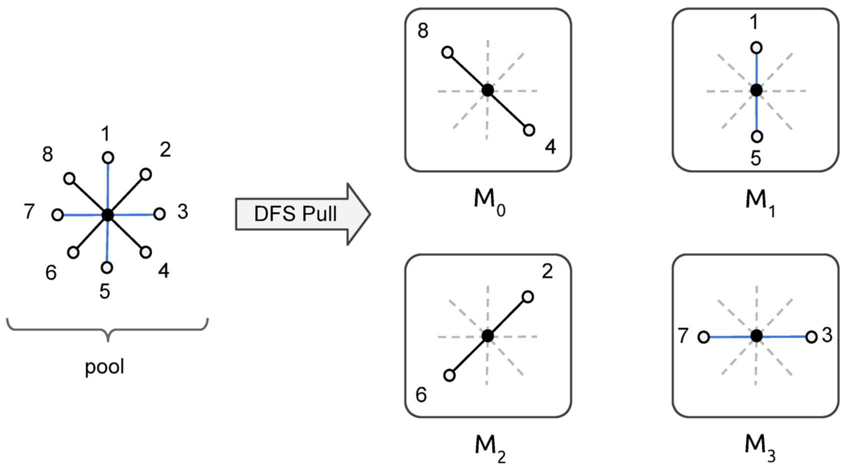


Figure 9. Sample range partitioning on a vertex with degree eight. There are four map tasks and all of them pull the same vertex record from the pool on distributed file system. Each map task processes a disjoint subset of neighbors where subsets are determined by a partitioning function.

The subset of neighbors that fall in the scope of map task M_i is determined by the partitioning function ϕ such that

$$\phi(M_i, v) = \{u : u \in \mathcal{N}(v),\ u \ \mathbf{mod} \quad m = i\} \tag{12}$$

The range partitioning mechanism is demonstrated on a sample vertex in Figure 9. Each task is responsible from sending a distance message to its own range of out-neighbors for a given `fat vertex`. This ensures that the computational load introduced by the high degree is distributed evenly among all tasks and solves the straggling process problem. Finally, the vertex record is also sent only once, based on a similar modulo operation `(lines 23-26)`. This avoids generating multiple copies of the same vertex record in all map tasks. After the `MAP` phase is completed, the `fat vertex pool` is cleared for the next iteration.

3.4. Complexity in MapReduce

The complexity of a MapReduce algorithm has two main components. `Communication cost` is the amount of data transmitted from mappers to reducers over the network. `Computation cost` is the total work done inside the `MAP` and `REDUCE` functions [25, 26]. In `Parallel-SSSP`, there is a one-to-one correspondence between the communication and computation costs, so we only analyze the former.

In an unweighted connected graph, each vertex becomes active exactly once. For each active vertex, at most, one distance message is sent to all neighbors. In practice, the bit vector optimization suppresses most edges after the first few iterations. Thus, the total number of distance messages sent from mappers to reducers is $O(m)$.

For a given iteration, sending one record per vertex from mappers to reducers yields $O(n)$ messages. Although the algorithm requires at most D iterations to converge, the majority (90%) of the vertices are generally discovered and removed from the input in less than 10 iterations because of small-world and shrinking diameter phenomena in large real-world graphs [22, 27]. As a result, the majority of the communication takes place within the first D' iterations where D' is the effective diameter.

Therefore, the communication and computation cost of `Parallel-SSSP` for unweighted graphs is $O(Dn + m) \approx O(D'n + m)$. Note that in scale-free networks, D' is small. Empirical studies on many real-world graphs typically suggest values between 4 and 7 while D is observed to decrease further with increasing graph size [28]. For comparison, a previously described MapReduce algorithm [29] for solving `SSSP` runs in $\Theta(Dn + Dm)$ time. This algorithm sends messages from all nodes to their neighbors at each iteration until convergence without avoiding any redundancy. The novelty and main advantage of `Parallel-SSSP` comes from the `Selective Push` and `IGS` optimizations. In Section 4, we show that these techniques speed-up the computation five to seven times in massive graphs.

3.5. *Distance estimation with PathCrawler*

The shortest paths discovered using `Parallel-SSSP` form the basis of distance estimation in large networks. In this section, we describe the `PathCrawler` algorithm, which is an extension of the `Landmark-BFS` method. The main idea is to use the extra pre-computation data to increase the accuracy of the results. Consider vertices `7` and `8` from the sample graph in Figure 4 when $\ell = 0$. There are three shortest paths between each pair `(0,7)` and `(0,8)`. The true distance between `7` and `8` can be calculated by using either `[{0,1,4,7} - {0,1,4,8}]` or `[{0,2,5,7} - {0,2,5,8}]` at the same time. If only one shortest path for each `(landmark, vertex)` pair is stored, the probability of finding the exact answer is $\frac{2}{9}$.

With `PathCrawler`, we increase the odds of finding the true distance by leveraging all shortest paths available from the pre-computation step. The algorithm essentially crawls the entire set of paths through the landmarks and obtains the shortest possible path from source to destination. The implementation is given in Algorithm 3.

Algorithm 3 PathCrawler

```
1: Input: {s,t} ∈ V, $\mathcal{L}$: Set of landmarks
2: $\Pi^*(s,\ell)$ and $\Pi^*(\ell,t)$ $\forall \ell \in \mathcal{L}$
3: function PATHCRAWLER($s,t$)
4:     $P \leftarrow \emptyset$
5:     for all $\ell \in L$ do
6:         for all $\pi \in \Pi^*(s,\ell)$ do
7:             $P \leftarrow P \cup \pi$                          ▷ paths from s to $\ell$
8:         end for
9:         for all $\pi \in \Pi^*(\ell,t)$ do
10:            $P \leftarrow P \cup \pi$                          ▷ paths from $\ell$ to t
11:        end for
12:    end for
13:    for all u ∈ s.out-neighbors do                         ▷ optional
14:        $e \leftarrow (s,u)$
15:        $P \leftarrow P \cup e$
16:    end for
17:    for all u ∈ t.in-neighbors do                          ▷ optional
18:        $e \leftarrow (u,t)$
19:        $P \leftarrow P \cup e$
20:    end for
21:    Let $G_P$ be the sub-graph of G induced by P
22:    $\pi^* \leftarrow$ DIJKSTRA($G_P$, s, t)
23:    return $\pi^*$
24: end function
```

Input is a pair of vertices $\{s,t\} \in V$ and shortest paths from/to the set of landmarks $\mathcal{L}$. All shortest paths are merged together between `lines 4-12`. In addition, there is an optional `two-hop guarantee` optimization[†], which adds all neighbors of s and t to the induced sub-graph G_P (`lines 12-20`). Finally, Dijkstra's algorithm (or BFS for unweighted graphs) is called as a sub-routine to find an estimated shortest path.

[†]Recommended when $m = O(n)$.

The `PathCrawler` algorithm deals with the potentially large induced sub-graph size by limiting the number of shortest paths between a vertex and a landmark, as discussed previously. Let τ be the maximum number of shortest paths stored for each (landmark, vertex) pair, that is, `MAX_PATHS`. The length of a shortest path has an upper bound of D, the graph diameter. The size of G_P is bounded above by $O(kD\tau)$, and the worst-case run time complexity of the algorithm is $O(k^2D^2\tau^2)$. In practice, the size of the induced sub-graph is quite smaller than $O(kD\tau)$ because multiple shortest paths from several landmarks highly overlap.

3.6. *Parallel-APSP*

The `PathCrawler` algorithm can be run in parallel to find an approximate solution for APSP. We assume that the graph is connected and undirected so $d(i,j) = d(j,i)\ \forall (i,j) \in V$. Note that regardless of how efficient the parallel implementation is, the lower bound for processing $\binom{n}{2}$ pairs of vertices is $\Theta(n^2)$. In this section, we present parallelization techniques and explain how to minimize the communication cost between cluster nodes, which has a significant effect on performance.

The input is the result of running `Parallel-SSSP` for all elements of $\mathcal{L}$. Each vertex is coupled with a compact output record consisting of shortest paths from the landmarks. Let r_i be the size of the record associated with vertex `i`. The total input size is given by $T(n) = \sum_{i=1}^{n} r_i$. For simplicity, we assume T grows on the order of n. That is, $|T(n)| \leqslant |n|\,c$ for some $c \in \mathbb{R}_{>0}$, so the total input size of `Parallel-APSP` is $O(n)$.

The number of reduce tasks required for a MapReduce job is `R`. In general, the rule of thumb is to choose `R` large enough to ensure each compute node executes one or more reducers to increase the amount of parallelism. In a balanced setting, each reducer is expected to process $O\left(\frac{n^2}{R}\right)$ vertices. Finally, a reducer is assumed to have enough physical memory to buffer μ vertex records on average where $\mu < n$.

3.7. *Naïve parallel implementation*

A simple parallel algorithm for solving APSP would be to generate all pairs of vertices (i,j) for $0 < i < j \leqslant n$ in mappers, and send them to reducers. Each vertex must be replicated $n-1$ times inside the map function and sent over the network. The communication cost of this algorithm is $O(n^2)$. Figure 10 shows the replication factor and record distribution among the reducers for a sample graph.

In a cluster environment, the interconnect speed is limited, and network bandwidth is shared by all compute nodes. Transferring a record over the network usually takes longer than processing it. In addition, most MapReduce implementations use the local disk as the default storage medium for intermediate records. Map outputs and reduce inputs are temporarily stored on disk and gradually fed into main memory during computation. This is consistent with the initial design premise of MapReduce and similar data intensive frameworks where data size exceeds the total cluster memory. There is a performance penalty for shipping intermediate records over the network and moving them from the local disk into memory. Therefore, minimizing the intermediate data size can considerably improve the throughput.

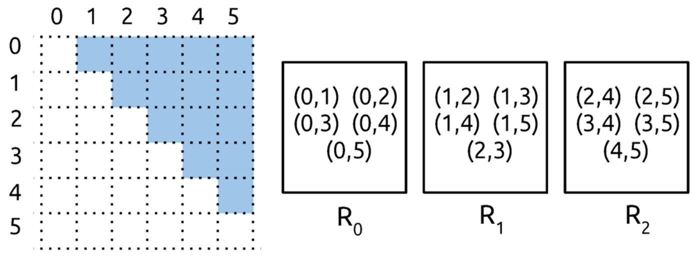


Figure 10. Map output for a small graph with six vertices. Blue squares correspond to vertex pairs sent from mappers to reducers. Each vertex is replicated five times, and the pairs are distributed among three reducers. In general, a graph with n vertices results in an intermediate output with $\binom{n}{2}$ pairs.

3.8. *Bucket-APSP*

The `Bucket-APSP` algorithm is designed to avoid replicating each vertex $n-1$ times. The idea is to group vertices in buckets and process a pair of buckets inside the reduce function. Each vertex is assigned to exactly one bucket. Reducers then compute an estimated shortest path between all pairs that contain one vertex from each bucket. Let `b` denote the number of buckets for the set $\{B_0, \ldots, B_{b-1}\}$. Each bucket contains $\frac{n}{b}$ vertices. A pair of buckets is labeled by a unique key (B_i, B_j) such that $0 \leqslant i \leqslant j < b$. Each bucket is paired with every other `b` buckets including itself. The total number of unique keys is $\binom{b+1}{2} = O(b^2)$. A reducer responsible from the bucket pair with key (B_i, B_j) stores all vertices from each bucket in main memory and computes the estimated shortest paths for each vertex pair `(u,v)` such that $u \in B_i$ and $v \in B_j$. Figure 11 shows a sample graph with six vertices and three buckets where each bucket contains two vertices.

The `Bucket-APSP` algorithm uses a fast data structure called the `BucketMap`. It is a hash map that associates a bucket with a list of bucket pairs that it participates in. That is, given a bucket B_i, `BucketMap[`B_i`]` returns an ordered list of bucket pairs B_xB_y such that $(0 \leqslant x \leqslant y < b)$ and $(x = i \vee y = i)$. For example, `BucketMap[`B_1`]` would return the ordered list $\{B_0B_1, B_1B_1, B_1B_2\}$ for the sample graph in Figure 11.

Algorithm 4 shows the pseudo-code for `Bucket-APSP`. Building and storing `BucketMap` takes $O(b^2)$ time and space `(lines 3-10)`. The input to `MAP` is the compact binary record that contains the list of shortest paths from a vertex v to the set of landmarks $\mathcal{L}$ computed by `Parallel-SSSP`. In `line 12`, v is assigned to one of the `b` buckets identified by b_v, and a copy of v is sent to the corresponding bucket pairs. The `REDUCE` function operates on a pair of buckets B_iB_j and all vertices that are assigned to B_i or B_j. Once the input vertices are saved in memory `(lines 20-27)`, a complete bipartite matching is performed between the two buckets and shortest paths are estimated for each vertex pair via `ALLPAIRS`. Note that when $i = j$, only a single bucket is processed inside the `REDUCE`,and such reducers are called mono-reducers.

`Bucket-APSP` has a lower replication factor than the naïve implementation. Each vertex goes exactly to one bucket. Because each bucket is replicated `b` times, total communication cost is $O(bn)$. Although a smaller `b` value indicates less communication, it should be chosen carefully to balance the amount of parallelism. During the reduce function, two buckets are stored in memory containing $\frac{2n}{b}$ vertices total. The maximum number of vertices that a reduce task can buffer is μ. Thus, $b = \frac{2n}{\mu}$ results in the lowest communication cost. This can be calculated easily on the basis of the average record size and the amount of physical memory in a compute node. However, as `b` decreases, the number of bucket pairs also goes down. A MapReduce job has `R` reduce tasks, and for maximum parallelism, none of them should be left idle. Ideally, the condition $\binom{b+1}{2}/R \geqslant 1$ should be satisfied to ensure that compute nodes have enough work to stay busy.

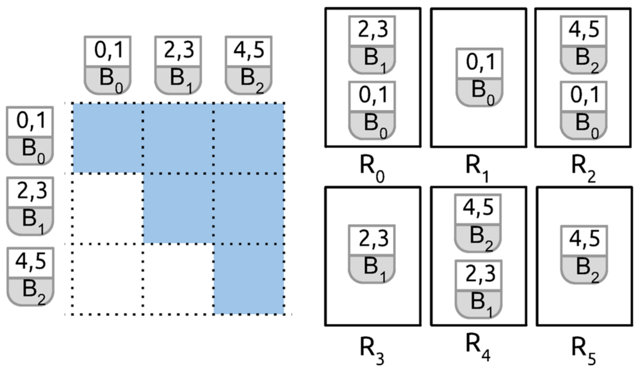


Figure 11. Map output of `Bucket-APSP` for the same graph in the previous figure. There are three buckets and six reducers. Each bucket contains two vertices. In this case, blue squares correspond to pairs of buckets processed by the same reduce task. Observe that each vertex is replicated only three times, and the communication cost is lower than the naïve implementation. Reducers R_1, R_3, and R_5 only contain a single bucket. Such reducers are called mono-reducers, and they compute all shortest paths within a single bucket.

Algorithm 4 Bucket-APSP

```
1: Input: Graph G=(V,E) with shortest paths from/to $\mathcal{L}$.
2: BucketMap ← Ø
3: function SETUP(b)                                   ▷ generate bucket pairs
4:     for $i = 0 \to b - 1$ do
5:         for $j = i \to b - 1$ do
6:             APPEND(BucketMap[i], $B_iB_j$)
7:             APPEND(BucketMap[j], $B_iB_j$)
8:         end for
9:     end for
10: end function
11: function MAP(v.id, v)
12:     $b_v$ = v.id mod b                             ▷ find the bucket v belongs with
13:     for all $B_iB_j \in$ BucketMap[$b_v$] do
14:         EMIT($B_iB_j$, v)
15:     end for
16: end function
17: function REDUCE($B_iB_j$, [$v_1$,...,$v_k$])
18:     $B_i$ ← Ø
19:     $B_j$ ← Ø
20:     for all v ∈ [$v_1$,...,$v_k$] do                ▷ save in memory
21:         bucket = v.id mod b
22:         if bucket = i then
23:             APPEND($B_i$, v)
24:         else
25:             APPEND($B_j$, v)
26:         end if
27:     end for
28:     if SIZE($B_i$) > 0 and SIZE($B_j$) > 0 then    ▷ compute all pairs
29:         ALLPAIRS($B_i$,$B_j$)
30:     else
31:         if SIZE($B_i$) > 0 then
32:             ALLPAIRS($B_i$,$B_i$)                   ▷ mono reducer
33:         else
34:             ALLPAIRS($B_j$,$B_j$)                   ▷ mono reducer
35:         end if
36:     end if
37: end function
38: function ALLPAIRS($B_x$,$B_y$)
39:     for all $v_x \in B_x$ do
40:         for all $v_y \in B_y$ do
41:             sp ← PATHCRAWLER($v_x$, $v_y$)
42:             EMIT($v_xv_y$, sp)
43:         end for
44:     end for
45: end function
```

The shortest paths between two disjoint subsets of vertices V_1 and V_2 can also be computed with `Bucket-APSP`. Copies of input vertices should partitioned into two sets of buckets S_1 and S_2, where $\mathrm{S_1}$ contains the buckets for $\mathrm{V_1}$ and $\mathrm{S_2}$ contains the buckets for $\mathrm{V_2}$. Then, each reducer can process a pair of buckets $\mathrm{B_iB_j}$ such that $\mathrm{B_i} \in \mathrm{S_1}$ and $\mathrm{B_j} \in \mathrm{S_2}$. The same idea can be generalized to k disjoint subsets for computing pairwise distances between vertex clusters or communities with a single pass of the input data.

3.9. Centrality estimation

Computing the closeness and betweenness centrality metrics both require solving APSP. Although `Bucket-APSP` reduces the communication cost of the naïve algorithm by a factor of $O\left(\frac{b}{n}\right)$, the quadratic computational cost of computing all pairwise distances makes application to large graphs infeasible. Instead of solving the exact APSP, we estimate these metrics in large graphs by adapting a sampling technique.

Given a sample set of vertices $S \subset V$, we estimate all shortest paths from S to V using the `PathCrawler` algorithm. The estimated closeness centrality of a vertex is then defined as the inverse of its average estimated distance to the sample dataset. Formally,

$$c'(v) = \frac{|S|}{\sum_{\substack{v \in V \\ u \in S}} d'(v,u)} \tag{13}$$

The parallel implementation of closeness centrality is fairly straightforward. The input is the same as `Parallel-APSP` and a sample dataset S. We assume the sample dataset can be buffered in local memory. That is, $|S| < \mu$. The `MAP` function simply reads the input from DFS and partitions it among `R` reducers. Each reducer gets $O(\frac{n}{R})$ vertices as its own share of input. Reducers also read S from DFS and buffer it in local memory. Therefore, the communication cost of this algorithm is $O(n + R|S|)$. The `REDUCE` function computes the estimated shortest paths between S and V_j for $0 \leqslant j < R$ in parallel using the `PathCrawler` algorithm.

The $|S| < \mu$ assumption is not a requirement and larger sample sizes can be handled by partitioning them and processing each partition as a separate MapReduce job. Alternatively, a similar grouping technique described in `Bucket-APSP` can be applied to run the entire computation in a single pass. In our experiments with large graphs, all sample sets were small enough to easily fit in memory.

Using this method, we can answer queries such as ‘Who are the most influential politicians of USA in Twitter?’ or ‘What are the most popular news websites in the Middle East?’ measuring influence or popularity by closeness centrality. Note that it is often a good practice to limit the total input size by region or category depending on context to achieve higher throughput and better performance.

Betweenness centrality is estimated similarly. Instead of counting how many times a vertex occurs in `all` shortest paths, we sample a large set of vertices $S \in V$ and compute the shortest paths for each pair in S using `Bucket-APSP`. Formally,

$$b(v) = \sum_{\substack{s \neq v \neq t \\ S \subset V}} \frac{\sigma_{svt}}{\sigma_{st}}, \qquad s,t \in S \quad \text{and} \quad v \in V \tag{14}$$

To increase accuracy, we slightly modify `PathCrawler` to return multiple estimated shortest paths between each `(s,t)` pair, rather than just one. Finally, we count how many times $\mathrm{v} \in \mathrm{V}$ occurs in the obtained shortest paths with another MapReduce job.

3.9.1. Suggesting a sample size. Determining the right sample size is important to make fast and accurate estimations. We use a geometric progressive sampling technique [30] to calculate the right sample size when estimating centrality. Let n_0 be the initial sample size and a be a constant. A schedule S_g of increasing sample sizes is defined as

$$S_g = a^i n_0 = \left\{n_0, a.n_0, a^2.n_0, \ldots, a^k n_0\right\} \tag{15}$$

At each step, the sample size is increased, and estimated closeness or betweenness centralities are computed. Then, vertices are sorted and ranked on the basis of their centrality values. This process continues until there is minimum change in vertex centrality rankings between two consecutive iterations. The change in rankings is determined by computing Spearman's rank correlation coefficient [31].

Rankings are usually more sensitive toward the lower end (less central part) of the distribution, and considering the ranks of the entire input set may require a large number of iterations. For fast convergence on large datasets, we compute the rank correlation among the top K vertices where K is a user supplied parameter generally ranging from a few hundreds to thousands. Typically, highest ranked vertices are less sensitive to changes, and this allows the algorithm to converge much faster. Another alternative for rapid convergence is to compute the fraction of overlapping vertices between two consecutive iterations. That is, the algorithm stops if the most central K vertices from the last two iterations overlap by more than $x\%$ where x is a user defined threshold.

4. EXPERIMENTS

4.1. Cluster setup

We ran the large scale experiments on a 20-node Hadoop cluster with 8 GB of memory and 4 Intel i7-2600K physical cores at 3.4 GHz in each machine. The main software stack includes Ubuntu 12.04 along with Java 1.6 64-bit and Hadoop 1.1.2.

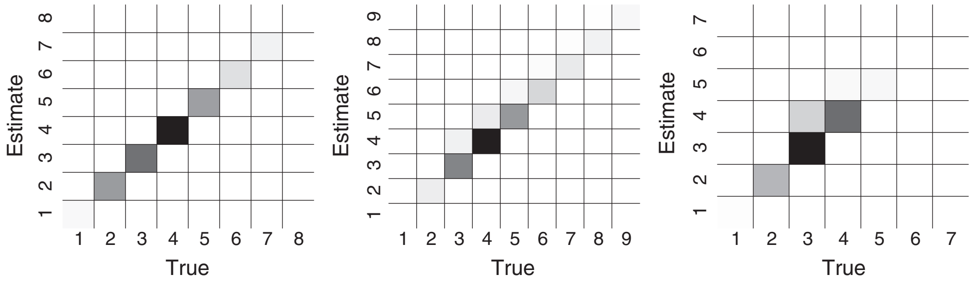
4.2. Accuracy

Figure 12 shows three heat maps that compare the true and estimated distance in medium-sized networks. For the pre-computation step, we used five landmarks with the highest degree in each network and stored up to 250 shortest paths between each (landmark, vertex) pair. For each graph, we calculated the exact and estimated shortest path lengths among half a million pairs. We computed the average error rate in distance estimation by the following formula:

$$\epsilon = \frac{\sum d_{approx}}{\sum d_{exact}} - 1 \tag{16}$$

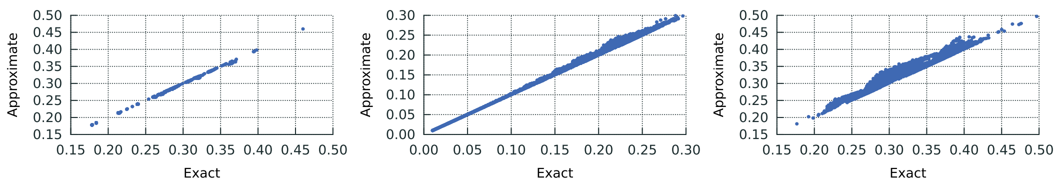
This metric is also called the `stretch` of the estimation. The Facebook graph resulted in the lowest stretch of 0.0002 followed by 0.0018 and 0.0030 in the Enron and Wikipedia graphs. We found the true shortest path length nine out of 10 times in all datasets.

The comparison of the true and estimated closeness centrality values are displayed in Figure 13. The near-linear correlation suggests that closeness centrality can be estimated highly accurately in



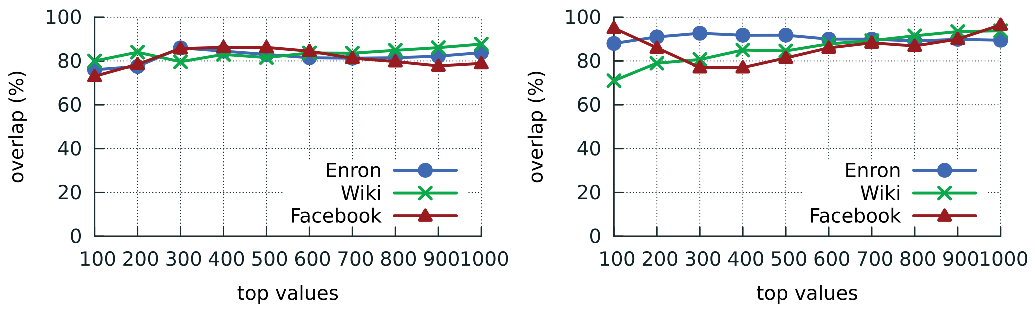
(a) Facebook (b) Enron (c) Wikipedia

Figure 12. Distance estimation accuracy: Darker squares correspond to denser areas of the pairwise distance distribution. The Facebook graph produced almost perfect results. A noticeable estimation error is seen in the Wikipedia graph at (3,4).



(a) Facebook: 0.9997 (b) Enron: 0.9968 (c) Wikipedia: 0.9724

Figure 13. Exact versus estimated closeness centrality and correlation coefficients.



(a) Betweenness Overlap (b) Closeness Overlap

Figure 14. Overlap of the most central nodes in the true and estimated results.

large graphs if pairwise distances are calculated with small error. Estimated closeness centrality values can replace the originals in large scale machine learning tasks such as a feature vector in a prediction algorithm.

We show the percentage overlap of the X most central nodes between the true and estimated results where X varies from 100 to 1000 in Figure 14. For example, we were able to identify the top 100 vertices with the highest closeness centrality in the Facebook graph with 96% overlap. As X increases, the overlap ratio stays mostly above 80%. Results from this experiment suggest that we can accurately answer social network analysis queries such as ‘Who are the top 500 most popular Twitter users?’ or ‘What are the top 100 most central websites with media content on the Internet?’.

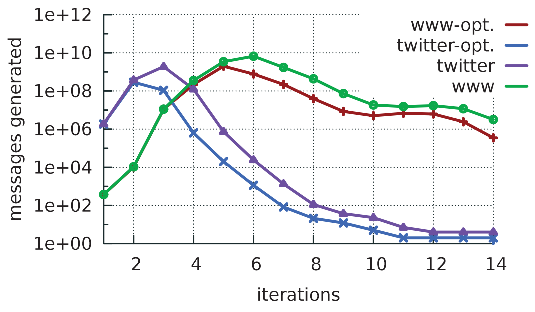
4.3. Performance

We evaluated the efficiency of the `Selective Push` optimization described in Section 3.2 by running the `Parallel-SSSP` algorithm on the Twitter and WWW datasets. We measured the number of distance messages sent from mappers to reducers with and without the bit vector optimization and plotted the results in Figure 15(a). Observe that the bit vector dramatically reduces the number of messages sent from mappers to reducers when the number of messages is over 100 million, resulting in an order of magnitude improvement. The `Selective Push + IGS` combination reduced the run time from 146 to 19 min in the Twitter graph and 581 to 115 min in the WWW graph for a single landmark.

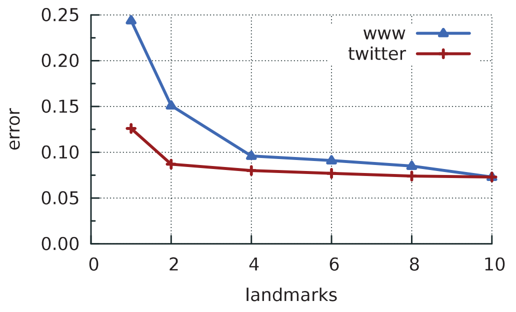
The accuracy of large scale distance estimation with increasing number of landmarks is reported in Figure 15(b). For this experiment, we chose 10 random vertices referred as the `oracles` and computed the length of the shortest path from each `oracle` to the rest of the graph using `Parallel-SSSP`. Then, we sampled 5 million vertices, estimated their distance to the `oracles`, and reported the error. Notice that even in the WWW graph, which contains 700 million nodes and 12 billion edges, we were able to calculate the shortest paths with 7% average error using only 10 landmarks. In both graphs, we picked out the highest degree vertices as the landmarks.

4.4. Effects of the small-world phenomena

In `Parallel-SSSP`, the time spent on a particular iteration varies significantly on the basis of the number of shortest paths computed. For example, the first iteration runs relatively fast because

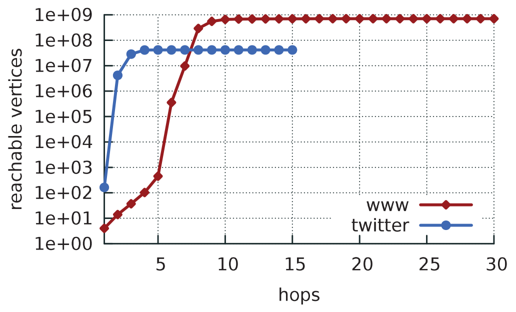


(a) Performance Benefits of the Bit-Vector

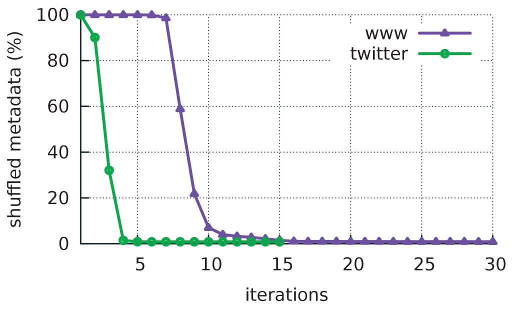


(b) Error Rate vs. Number of Landmarks

Figure 15. (a) Number of distance messages with and without the bit vector for the first 15 iterations of `Parallel-SSSP`. Note that the WWW graph converges in approximately 60 iterations. (b) Accuracy of distance estimation versus number of landmarks.



(a) Hop plot for a randomly chosen vertex



(b) Shuffled vertex records in Parallel-SSSP

Figure 16. The booming effect in Parallel-SSSP on large graphs due to the small-world phenomena. The figures are plotted by sampling a set of nodes from both graphs and averaging the results to represent a single random vertex.

only the neighbors of the source vertex are discovered. The second one takes longer because all vertices that are two hops further from the source are identified. This increase continues exponentially until the majority of the vertices are discovered. Because of the small-world and shrinking diameter phenomena in large graphs [22, 27], the effective diameter is small and over 90% of the vertices are discovered rapidly. In practice, there are usually two or three `booming` iterations where a large portion of the shortest paths are computed. When a high-degree vertex is chosen as the source, a booming effect is often observed within two or three hops around it. This behaviour can be examined in Figure 15(a). There is a steep increase in the number of messages generated during the second and third iterations for the Twitter and WWW graphs, respectively. The number of messages begins to go down in both graphs soon after reaching the peak, as most of the shortest paths are already computed at that point. The overall run time of `Parallel-SSSP` is dominated by the first D' iterations where D' is the effective diameter.

It may take longer to observe the booming effect when the algorithm is started from a random low-degree vertex, which is loosely connected to the rest of the graph. Figure 16(a) shows the total number of reachable vertices from a random source within increasing number of hops. Notice that a rapid jump starts with the fifth hop in the WWW graph. Almost all vertices are reachable by the 10th hop in both graphs except for the long and sparse chain-like sequences.

Figure 16(b) shows the percentage of records sent from mappers to reducers when `Parallel-SSSP` is started from a random vertex. Another interpretation of this figure is the decrease of the problem size over time. As we explained the `IGS` technique in Section 3.1, recently discovered vertices are sent to the final output location on DFS and discarded from the remaining iterations when the graph is unweighted `(line 12 of Algorithm 1)`. Consequently, the records sent from mappers to reducers correspond to those vertices that have not been discovered yet. As the figure suggests, large percentage rates are observed within the first few iterations and the algorithm starts converging rapidly afterwards. The input size reduces to a small fraction of the initial value after reaching the effective diameter. Notice that Figures 16(a) and 16(b) is reverse-shaped plots emphasizing on two different measures that are inversely proportional. Both figures indicate that the majority of the computation and communication takes place within the first D' iterations.

4.5. *Performance optimizations in parallel single source shortest path problem*

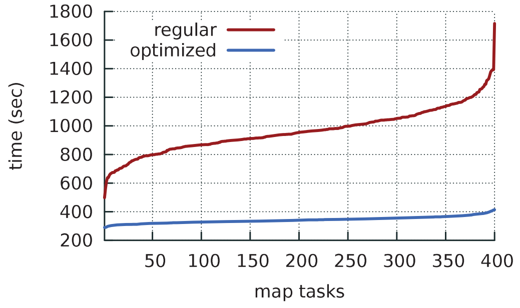
The efficiency of `Parallel-SSSP` comes from three novel optimization techniques:

- `IGS`: sends vertices to the final output location on DFS soon after they are discovered–`Section 3.1`
- `Selective Push`: uses bit vectors to suppress redundant messages sent from mappers to reducers–`Section 3.2`
- `Fat Vertex Optimization`: handles high-degree vertices exclusively to avoid the straggling map task problem–`Section 3.3`

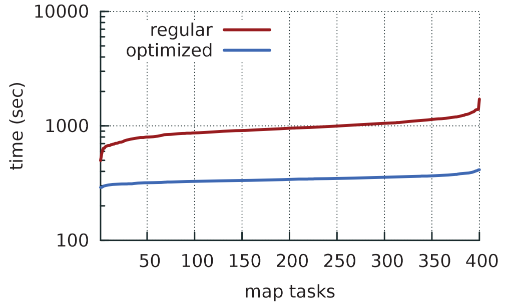
Figures 17 and 18 show the combined effect of these optimizations on the WWW and Twitter datasets. We compare the optimized algorithm with a *regular* version, which does not contain any of the optimizations earlier. We plot the run time distribution of the map tasks for one of the long-running iterations that dominate the total run time in both versions of the algorithm. Both figures contain 400 map tasks sorted in ascending order of execution time.

The straggling map tasks are easily observable in Figures 17(a) and 18(a) with the long tail rising steeply toward the end. The longest running map task for the WWW graph takes about 1750 s to complete, whereas the shortest one runs in approximately 500 s with the regular algorithm, which does not use any of the optimization techniques. The gap is larger for the Twitter graph with the slowest and fastest map tasks taking 1300 and 6250 s, respectively. The main reason for the increasing tail is the presence of very few high-degree vertices in both graphs because of the skewed degree distribution. The more edges a map task processes, the longer it takes to complete. Map tasks assigned to process the highest degree vertices fall behind and result in underutilization of the compute cluster. A large number of idle CPU cores hold onto the few busy ones without doing any useful work until the `REDUCE` phase starts. The underutilization can be observed more clearly in Figure 18(a). The first 200 tasks finish in approximately 2000 s, whereas the second half takes over 6000 s to complete. If there were 400 CPU cores to run all tasks simultaneously, half of the cluster would stay idle for over an hour.

The optimized algorithm removes the long tail at the end and balances individual task execution times. Load balancing helps increasing the cluster utilization and decreasing the total run time dramatically. The fastest and slowest map tasks take 290 and 414 s for the WWW graph. The gap is smaller for the Twitter network with task run times ranging from 56 to 128 s. The overall effect

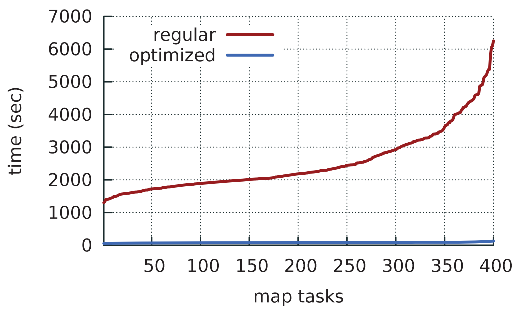


(a) Normal scale

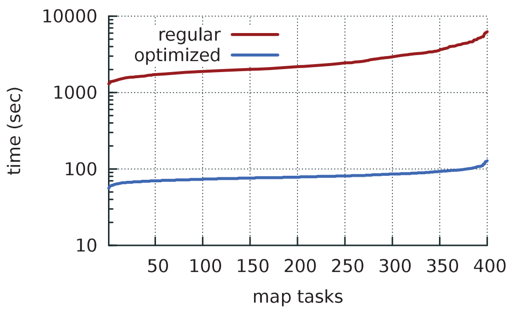


(b) Logarithmic scale

Figure 17. Map task run times for iteration #4 over the WWW graph. Median task completion times are 954 and 340 s for the regular and optimized versions, respectively.



(a) Normal scale



(b) Logarithmic scale

Figure 18. Map task run times for iteration #3 over the Twitter graph. Median task completion times are 2183 and 78 s for the regular and optimized versions, respectively.

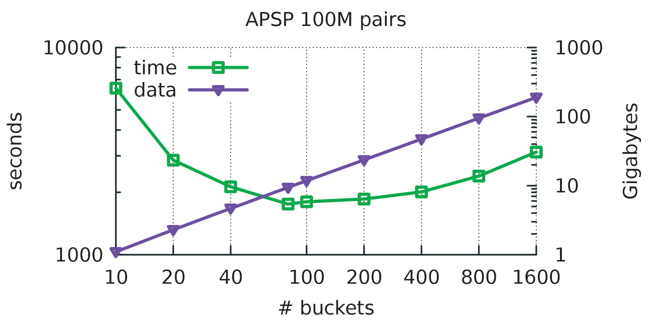


Figure 19. Number of buckets versus run time and intermediate data in Bucket-APSP.

of the optimization techniques for a single iteration is observable in Figures 17(b) and 18(b). There is a noticeable gap between the regular and optimized versions with median task run times going down from 954 to 340 s for the WWW graph and 2183 to 78 s for the Twitter graph.

To understand why the optimized algorithm yields better speed-up for the Twitter dataset, we look at the individual contribution of each technique. Figure 15(a) shows that `Selective Push` kicks in starting iterations #6 and #3 for the WWW and Twitter datasets, respectively. The difference between the number of messages generated with and without the bit vector is negligible in iteration #4 of the WWW graph. However, there is over an order of magnitude difference between the two quantities in iteration #3 of the Twitter dataset. As a result, `Selective Push` yields considerable speed-up on the Twitter dataset, whereas it is not as effective on the WWW graph for the particular iterations considered. The three times speed-up in iteration #4 of the WWW graph mainly results from handling high-degree vertices efficiently and avoiding straggling map tasks. The Twitter dataset benefits from all optimizations so a much higher speed-up is observed during iteration #3. A similar speed-up for the WWW dataset can be observed once `Selective-Push` becomes more effective starting iteration #6.

4.6. Intermediate data versus run time in Bucket-APSP

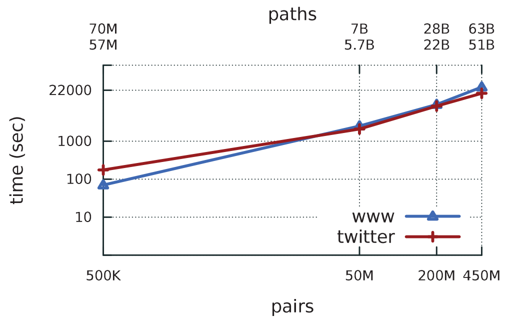
The number of buckets should be chosen carefully to maximize cluster utilization and minimize the run time in `Bucket-APSP`. The communication cost of the algorithm is $O(bn)$ where b is the number of buckets. Intuitively, as b decreases, the run time of the algorithm is expected to go down because less intermediate data are generated. However, if the total number of unique bucket pairs is less than the number of available CPU cores, the cluster utilization goes down. That is, in general, $\binom{b+1}{2}$ should be a multiple of R where R is the total number of reduce tasks. This ensures that each reduce task receives multiple bucket pairs as input. Increasing b can have a positive effect on the run time performance because it results in higher record granularity, that is, more even distribution of vertex pairs among the reduce tasks. On the other hand, very large b values can quickly inflate the intermediate data and reduce the performance.

We plot the change in run time and the amount of intermediate data with varying number of buckets while estimating the distance between 10,000 vertices. Figure 19 shows the results for 100 million vertex pairs. As expected, the size of the intermediate data increases with larger b values, although the run time varies depending on the cluster utilization. Observe that increasing b from 10 to 80 results in a noticeable performance improvement. This is because more CPU cores are utilized, and the total work is distributed evenly among the reduce tasks. As the number of buckets goes beyond 200, generating and shuffling excessive amounts of intermediate records (bucket pairs) starts to dominate the total execution time of the algorithm and affects the performance negatively.

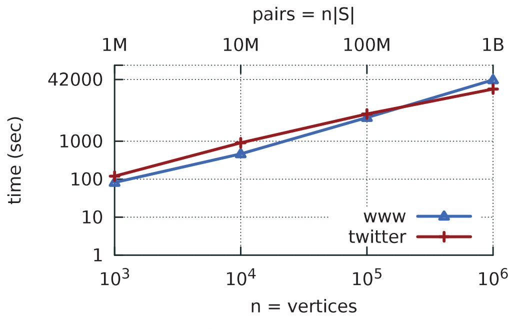
4.7. Scalability

Figure 20 shows how the parallel closeness and betweenness centrality algorithms scale with increasing input size. For betweenness centrality, we sampled 30,000 vertices and computed up to 250 shortest paths between each pair in the largest setting. This resulted in approximately 450 million pairs of vertices. We calculated a maximum of 63 and 51 billion shortest paths in the WWW and Twitter graphs, respectively. For closeness centrality, we sampled the top one million vertices with the highest degree and ranked them on the basis of the estimated values. In the largest setting, we estimated the distance between one billion vertex pairs in 11.5 h. Observe that both algorithms scale almost linearly with increasing input.

We measured an average system throughput of 17,600 pairs/s for betweenness and 20,300 pairs/s for closeness centrality, which includes the overhead to start a MapReduce job and initialize all tasks. Note that the betweenness centrality algorithm has lower throughput because for a given pair of

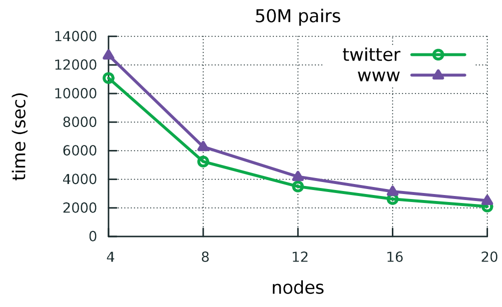


(a) Betweenness Scalability

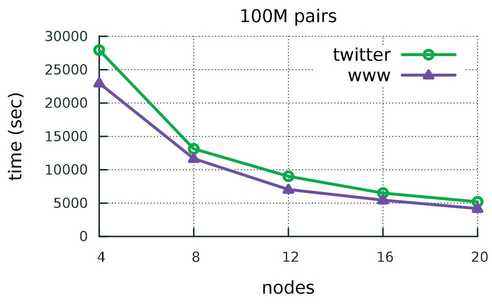


(b) Closeness Scalability

Figure 20. Estimating centrality in billion-scale: change in run time with increasing input size.



(a) Betweenness Scalability



(b) Closeness Scalability

Figure 21. Estimating centrality in billion-scale: change in run time with increasing hardware resources.

Table II. Top 10 most central nodes in large networks.

a. Closeness centrality		b. Betweenness centrality	
Twitter	WWW-2002	Twitter	WWW-2002
BarackObama	google.com	Ashton Kutcher	google.com
Ashton Kutcher	yahoo.com	CNN Breaking News	yahoo.com
Ellen DeGeneres	dmoz.org	Barack Obama	dmoz.org
CNN Breaking News	rambler.ru	Britney Spears	rambler.ru
Oprah	hypermail.org	Ellen DeGeneres	hypermail.org
Ryan Seacrest	macromedia.com	Twitter	dmoz.org
Britney Spears	refdesk.com	SHAQ	careerbank.com
SHAQ	wunderground.com	Oprah	adobe.com
Twitter	commondreams.org	Ryan Seacrest	sourceforge.net
@shravandude	blogger.com	Lance Armstrong	phpb.com

vertices, it computes up to 250 estimated shortest paths, whereas the closeness centrality algorithm only computes a scalar distance estimate for each pair.

The average distance estimation time for a pair of vertices inside a single computer is under 8 ms for both input graphs. For comparison, the `Landmark-BFS` algorithm is reported to estimate the distance between a pair of vertices on a comparable Twitter graph in 889 ms. This yields over two orders of magnitude improvement for a single compute node. Our distance estimation algorithm runs in parallel where each compute node can work independently on different pairs. Thus, we get an additional speed-up from parallelism that increases with cluster size. The scalability of the closeness and betweenness centrality algorithms with increasing number of compute nodes is shown in Figure 21. Higher number of hardware resources results in increased system throughput and faster application run time with linear scalability. We finally report the top 10 vertices with the highest betweenness and closeness centralities from the Twitter and WWW datasets in Table II.

5. DISCUSSION AND RELATED WORK

The optimization techniques described for `Parallel-SSSP` are generalizable to other `vertex-centric` distributed graph algorithms and frameworks. The `fat-vertex` optimization can be used on any large graph that exhibits skewed degree distribution where the performance is affected by a small number of high-degree vertices. This is a common problem in graph processing as the degree distributions of many real-world graphs follow power laws. The `Selective Push` optimization can be used to target a particular subset of vertices with higher priority for fast convergence of iterative graph algorithms. Recent examples to such algorithms include computing the incremental PageRank and identifying connected components and graph-based label propagation for recommendation systems using prioritized iterative computations [32].

Parallel distance estimation has many applications in graph mining. The `neighborhood function` $N(h)$ of a graph is defined as the number of vertex pairs that are within h hops of each other [33]. The definition is extended further for individual nodes and sub-graphs [34]. The `individual neighborhood function` $IN(u,h)$ of a vertex u is defined as the number of nodes that are reachable from u in at most h hops. For small graphs, $N(h)$ can be computed by summing the `individual neighborhood function` over all $u \in V$. Given two subsets of graph vertices V_1 and V_2, `the generalized neighborhood function` $N(h, V_1, V_2)$ gives the number of pairs between V_1 and V_2 that are within distance h or less. Approximate neighborhood functions have been used previously for measuring vertex importance, detecting sub-graph similarity and clustering sub-graphs in large networks. Hoba can be used to compute exact or estimated values for these functions in billion-scale graphs. Exact answers to $IN(u,h)$ can be calculated efficiently with `Parallel-SSSP` without the requirement to store the entire graph structure in cluster memory. Fast estimations to $IN(u,h)$ can be provided by running `Path-Crawler` in parallel, eliminating the need for a complete graph traversal from arbitrary u. The `neighborhood` `function` $N(h)$ can be estimated by sampling a set of vertices, summing the exact or estimated individual neighborhood functions for each element of the sample set and extrapolating the results. Finally, the `Bucket-APSP` algorithm can be used to estimate the `generalized neighborhood function` by computing all pairwise distances between V_1 and V_2.

Centrality has several use cases for measuring the structural importance of a vertex. It can be used for identifying critical hubs in a road network, detecting vulnerabilities in router networks, or finding the most influential people in social networks. Another application area of centrality is graph clustering and community detection. Communities are dense sub-graphs that are loosely connected with each other. Previously, closeness and betweenness centralities have been used as building blocks of the k-medoids and Girvan–Newman algorithms for clustering medium-sized networks [35–37]. The parallel algorithms we presented for estimating centrality can be incorporated into similar techniques to cluster massive graphs using cheap commodity hardware.

Several frameworks have been proposed to solve problems that concern large graphs. GraphLab [38, 39] is a programming abstraction to express dynamic graph-parallel computations asynchronously in shared and distributed memory environments. The computation is carried out as a distributed loop that processes vertices in parallel, based on a prioritized ordering scheme. Pregel [40] is a functional programming based model that aims to solve large scale graph problems. The unit of computation is called a `superstep` during which a vertex exchanges messages with its neighbors. The graph structure including the vertices, edges, and the computation state is stored in distributed memory. In graph mining, MapReduce has been used for detecting connected components, enumerating triangles and sub-graphs and diameter estimation, and finding sub-graphs of high connectivity [25, 41–43].

From a distributed graph computation standpoint, the fundamental idea behind all abstractions earlier is the same: partition the graph, process each vertex independently, and send/receive messages between neighboring vertices. In this paper, our main concentration is on distance and centrality estimation in large graphs rather than potential speed-ups in execution time resulting from the platform of choice. We address performance issues related to large scale graph mining algorithms common to all distributed platforms. Although Hoba is built on MapReduce, the algorithms described in this paper can be implemented using any other framework optimized for large scale graph processing with minor modifications.

6. CONCLUSION

We have motivated, described, and evaluated Hoba, a scalable and efficient library for estimating distance and centrality on shared nothing architectures. Hoba runs on top of Hadoop, and it can handle graphs with hundreds of millions of nodes and billions of edges using a small computing cluster. To the best of our knowledge, there is no other work on batch computation of shortest paths in large graphs. We described methods for optimizing the single source shortest path and APSP problems in MapReduce and proposed a novel parallel algorithm for estimating shortest paths in shared nothing architectures. We used these algorithms to estimate closeness and betweenness centrality metrics and identified the vertices with top centrality scores in the largest publicly available real-world graphs. All algorithms described in this paper can run on weighted and directed graphs with minor modifications. Hoba is an open source and compatible with the Amazon Elastic MapReduce service. It allows any network scientist to mine billion-scale graphs for moderate costs. Our approach avoids the overhead of using high-end servers that are expensive and hard to maintain. We encourage using Hoba for graph clustering, social network analysis, and Web mining.

ACKNOWLEDGEMENT

This research was supported by the National Institutes of Health under grant R01LM010132 and the US Department of Energy under contract DE-AC02-06CH11357. We thank Bob Barlett and Philip Kaufman from The University of Chicago's Computer Science Department for their help in configuring the hardware for the large scale experiments.

REFERENCES

1. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms* (3rd edn). The MIT Press, 2009.
2. Floyd RW. Algorithm 97: shortest path. *Communications of the ACM* 1962; **5**(6):345.
3. Potamias M, Bonchi F, Castillo C, Gionis A. Fast shortest path distance estimation in large networks. *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, Honkong, China, 2009; 867–876.
4. Mao Y, Saul LK, Smith JM. Ides: an Internet distance estimation service for large networks. *IEEE Journal on Selected Areas in Communications* 2006; **24**(12):2273–2284.
5. Das Sarma A, Gollapudi S, Najork M, Panigrahy R. A sketch-based distance oracle for Web-scale graphs. *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, New York City, USA, 2010; 401–410.
6. Tretyakov K, Armas-Cervantes A, García-Bañuelos L, Vilo J, Dumas M. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, Glaskov, UK, 2011; 1785–1794.
7. Gubichev A, Bedathur S, Seufert S, Weikum G. Fast and accurate estimation of shortest paths in large graphs. *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, Toronto, Canada, 2010; 499–508.
8. Maier M, Rattigan M, Jensen D. Indexing network structure with shortest-path trees. *ACM Transactions on Knowledge Discovery from Data* 2011; **5**(3):15:1–15:25.
9. Diamond J, Burtscher M, McCalpin JD, Kim Byoung-Do, Keckler SW, Browne JC. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, 2011; 32–43.
10. Liu L, Li Z, Sameh AH. Analyzing memory access intensity in parallel programs on multicore. *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, Island of Kos, Greece, 2008; 359–367.
11. Bader DA, Madduri K. Parallel algorithms for evaluating centrality indices in real-world networks. *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, Columbus Ohio, 2006; 539–550.
12. Ediger D, Jiang K, Riedy J, Bader DA, Corley C, Farber RM, Reynolds WN. Massive social network analysis: mining twitter for social good. *ICPP*, San Diego, CA, 2010; 583–593.
13. Madduri K, Ediger D, Jiang K, Bader DA, Chavarria-Miranda D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, Rome, Italy, 2009; 1–8.
14. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
15. Siganos G, Tauro SL, Faloutsos M. Jellyfish: a conceptual model for the as internet topology. *Journal of Communications and Networks* 2006; **8**(3):339–350.
16. Freeman LC. Centrality in social networks: conceptual clarification. *Social Networks* 1979; **1**(3):215–239.
17. Freeman LC. A set of measures of centrality based on betweenness. *Sociometry* 1977; **40**(1):35–41.
18. Brandes U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 2001; **25**:163–177.
19. Kang U, Papadimitriou S, Sun J, Tong H. Centralities in large networks: algorithms and observations. *SDM*, Mesa, Arizona, 2011; 119–130.
20. Kleinberg JM, Kumar R, Raghavan P, Rajagopalan S, Tomkins AS. The Web as a graph: measurements, models, and methods. *Proceedings of the 5th Annual International Conference on Computing and Combinatorics*, COCOON'99, Springer-Verlag, Berlin, Heidelberg, 1999; 1–17.
21. Redner S. How popular is your paper? An empirical study of the citation distribution. *The European Physical Journal B* 1998; **4**(2):131–134.
22. Leskovec J, Kleinberg J, Faloutsos C. Graphs over time: densification laws, shrinking diameters and possible explanations. *KDD*, Chicago IL, 2005; 177–187.
23. Kang U, Tsourakakis CE, Appel AP, Faloutsos C, Leskovec J. Hadi: mining radii of large graphs. *ACM Transactions on Knowledge Discovery from Data* 2011; **5**(2):8:1–8:24.
24. Croft B, Metzler D, Strohman T. *Search Engines: Information Retrieval in Practice* (1st edn). Addison-Wesley Publishing Company: USA, 2009.
25. Afrati FN, Fotakis D, Ullman JD. Enumerating subgraph instances using map-reduce. *CoRR* 2013; **abs/1208.0615**:62–73.
26. Afrati FN, Sarma AD, Menestrina D, Parameswaran AG, Ullman JD. Fuzzy joins using mapReduce. *ICDE*, Arlington, Virginia, 2012; 498–509.
27. Reka A, Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics* 2002; **74**:47–97.
28. Chakrabarti D, Faloutsos C. *Graph Mining: Laws, Tools, and Case Studies*, Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan & Claypool Publishers, 2012.
29. Lin J, Dyer C. Data-intensive text processing with mapReduce. *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, Association for Computational Linguistics, Stroudsburg, PA, USA, 2009; 1–2.
30. Provost F, Jensen D, Oates T. Efficient progressive sampling. *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, San Diego, CA, 1999; 23–32.

31. Spearman C. The proof and measurement of association between two things. By C. Spearman, 1904. *The American Journal of Psychology* 1987; **100**(3-4):441–471.
32. Zhang Y, Gao Q, Gao L, Wang C. Priter: a distributed framework for prioritized iterative computations. *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, Cascais, Portugal, 2011; 13:1–13:14.
33. Faloutsos M, Faloutsos P, Faloutsos C. On power-law relationships of the Internet topology. *SIGCOMM Computer Communication Review* 1999; **29**(4):251–262.
34. Palmer CR, Gibbons PB, Faloutsos C. ANF: a fast and scalable tool for data mining in massive graphs. *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, Edmonton, Alberta, Canada, 2002; 81–90.
35. Kaufman L, Rousseeuw PJ. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, 1990.
36. Girvan M, Newman MEJ. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 2002; **99**:7821–7826.
37. Rattigan MJ, Maier M, Jensen D. Graph clustering with network structure indices. *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, Corvallis, OR, 2007; 783–790.
38. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. Graphlab: a new parallel framework for machine learning. *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010; 340–349.
39. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB* 2012; **5**(8):716–727.
40. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 International Conference on Management of Data*, Sydney, Australia, 2010; 135–146.
41. Kang U, Tsourakakis CE, Faloutsos C. Pegasus: a peta-scale graph mining system implementation and observations. *Ninth IEEE International Conference on Data Mining, 2009. ICDM '09*, Miami FL, 2009; 229–238.
42. Cohen J. Graph twiddling in a mapReduce world. *Computing in Science & Engineering* 2009; **11**(4):29–41.
43. Suri S, Vassilvitskii S. Counting triangles and the curse of the last reducer. *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, Hyderabad, India, 2011; 607–614.