Massively Parallel Algorithms for Personalized PageRank

Guanhao Hou
Xingguang Chen
Sibo Wang
The Chinese University of Hong Kong
ghhou,xgchen,swang@se.cuhk.edu.hk

Zhewei Wei
Gaoling School of Artificial Intelligence
Renmin University of Chia
zhewei@ruc.edu.cn

ABSTRACT

Personalized PageRank (PPR) has wide applications in search engines, social recommendations, community detection, and so on. Nowadays, graphs are becoming massive and many IT companies need to deal with large graphs that cannot be fitted into the memory of most commodity servers. However, most existing state-of-the-art solutions for PPR computation only work for single-machines and are inefficient for the distributed framework since such solutions either *(i)* result in an excessively large number of communication rounds, or *(ii)* incur high communication costs in each round.

Motivated by this, we present *Delta-Push*, an efficient framework for single-source and top-k PPR queries in distributed settings. Our goal is to reduce the number of rounds while guaranteeing that the load, i.e., the maximum number of messages an executor sends or receives in a round, can be bounded by the capacity of each executor. We first present a non-trivial combination of a redesigned parallel push algorithm and the Monte-Carlo method to answer single-source PPR queries. The solution uses pre-sampled random walks to reduce the number of rounds for the push algorithm. Theoretical analysis under the *Massively Parallel Computing (MPC)* model shows that our proposed solution bounds the communication rounds to $O(\log \frac{n^2 \log n}{\epsilon^2 m})$ under a load of $O(m/p)$, where m is the number of edges of the input graph, p is the number of executors, and ϵ is a user-defined error parameter. In the meantime, as the number of executors increases to $p' = \gamma \cdot p$, the load constraint can be relaxed since each executor can hold $O(\gamma \cdot m/p')$ messages with invariant local memory. In such scenarios, multiple queries can be processed in batches simultaneously. We show that with a load of $O(\gamma \cdot m/p')$, our Delta-Push can process γ queries in a batch with $O(\log \frac{n^2 \log n}{\gamma \epsilon^2 m})$ rounds, while other baseline solutions still keep the same round cost for each batch. We further present a new top-k algorithm that is friendly to the distributed framework and reduces the number of rounds required in practice. Extensive experiments show that our proposed solution is more efficient than alternatives.

PVLDB Reference Format:
Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. Massively Parallel Algorithms for Personalized PageRank. PVLDB, 14(9): 1668 - 1680, 2021.
doi:10.14778/3461535.3461554

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461554

1 INTRODUCTION

Given a directed graph G, a source vertex s, and a decay factor α, a random walk (more precise, a random walk with restart [11]) from s is a random traversal on G such that *(i)* it starts from s; *(ii)* at each step it terminates at the current vertex with α probability and randomly jumps to one of the out-neighbors of the current vertex with $1 - \alpha$ probability. The *personalized PageRank (PPR)* of vertex t with respect to source s is the probability that a random walk from s terminates at vertex t. Intuitively, the personalized PageRank $\pi_s(t)$ of vertex t with respect to vertex s indicates the importance of vertex t from the perspective of s. Hence, PPR is widely used as a vertex proximity in data mining area and has many applications in search engines [19, 30], social recommendations [17], community detection [2], spam detection [18], and so on.

Due to the important applications of PPR, it has attracted a plethora of research work [25, 26, 36, 37, 39–41] to improve the efficiency of PPR computation. Most of the solutions are designed for single-machines and hardly work in distributed settings. However, graphs are becoming massive and growing rapidly in the era of big data and many IT companies need to handle huge graphs that cannot be fitted into the main memory of most commodity servers. In such scenarios, single-machine algorithms no longer work and distributed algorithms become the paradigm.

Bahmani et al. [6] utilize MapReduce to calculate PPR queries in distributed setting and propose the *Doubling* algorithm to reduce the communication round over the valiant Monte-Carlo approach. The state-of-the-art Monte-Carlo based solution is *DistPPR* proposed by Lin [24]. DistPPR still optimizes the Monte-Carlo method by exploiting the parallel pipeline framework. They further introduce optimization techniques to avoid exploding the memory capacity by hierarchical sampling on large vertices and save sampling cost by pre-storing short random walks on vertices with a small degree. However, Monte-Carlo based methods [6, 24] incur a large number of random walks and can hardly measure the workload of each vertex, making it difficult to design a load-balancing partition scheme. Besides, the pipeline method makes the load, i.e., the maximum number of messages an executor sends or receives in a round, and the communication cost per round to be determined by expected values instead of a fixed value. This incurs huge overheads (reshuffling and repeating the task) if the load exceeds the memory of some executors. Guo et al. [15] explore the linearity of PPRs and pre-compute partial results for hub vertices on each executor. When a query comes, it makes use of the pre-stored partial results to speed up the query processing. The solution in [15], however, is still not scalable to graphs with billion edges under the $O(m/p)$ load constraint since *(i)* its load is $O(n \cdot p)$, which increases with the

Table 1: Frequently used notations

Notation	Description
$G = (V, E)$	A graph with vertex set V & edge set E
n, m	The number of vertices and edges
p	The number of processor in the cluster
γ	The batch size to fit the scale of cluster
$N_{out}(v)$	The set of out-neighbors of vertex v
α	The random walk termination probability
$\pi_s(v)$	The exact PPR of v with respect to s
$\widetilde{\pi}_s(v)$	The estimation of PPR of v with respect to s
ϵ	The relative error bound in Definitions 2.1-2.2
δ	The threshold in Definitions 2.1-2.2
p_f	The failure probability of the estimation
$\hat{\pi}_s(v)$	The reserve of v with respect to s
$r_s(v)$	The residue of v with respect to s

number of executors; *(ii)* it prepossesses the PPR vector of a considerable amount of hub nodes whose sizes are difficult to bound. This further makes it difficult to bound the load to $O(m/p)$. In this paper, we focus on methods that can achieve a load of $O(m/p)$.

Motivated by the limitations of existing solutions, we present *Delta-Push*, a new distributed framework for efficient single-source and top-k PPR query processing. In distributed settings, the communication costs among different executors and the number of rounds are the main factors that affect the overall performance. On one hand, each executor has limited memory and cannot hold messages with a size larger than its capacity. Therefore, the size of messages received on each executor usually needs to be limited by a threshold. In the *Massively Parallel Computing (MPC)* model, two distributed algorithms are compared by the number of rounds required under the same load. The less the number of rounds the algorithm requires, the more efficient the algorithm is. We show that the number of rounds of our proposed Delta-Push can be bounded by $O(\log \frac{n^2 \log n}{\epsilon^2 m})$, which performs better, i.e., requires less number of rounds, than existing alternatives when the required memory for each executor is $O(m/p)$, where m is the number of edges and p is the number of executors. Moreover, as the number of executors increases to $p' = \gamma \cdot p$, the load can be relaxed since each executor can support up to $O(\gamma \cdot m/p')$ messages with invariant local memory, and the amortized number of rounds for one query will become $O(\frac{1}{\gamma} \log \frac{n^2 \log n}{\gamma \epsilon^2 m})$. With a reasonable assumption that $\gamma = \Omega(\log n/\epsilon)$ and $m = \Omega(n)$, we finally reach amortized $O(\frac{1}{\gamma} \log(n/\epsilon))$ rounds for each query.

The main idea of the distributed single-source PPR query algorithm is to combine a redesigned push algorithm with the Monte-Carlo method to reduce the number of rounds. We will show that by pre-storing random walks, it is possible to reduce the number of rounds of the push algorithm. However, the larger number of the random walks we pre-store, the larger communication overhead it brings when we combine the push results and the Monte-Carlo results by a join operation. How to pre-sample a proper number of random walks while still guaranteeing that the load of each executor can be bounded? We tackle this challenging issue by a careful theoretical analysis. Next, we further present a new top-k algorithm. To our knowledge, none of existing distributed algorithms for PPR provide any efficient top-k algorithm but simply adopts the single-source query algorithm to answer the top-k queries, causing unnecessarily high running costs.

Moreover, our method is more friendly to the distributed setting compared to the existing test-and-trial method. In particular, how to examine if the top-k answer is accurate enough (to provide approximation guarantee)? Typically, we need to either gather the top-k lower/upper bounds for the PPR estimations or the top-k PPR estimation scores. In this case, we need to first combine the push result and the pre-stored random walks by a join operation, and then invoke a sorting procedure to derive the top-k answers. This incurs unnecessarily high number of rounds and is not expected. Our designed top-k algorithm avoids such additional overheads and requires (practically) less number of rounds compared to the single-source counterparts. Extensive experiments on large datasets demonstrate that our proposed Delta-Push is up to an order of magnitude faster than alternatives.

2 PRELIMINARIES

2.1 Problem Definition

Given a directed graph G[1], the personalized PageRank (PPR) $\pi_s(v)$ of any $v \in V$ with respect to s is defined as the probability that a random walk starting from s terminates at v. An important type of PPR query is the single-source PPR query where we are given a source s, and the goal is to return the PPR $\pi_s(v)$ of each vertex $v \in V$ with respect to the source s. Computing the exact PPR is computational expensive and most existing studies, e.g., [6, 24, 39], consider approximate single-source PPR query defined as follows.

Definition 2.1. ((ϵ, δ)-approximate single-source PPR) Given a source vertex s, a threshold δ, an error bound $\epsilon \in (0, 1]$, and a failure probability p_f, an approximate single-source PPR query returns an estimated PPR $\widetilde{\pi}_s(v)$ for each vertex $v \in V$, such that the following equations hold:

$$|\pi_s(v) - \widetilde{\pi}_s(v)| \leq \epsilon \cdot \pi_s(v) \quad \forall \pi_s(v) \geq \delta; \tag{1}$$

$$|\pi_s(v) - \widetilde{\pi}_s(v)| \leq \epsilon \cdot \delta \quad \forall \pi_s(v) < \delta. \tag{2}$$

with at least $1 - p_f$ probability. □

Besides, many applications, e.g. recommender system, may not require all PPR values with respect to s. Instead, they only need to return the k vertices with the top-k highest PPR values with respect to s. Such queries are denoted as the single-source top-k PPR queries, or simply top-k PPR queries. Again, returning the exact top-k answers may incur high computational costs and existing studies focus on approximate top-k queries [39] defined as follows.

Definition 2.2. ((ϵ, δ)-approximate top-k PPR) Given a source vertex s, a threshold δ, an error bound $\epsilon \in (0, 1]$, a failure probability p_f, and a positive integer k, an approximate top-k PPR query returns a sequence of k vertices, $v_1, v_2, \ldots, v_k$, such that for any $i \in [1, k]$ with $\pi_s(v_i^*) \geq \delta$, the following equations hold with at least $1 - p_f$ probability:

$$|\pi_s(v_i) - \widetilde{\pi}_s(v_i)| \leq \epsilon \cdot \pi_s(v_i); \tag{3}$$

[1]We treat an undirected graph as a special case of a directed graph where each undirected edge (u, v) corresponds to two directed edges $\langle u, v \rangle$ and $\langle v, u \rangle$

Algorithm 1: Forward-Push

Input: Graph G, source node s, termination probability α, residue threshold r_{max}
Output: $\hat{\pi}_s(v)$, $r_s(v)$ for all $v \in V$

1 $r_s(s) \leftarrow 1; r_s(v) \leftarrow 0$ for all $v \neq s$;
2 $\hat{\pi}_s(v) \leftarrow 0$ for all v;
3 **while** $\exists v \in V$ *such that* $r_s(v)/|N_{out}(v)| > r_{max}$ **do**
4 **for** *each* $u \in \mathcal{N}^{out}(v)$ **do**
5 $r_s(u) \leftarrow r_s(u) + (1-\alpha) \cdot \frac{r_s(v)}{|N_{out}(v)|}$
6 **end**
7 $\hat{\pi}_s(v) \leftarrow \hat{\pi}_s(v) + \alpha \cdot r_s(v)$;
8 $r_s(v) \leftarrow 0$;
9 **end**

$$\pi_s(v_i) \geq (1-\epsilon) \cdot \pi_s(v_i^*), \tag{4}$$

where v_i^* has the i-th largest exact PPR score with respect to s. □

In Definition 2.2, the Equation 3 ensures the accuracy of the estimated PPR values and Equation 4 guarantees that the i-th estimated PPR value returned will be close to the exact i-th largest PPR score. In this paper, we focus on the estimation with high quality, thus we fix $\delta = 1/n$ and $p_f = 1/n$, where n is the number of vertices in G. That is, we provide approximation guarantees for above-average PPR values with high probability.

Table 1 lists the frequently used notations in the paper.

2.2 Distributed Computing

As the scale of graphs becomes more and more massive, many IT companies need to handle huge graphs that cannot be fitted into the memory of most commodity servers. In such scenarios, single-machine algorithms no longer work.

Therefore, we utilize the distributed computing to handle the PPR problem for massive graphs. In particular, we implement our method with Spark [45], a distributed computing framework based on Hadoop [9], whose fundamental computing unit is MapReduce. A MapReduce algorithm proceeds in three phases: *(i) map* phase takes as input the data and emits key-value pairs by the map function, *(ii) shuffle* phase distributes the key-value pairs produced in the map phase and ensures pairs with the same key will be delivered to the same executor, *(iii) reduce* phase aggregates the key-value pairs by the key and then applies the reduce function on the aggregated data to result in new key-value pairs.

Furthermore, we theoretically analyze our proposal and existing solutions under the *Massively Parallel Computing* (MPC) model [3, 7, 14, 21]. MPC is an original model of computation for MapReduce which gives more local computation power (in principle unbounded) compared with other distributed computing models. It appraises an algorithm from two aspects: *(i)* the *round* is the number of times the executors communicate with each other and compute locally, indicates the number of phases during MapReduce job proceeding, *(ii)* the *load* represents the maximum number of messages a executor receives, proceeds and sends in each round.

For graph problems with number n of vertices and number m of edges, the input size is m so that the total space under the MPC model is usually set to be $O(poly(m))$. Without loss of generality, we assume that $m = \Theta(n^{1+\lambda})$ where $0 \leq \lambda \leq 1$ and bound the total space of the cluster (also the number of messages will be transferred during each round) by $O(m) = O(n^{1+\lambda})$.

To partition the input graph, we adopted the Longest Processing Time (LPT) scheduling algorithm [42], which is originally designed for balanced scheduling on parallel machines. The partitioning result with LTP algorithm provides strong theoretical guarantees on the load. In particular, it guarantees that the maximum number of edges a partition holds will be no greater than 1.5 times of the optimal vertex partition scheme which will be $O(m/p)$ if no vertex with a degree (in degree + out degree) above m/p. In case we have a node v with degree above m/p, we divide the edges incident to v to several disjoint subset and each subset has no more than m/p edges. By this way, the adopted LPT partition algorithm still guarantees that each executor stores $O(m/p)$ edges so that we can bound the load by $O(m/p)$ for the cluster which consists of p executors.

2.3 Existing Solutions Revisited

FORA. FORA [38] is a solution for approximate single-source and top-k PPR query on single-machines. It consists of two phases: the *Forward-Push* [2] and the *Random Walk* [10].

Algorithm 1 shows the pseudo-code of the Forward-Push algorithm. The Forward-Push phase takes as input G, a source vertex s, a termination probability α, and a threshold r_{max}. For each vertex v, it maintains a reserve $\hat{\pi}_s(v)$ and a residue $r_s(v)$. Intuitively, the reserve indicates the portion of the random walks that have stopped at vertex v and $r_s(v)$ indicates the portion of the random walks that currently stay at vertex v but have not stopped yet. If residues of all vertices are zero, then reserve $\hat{\pi}_s(v)$ is exactly the PPR value $\pi_s(v)$.

In the beginning of the algorithm, the Forward-Push sets $r_s(s) = 1$ and $r_s(v) = 0$ for all $v \in V \setminus \{s\}$; $\hat{\pi}_s(v) = 0$ for all v (Algorithm 1 Lines 1-2), indicating that all random walks currently stay at node s. Then, the Forward-Push applies a push operation to a vertex v such that it converts $\alpha \cdot r_s(v)$ to the reserve of node v, since among $r_s(v)$ of the random walks $\alpha \cdot r_s(v)$ of them will stop at node v. Then, for the remaining $(1-\alpha) \cdot r_s(v)$, it propagates $(1-\alpha) \cdot r_s(v)/|N_{out}(v)|$ to each out-neighbor of v. Finally, it clears the residue of itself (Algorithm 1 Lines 4-8). If the residues of all nodes are depleted, then the exact PPR values are derived. However, this incurs enormous computational costs. Hence, in Forward-Push, it imposes a threshold r_{max} and only pushes the nodes whose residue is no smaller than r_{max} times its out-degrees. By this strategy, the computational cost can be bounded by $O(1/r_{max})$. The Forward-Push preserves the following invariant after each push operation.

$$\pi_s(t) = \hat{\pi}_s(t) + \sum_{v \in V} r_s(v) \cdot \pi_v(t) \tag{5}$$

However, $\pi_v(t)$ is still unknown and in FORA, it further includes a Random Walk phase to sample $\omega_v = \left\lceil r_s(v) \cdot \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2 \delta} \right\rceil$ random walks from each $v \in V$. Then, it uses the fraction of random walks among the ω_v random walks stopped at t as an estimation $\widetilde{\pi}_v(t)$ of $\pi_v(t)$. Finally, FORA takes each estimation $\widetilde{\pi}_v(t)$ into Equation 5 and outputs an unbiased estimation of $\pi_s(v)$ for each $v \in V$ satisfying Definition 2.1. FORA works efficiently on

single-machines but does not suit to the distributed computing well and we will show that in Section 3.1.

Power Method. The Power method [30] is widely used in existing distributed graph processing systems such as Spark GraphX [13] as a solution for PPR problems. Power method calculate the PPR directly based on another form of definition of PPR, where

$$\boldsymbol{\pi_s} = \alpha \cdot \boldsymbol{e_s} + (1-\alpha) \cdot \boldsymbol{\pi_s} \cdot D^{-1}A. \tag{6}$$

Here, $\boldsymbol{\pi_s}$ is a vector whose j-th element equals $\pi_s(v_j)$, $A \in \{0,1\}^{n\times n}$ is the adjacency matrix of G, $D \in R^{n\times n}$ is a diagonal matrix in which each i-th element on its main diagonal equals the out-degree of v_i, and e_s is an one-hot vector with the position at s to be 1. It starts with $\pi_s^0 = e_s$ and calculates $\pi_s^i = \alpha e_s + (1-\alpha) \cdot \boldsymbol{\pi_s}^{i-1} \cdot D^{-1}A$ iteratively. With i iterations, the L_1 error between $\boldsymbol{\pi}_s^i$ and $\boldsymbol{\pi}_s$ can be bounded by $(1-\alpha)^i$. To answer the approximate queries, we set l such that $(1-\alpha)^l \leq \epsilon\delta$ to provide the approximation guarantee.

Monte-Carlo. Many existing solutions for PPR on distributed system are based on the Monte-Carlo method [10]. Given a source node s, a Monte-Carlo based algorithm generates ω random walks from s, and use the fraction of random walks $f_s(v)$ that terminate at v as an estimation of the PPR $\pi_s(v)$ of v with respect to s. According to [10], the Monte-Carlo based algorithm satisfies Definition 2.1 with a sufficiently large number $\omega = \left\lceil \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2\delta} \right\rceil$ of random walks. The first efficient Monte-Carlo based algorithm on distributed system is the *Doubling* algorithm proposed by Bahmani et al. [6] which merges the segments of short random walks to increase the length of random walks doubly. However, it wastes computational resources when the merge method runs several times and remains just a few number of long walks.

A more optimized Monte-Carlo based algorithm is *DistPPR* proposed by Lin [24] which exploits the parallel pipeline framework to improve the performance. However, the pipeline framework also brings a risk of failure under a strict workload constraint and the extremely large amount of random walks is still the bottleneck of all the Monte-Carlo based algorithms. Moreover, DistPPR cannot handle the approximate top-k queries satisfying Definition 2.2 without increasing the asymptotic time complexity. This is because that to check an interim result within DistPPR, it needs to gather all the random walks pushed into the pipeline before the checkpoint, making the amortized analysis for pipeline framework ineffective.

3 SINGLE-SOURCE PPR QUERY ALGORITHM

3.1 Main Idea

As described in Section 2.3, the Monte-Carlo method is inefficient due to a large number of random walks required to satisfy the approximation guarantee. Furthermore, in distributed frameworks, the Monte-Carlo method can hardly measure the workload of each vertex or edge and thus makes it impossible to design a load balanced partition scheme in some situation.

The index-free FORA [38] method works well on single-machines by reducing the number of required random walks with the Forward-Push method. However, the distributed computing model is so much different from the single-machine setting that it *(i)* will waste massive computational resources by partial pushing results $O(1/r_{max})$ rounds in worst case, *(ii)* it will be challenging to design efficient

Algorithm 2: Pre-Sample(G, α)

Input: Graph $G = (V, E)$, termination probability α
Output: Pre-sampled random walks

1 $\omega_p \leftarrow \lfloor m/n \rfloor$;
2 $W_t \leftarrow \emptyset$;
3 **for** $1 \ldots p$ **do**
4 **foreach** $v \in V$ **in parallel do**
5 $W_a^0(v) \leftarrow \emptyset$;
6 **for** $1 \ldots \lceil \omega_p/p \rceil$ **do**
7 $W_a^0(v) \leftarrow W_a^0(v) \cup [\langle v, v\rangle]$;
8 **end**
9 **end**
10 $W_a \leftarrow \bigcup_{v\in V} W_a^0(v)$;
11 **while** $W_a \neq \emptyset$ **do**
12 $W_t^* \leftarrow W_a$ ***filter***$\{\ p \mapsto (\ q \xleftarrow{\$} [0,1)\ ) < \alpha\ \}$;
13 $W_t \leftarrow W_t \cup W_t^*$;
14 $W_a \leftarrow W_a \setminus W_t^*$;
15 $W_a \leftarrow W_a$ ***map-value***$\{\ v \mapsto u \xleftarrow{\$} N_{out}(v)\ \}$;
16 **end**
17 **end**
18 **return** W_t;

algorithms that samples the random walks dynamically whilst fulfilling the $O(m/p)$ load constraint.

We draw on the experience from FORA+ which combines the Forward-Push and Monte-Carlo methods to reduce the number of random walks, but ponder the problem from the opposite perspective. The proposed Delta-Push method was come up with the consideration to do the push method globally and reduce the rounds of push by pre-sampling a reasonable amount of random walks. In the meantime, the main concern now is how to reduce the number of rounds under the load constraint. Next, we elaborate on the algorithm details of our Delta-Push.

Algorithm details. Delta-Push includes a pre-sample phase to sample $\omega_p = \lfloor m/n \rfloor$ random walks from each node. Algorithm 2 shows the pseudo-code of the pre-sample phase [2]. Basically, it issues $\lceil \omega_p/p \rceil$ random walks from each vertex v (Algorithm 2 Lines 4-9) in each iteration and then simulates the random walk in the distributed setting. In particular, we maintain a key-value pair $p = \langle s, v\rangle$ to represent a random walk from s currently staying at vertex v. Then, for each random walk that has not stopped yet, i.e., each key-value pair p, it randomly samples a random number q. If $q < \alpha$, then we mark the random walk as a terminated walk, and collect all terminated walks to the result set W_t (Algorithm 2 Lines 12-13). Otherwise, the random walk randomly jumps to one of the out-neighbors and updates the key-value pair correspondingly (Algorithm 2 Lines 15-16). The process repeats until all the random walks have stopped. We repeat this process p times where p is the number of processor in the cluster, and finally we have at least $\lfloor m/n \rfloor$ sampled random walks. This finishes the pre-sample phase. Notice that the pre-sample can

[2] $x \xleftarrow{\$} S$ means x is selected from S randomly with uniform distribution

Algorithm 3: Global-Push($G, \alpha, r_s, \hat{\pi}_s$)

Input: Graph $G = (V, E)$, termination probability α, residues r_s, reserves $\hat{\pi}_s$

Output: The maximum of residues r'_{max}, residues r'_s, reserves $\hat{\pi}'_s$

1 $\pi^*_s \leftarrow r$ ***map-value***$\{ r_s(v) \mapsto \alpha \cdot r_s(v) \}$;
2 **foreach** $\langle v, r_s(v) \rangle \in r_s$ **in parallel do**
3 $\quad r^*_s(v) \leftarrow \left[\left\langle u, (1-\alpha) \cdot \frac{r_s(v)}{|N_{out}(v)|} \right\rangle \;\middle|\; u \in N_{out}(v) \right]$;
4 **end**
5 $r'_s \leftarrow (\bigcup_{v \in V} r^*_s(v))$ ***reduce-by-key***$\{+\}$;
6 $\hat{\pi}'_s \leftarrow (\hat{\pi}'_s \cup \pi^*_s)$ ***reduce-by-key***$\{+\}$;
7 $r'_{max} \leftarrow r'_s$ ***reduce-value***$\{ max \}$;
8 **return** $r'_{max}, r', \hat{\pi}'$;

be then used by all single-source/top-k PPR queries from any source s since they are independent with respect to this source s.

Next, for a single-source PPR query from source s, a redesigned push algorithm is adopted by Delta-Push to make full use of the local computational power. To explain, local computations are now no longer the major bottleneck if the time to complete a local task varies as a polynomial function on the size of the input. Therefore, it is desired to make full use of the local computations so as to reduce the number of rounds and communication costs if possible. Our later analysis gives an affirmative answer to this and we will elaborate on it later in Section 5. The pseudo-code of our redesigned push algorithm *Global-Push* is shown in Algorithm 3. Instead of pushing the vertices with a residue larger than r_{max} times its out-degree (Algorithm 1 Line 3), the Global-Push simply push on all vertices with non-zero residue in parallel (Algorithm 3 Lines 2-4). Then, it reduces the reserve and residue values according to the key (the ID of each vertex).

The pseudo-code of Delta-Push is shown in Algorithm 4. Notice that our Delta-Push takes a set of queries as the input. This is expected since in real-life applications we usually need to handle a set Q of single-source/top-k queries instead of a single query. Initially, Delta-Push loads pre-sampled random walks generated by Algorithm 2 (Algorithm 4 Line 1). Next, Delta-Push processes each query in Q: For each source s, it first appends a key-value pair $\langle s, 1 \rangle$ to r (Algorithm 4 Line 5), indicating that we are pushing from source s with full residue 1. Then, it iteratively invokes the Global-Push until the maximum of the residues of all vertices with respect to s is smaller than $\omega_p/\omega_{\epsilon,\delta}$, where ω_p is the number of random walks pre-sampled at each vertex and $\omega_{\epsilon,\delta}$ is the number of random walks required by the pure Monte-Carlo method to fulfil Definition 2.1 (Algorithm 4 Lines 7-10). When the Global-Push ends, it combines the results with pre-sampled results to derive the PPR estimation of each node v with respect to s. Specifically, for a given source s, we compute an estimation of $\pi_s(t)$ (denoted as $\widetilde{\pi}_s(t)$) with reserve $\hat{\pi}_s(t)$ and a combination of residues $r_s(v)$ and a rough approximation $\pi'_v(t)$ for each node v (Algorithm 5 Lines 1-6):

$$\widetilde{\pi}_s(t) = \hat{\pi}_s(t) + \sum_{v \in V} r_s(v) \cdot \pi'_v(t). \tag{7}$$

Here, $\pi'_v(t)$ is derived based on a portion of the pre-sampled random walks. More precisely, we sample $\omega_v = \lceil r_s(v) \cdot \omega_p / r_{sum} \rceil$ random

Algorithm 4: Delta-Push(G, Q, α, ϵ)

Input: Graph $G = (V, E)$, queries Q, termination probability α, relative accuracy guarantee ϵ

Output: Approximate personalized Page Ranks $\widetilde{\pi}$

1 Load pre-sampled random walks generated by Algorithm 2, and group them by sources (denoted as $W_t(v)$);
2 $\delta \leftarrow 1/n$; $p_f \leftarrow 1/n$;
3 $\omega_p \leftarrow p \cdot \left\lceil \frac{\lfloor m/n \rfloor}{p} \right\rceil$; $\omega_{\epsilon,\delta} \leftarrow \left\lceil \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2 \delta} \right\rceil$;
4 **foreach** $s \in Q$ **do**
5 $\quad r_s \leftarrow [\langle s, 1 \rangle]$; $\hat{\pi}_s \leftarrow \emptyset$;
6 $\quad r_{sum} \leftarrow 1$; $r_{max} \leftarrow 1$;
7 $\quad$ **while** $r_{max} > \omega_p/\omega_{\epsilon,\delta}$ **do**
8 $\quad\quad r_{sum} \leftarrow (1-\alpha) r_{sum}$;
9 $\quad\quad (r_{max}, r_s, \hat{\pi}_s) \leftarrow$ Global-Push$(G, \alpha, r_s, \hat{\pi}_s)$;
10 $\quad$ **end**
11 $\quad$ **output** Combine$(r_{sum}, r_s, \hat{\pi}_s, \omega_p, W_t)$;
12 **end**

walks from pre-stored random walks starting from v. Then, if among the ω_v random walks, x_t of them stops and t, $\pi'_v(t)$ is estimated as x_t/ω_v. This finishes one estimation of the single-source PPR. It then turns to another query until all the queries in Q are processed.

Approximation guarantee. The remaining question is how to determine r_{max} so that Algorithm 4 returns query answers satisfying Definition 2.1. Consider the bound of the total space we described in Section 2.2 which is $O(m)$, so we can derive the number of pre-sampled walks for each vertex is $\omega_p = \lfloor m/n \rfloor$, and $\omega_{\epsilon,\delta}$ is the number of random walks required by the pure Monte-Carlo method to provide approximation guarantee which is $\left\lceil \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2 \delta} \right\rceil$.

Theorem 3.1. *For any vertex $v \in V$, Delta-Push returns an approximate PPR $\widetilde{\pi}_s(v)$ satisfying Definition 2.1 if $r_{max} \leq \omega_p/\omega_{\epsilon,\delta}$, where r_{max} is the maximum of residues $r_s(v)$ for all $v \in V$.*

Proof. Consider the ω_v random walks generated from vertex v. Let $X^i_v(t)$ be a Bernoulli variable that takes value 1 if the i-th random walk terminates at t, and value 0 otherwise. By definition:

$$E[X^i_v(t)] = \pi(v, t)$$

If a random walk ends at a vertex t, then our method increases $\widetilde{\pi}_s(v)$ by $r_s(v)/\omega_v$ (denoted as μ_v), and we have:

$$E\left[\sum_{i=1}^{\omega_v} (\mu_v \cdot X^i_v(t)) \right] = r_s(v) \cdot \pi_v(t)$$

Observing that $\sum_{i=1}^{\omega_v} (\mu_v \cdot X^i_v(t))$ is exactly the amount of increment that $\widetilde{\pi}_s(v)$ receives from v, we denote this increment as ψ_v. Then:

$$E\left[\sum_{v \in V} (\psi_v) \right] = \sum_{v \in V} r_s(v) \cdot \pi_v(t)$$

Besides, we have following conclusion derived from FORA [38]:

Lemma 3.2. *The combination of $\hat{\pi}_s(v)$ and $\sum_{v \in V}(\psi_v)$ gives an approximated PPR $\widetilde{\pi}_s(v)$ that satisfies Definition 2.1, with the number of random walks $\omega_v = r_s(v) \cdot \omega_{\epsilon,\delta}$ for each vertex $v \in V$.*

Algorithm 5: Combine($r_{sum}, r_s, \hat{\pi}_s, \omega_p, W_t$)

Input: Sum of residues r_{sum}, residues r_s, reserves $\hat{\pi}_s$, samples count ω_p, pre-sampled walks W_t
Output: Approximate PPR $\widetilde{\pi}_s$

1 **foreach** $\langle v, r_s(v)\rangle \in r_s$ **in parallel do**
2 $\quad \omega_v \leftarrow \left\lceil \frac{r_s(v)}{r_{sum}} \cdot \omega_p \right\rceil$;
3 $\quad \widetilde{\pi}^r_s(v) \leftarrow \left[\left\langle t_i \xleftarrow{\$} W_t(v), \frac{r_s(v)}{\omega_v} \right\rangle \middle| \; i \in 1 \dots \omega_v \right]$;
4 **end**
5 $\widetilde{\pi}^*_s \leftarrow \bigcup_{v \in V} \widetilde{\pi}^r_s(v)$;
6 $\widetilde{\pi}_s \leftarrow (\hat{\pi}_s \cup \widetilde{\pi}^*_s)$ ***reduce-by-key***$\{+\}$;
7 **return** $\widetilde{\pi}$;

Recall that now $r_{max} \le \omega_p/\omega_{\epsilon,\delta}$. Hence $r_{max} \cdot \omega_{\epsilon,\delta} \le \omega_p$, indicating that for an arbitrary node v, it has pre-stored a sufficient number of random walks to be used in Delta-Push. Proof done. □

Theorem 3.3. *Algorithm 4 completes in* $O(\log \frac{n^2 \log n}{\epsilon^2 m})$ *rounds.*

Proof. We first note that the Global-Push algorithm and the Combine method only take $O(1)$ round. In the main loop (Lines 4-11) of Algorithm 4, we consider the number of iterations for each query. Notice that with each invocation of the Global-Push algorithm, r_{sum} will decrease by the rate of $1-\alpha$ and r_{max} is obviously no greater than r_{sum}. The main loop repeatedly invokes the Global-Push until $r_{max} \le \omega_p/\omega_{\epsilon,\delta}$. Note that $r_{max} \le r_{sum}$ by their definition and the worst case which will take the most round to complete is that r_{max} keeps being equal to r_{sum}. Recap that $\omega_{\epsilon,\delta} = \left\lceil \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2\delta} \right\rceil$, $\omega_p = \lfloor m/n \rfloor$, $\delta = 1/n$ and $p_f = 1/n$, the number of times the Global-Push get invoked is bounded by:

$$O\left(\log_{1-\alpha} \frac{\omega_p}{\omega_{\epsilon,\delta}}\right) = O\left(\log \frac{\omega_{\epsilon,\delta}}{\omega_p}\right) = O\left(\log\left(\frac{\log n}{\epsilon^2/n} \cdot \frac{n}{m}\right)\right) = O\left(\log \frac{n^2 \log n}{\epsilon^2 m}\right)$$

This finishes the proof. □

It is easy to see that the memory cost of Algorithm 4 is $O(m)$ as there will be at most one share of residues transferred across each edge in Algorithm 3, and the pre-sampled random walks used in Algorithm 5 is $O(n \cdot \omega_p) = O(n \cdot \lfloor m/n \rfloor) = O(m)$. We just use the conclusion of total memory for introducing our solution, and we will give the analysis of load under the MPC model in Section 5.

3.2 Processing with Batches

Assume that we extend the cluster so that the number of executors is γ times that of the original cluster. Then there are $p' = \gamma \cdot p$ executors and $O(\gamma \cdot m)$ space in total. We can apply a minor modification to Algorithms 3-5 which changes the *key* of reserves $\hat{\pi}$ and residues r from only the staying vertex v to a pair of source and staying vertex $\langle s, v\rangle$ (So a residue will be $\langle\langle s, v\rangle, r_s(v)\rangle$ and reserves are similar). Then we append γ sources $s \in Q$ as $\langle\langle s, s\rangle, 1\rangle$ into the initial residues in parallel instead of appending one source (Algorithm 4 Line 5).

Algorithm 6: Batched Main Loop of Delta-Push

1 $r_{sum} \leftarrow 1$; $r_{max} \leftarrow S$ ***map***$\{ s \to \langle s, 1\rangle \}$;
2 $r \leftarrow S$ ***map***$\{ s \to \langle\langle s, s\rangle, 1\rangle \}$; $\hat{\pi} \leftarrow \emptyset$;
3 **while** $S \neq \emptyset$ **do**
4 $\quad$ **if** $r_{sum} \le m/\omega_{\epsilon,\delta}$ **then**
5 $\quad\quad r^\circ \leftarrow r_{max}$ ***filter***$\{ \langle s, r_s\rangle \mapsto r_s \le \omega_p/\omega_{\epsilon,\delta} \}$;
6 $\quad\quad S_f \leftarrow$ ***get-keys***$\{ r^\circ \}$;
7 $\quad\quad r_f \leftarrow r$ ***filter***$\{ \langle\langle s, v\rangle, r_s(v)\rangle \mapsto s \in S_f \}$;
8 $\quad\quad \hat{\pi}_f \leftarrow \hat{\pi}$ ***filter***$\{ \langle\langle s, t\rangle, \hat{\pi}_s(t)\rangle \mapsto s \in S_f \}$;
9 $\quad\quad S \leftarrow S \setminus S_f$; $r \leftarrow r \setminus r_f$; $\hat{\pi} \leftarrow \hat{\pi} \setminus \hat{\pi}_f$;
10 $\quad\quad$ **output** Combine($r_{sum}, r_f, \hat{\pi}_f, \omega_p, W_t$);
11 $\quad$ **end**
12 $\quad r_{sum} \leftarrow (1-\alpha) r_{sum}$;
13 $\quad (r_{max}, r_s, \hat{\pi}_s) \leftarrow$ Global-Push($G, \alpha, r_s, \hat{\pi}_s$);
14 **end**

In addition, with the increment of total space, we can also store γ times the number of pre-sampled random walks and reduce the push rounds for each batch. In particular, we have $\omega'_p = \gamma \cdot \omega_p$ pre-sampled random walks in the batch method. It is valid that the sets of selected pre-sampled walks to estimate different sources have a non-empty intersection, so we can consider the r_{sum} and r_{max} with respect to each source s individually. By Theorem 3.1, the process for a query completes when $r_{max} \le \omega'_p/\omega_{\epsilon,\delta}$. However, there may exist multiple queries that finish the push phase at the same round in the batch method. Hence, we must additionally bound $r_{sum} \le m/\omega_{\epsilon,\delta}$, to guarantee that the total messages in the combine phase will not exceed $O(\gamma \cdot m)$ space. Note that for any source s, $r^s_{max} \le r^s_{sum}$, and r^s_{sum} for each source s decreases by the same rate $1-\alpha$ in the batch method. By the proof of Theorem 3.3, the number of push rounds for each batch can be bounded by:

$$O\left(\log_{1-\alpha} \frac{\omega'_p}{\omega_{\epsilon,\delta}}\right) = O\left(\log \frac{n^2 \log n}{\gamma \epsilon^2 m}\right)$$

Assume that $\gamma = \Omega(\log n/\epsilon)$, a reasonable scale factor for a real-life cluster. The number of rounds for each batch can be bounded by:

$$O\left(\log \frac{n^2 \log n}{\gamma \epsilon^2 m}\right) = O\left(\log \frac{n^{1-\lambda} \log n}{(\log n/\epsilon) \cdot \epsilon^2}\right) = O\left(\log\left(n^{1-\lambda}/\epsilon\right)\right)$$

where $m = \Omega(n^{1+\lambda})$ according to our assumption in Section 2.2.

Algorithm 6 shows the pseudo-code of the main loop of Delta-Push for batch processing. All parameters are the same as Algorithm 4 except that $\omega_p = \gamma p \cdot \lceil \lfloor m/n \rfloor / p \rceil$ instead of $p \cdot \lceil \lfloor m/n \rfloor / p \rceil$. To maximize the efficiency in practice, PPRs with respect to a source s will be estimated as soon as $r_{sum} \le m/\omega_{\epsilon,\delta}$ and $r_{max} \le \omega_p/\omega_{\epsilon,\delta}$ (Lines 4-11), then the Global-Push is invoked with the reserves and residues with respect to remained sources (Lines 12-13). As described above, this method takes $O(\frac{1}{\gamma} \log \frac{n^2 \log n}{\gamma \epsilon^2 m})$ amortized rounds for each query with $O(\gamma \cdot m)$ space in total.

Algorithm 7: Delta-Push-Top-$k(G, Q, \alpha, \epsilon, k)$

Input: Graph $G = (V, E)$, queries Q, termination probability α, relative accuracy guarantee ϵ, results count k

Output: Approximate top-k personalized Page Ranks $\widetilde{\pi}$

1 Load pre-sampled random walks generated by Algorithm 2, and group them by sources (denoted as $W_t(v)$);
2 $\delta \leftarrow 1/n$; $p_f \leftarrow 1/n$;
3 $\omega_p \leftarrow p \cdot \left\lceil \frac{\lfloor m/n \rfloor}{p} \right\rceil$; $\omega_{\epsilon,\delta} \leftarrow \left\lceil \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2\delta} \right\rceil$;
4 $\epsilon' \leftarrow \epsilon/2$; $p'_f \leftarrow p_f/n$;
5 **foreach** $s \in Q$ **do**
6 $r_s \leftarrow [\langle s, 1\rangle]$; $\hat{\pi}_s \leftarrow \emptyset$;
7 $r_{sum} \leftarrow 1$; $r_{max} \leftarrow 1$;
8 **while** $r_{max} > \omega_p/\omega_{\epsilon,\delta}$ **do**
9 $\delta' \leftarrow r_{max} \cdot \left\lceil \frac{(2\epsilon'/3+2)\log(2/p'_f)}{\epsilon'^2\omega_p} \right\rceil$;
10 $c_{\delta'} \leftarrow \textbf{\textit{count}}\{\textbf{\textit{filter}}\{\langle t, \hat{\pi}_s(t)\rangle \mapsto \hat{\pi}_s(t) \geq \delta'\}\}$;
11 **if** $c'_\delta \geq k$ **then**
12 break;
13 **end**
14 $r_{sum} \leftarrow (1-\alpha) r_{sum}$;
15 $(r_{max}, r_s, \hat{\pi}_s) \leftarrow$ Global-Push$(G, \alpha, r_s, \hat{\pi}_s)$;
16 **end**
17 **output** Combine$(r_{sum}, r_s, \hat{\pi}_s, \omega_p, W_t)$;
18 **end**

4 TOP-K PPR QUERY ALGORITHM

An extra advantage of Delta-Push is that it can handle approximate top-k queries without bringing additional overheads when retrieving the top-k answers. To explain, in top-k queries, typically we need to either retrieve the top-k answer or the top-k upper/lower bounds. In either scenarios, it requires that the algorithm goes through a sorting process, incurring additional round overheads. Next, we explain the details of our top-k algorithm and show how to avoid such additional rounds.

To return an approximate top-k query satisfying Definition 2.2, we need to consider the k-th largest PPR value of s (denoted as $\pi_s(v_k)$) and it seems that we can provide a guarantee as Equation 1 described if we set $\delta = \pi_s(v_k)$. However, we have no information about this value, unless we estimated it with high quality guarantees. The naive solution for approximate top-k PPR is to keep $\delta = 1/n$ as the original Delta-Push for single-source queries, which would lead to unnecessary round overheads.

Looking back to Algorithm 4, we push the residues of all vertices with respect to a source s in each iteration, until r_{max}, i.e., the maximum of all residues with respect to source s, is no larger than $\omega_p/\omega_{\epsilon,\delta}$ and thus provide a guarantee with threshold δ. So, is there any quality guarantee when $r_{max} = r' > \omega_p/\omega_{\epsilon,\delta}$? In fact, there is also a guarantee in this case. If we denote $\omega_{\epsilon,\delta'}$ as replacing δ in $\omega_{\epsilon,\delta}$ with δ', it is clear that when $r_{max} = \omega_p/\omega_{\epsilon,\delta'}$, the results will satisfy Definition 2.1 with a threshold δ'.

Based on the feature we mentioned above, we can first come up with a test-and-trial method. In particular, we may change the failure probability of each estimation to $p'_f = p_f/(n \log n)$, and then decide whether the current top-k results are sufficiently accurate with the decrease of the δ' as FORA did for approximate top-k PPR in [38]. By union bound, we can prove that this method can provide results satisfying Definition 2.1. However, for such a basic solution, it requires to combine/gather the push result and the pre-sampled random walks in every iteration and then retrieve the top-k results to see if the answer satisfies Definition 2.2 or not, which incurs both high communication costs and much larger number of rounds.

Instead of adopting the test-and-trial strategy in [38, 39], we propose a significantly simpler method that provides the same guarantee. In this new algorithm, we only gather push results and pre-sampled random walks in the last iteration, which significantly reduces communication costs compared to the above basic solution. Besides, we may require much less number of Global-Push to the source s compared with Algorithm 4, making the algorithm stop earlier than the single-source counterpart and more efficient for top-k queries. Next, we elaborate on the details of our top-k algorithm.

Algorithm details. Algorithm 7 shows the pseudo-code of our proposed top-k algorithm. At the beginning, we set $\epsilon' = \epsilon/2$ and $p'_f = p_f/n$ which we will explain later. The process naturally completes when $r_{max} \leq \omega_p/\omega_{\epsilon,\delta}$ just like the original Delta-Push does. It will satisfy Definition 2.1 with given ϵ for $\delta = 1/n$ and $p_f = 1/n$ so that it satisfies Definition 2.2 as well because we just provide a guarantee for the vertices v_i with $\pi_s(v_i) \geq \delta$ for the approximate top-k PPR problem. In every iteration, our method first checks whether there exist at least k vertices $v \in V$ with reserve $\hat{\pi}_s(v) \geq \delta'$ (Lines 8-13), and δ' is derived by the following equation:

$$r_{max} \cdot \omega_{\epsilon',\delta'} = r_{max} \cdot \left\lceil \frac{(2\epsilon'/3+2)\log(2/p'_f)}{\epsilon'^2\delta'} \right\rceil = \omega_p \qquad (8)$$

The reserve $\hat{\pi}_s(v)$ is a deterministic value as Global-Push is a deterministic algorithm, and in fact, it is a biased estimation of $\pi_s(v)$ which keeps less than or equal to $\pi_s(v)$. Thus if there are at least k vertices $v \in V$ with reserve $\hat{\pi}_s(v) \geq \delta'$, $\pi_s(v) \geq \delta'$ will also hold for these vertices, and the method stops iterating and estimates $\widetilde{\pi}_s(v)$ for all $v \in V$ in this case. Otherwise, by the observation of Equation 8, δ' has a positive linear correlation with r_{max}. Note that even though r_{max} may increase or decrease after a push step, its upper bound r_{sum} monotonically decreases by a rate of $1-\alpha$, and thus δ' tends to decline. It looks like that we push the upper bound of threshold δ' in each iteration so that more vertices v will satisfy $\pi_s(v) \geq \delta'$. This is why we call our method Delta-Push.

It is clear that Algorithm 7 takes the same asymptotic rounds as Algorithm 4 with the same memory constraint because both applying a filtering to $\hat{\pi}_s$ and counting the number of elements (Algorithm 7 Line 10) take $O(1)$ rounds and will not create new data or lead to a shuffling. The top-k method can also apply a batch processing as we discussed in Section 3.2. As we described in Section 3.2, it can complete in $O(\log \frac{n^2 \log n}{\gamma \epsilon^2 m})$ rounds for each batch with $O(\gamma \cdot m)$ space in total. Theoretically, Algorithm 7 imports more calculating steps compared with Algorithm 4 in each iteration so it will work worse in the worst case if there are no k vertices with significant PPRs with respect to s. But we will show that it work well in practical experiments in Section 6.

Correctness. The correctness of Algorithm 7 can be summarized by the following theorem.

Theorem 4.1. *The Delta-Push-Top-k provides ϵ-approximate top-k PPRs satisfying Definition 2.2 with probability $1-p_f$ where $p_f = 1/n$.*

Proof. As we described above, when Algorithm 7 completes with δ', the Definition 2.1 is satisfied when replacing δ with δ' and there exist at least k vertices $v \in V$ with $\pi_s(v) \geq \delta'$. To prove that Equation 3 will be satisfied, we consider the vertex v_i which has the i-th largest estimated PPR $\widetilde{\pi}_s(v_i)$ in the following three cases:

- $\boldsymbol{\pi_s(v_i) \geq \delta'}$: Equation 3 is satisfied since for any $\pi_s(v) \geq \delta'$, $|\pi_s(v) - \widetilde{\pi}_s(v)| \leq \epsilon' \cdot \pi_s(v) = \epsilon \cdot \pi_s(v)/2 \leq \epsilon \cdot \pi_s(v)$.
- $\boldsymbol{\delta' > \pi_s(v_i) \geq \delta' - \epsilon'\delta'}$: By Equation 2, we have that $|\pi_s(v) - \widetilde{\pi}_s(v)| \leq \epsilon' \cdot \delta'$ for any $\pi_s(v) < \delta'$. Recap that $\epsilon' = \epsilon/2$ and $\pi_s(v_i) < \delta$, we have:

  $$|\widetilde{\pi}_s(v_i) - \pi_s(v_i)| \leq \epsilon'\delta' \leq \frac{1}{2}\epsilon\delta'$$

  So, to prove the estimated PPR satisfies Equation 3, we must prove that $\epsilon\delta'/2 \leq \epsilon \cdot \pi_s(v_i)$. Notice that:

  $$\epsilon \cdot \pi_s(v_i) \geq \epsilon \cdot (\delta' - \epsilon'\delta') = \epsilon\delta' - \frac{1}{2}\epsilon^2\delta' \geq \frac{1}{2}\epsilon\delta'$$

  where the last inequality holds due to $\epsilon - \epsilon^2 \geq 0$ for $\epsilon \in (0, 1]$ and $\delta \geq 0$ for their definitions. Thus, $\epsilon \cdot \pi_s(v_i) \geq |\widetilde{\pi}_s(v_i) - \pi_s(v_i)|$ satisfies Equation 3.
- $\boldsymbol{\delta' - \epsilon'\delta' > \pi_s(v_i)}$: In this case, it is impossible for v_i to be a top-k candidate. When the algorithm completes, there are at least k vertices v^* with $\hat{\pi}_s(v^*) \geq \delta'$. By Equation 7 we have $\widetilde{\pi}_s(v^*) \geq \hat{\pi}_s(v^*)$, and for $\widetilde{\pi}_s(v_i)$, by Equation 2 we have that:

  $$\widetilde{\pi}_s(v_i) \leq \pi_s(v_i) + \epsilon'\delta' < \delta'$$

  Recall that there exist at least k vertices v^* with $\widetilde{\pi}_s(v^*) > \widetilde{\pi}_s(v_i)$, so v_i will not become a top-k vertex.

Next, we show that Equation 4 is satisfied for each returned node as well. Denote the actual vertex with i-th largest PPR as v_i^*. We also consider v_i and v_i^* in three cases:

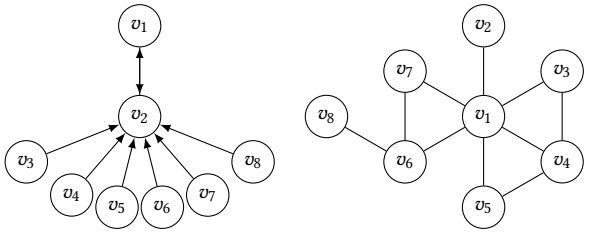
- $\boldsymbol{\pi_s(v_i^*) \leq \pi_s(v_i)}$: Equation 4 is straightforwardly satisfied according to the definition.
- $\boldsymbol{\delta' \leq \pi_s(v_i) < \pi_s(v_i^*)}$: It indicates that there exists some vertex v^* with $\pi_s(v^*) \geq \pi_s(v_i^*)$ that is overtaken by v_i because of the error of estimation. Consider the case $v^* = v_i^*$ and assume that $\pi_s(v_i^*) - \pi_s(v_i) = d$. Then, we know that the estimation of v_i and v_i^* does not exceed their upper and lower error bound with p_f' probability. By Equation 1, the upper bound of $\widetilde{\pi}_s(v_i)$ is $(1 + \epsilon') \cdot \pi_s(v_i)$ and the lower bound of $\widetilde{\pi}_s(v_i^*)$ is $(1 - \epsilon') \cdot \pi_s(v_i^*)$. If v_i overtakes v_i^*, we have $\widetilde{\pi}_s(v_i) > \widetilde{\pi}_s(v_i^*)$, therefore:

  $$d < \epsilon' \cdot \pi_s(v_i) + \epsilon' \cdot \pi_s(v_i^*) < 2\epsilon' \cdot \pi_s(v_i^*)$$

  Thus, we have:

  $$\pi_s(v_i) = \pi_s(v_i^*) - d > (1 - 2\epsilon') \cdot \pi_s(v_i^*) = (1 - \epsilon) \cdot \pi_s(v_i^*)$$

  It is the necessary condition that v_i can take place with the i-th largest estimated value because if $\pi_s(v_i)$ is not large enough to overtake v_i^*, then it is impossible that v_i can overtake other v^* which with $\pi_s(v^*) > \pi_s(v_i^*)$ unless the upper bound or lower bound is exceeded with at most $p_f' = 1/n^2$ probability. By union bound, we have the fail probability to be $p_f = 1/n$.



(a) The worst case of directed graph (b) An undirected graph

Figure 1: Example graphs for load analysis

- $\boldsymbol{\pi_s(v_i) < \delta'}$: We also consider that v_i^* is overtaken by v_i and assume that $\pi_s(v_i^*) - \pi_s(v_i) = d$. By Equation 1, the lower bound of $\widetilde{\pi}_s(v_i^*)$ is $(1-\epsilon') \cdot \pi_s(v_i^*)$ with at least $1-p_f'$ probability. Then, by Equation 2, the upper bound of $\widetilde{\pi}_s(v_i)$ is $\pi_s(v_i) + \epsilon' \cdot \delta'$ with at least $1 - p_f'$ probability. As we described in the second case, $\widetilde{\pi}_s(v_i) > \widetilde{\pi}_s(v_i^*)$ when v_i overtakes v_i^*, we have that:

  $$d < \epsilon' \cdot \delta' + \epsilon' \cdot \pi_s(v_i^*) < 2\epsilon' \cdot \pi_s(v_i^*)$$

  Thus, similar to the second case, we have $\pi_s(v_i) > (1-\epsilon) \cdot \pi_s(v_i^*)$ with at least $1 - p_f$ probability.

  This finishes the proof. □

5 THEORETICAL ANALYSIS

In this section, we will show that our proposed solution asymptotically reduces the communication rounds over existing alternatives with the same load constraint under the MPC model.

Power method. The power method is widely used in existing distributed graph processing systems such as Spark GraphX as the built-in solution for single-source PPR problem. As we analyzed in Section 2.3, the number of rounds can be bounded with $O(\log_{1-\alpha}(\epsilon\delta)) = O(\log n/\epsilon)$. Next, we analyse the load of the Power method. In every round, each entry v of π_s^{i-1} generates $|N_{out}(v)|$ shares and passes them to the target vertices across edges. Therefore there will be at most one share transferred across each edge. As we described in Section 2.2 that when the input graph is well partitioned, each machine handles $O(m/p)$ edges, therefore each machine will send and receive $O(m/p)$ messages which satisfies the load constraint exactly.

Monte-Carlo method. Before we start the analysis for another main competitor DistPPR, there is still an issue that needs to be clarified. Specifically, DistPPR, the best existing distributed algorithm based on Monte-Carlo is originally for the Fully Approximate PPR problem (Definition 2.2 in [24]), so it starts the same number of random walks for all the vertices in each round. This original implementation can hardly collect the sampled walks to result in an estimation of PPR with high precision for each source under a reasonable load constraint. To explain, there will be $O(n^2)$ pairs of PPR if we start estimating for all sources at the same time. For this reason, we consider a variant of the DistPPR for the SSPPR problem (Definition 2.1) to bound the total number of messages to $O(m)$ in each round and we assume that the messages can be evenly partitioned to p parts so that the load of each executor can be bounded as $O(m/p)$. However, this assumption may not hold and we will discuss the worst case later.

Recap that in a pure Monte-Carlo based method, it needs to sample $\omega_{\epsilon,\delta} = O(\frac{\log(1/p_f)}{\epsilon^2\delta})$ random walks for each query to provide an approximate answer satisfying Definition 2.1. For each query, we let the variant method append $\lfloor \alpha \cdot m \rfloor$ random walks into the pipeline in each iteration to guarantee that the workload of the pipeline will be $O(m)$ in expectation. We have that the variant DistPPR for SSPPR problem requires $O(\omega_{\epsilon,\delta}/(\alpha \cdot m))$ rounds to start all $\omega_{\epsilon,\delta}$ random walks. After pushing all random walks into the pipeline, there will still remain $O(m)$ active random walks in pipeline in expectation. It takes $O(\log m)$ rounds which is necessary for the guarantee of failure probability by the Chernoff bound as we will derive in Lemma 5.1, to complete remained random walks.

Recall that $\delta = 1/n$ and $p_f = 1/n$. The number of rounds required by DistPPR to solve the SSPPR defined as Definition 2.1 is:

$$O\left(\frac{\omega_{\epsilon,\delta}}{\alpha \cdot m} + \log m\right) = O\left(\frac{n \log n}{\epsilon^2 m} + \log m\right)$$

Moreover, the asymptotic cost of rounds for DistPPR is an optimistic estimation because we utilize an assumption of load balance to guarantee the load constraint but it is impossible in some situation. The first case is Figure 1(a). When we estimate PPR with respect to a source s, DistPPR appends $\lfloor \alpha \cdot m \rfloor = O(m)$ random walks starting at s in each iteration, and each vertex in Figure 1(a) has only one out-edge. Thus only one executor who handles this out-edge of s must process all random walks which are just appended in this iteration. In the meantime, random walks starting from any source s will reach v_2 by at most one step in Figure 1(a), and then they will pass through $\langle v_2, v_1 \rangle$ or $\langle v_1, v_2 \rangle$ repeatedly. Thus at most two executors who handle these two edges respectively must process all random walks which are appended before this iteration. Therefore, at most three executors are busy to process $O(m)$ messages and the other executors are idle in this case. Even worse, the Monte-Carlo method not only takes bad load balance under extreme conditions as Figure 1(a), but also works badly in some common situations. Consider Figure 1(b) which is a common undirected graph in practice. The vertices v_2 in Figure 1(b) has only one edge. As we describe above, only one executor which handles the edge must process the first step of all random walks and the load will be $O(m)$. Therefore, it is impossible for DistPPR to keep load balance unless an arbitrary vertex in the graph has $\Omega(p)$ out-edges based on the above analysis.

Delta-Push. Recap that our Delta-Push includes a pre-sample phase and a global-push phase. We first analyze the number of rounds for the pre-sample phase and have the following lemma.

Lemma 5.1. *Algorithm 2 completes in $O(p \log(m/p))$ rounds under $O(m/p)$ load constraint, with at least $1 - 1/n$ probability.*

Proof. In Algorithm 2, we pre-sample $\lceil \lfloor m/n \rfloor / p \rceil$ random walks from all vertices each iteration, so that the number of random walks will be $\omega_a = n \cdot \lceil \lfloor m/n \rfloor / p \rceil = O(m/p)$. First, it must be clarified that the initial phase (Algorithm 2 Lines 4-9) which has a local loop still takes only 1 round under the MPC model because the MPC model allow the executor do almost unlimited local calculation in a round. Now, let X_v^i be the steps of the i-th random walk starting from v until it terminate, then $X \sim Geom(\alpha)$. We have that:

$$E[X_v^i] = \frac{1}{\alpha}$$

Then, we have the following result by Chernoff bound:

$$\begin{aligned} Pr[X_v^i \geq (1 + 4\ln\omega_a)\frac{1}{\alpha}] &\leq \exp\left(-\frac{(4\ln\omega_a)^2}{2 + 4\ln\omega_a} \cdot \frac{1}{\alpha}\right) \\ &\leq \exp\left(-\frac{16(\ln\omega_a)^2}{6\ln\omega_a}\right) < \omega_a^{-2} \end{aligned}$$

Finally, by union bound we have:

$$Pr[Max\{X_v^i\} \geq (1 + 4\ln\omega_a)\frac{1}{\alpha}] < \omega_a^{-1} < \left(\frac{m}{p}\right)^{-1} \leq \left(\frac{n}{p}\right)^{-1} \quad (9)$$

Therefore, each iteration in Algorithm 2 finishes in $O(\log\omega_a) = O(\log(m/p))$ rounds with at least $1 - 1/(n/p)$ probability. Applying union bound to Equation 9 again, Algorithm 2 will complete in $O(p \log(m/p))$ rounds with least $1 - 1/n$ probability.

Pay attention that we only pre-sample $O(m/p)$ random walks in each iteration and it is still under the load constraint in total, so it cannot exceed the load in any situation. In addition, this bound of the number of random walks is asymptotically tight. Consider Figure 1(a), and assume that we start ω' random walks from each vertex in an iteration. There are $\alpha \cdot \omega'$ random walks terminated immediately at their source, and the remained random walks (except which start at v_2) will concentrate on v_2. In the next step, there will be $(n-1) \cdot (1-\alpha) \cdot \omega'$ random walks on v_2 and the $(1-\alpha)$ of them will pass through the edge (v_2, v_1). Thus, to guarantee the executor who handles this edge will not process above $O(m/p)$ random walks, the ω' must be $O((m/p)/n)$. □

Up to now, we have proved that the pre-sample method completes in $O(p \log(m/p))$ rounds. The rounds of pre-sample method will become $O(1)$ in amortization as long as we need to process $O(p \log(m/p))$ queries, which can be easily satisfied. Therefore, the pre-sample cost becomes insignificant compared to the rounds taken by the main loop of Algorithm 4 (Lines 4-12) with numerous queries, such that this cost can be almost ignored in the final analysis. For this phase, we have the following lemma to bound the number of its rounds for each query.

Lemma 5.2. *The main loop (Algorithms 4 Lines 4-12) completes in $O(\log \frac{n^2 \log n}{\epsilon^2 m})$ rounds for each query under $O(m/p)$ load constraint, and gives an approximated PPR that satisfies Definition 2.1.*

Proof. We have proved that the main loop will complete in $O(\log \frac{n^2 \log n}{\epsilon^2 m})$ rounds with $O(m)$ memory cost in total. Now we prove that if the graph is well partitioned, the data stored in the memory and all messages transferred in each round are balanced among partitions. At the beginning of Algorithm 4, we stored pre-sampled random walks. Notice that all vertices have the same number of stored samples. Therefore we can simply store a random walk $\langle s, t \rangle$ which starts at s and terminates at t on the $(s\%p)$-th partition so that each partition will handle the same number of samples.

The load of push phase is similar to the Power method we discussed above. In every round, there will be at most one share of residues transferred across each edge and the load constraint is exactly satisfied. When the Global-Push ends, it further invokes Algorithm 5 to combine the pre-sample and the push results.

As we have described in Section 3.1, Algorithm 5 will use $O(m)$ sampled walks in total for each query to generate $O(m)$ shares of residues and there is one reserve of each terminated vertex t with

respect to s. Then, as we described above, each partition can handle exactly $O(m/p)$ pre-sampled random walks and output $O(m/p)$ shares. Next, the estimation of PPR of t with respect to s will be the combination of some shares of residues and the reserve. Thus, the number of shares to be combined can be bounded as $O(m+n) = O(m)$. In addition, the reserves and the shares of residues in this phase can be simply seen as key-value pairs. Therefore it is easy to apply the reduce method in $O(1)$ rounds with balanced load in existing distributed framework, so that it will not exceed the $O(m/p)$ load constraint for each executor. Proof done. □

Given Lemmas 5.1 and 5.2, we have the following theorem to bound the round to process each query.

Theorem 5.3. *The Delta-Push algorithm provides approximate PPRs satisfying Definition 2.1 with amortized $O(\log(n/\epsilon))$ rounds for each query in Q under the $O(m/p)$ load constraint.*

Proof. Since the pre-sampled random walks can be proceeded once and then used for any subsequent single-source queries. Then, as long as we have at least $p\log(m/p)$ queries (which can be easily satisfied), the amortized cost for the pre-sample phase can be then bounded by $O(1)$. For the Global-Push phase, the number of rounds to process each query can be bounded by $O(\log\frac{n^2\log n}{\epsilon^2 m})$. Recall that we assume $m = \Omega(n^{1+\lambda})$ in Section 2.2, and therefore the amortized round to process one query can be bounded by:

$$O\left(1+\log\frac{n^2\log n}{\epsilon^2 m}\right) = O\left(\log\frac{n^2\log n}{\epsilon^2 m}\right) = O\left(\log\frac{n^{1-\lambda}\log n}{\epsilon^2}\right) = O\left(\log\frac{n}{\epsilon}\right)$$

This finishes the proof. □

In addition, a power method provides the same guarantee with the same asymptotic rounds which is $O(\log_{1-\alpha}(\epsilon\delta)) = O(\log(n/\epsilon))$. However, as we showed in Section 3.2, Delta-Push is more scalable in batch query processing, which takes less rounds when the number of executors increases such that total space of the cluster becomes larger, while the Power method still keeps the same round, which is inferior to our solution.

6 EXPERIMENTS

In this section, we experimentally evaluate our proposed Delta-Push against alternatives. All experiments are conducted on an AWS cluster consisting of 20 processors, each with 16 VCPUs clocked at 2.5GHz and 64GB memory. The network bandwidth among the processors is 10Gbps. We compare our method against *DistPPR* [24], which is the state-of-the-art Monte-Carlo based method and outperforms other methods, e.g. *Doubling* [6]. We further include Power method as our baseline. In addition, we report the query performance of FORA+ for single-source query (denoted as *FORA+*) and the query performance of FORA+ of top-k query (denoted as *FORA+-Top-k*) [38, 39] on a single machine.

Implementation. All source codes are implemented with Scala 2.12 on Spark version 3.0 and Hadoop version 3.2. We partition the input graph with the LPT method as mentioned in Section 2.2 and maintain the graph with Spark GraphX [13]. For our pre-sampling phase, following the pipeline solution in DistPPR, we pipeline the p different iterations to improve the practical performance. However, to bound the load, we will flush the pipeline and redo the task when the load constraint is not satisfied. This method works well and guarantees that load is bounded by $O(m/p)$.

Table 2: Datasets. ($K = 10^3, M = 10^6, B = 10^9$)

Name	n	m	Type
GrQc	5.2K	29.0K	undirected
DBLP	613.6K	2.0M	undirected
Stanford	281.9K	2.3M	directed
Pokec	1.6M	30.6M	directed
LJ	4.8M	69.0M	directed
Orkut	3.1M	117.2M	undirected
Twitter	41.7M	1.5B	directed
Friendster	65.6M	1.8B	undirected

Datasets. To show the scalability of our proposed solution, we test on 8 publicly available datasets: *GrQc*, *DBLP*, *Stanford*, *Pokec*, *LJ*, *Orkut*, *Twitter*, and *Friendster*, with varying graph size from 29 thousand edges to 1.8 billion edges. All datasets except Twitter can be downloaded from Stanford Large Network Dataset Collection [3]. The Twitter dataset can be found from KONECT [22] project [4]. Table 2 shows the details of these 8 datasets.

Parameters. In our experiment, we fix $\delta = 1/n$ and $p_f = 1/n$ to provide guarantees for all above-average PPR values with high probability. By default, we set $\alpha = 0.2$ and $\epsilon = 0.5$ following [24, 39].

6.1 Comparisons with Previous Work

Average query time. In the first set of experiments, we examine the efficiency of our proposed method against alternatives. Following [25], we tune the parameter ϵ so that all methods return the top-500 query with an average precision no smaller than 99.5%. For the ground-truth, we use the Power method and stop until the absolute error is no more than 10^{-20}. In such a case, even if we obtained the exact ground-truth values, as long as we store them in double-precision, the precision loss will be more than 10^{-20}. Hence, the values that we have calculated can be treated as the ground-truth values.

For distributed algorithms, we compare with DistPPR [24] and the Power method using the GraphX Pregel API. We further include the single-source PPR algorithm FORA and run it with a single-core on a single-machine to see the cost-effectiveness. Figure 2 reports the average running time of all methods. Notice that the y-axis is log-scale and we terminate the algorithm if it runs with more than 24 hours or some of executors crash with the Out-Of-Memory(OOM) error. As we can observe, our Delta-Push is orders of magnitude faster than the Monte-Carlo method, which is as expected since it requires far less round complexity than the Monte-Carlo method. In addition, our Delta-Push is 3x to 5x faster than the Power method on most datasets. Remarkably, our Delta-Push-Top-k algorithm is further several times faster than our Delta-Push algorithm, and is up to an order of magnitude faster than the Power

[3] 2021, SNAP. http://snap.stanford.edu/data/index.html
[4] 2021, KONECT. http://konect.cc/

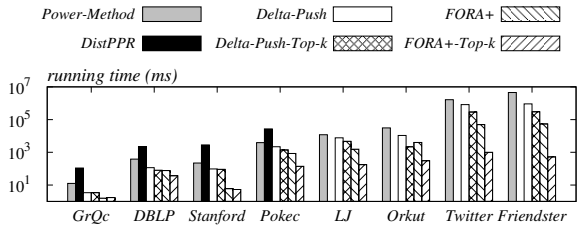


Figure 2: Average running time

method. This demonstrates the effectiveness of our Delta-Push and Delta-Push-Top-k algorithm. Our Delta-Push (resp. Delta-Push-Top-k) is slower than the single-machine FORA+ (resp. FORA+-Top-k), which is expected. The main reason is that Delta-Push and Delta-Push-Top-k need to communicate across multiple executors through the network while the single-machine counterparts do not need. However, we note that the single-machine FORA+ and FORA+-Top-k are limited by the memory capacity of the machine. When the graph size is larger than the memory capacity, FORA+ and FORA+-Top-k no longer work. In contrast, our proposed distributed algorithms can still work by piling up enough machines and running in the distributed environment.

Communication round. In the second set of experiments, we examine the communication round of all distributed algorithms. The results are shown in Figure 3. As we can observe, our Delta-Push-Top-k algorithm requires the least number of rounds among all methods and the number of rounds is orders of magnitude smaller than DistPPR and the Power method. Delta-Push requires the second least number of rounds as it pre-samples random walks and can reuse it during the query processing. The observation in Figure 3 is generally consistent with that in Figure 2, which means the number of rounds is indeed an important indicator of the efficiency for distributed algorithms.

Communication cost. Next, we examine the total communication cost of all distributed algorithms. Our Delta-Push-Top-k achieves the least total communication costs on almost all methods. The main reason is that our top-k algorithm requires only very small rounds, which avoids a lot of communication costs. The Monte-Carlo method actually requires less number of total communication costs than Delta-Push. The reason is that Monte-Carlo method only stores integers while Delta-Push needs to store non-integer values, like residue and reserve. The Power method requires the highest communication cost among all methods. However, the communication cost is not a direct reflection of the efficiency since the performance of a cluster mainly depends on the executor which is the last to complete its task. The Monte-Carlo method is difficult to further scale up because the communication may be quite skewed, resulting in excessively high load as we will show shortly.

Memory peak. Next, we examine the memory peak of all distributed algorithms. The Monte-Carlo method has the highest memory peak while all other methods have similar memory peaks. The main reason is that the graph is well partitioned and the messages get sent/received on each executor is balanced. Therefore, the memory peak on Delta-Push, Delta-Push-Top-k, and Power method are quite similar. However, even if the graph is well partitioned, the

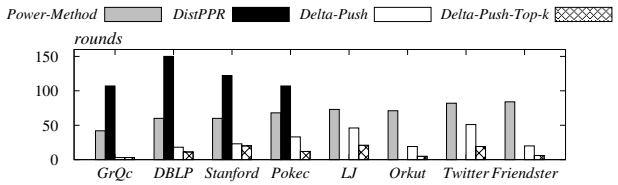


Figure 3: Average rounds

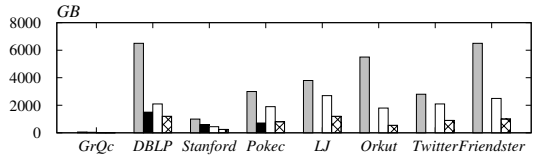


Figure 4: Total communication cost

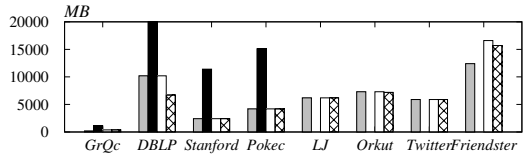


Figure 5: Memory peak

Table 3: Statistics of pre-sample phase.

Name	γ	Pre-sample Time (min)
GrQc	50000	4.8
DBLP	5000	32.6
Stanford	2500	28.4
Pokec	200	22.0
LJ	100	25.2
Orkut	50	44.8
Twitter	5	29.2
Friendster	5	68.4

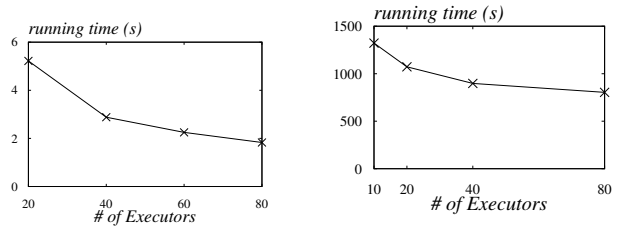
Monte-Carlo method may end up with skewed message distributions on different executors as we analyzed in Section 5, resulting in high memory peaks.

Pre-sampling phase. Table 3 shows the batch size and pre-sample time on each dataset. As we can observe, the number of nodes handled per batch decreases when the size of the input graph increases. To explain, the cluster has a limited memory and bandwidth while γ is a factor to make $\gamma \cdot m$ fit the scale of our cluster. Therefore the batch sizes of different datasets are different since m is different on different datasets. The larger the size of the graph is, the smaller the batch size is. On all datasets, the pre-sample time is moderate and is not a bottleneck as we analyze in Section 5. The cost can be further compensated by the large number of queries.

6.2 Scalability of Delta-Push

In the last set of experiment, we evaluate the scalability of our proposed method. Figure 6(a) reports the average running time to evaluate 200 queries per batch on Pokec with $20 \sim 80$ executors. With an increasing number of executors, it allows more pre-sampled random walks to be stored and we can observe that the average running time keeps decreasing since it reduces the rounds.

Figure 6(b) shows the average running time to evaluate PPR on Friendster with different numbers of executors but with the same



(a) Varying executors on Pokec (b) Varying executors on Friendster

Figure 6: Scalability test

total number of cores and total memory. This compares the settings where we pile up standard machines in contrast with the setting where we upgrade the hardware of the machine, e.g., increasing the memory or number of cores. The leftmost setting is the case when we have 10 machines and each machine has 128GB memory and 32 cores. The rightmost setting is the case when we have 80 machines where each machine has 16GB memory and 4 cores. As we can observe, with a large number of non-powerful machines, the performance is actually better than that with a few powerful servers. To explain, the bottleneck is the bandwidth and the larger number of machines we have, the less number of messages each machine will send or receive, and the more efficient the communication is.

7 RELATED WORK

Single-machine PPR algorithms. Personalized PageRank was first proposed by Page et al. [30]. The first category of solution is matrix-based methods. The classic solution is the power method as described in Section 2.3. However, the power method is usually slow when the graph becomes huge. Some follow-up research work [12, 20, 28, 33, 48] then focuses on improving the matrix multiplication process by decomposing the input graph into tree structures or sub-matrices, and utilizing the decomposition to speed up the PPR queries. However, all such optimization techniques are designed for single-machines and cannot be applied to distributed settings.

There also exist methods that use the local push algorithm to derive the single-source PPR queries (Foward Push [2]) and to derive the single-target PPR queries (Backward Push [1, 19]). Berkin et al. [8] propose to pre-compute the Forward Push resulting from several important nodes, and then use these results to speed up the query performance. Ohsaka et al. [29] further design algorithms to update the stored Forward Push results on dynamic graphs. Jeh et al. [19] propose the backward search algorithm to answer single-target PPR queries. Zhang et al. [46] design algorithms to update the stored forward and backward push results on dynamic graphs. Wang et al. [34] further present randomized backward push to improve the query performance with the same accuracy. However, all such algorithms cannot be applied to distributed settings.

Another branch of research work combines the local push algorithms with Monte-Carlo method to improve the query efficiency. Lofgren et al. [25, 26] and Wang et al. [36, 37] combine random walk and backward push to improve the query performance of pairwise PPR queries. Wang et al. [38, 39] further propose FORA to combine the forward push and random walks to improve the query efficiency. ResAcc [23] accelerates FORA by accumulating the residues that returned to the source node in the forward push phase and "distribute" this residue to other nodes proportionally based on the reserve values prior to the Monte-Carlo phase.

Finally, a plethora of research work [4, 6, 11, 12, 16, 25, 41] study how to efficiently process the top-k PPR queries. Gupta et al. [16] propose to use Forward Push to return the top-k answers. However, their solutions do not provide any approximation guarantee. Avrachenkov et al. [4] study how to use Monte-Carlo approach to find the top-k nodes. Nevertheless, the solution does not return estimated PPR values and does not provide any worst-case assurance. Fujiwara et al. [11, 12] and Shin et al. [33] investigate how to speed up the top-k PPR queries with the matrix decomposition approach. These approaches provide no approximation guarantees. Wei et al. propose the index-free TopPPR [41], which combines Forward Push, random walk, and backward push to answer top-k PPR queries with precision guarantees.

Distributed algorithms. There exist numerous of existing scalable distributed algorithms based on the Monte-Carlo method. Bahmani et al. [5] propose Doubling to merge segments of short random walks and then increase the length of random walks doubly. Lin [24] proposes to further integrate the pipeline to improve the PPR query performance. Sarma et al. [32] and Luo [27] investigate the acceleration of PageRank computation in congestion model. However, they only focus on reducing the communication round for the random walks but discard the load which may significantly affects the query performance. In contrast, in our setting, random walks are pre-sampled and can be easily amortized to the enormous incoming queries. Guo et al. [15] explore the linearity of PPRs and pre-compute partial results for hub vertices on each machine. When a query comes, it makes use of the pre-stored partial results to speed up the query processing. However, such a solution is still not scalable to graphs with billion edges. There are also several existing research works focusing on concurrent PPR [35] or general graph query processing on a single machine, e.g., [31, 43, 44, 47]. These works are orthogonal to our study and can be further applied to optimize our query processing handled on the same executor.

8 CONCLUSIONS

In this paper, we present an efficient distributed framework for single-source and top-k PPR queries. Theoretical analysis shows that our proposed solutions are asymptotically better than alternatives and experiments show that they are faster than competitors.

ACKNOWLEDGMENTS

Sibo Wang is supported by Hong Kong RGC ECS (No. 24203419), RGC CRF (No. C4158-20G), CUHK Direct Grant (No. 4055114), and NSFC (No. U1936205). Zhewei Wei is supported in part by National Natural Science Foundation of China (No. 61972401, No. 61932001 and No. 61832017), by Beijing Outstanding Young Scientist Program NO. BJJWZYJH012019100020098, and by Alibaba Group through Alibaba Innovative Research Program. Zhewei Wei also works at Beijing Key Laboratory of Big Data Management and Analysis Methods, MOE Key Lab of Data Engineering and Knowledge Engineering, and Pazhou Lab, Guangzhou, 510330, China. This work is supported by Public Computing Cloud, Renmin University of China. Sibo Wang and Zhewei Wei are the corresponding authors.

REFERENCES

[1] Reid Andersen, Christian Borgs, Jennifer T. Chayes, John E. Hopcroft, Vahab S. Mirrokni, and Shang-Hua Teng. 2007. Local Computation of PageRank Contributions. In *WAW*. 150–165.

[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.

[3] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel Algorithms for Geometric Graph Problems. In *STOC*. 574–583.

[4] Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, Elena Smirnova, and Marina Sokol. 2011. Quick Detection of Top-k Personalized PageRank Lists. In *WAW*. 50–61.

[5] Lars Backstrom and Jure Leskovec. 2011. Supervised random walks: predicting and recommending links in social networks. In *WSDM*. 635–644.

[6] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. 2011. Fast personalized PageRank on MapReduce. In *SIGMOD*. 973–984.

[7] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication steps for parallel query processing. In *PODS*. 273–284.

[8] Pavel Berkhin. 2005. Survey: A Survey on PageRank Computing. *Internet Mathematics* 2, 1 (2005), 73–120.

[9] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

[10] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[11] Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Takeshi Mishima, and Makoto Onizuka. 2013. Efficient ad-hoc search for personalized PageRank. In *SIGMOD*. 445–456.

[12] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012. Efficient personalized pagerank with accuracy assurance. In *KDD*. 15–23.

[13] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[14] Michael Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, Searching, and Simulation in the MapReduce Framework. In *ISAAC*. 374–383.

[15] Tao Guo, Xin Cao, Gao Cong, Jiaheng Lu, and Xuemin Lin. 2017. Distributed Algorithms on Exact Personalized PageRank. In *SIGMOD*. 479–494.

[16] Manish S. Gupta, Amit Pathak, and Soumen Chakrabarti. 2008. Fast algorithms for topk personalized pagerank queries. In *WWW*. 1225–1226.

[17] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. Wtf: The who to follow service at twitter. In *WWW*. 505–514.

[18] Zoltán Gyöngyi, Pavel Berkhin, Hector Garcia-Molina, and Jan O. Pedersen. 2006. Link Spam Detection Based on Mass Estimation. In *VLDB*. 439–450.

[19] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *WWW*. 271–279.

[20] Jinhong Jung, Namyong Park, Lee Sael, and U Kang. 2017. BePI: Fast and Memory-Efficient Method for Billion-Scale Random Walk with Restart. In *SIGMOD*. 789–804.

[21] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *SODA*. 938–948.

[22] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350.

[23] Dandan Lin, Raymond Chi-Wing Wong, Min Xie, and Victor Junqiu Wei. 2020. Index-Free Approach with Theoretical Guarantee for Efficient Random Walk with Restart Query. In *ICDE*. 913–924.

[24] Wenqing Lin. 2019. Distributed Algorithms for Fully Personalized PageRank on Large Graphs. In *WWW*. 1084–1094.

[25] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2016. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*. 163–172.

[26] Peter A Lofgren, Siddhartha Banerjee, Ashish Goel, and C Seshadhri. 2014. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*. 1436–1445.

[27] Siqiang Luo. 2019. Distributed PageRank Computation: An Improved Theoretical Study. In *AAAI*. 4496–4503.

[28] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. 2014. Computing personalized PageRank quickly by exploiting graph structures. *PVLDB* 7, 12 (2014), 1023–1034.

[29] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. Efficient PageRank Tracking in Evolving Networks. In *SIGKDD*. 875–884.

[30] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).

[31] Peitian Pan and Chao Li. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *ICCD*. 217–224.

[32] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. 2013. Fast Distributed PageRank Computation. In *ICDCN*. 11–26.

[33] Kijung Shin, Jinhong Jung, Lee Sael, and U. Kang. 2015. BEAR: Block Elimination Approach for Random Walk with Restart on Large Graphs. In *SIGMOD*. 1571–1585.

[34] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. 2020. Personalized PageRank to a Target Node, Revisited. In *SIGKDD*. 657–667.

[35] Runhui Wang, Sibo Wang, and Xiaofang Zhou. 2019. Parallelizing approximate single-source personalized PageRank queries on shared memory. *VLDB J.* 28, 6 (2019), 923–940.

[36] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. HubPPR: Effective Indexing for Approximate Personalized PageRank. *PVLDB* 10, 3 (2016), 205–216.

[37] Sibo Wang and Yufei Tao. 2018. Efficient Algorithms for Finding Approximate Heavy Hitters in Personalized PageRanks. In *SIGMOD*. 1113–1127.

[38] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. 2019. Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries. *Trans. Database Syst.* 44, 4 (2019), 18:1–18:37.

[39] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *SIGKDD*. 505–514.

[40] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Yu Liu, Xiaoyong Du, and Ji-Rong Wen. 2019. PRSim: Sublinear Time SimRank Computation on Large Power-Law Graphs. In *SIGMOD*. 1042–1059.

[41] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. 2018. TopPPR: Top-k Personalized PageRank Queries with Precision Guarantees on Large Graphs. In *SIGMOD*. 441–456.

[42] David P Williamson and David B Shmoys. 2011. *The design of approximation algorithms*. Cambridge university press.

[43] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. 2017. Processing Concurrent Graph Analytics with Decoupled Computation Model. *Trans. Computers* 66, 5 (2017), 876–890.

[44] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: an efficient, low-cost system for concurrent graph processing. In *HPDC*. 227–238.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.

[46] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate Personalized PageRank on Dynamic Graphs. In *KDD*. 1315–1324.

[47] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *SC*. 3:1–3:14.

[48] Fanwei Zhu, Yuan Fang, Kevin Chen-Chuan Chang, and Jing Ying. 2013. Incremental and Accuracy-Aware Personalized PageRank through Scheduled Approximation. *PVLDB* 6, 6 (2013), 481–492.