Realtime Top-k Personalized PageRank over Large Graphs on GPUs

Jieming Shi[†], Renchi Yang[*], Tianyuan Jin[‡], Xiaokui Xiao[†], Yin Yang[§]
[†]School of Computing, National University of Singapore, Singapore
[*]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[‡]University of Science and Technology of China, Hefei, China
[§]College of Science and Engineering, Hamad Bin Khalifa University, Qatar
[†]{shijm, xkxiao}@nus.edu.sg, [*]yang0461@e.ntu.edu.sg,
[‡]jty123@mail.ustc.edu.cn, [§]yyang@hbku.edu.qa

ABSTRACT

Given a graph G, a source node $s \in G$ and a positive integer k, a top-k *Personalized PageRank* (*PPR*) query returns the k nodes with the highest PPR values with respect to s, where the PPR of a node v measures its relevance from the perspective of source s. Top-k PPR processing is a fundamental task in many important applications such as web search, social networks, and graph analytics. This paper aims to answer such a query in *realtime*, i.e., within less than 100ms, on an Internet-scale graph with billions of edges. This is far beyond the current state of the art, due to the immense computational cost of processing a PPR query. We achieve this goal with a novel algorithm kPAR, which utilizes the massive parallel processing power of GPUs.

The main challenge in designing a GPU-based PPR algorithm lies in that a GPU is mainly a *parallel computation* device, whereas PPR processing involves graph traversals and value propagation operations, which are inherently *sequential* and *memory-bound*. Existing scalable PPR algorithms are mostly described as single-thread CPU solutions that are resistant to parallelization. Further, they usually involve complex data structures which do not have efficient adaptations on GPUs. kPAR overcomes these problems via both novel algorithmic designs (namely, *adaptive forward push* and *inverted random walks*) and system engineering (e.g., load balancing) to realize the potential of GPUs. Meanwhile, kPAR provides rigorous guarantees on both result quality and worst-case efficiency. Extensive experiments show that kPAR is usually 10x faster than parallel adaptations of existing methods. Notably, on a billion-edge Twitter graph, kPAR answers a top-1000 PPR query in 42.4 milliseconds.

PVLDB Reference Format:
Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. Realtime Top-k Personalized PageRank over Large Graphs on GPUs. *PVLDB*, 13(1): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3357377.3357379

Proceedings of the VLDB Endowment, Vol. 13, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3357377.3357379

1. INTRODUCTION

Given a graph G, a source node s, and a target node t, the *Personalized PageRank* (*PPR*) [28] $\pi(s,t)$ of t with respect to s is defined as the probability that a random walk starting from s terminates at t. Intuitively, the PPR of t reflects its relevance from the perspective of source node s, which is meaningful in many important applications. For instance, search engines use PPR to rank web pages based on individual users' preferences [45]; social networking sites utilize PPR to recommend contents and new connections to users [24, 32]; biologists apply PPR to analyze the relationships in protein networks [26]. Recently, PPR has also been applied to deep graph convolutional networks [30], which achieves state-of-the-art performance in semi-supervised classification tasks.

In many application, often the user is mainly interested in the most relevant nodes with respect to the source node s [54], rather than ones with little relevance to s. This motivates top-k PPR queries, which return the k nodes with the highest PPR scores with respect to s. Finding the exact top-k PPR results is known to be computationally intensive [52, 54], due to the large number of nodes that need to be accessed. Hence, practical solutions that scale to large graphs (e.g., [52,54]) usually focus on answering *approximate* top-k PPR queries, which provides a controllable trade-off between result quality and processing efficiency. Even so, processing a top-k on an Internet-scale graph with billions of edges is still rather expensive in terms of computations, and existing work (e.g., [31]) has used a computing cluster with hundreds of nodes for this purpose.

In this paper, we set the ambitious goal of answering a top-k PPR query in *realtime* (in particular, within 100 milliseconds, as suggested in [1, 44]), on a single commodity server. Such a solution can be a key enabler for applications where users wait online for top-k PPR results, and graph analysis at scale using deep neural networks [30]. This objective, however, is far beyond the current state of the art, which, though asymptotically optimal, often neglect system aspects such as parallelism and memory access. We achieve it through a novel solution, namely kPAR, which utilizes the massively parallel processing power of a graphical processing unit (GPU). Compared to a CPU, a GPU has a significantly higher number of threads; however, unlike a CPU thread that can individually carry out instructions with complex logic, GPU threads are lightweight ones limited to simple tasks. Further, GPU threads are grouped into *warps* (e.g., 32 threads per warp), such that all threads in the same warp can only perform the same execution path concurrently. If multiple paths exist in a warp (e.g., if-else clauses), these paths are executed sequentially rather than in parallel, which is called *warp divergence*. Due to such architectural characteristics, common data structures, e.g., sets and heaps, are considerably less efficient on GPUs than on CPUs; hence, adapting a CPU-based algorithm to GPUs often requires non-trivial algorithmic re-designs. Designing a GPU-based top-k PPR solution is particularly challenging, as PPR computation relies on graph traversal and value propagations that involve complex logic and random memory accesses, which would lead to severe warp divergence and high GPU memory access overhead. In addition, real-world graphs often exhibit high skewness, e.g., in terms of node degrees and random walk costs, which may cause load-imbalance among warps, further hampering GPU parallelism. In our experiments, we have adapted a recent approach FORA+ [52] to run on a GPU, and found that its performance is no better than on a multiple-core CPU.

Addressing the above challenges for using GPUs to answer top-k PPR queries, we propose kPAR (short for Top-k PPR with Adaptive Forward Push and Inverted Random Walks), a GPU-based top-k PPR algorithm that achieves high efficiency through both algorithmic designs and system engineering specific to GPUs. In terms of algorithmic design, kPAR includes two main techniques, *adaptive forward push* (AFP) and *inverted random walks* (IRW). AFP differentiates the push strategies of nodes according to their global PageRank values [45], to remedy the deficiency of the original Forward Push [9] that applies a unified push strategy for all nodes, which would cause most GPU threads to stay idle in the iterations with few nodes to push. IRW maintains a start-list for each node v, which stores all the nodes with at least one random walk stopping at v, as well as their frequencies to v, differing from the conventional way that stores, for each node v, a list of end nodes of the random walks starting from v [51, 52]. To store the same number of random walks, IRW requires less space, which is especially important on GPUs with a relatively small amount of high-bandwidth video RAM (16GB in our experiments). More importantly, utilizing the IRW technique, kPAR develops new PPR bounds (elaborated in Section 3.4) to help find top-k results by only scanning promising candidates instead of all nodes. Finally, we develop parallel optimizations for PPR computation on GPUs, which take into account load balancing, atomic operations, and memory access patterns.

We experimentally evaluate our algorithm against the state-of-the-art methods, including their parallel versions on CPUs and GPUs over large graphs. The results show that kPAR is faster than existing methods by one to three orders of magnitude. In particular, kPAR answers top-1000 PPR queries using 42.4 milliseconds on a large Twitter graph with 42 million nodes and 1.5 billion edges.

2. PRELIMINARIES

2.1 Problem Definition

Let $G = (V, E)$ be a directed graph, where V is the set of nodes and E is the set of edges ($n = |V|$ and $m = |E|$). Given a source node $s \in V$ and a jump factor α, a walker starts from s to traverse the graph G, and at each step, the walker either (i) terminates at the current node with α probability, or (ii) jumps to a randomly selected out-neighbor of the current node. For any node $v \in V$, the personalized PageRank (PPR) $\pi(s, v)$ is the probability that a random walk from s terminates at v [45]. Note that if a graph is undirected, we can convert it to a directed graph by regarding each edge as two directed edges of opposite directions.

Table 1: Frequently used notations.

Notation	Description
G=(V, E)	Input graph G with node set V and edge set E
n, m	$n = \|V\|, m = \|E\|$
$N_{out}(v)$	The sets of out-neighbors of node v
$d_{out}(v)$	The out-degree of node v
$\pi(s, t)$	The exact PPR value of t w.r.t., s
$\hat{\pi}(s, t)$	The estimated PPR value of t w.r.t., s
α	The probability for a walker to stop per step
δ, ϵ, p_f	Parameters of an approximate top-k PPR query (Definition 1)
δ_{init}	The initial value of δ for a query
δ_{min}	The pre-computation setting of δ; during a query, $\delta \geq \delta_{\text{min}}$
r_{max}	The global residue threshold for forward push
$r_{max}(v)$	The residue threshold of node v
$\beta(v)$	The scale factor of v, $r_{max}(v) = \beta(v) \cdot r_{max}$
$r(s, v)$	The residue of v during forward push from s
$\pi^{\circ}(s, v)$	The reserve of v during forward push from s
τ	The candidate threshold
ρ	The maximum possible increment of PPR value caused by one random walk in Eq. (10)
$\omega(v, t)$	The total number of random walks starting from node v and stopping at t
$\omega(v)$, $\omega_I(v)$	The total number of random walks starting from (resp. stopping at) node v

A top-k PPR query takes as input a graph G, a source node s, and a parameter k, and returns the top-k nodes with the highest PPR with respect to s, together with their respective PPR values. This paper focuses on *approximate* top-k PPR query processing, defined as follows [51, 52]:

Definition 1. (Approximate Top-k PPR Queries) Given a source node s, a threshold δ, an error bound ϵ, a failure probability p_f, and a positive integer k, an approximate top-k PPR query returns a sequence of k nodes $\{v_1, \ldots, v_i, \ldots, v_k\}$, associated with their estimated PPR $\hat{\pi}(s, v_i)$, such that with probability $1 - p_f$, for any $i \in [1, k]$, when $\pi(s, v_i^*) > \delta$,

$$|\hat{\pi}(s, v_i) - \pi(s, v_i)| \leq \epsilon \cdot \pi(s, v_i) \tag{1}$$

$$\pi(s, v_i) \geq (1 - \epsilon) \cdot \pi(s, v_i^*) \tag{2}$$

hold, where v_i^* is the node with i-th largest actual PPR.

The accuracy of the estimated PPR values is ensured by Equation 1, and Equation 2 guarantees that the i-th returned node has a PPR close to the actual i-th largest PPR. Following previous work [36, 37, 51, 52], we assume that $\delta = O(1/n)$, to provide approximation guarantees for nodes with above-average PPR. Table 1 lists the notations that are frequently used in our paper.

2.2 Graphics Processing Units

GPUs are widely used for general-purpose parallel computing [22, 33, 42, 43, 49, 53]. Here we introduce the key architectural characteristics of GPUs with the popular Nvidia CUDA programming framework.

Kernels and Thread Hierarchy. A GPU consists of tens of Streaming Multiprocessors (SMs), each of which has more than one thousand threads. The threads are managed in a *grid-block-warp* hierarchy. When a GPU program, called a *kernel*, is launched with a user-specified number of threads, the GPU creates a *grid* of threads for it. The grid partitions the threads into a number of *blocks* with same size (*e.g.*, 1024). A block consists of dozens of *warps*, the basic unit of execution in an SM. Note that the user-specified number of threads and the block size cannot be changed at runtime.

Single Instruction Multiple Thread (SIMT). A warp consists of 32 consecutive threads. SIMT means all threads in a warp execute the same instruction at the same time, and each thread carries out that operation on its own private data. If two threads in a warp have different execution paths, *warp divergence* occurs, and the GPU executes these paths sequentially. GPU threads are extremely lightweight and optimized for data-parallel tasks with simple control logic, unlike a CPU thread that is capable of handling complex control logic individually. Hence, complex control logic on GPUs would downgrade the efficiency.

Memory Hierarchy and Coalescing. Threads on GPUs cannot communicate with each other directly. Threads in a warp can communicate via *registers*; threads in a block can share data via *shared memory*; threads across blocks can only share data via *global memory*. Among these, global memory is the largest but slowest storage, while registers and shared memory are scarce resources in SMs. GPUs support *memory coalescing* to reduce global memory access overhead: if multiple threads in a warp access consecutive addresses in global memory, the SM for the warp issues one request to get the consecutive data for these threads, instead of issuing separate requests; if the memory accesses are not consecutive, multiple random accesses are issued.

Atomic Operations. The numerical operations on parallel hardware must be atomic since multiple threads may update the same value simultaneously. GPUs support necessary atomic operations. Atomic operations need to be used carefully. If several atomic operations update the same value simultaneously, the operations would be executed sequentially. Even worse, the operations will keep trying to update until success, which means a lot of repeated control logic, *e.g.*, while loops, leading to warp divergence.

2.3 Existing CPU-based Methods

We first explain two basic techniques for estimating PPR, Monte-Carlo and ForwardPush, which are closely related to this work, and then review the state-of-the-art methods, FORA and TopPPR, as well as their deficiencies for GPUs.

Monte-Carlo. Monte-Carlo (MC) [17] is a classic solution for PPR estimation. MC generates $\omega(s)$ random walks from source node s. For each node v, there are $\omega(s,v)$ random walks stopping at v. The fraction $\frac{\omega(s,v)}{\omega(s)}$ is regarded as the estimated PPR $\hat{\pi}(s,v)$. When $\omega(s) = \Omega\left(\frac{\log(1/p_f)}{\epsilon^2\delta}\right)$, MC satisfies Equation 1 with time complexity $O\left(\frac{\log(1/p_f)}{\epsilon^2\delta}\right)$ [17]. MC is inefficient as shown in [36, 51, 52].

Forward Push. Forward Push [9] (see Algorithm 1) can approximate the PPR scores of all the nodes w.r.t., source s, but without any accuracy guarantee. Every node v has two values, a *reserve* $\pi^\circ(s,v)$ and a *residue* $r(s,v)$, with initial values $r(s,s) = 1$, $\pi^\circ(s,s) = 0$, and $r(s,v) = \pi^\circ(s,v) = 0$ for any $v \neq s$. Forward Push sets a threshold r_{max} for all the nodes, and then keeps *pushing* the residues of the nodes v satisfying $\frac{r(s,v)}{d_{out}(v)} > r_{max}$ (called frontiers) to their reserves and their out-neighbors' residues at Lines 6-12, where $N_{out}(v)$ denotes the set of out-neighbors of v and $d_{out}(v)$ is the number of out-neighbors. Specifically, it increases the residue of each out-neighbor u's residue by $(1-\alpha)\cdot\frac{r(s,v)}{d_{out}(v)}$. Then, it increases v's reserve by $\alpha \cdot r(s,v)$, and resets v's residue to $r(s,v) = 0$. This iterative process terminates when there is no node to push, i.e., when the frontier set S becomes empty. Finally, Forward Push returns $\pi^\circ(s,v)$ as an approximation of $\pi(s,v)$. Forward Push runs in $O(1/r_{max})$ time. The estimated $\pi^\circ(s,v)$ does not offer any worst-case error guarantee, meaning that Forward Push by itself is insufficient for the problem in Definition 1.

Algorithm 1: Forward Push

```
   Input: $G$, $s$, $\alpha$, $r_{max}$
   Output: $\pi^\circ(s,v)$, $r(s,v)$ for all $v \in V$
1  $r(s,s) \leftarrow 1; r(s,v) \leftarrow 0$ for all $v \neq s$;
2  $\pi^\circ(s,v) \leftarrow 0$ for all $v$;
3  Frontier Set $S \leftarrow \{s\}$;
4  while $S \neq \emptyset$ do
5      Temporary Set $S' \leftarrow \emptyset$;
6      for each $v \in S$ do
7          for each $u \in N_{out}(v)$ do
8              $r(s,u) \leftarrow r(s,u) + (1-\alpha)\cdot\frac{r(s,v)}{d_{out}(v)}$;
9              if $r(s,u)/d_{out}(u) > r_{max}$ then
10                 $S' \leftarrow S' \cup \{u\}$;
11         $\pi^\circ(s,v) \leftarrow \pi^\circ(s,v) + \alpha\cdot r(s,v)$;
12         $r(s,v) \leftarrow 0$;
13     $S \leftarrow S'$;
```

FORA. FORA [52] leverages the power of both Forward Push and Monte-Carlo, and applies an iterative framework to answer top-k PPR queries. In each iteration, it first performs forward push from s, and then generates random walks from the nodes with non-zero residues. Specifically, Forward Push has the following invariant [9]:

$$\pi(s,t) = \pi^\circ(s,t) + \textstyle\sum_{v\in V} r(s,v)\cdot\pi(v,t), \tag{3}$$

After forward push, FORA uses Monte-Carlo to derive $\pi'(v,t)$, approximating the unknown $\pi(v,t)$ for each node v, and then replaces $\pi(v,t)$ in Equation 3 with $\pi'(v,t)$, obtaining the following estimation:

$$\hat{\pi}(s,t) = \pi^\circ(s,t) + \textstyle\sum_{v\in V} r(s,v)\cdot\pi'(v,t). \tag{4}$$

Denote the sum of the residues of all the nodes after forward push as r_{sum}. Since Forward Push runs in $O(1/r_{max})$ and the expected runtime complexity of the Monte-Carlo phase is $O(r_{sum}\cdot\frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2\cdot\delta})$, FORA minimizes the total time complexity by setting r_{max} as follows:

$$r_{max} = \frac{\epsilon}{\sqrt{m}}\cdot\sqrt{\frac{\delta}{(2\epsilon/3+2)\cdot\log(2/p_f)}}. \tag{5}$$

On CPUs, FORA performs forward push as shown in Algorithm 1. Forward Push on CPUs can efficiently maintain a *set* of unique frontiers to push (Lines 3, 5 and 10), since sets (or heaps, queues) on CPUs can be efficiently updated.

In contrast, the only data structure on GPUs that we can rely on to design our algorithm is just *arrays*. When migrated to GPUs, in each iteration, forward push on GPUs atomically pushes residues of all the frontiers and generates an array of new frontiers for next iteration. In Section 3.2, we show that simply extending forward push on GPUs is rather inefficient since there are many iterations with few frontiers, indicating that most of the threads are idle during these iterations, which slows down the performance. Furthermore, most real-world graphs are *scale-free* (*i.e.*, following power law distributions with some nodes having extremely large out-degrees but the remaining nodes having small out-degrees), which generates *imbalanced workloads* on GPUs among thread blocks. A similar situation occurs during Monte Carlo, though random walks can be pre-computed and parallelized. In our experiments, CPU-parallelized and GPU-parallelized FORA are inferior to our algorithm.

TopPPR. TopPPR [54] is an algorithm for top-k PPR queries that ensures at least p-precision (i.e., at least p fraction of the actual top-k nodes are returned). Note that TopPPR solves a different problem from ours, in the sense that it does not provide guarantees about the PPR values, nor the order of the nodes returned. TopPPR integrates three techniques: Forward Push (FP), Monte Carlo (MC), and Backward Search (BS) [8], while FORA only uses FP and MC. In a nutshell, TopPPR iteratively maintains and updates a candidate set by performing the three techniques one by one, until the candidate set is certain to satisfy the precision requirements. It is non-trivial to extend TopPPR to GPUs. First, TopPPR needs to store two copies of a graph in memory, one for the out-neighbor lists of all nodes (used by FP and MC), and the other for the in-neighbor lists of all nodes (used by BS). This doubles the memory consumption of the algorithm, which can be problematic for large graphs. For instance, for the billion-edge Twitter graph used in our experiments, TopPPR requires 24GB memory, which exceeds the capacity (16GB) of the Tesla P100 GPU in our experimental setting. Second, TopPPR suffers the same issues that FORA has during FP and MC stages, as analyzed above for FORA. Third, for every iteration, TopPPR needs to create and maintain data structures that are difficult to parallelize on a GPU due to the latter's architectural characteristics, which include the Alias structure [50] for weighted sampling, inverted lists for group Backward Search, and priority queues for candidate update. In our experiments, we found that the GPU version of TopPPR is no faster than the multi-thread CPU version of TopPPR.

3. ALGORITHM

We provide an overview of our algorithmic techniques and then present the details.

3.1 Overview

Our algorithm combines Forward Push and Monte-Carlo , and utilizes the forward push invariant (Equations 3 and 4) to answer top-k PPR queries defined in Definition 1. Observing that there are many *ineffective iterations* (*i.e.*, iterations with few frontiers, causing GPUs mostly idle) during forward push, we propose an *adaptive forward push* (AFP) technique in Section 3.2 to tackle the issue. Instead of using one r_{max} threshold for all the nodes in the original forward push, we use *global PageRank* [45] to assign different residue thresholds $r_{max}(v)$ to different nodes v. For nodes that tend to become frontiers in the ineffective iterations, AFP assigns larger residue thresholds to them, aiming to decrease their chance to become frontier, and thus reduce ineffective iterations. With respect to the Monte-Carlo phase, in Section 3.3, we design a new indexing scheme, called *inverted random walks* (IRW), to organize the pre-computed random walks in their inverted form. In a nutshell, each node v maintains a start-list containing all the nodes that have random walks stopping at v, as well as the count of such random walks. By aggregating random walks with the same starting and ending nodes together, the index size is reduced, which offers flexibility to design more optimizations that may require extra space, especially on GPUs with limited global memory. Utilizing IRW, we derive new theoretical bounds and present our parallel algorithm, kPAR, for top-k PPR queries in Section 3.4.

Algorithm 2: ParallelAFP

```
  Input: Set of frontiers $F, s, \alpha, r_{max}(v)$ for all $v \in V$
  Output: $\pi^{\circ}(s,v)$, $r(s,v)$ for all $v \in V$
1 while $F \neq \emptyset$ do
2     parallel for each $v \in F$ do
3         $\pi^{\circ}(s,v) \leftarrow \pi^{\circ}(s,v) + \alpha \cdot r(s,v)$;
4         $F(v) \leftarrow r(s,v)$; $r(s,v) \leftarrow 0$;
5     $F \leftarrow$ ParallelResiduePush$(F, r_{max}(\cdot))$;
```

3.2 Adaptive Forward Push

Algorithm 2 shows the pseudo code of AFP on GPUs, ParallelAFP, with new techniques to be presented shortly. In each iteration, the frontiers in F push their residues to their reserves (Line 3) and to the residues of their out-neighbors using procedure ParallelResiduePush at Line 5, which also returns the new frontiers for next iteration. Algorithm 2 iterates until F is empty. ParallelResiduePush will be explained in Section 4.1 since it involves mainly parallel optimizations.

We observe that the number of frontiers increases abruptly during the first few iterations, but then decreases slowly by many iterations until reaching zero. Lots of the iterations in the late stage have just dozens of frontiers, which means that most of the threads are idle in these iterations. We call such iterations as *ineffective iterations*. Ineffective iterations are harmful to efficiency, since (i) GPU resources are mostly idle during the ineffective iterations; (ii) the launch of kernels has a minimum cost regardless of the number of frontiers.

AFP is designed to reduce such ineffective iterations. Instead of one residue threshold r_{max} for all nodes, AFP maintains an individual threshold $r_{max}(u)$ for each node $u \in V$, and $r_{max}(u)$ is larger if u is easier to become frontier in the ineffective iterations. One question is how to identify such nodes before any query starts.

We employ global PageRank [45], a measure that evaluates the global importance of a node $u \in V$ with respect to the whole graph. Denote the PageRank of node u as $pr(u)$. Generally, a node u with large $pr(u)$ has a large impact on others (*i.e.*, many incoming followers). Further, observe that residues are pushed out along the out-going edges of a frontier t, equivalent to being pushed along the incoming-edge of the out-neighbors u of the frontier. Therefore, node u with larger $pr(u)$ tends to accumulate higher residue. Note that u becomes frontier if $r(s,u)/d_{out}(u) > r_{max}$ holds. Hence, among the nodes with large PageRank, those with small out-degrees are the easiest ones to become frontiers. Therefore, we pre-compute the PageRank of all the nodes u in graph G and sort the nodes in decreasing order based on score $pr(u)/d_{out}(u)$, to obtain a ranked list called *PRO-rank list*. Then given the r_{max} in Equation 5, AFP assigns a larger residue threshold $r_{max}(u)$ for u by assigning a larger scale factor $\beta(u)$ if u ranks higher in the list:

$$r_{max}(u) = \beta(u) \cdot r_{max}, \text{ where } \beta(u) \geq 1. \qquad (6)$$

AFP increases the difficulty for the highly ranked nodes to become frontiers, without requiring any information about specific queries and only using PageRank and out-degrees. Method ParallelAFP in Algorithm 2 takes as input a vector of residue thresholds $r_{max}(v)$ for all $v \in V$, instead of one r_{max} for all in the original Forward Push (Algorithm 1). For simplicity, we use $r_{max}(\cdot)$ to represent $r_{max}(v)$ thresholds of all $v \in V$; similar notations are used for other variables too.

Remark. The increase of residue threshold $r_{max}(u)$ should be compensated by more random walks from u in order to guarantee the accuracy, meaning that the choice of $r_{max}(u)$ is closely related to the number of random walks precomputed by u. Therefore, we explain how to decide $r_{max}(u)$ in Section 3.3.2 after introducing the inverted random walks.

3.3 Inverted Random Walks

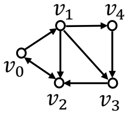
Given a random walk from node v to t, we call v as the start and t as the end of the random walk. Existing methods [51, 52] maintain an end-list for each node v, storing all the end nodes of the random walks starting from v, in their randomly generated order (*e.g.*, left part of Figure 1). Observe that there are many random walks from the same start to the same end. For instance, v_0 stops at v_2 three times and consequently, v_2 is stored three times in the end-list of v_0, obviously wasting space that is precious on GPUs. We propose to organize random walks in their inverted form, called Inverted Random Walks (IRW). IRW maintains a start-list for each node v, and the list stores all the nodes that have at least one random walk stopping at v, as well as their frequencies to v. When storing the same number of random walks, IRW uses less space than the conventional index, which offers us the space flexibility on GPUs to design more optimizations for large graphs. More importantly, IRW assists the algorithm kPAR in Section 3.4 to develop new PPR bounds, which helps to find top-k results on GPUs.

3.3.1 Doubled-CSR IRW Structure

Specifically, every node t maintains a *start-list*, which stores the nodes that have at least one random walks stopping at t. Denote $\omega(v)$ as the number of random walks starting from node v, and $\omega(v,t)$ as the number of random walks starting from v and stopping at t. For node t, its start-list consists of $(v, \omega(v,t))$ pairs. The right side of Figure 1 presents the corresponding IRW scheme example. The start-list of v_3 indicates that node v_1 stops at v_3 4 times (frequencies in brackets), v_4 stops at v_3 3 times, and v_0 stops at v_3 once (no number in brackets means frequency 1). IRW also stores the total number of random walks that stop at v, $\omega_I(v)$, *e.g.*, v_3 has 8 random walks stopping at it. If a frequency is larger than one, we call it *significant*. Within a start-list, we place the starting nodes with significant frequencies ahead of those insignificant ones, which helps organize IRW into a compact *doubled-CSR* format (Compressed Sparse Row).

$\omega(v)$

$v_0(8) \rightarrow v_3, v_1, v_1, v_2, v_0, v_1, v_2, v_2$
$v_1(10) \rightarrow v_1, v_2, v_3, v_4, v_2, v_4, v_4, v_3, v_3, v_3$
$v_2(4) \rightarrow v_1, v_0, v_0, v_0$
$v_3(4) \rightarrow v_2, v_2, v_0, v_2$
$v_4(4) \rightarrow v_3, v_2, v_3, v_3$
RW index



$\omega_I(v)$

$v_0(5) \leftarrow v_2(3), v_0, v_3$
$v_1(5) \leftarrow v_0(3), v_1, v_2$
$v_2(9) \leftarrow v_0(3), v_1(2), v_3(3), v_4$
$v_3(8) \leftarrow v_1(4), v_4(3), v_0$
$v_4(3) \leftarrow v_1(3)$
IRW index (start-lists)

Figure 1: Example of Inverted Random Walks

Nodes	v_0	v_1	v_2	v_3	v_4	
$\omega(v)$	8	10	4	4	4	
$\omega_I(v)$	5	5	9	8	3	
src_begin	0	3	6	10	13	14
sigFreq_begin	0	1	2	5	7	8

	0			3			6				10			13
CSR_1	v_2	v_0	v_3	v_0	v_1	v_2	v_0	v_1	v_3	v_4	v_1	v_4	v_0	v_1

	0	1	2			5		7
CSR_2	3	3	3	2	3	4	3	3

Figure 2: Doubled-CSR Inverted Random Walks

Figure 2 displays the doubled-CSR of the example in Figure 1: CSR$_1$ stores all the start-lists and CSR$_2$ stores the corresponding significant frequencies.

Then we can use IRW to estimate $\hat{\pi}(s,t)$. After forward push from query s, the reserves and residues of all the nodes are known, and $\pi'(v,t)$ is estimated by $\frac{\omega(v,t)}{\omega(v)}$. Then we can estimate $\hat{\pi}(s,t)$ based on Equation 4 by parallel-scanning the start-list of t. Take Figure 1 as an example with source node v_0 and target node v_3. According to Equation 4, $\hat{\pi}(v_0, v_3) = \pi^{\circ}(v_0, v_3) + \sum_{v \in V} r(v_0, v) \cdot \pi'(v, v_3)$. Suppose that after forward push, the residues are $r(v_0, v_0) = 0, r(v_0, v_1) = 0.128, r(v_0, v_2) = 0.235, r(v_0, v_3) = 0.107, r(v_0, v_4) = 0.107$. The start-list of v_3 in Figure 1 shows that only v_1, v_4, v_0 have random walks stopping at v_3. Therefore, $\sum_{v \in V} r(v_0, v) \cdot \pi'(v, v_3) = 0.128 \cdot \frac{4}{10} + 0.107 \cdot \frac{3}{4} + 0 \cdot \frac{1}{8} = 0.131$. Then $\hat{\pi}(v_0, v_3)$ can be easily obtained since $\pi^{\circ}(v_0, v_3)$ is already produced by forward push. Note that although there are 4 random walks from v_1 to v_3, they are processed together, due to the frequency aggregation.

3.3.2 Offline Indexing and Deciding $r_{max}(v)$

For each node $v \in V$, we pre-compute $\omega(v)$ random walks:

$$\omega(v) = \left\lceil d_{out}(v) \cdot r_{max}(v) \cdot \frac{(2\epsilon/3 + 2) \cdot \log \frac{2n \log n}{p_f}}{\epsilon^2 \cdot \delta_{\min}} \right\rceil, \qquad (7)$$

where $\delta_{\min}$ is the pre-computation setting for δ.

According to Lemma 1 and Corollary 1 introduced later in Section 3.4.5, with the above number of random walks pre-computed per node, given any nodes s and t in V, we can guarantee that $(1 - \epsilon) \cdot \pi(s,t) \leq \hat{\pi}(s,t) \leq (1 + \epsilon) \cdot \pi(s,t)$ when $\pi(s,t) > \delta_{\min}$. For simplicity, denote $\psi = \frac{(2\epsilon/3+2) \cdot \log(2n \log n / p_f)}{\epsilon^2}$.

As mentioned in Section 3.2, to guarantee the same accuracy, the increase of residue thresholds must be compensated by more random walks. Given the fact that GPUs have very limited memory, we decide $\beta(v)$ and $r_{max}(v)$ in Equation 6 by a space budget. At the beginning, assume that $\beta(v) = 1$ for all $v \in V$ (*i.e.*, $\forall v \in V, r_{max}(v) = r_{max}$), then we can have the initial total number of random walks

w: $\omega = \sum_{v \in V} \omega(v)$. Suppose that we can afford a budget of 20% more random walks. Choosing a factor $a = 2^i$ ($i \geq 1$), we scan the PRO-rank list built in Section 3.2 from top down, and increase $\omega(v)$ to $a \cdot \omega(v)$ for the scanned nodes v, until half of the budget is consumed. Then we decrease a by half, and keep scanning and increasing $\omega(v)$ to $a \cdot \omega(v)$, until a quarter of the budget is used up. This process continues until a becomes 2. When $a = 2$, we keep scanning and deplete all remaining budget. If node v has $a - 1$ times more random walks, $\beta(v)$ is calculated as in Equation 8, derived based on Equation 7. Then we get $r_{max}(v)$ in Equation 6.

$$\beta(v) = \frac{a \cdot \delta_{\min} \cdot \lceil d_{out}(v) \cdot r_{max} \cdot \psi / \delta_{\min} \rceil - \delta_{\min}}{d_{out}(v) \cdot r_{max} \cdot \psi} \tag{8}$$

The space complexity of IRW index is presented below.

THEOREM 1. *The space complexity of the* IRW *index is*

$$O\left(n + \tfrac{1}{\epsilon}\sqrt{\frac{m \log(1/p'_f)}{\delta_{\min}}}\right).$$

When $\delta_{\min} = O(1/n)$, $\log p'_f = O(\log n)$, *for scale-free graphs, i.e.,* $m/n = O(\log n)$, *the complexity becomes* $O(\frac{1}{\epsilon} n \cdot \log n)$.

PROOF.

$$\begin{aligned} & O\left(\textstyle\sum_v \left\lceil r_{max}(v) \cdot d_{out}(v) \cdot \frac{\psi}{\delta_{\min}} \right\rceil\right) \\ & = O\left(\textstyle\sum_v \left\lceil r_{max} \cdot d_{out}(v) \cdot \frac{\psi}{\delta_{\min}} \right\rceil\right) \\ & = O\left(n + \sqrt{\tfrac{m \cdot \psi}{\delta_{\min}}}\right) = O\left(n + \tfrac{1}{\epsilon}\sqrt{\frac{m \log(1/p'_f)}{\delta_{\min}}}\right) \end{aligned} \tag{9}$$

□

3.4 Parallel Top-k PPR Processing

Algorithm 3 shows the pseudocode of the main algorithm kPAR. Given a graph G, a source node s, a jump factor α, a parameter k, kPAR finds the top-k nodes associated with their approximated PPR scores. kPAR works in an iterative way, starting by setting δ with a relative large δ_{init}, and reducing δ by half per iteration, until δ reaches $\delta_{\min}$ or the maximum iterations L is reached (Line 6), where L is a constant with $L \leq \log n$. The failure probability of each iteration is $p'_f = \frac{p_f}{n \log n}$ (Line 1). In each iteration, all n nodes have the possibility to be in the top-k result. By applying union bound, when $p'_f = \frac{p_f}{n \log n}$, the failure probability of the whole algorithm is exactly p_f (*i.e.*, p_f in Equation 5 needs to be replaced with p'_f). In each iteration, parallel AFP is performed at Line 7, followed by candidate set generation (Line 8) and a technique called parallel bound reduction (Line 10) that helps get a tighter bound. From Lines 12-23, we probe the candidate set C and check if top-k nodes satisfying Definition 1 are found. Specifically, we first partition the candidates in C into shards C_i with size that can exactly utilize all the threads of a GPU, then calculate the PPR scores of the candidates in C_i in parallel (Line 13). Then, the processed candidates with their scores are used to calculate the bounds derived in Section 3.4.2 (Lines 14-23). If the current iteration fails, δ is reduced by half and other parameters are updated accordingly at Lines 24-25. Under the new parameters, new frontiers are generated for next iteration (Line 26). Following convention [52], under extreme cases that rarely happen, if no top-k nodes are reported after L iterations, parallel single source query is performed to get the top-k result (Lines 27-32).

Algorithm 3: Parallel Top-k PPR: kPAR

```
   Input: Graph $G$, source node $s$, jump factor $\alpha$, parameter $k$
   Output: Top-$k$ nodes with approximate PPR scores
 1 $\delta \leftarrow \delta_{\text{init}}$, $p'_f \leftarrow \frac{p_f}{n \cdot \log n}$, $\ell \leftarrow 0$;
 2 $r(s,s) \leftarrow 1, r(s,v) \leftarrow 0$ for all $v \neq s$;
 3 $\pi^\circ(s,v) \leftarrow 0$, $\hat{\pi}(s,v) \leftarrow 0$ for all $v$;
 4 Processed nodes $C_p \leftarrow \{\}$; Current top-$k$ nodes $C_k \leftarrow \{\}$;
 5 Current $k$-th largest PPR score $\hat{\pi}_k \leftarrow 0$;
 6 while $\delta \geq \delta_{\min}$ and $\ell < L$ do
 7     ParallelAFP$(F, s, a, r_{max}(\cdot))$;
 8     $C \leftarrow$ ParallelGetCandSet$(\pi^\circ(\mathsf{s}, \cdot), \tau)$;
 9     $\hat{\pi}(s,\cdot) \leftarrow \pi^\circ(s,\cdot)$;
10     $\omega_I^{ub} \leftarrow$ ParallelBoundReduction$(H)$;
11     $C_p \leftarrow C_p \cup H$;
12     for each shard $C_i \subseteq C$ do
13         ParallelComputePPR$(C_i)$;
14         Update $C_k$, $C_p$, and get $\hat{\pi}^{ub}$ by Eq. (11);
15         $LB_\ell(\hat{\pi}'_k) \leftarrow \forall v \in C_k$, the smallest $LB_\ell(v) \geq \delta_{\min}$;
16         success← false;
17         if $LB_\ell(\hat{\pi}'_k) \geq (1-\epsilon) \cdot UB_\ell(\hat{\pi}^{ub})$ then
18             success← true;
19             parallel for $v_i \in C_k$ that $LB_\ell(v_i) \geq \delta_{\min}$ do
20                 if $LB_\ell(v_i) \leq (1-\epsilon) \cdot UB_\ell(v_{i+1})$ then
21                     success← false;
22         if success then
23             return top-$k$ nodes with their PPR $\hat{\pi}(s,\cdot)$ ;
24     $\delta \leftarrow \delta/2$, $\ell \leftarrow \ell + 1$;
25     Update $r_{max}$, $r_{max}(v)$ , $\rho$ by Eq. (5), (6), and (10);
26     $F \leftarrow$ ParallelGenFrontiers$(r(s,\cdot), r_{max}(\cdot))$;
27 parallel for each node $v$ with $r(s,v) > 0$ do
28     $\omega(v) = \left\lceil r(s,v) \cdot 4(\epsilon/3 + 2) \cdot \log(2/p'_f)/(\epsilon^2 \cdot \delta) \right\rceil$;
29     parallel for $i = 1, \ldots, \omega(v)$ do
30         $t \leftarrow$ RandomWalk$(s)$;
31         atomicAdd$(\hat{\pi}(s,t), \frac{r(s,v)}{\omega(v)})$;
32 return top-$k$ nodes with their PPR $\hat{\pi}(s,\cdot)$ ;
```

In the following, we explain the techniques mentioned above one by one, except procedures ParallelComputePPR (Line 13) and ParallelGenFrontiers (Line 26) that are mainly parallel optimizations and are introduced in Section 4.

3.4.1 Parallel Candidate Set Generation

The first step is to obtain a candidate set in each iteration. Since Algorithm 3 starts with $\delta = \delta_{\text{init}}$ and iteratively reduces δ by half until top-k nodes are found, in order to reduce unnecessary overhead, the number of promising candidates per iteration is related to δ. Naturally we can set a candidate threshold τ that is proportional to δ, *e.g.*, 0.5δ, such that after forward push, all the nodes with $\pi^\circ(s,v) \geq \tau$ are selected as candidates into C. Since C is chosen according to δ, the size of C would not be large. The candidates are sorted in decreasing order of their reserves. Note that given a candidate t, after estimating its PPR at Line 13 of Algorithm 3, if its $\hat{\pi}(s,t)$ is smaller than δ, it is excluded from the computation afterwards.

There are several possible ways to implement procedure ParallelGetCandSet to collect the candidates. One way is to maintain a candidate flag for each node v (initialized as 0). And during forward push, whenever $\pi^\circ(s,v)$ becomes larger than τ, we set the flag to 1. After forward push, a standard parallel select is performed over the flags to get all the

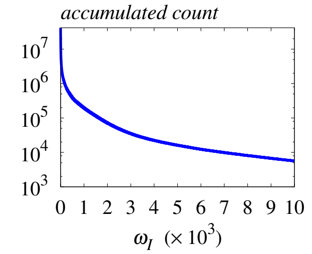


Figure 3: Accumulated Count of Nodes v with $\omega_I(v)$

flagged nodes into C. Another way maintains the candidate flags as well, but detects and inserts candidates into C during forward push. Specifically, C is an array with next available index $idx = 0$; whenever $\pi^\circ(s, v)$ becomes larger than τ and v's candidate flag is 0, the thread handling v atomically places v at the idx-th position of C and atomically increases idx by 1. When there are few candidates, the second way is preferred, otherwise, the first way.

3.4.2 PPR Bounds

For any top-k PPR queries with δ ($\delta \geq \delta_{\min}$), combining Equations 5, 6, and 7, we derive that the maximum possible increment of PPR value caused by one random walk from any node $v \in V$, denoted as ρ, is:

$$\rho = \frac{\sqrt{\delta \cdot \delta_{\min}}}{\psi} \tag{10}$$

After forward push from s, given the reserve $\pi^\circ(s, v)$ of any node $v \in V$, the following inequality holds:

$$\hat{\pi}(s, v) \leq \hat{\pi}^{ub}(s, v), \text{ where } \hat{\pi}^{ub}(s, v) = \pi^\circ(s, v) + \omega_I(v) \cdot \rho$$

Suppose that the current k-th largest score $\hat{\pi}_k$ is obtained after processing a subset of candidate set C, denoted as C_p. The unprocessed candidates are in set $C \setminus C_p$. We derive $\hat{\pi}^{ub}$ as an upper bound of $\hat{\pi}(s, v)$ for any unprocessed node $v \in V \setminus C_p$ as follows:

$$\hat{\pi}^{ub} = \max(\tau, \max_{v \in C \setminus C_p} \pi^\circ(s, v)) + \rho \cdot \omega_I^{ub}, \tag{11}$$

where $\omega_I^{ub} = \max_{v \in V} \omega_I(v)$.

For each top-k node $v \in C_k$ in ℓ-th iteration, given its $\hat{\pi}(s, v)$, we define its upper bound and lower bound as follows (the bounds are derived in Section 3.4.5 when proving the correctness of our algorithm):

$$\begin{aligned} UB_\ell(v) &= (1 + \epsilon') \cdot \hat{\pi}(s, v), \\ LB_\ell(v) &= (1 - \epsilon') \cdot \hat{\pi}(s, v), \end{aligned} \tag{12}$$

where ϵ' satisfies $\frac{2\epsilon'/3+2}{\epsilon'^2 \cdot \hat{\pi}(s,v)} = \frac{2\epsilon/3+2}{\epsilon^2 \cdot \sqrt{\delta \cdot \delta_{\min}}}$.

3.4.3 Bound Reduction in Parallel

Observe that ω_I^{ub} in Equation 11 can be extremely large when considering all the nodes in a graph, leading to a loose $\hat{\pi}^{ub}$. In the following, we develop a practical technique to reduce it by excluding a set H of nodes v with large $\omega_I(v)$. Then we calculate ω_I^{ub} as $\max_{v \in V \setminus H} \omega_I(v)$.

Figure 3 displays the accumulated count of nodes with $\omega_I(v)$ no smaller than a value at x-axis v (y-axis in log scale), on Twitter dataset. For example, when $\omega_I(v) = 3 \times 10^3$ at x-axis, the corresponding y-axis value is about 3.5×10^4, which means that there are 3.5×10^4 nodes with $\omega_I(v) \geq 3000$. The distribution in the figure is highly skewed. We call the set of nodes with $\omega_I(v) \geq 3000$ as the set of high-irw nodes, denoted as H. Compared with the 41.6 million nodes on Twitter, the high-irw nodes are few. During top-k query processing, it would be fruitful to compute the estimated PPR scores of all the nodes in H on the fly, such that it can help reduce ω_I^{ub} greatly. Note that this technique serves as a practical choice to help derive a tighter bound. For real graphs, it is easy to choose a size of H by studying the distributions as displayed in Figure 3.

Algorithm 4: ParallelBoundReduction

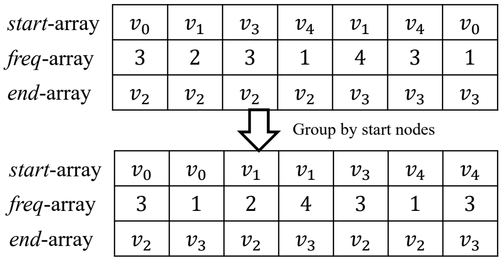
```
  Input: $H$, $r(s, v)$ and $\omega(v)$ for all $v \in V$
  Output: $\hat{\pi}(s, u)$ for any $u \in H$
1 parallel for each array index $h$ of $H$ do
2     $t \leftarrow end[h]$; $u \leftarrow start[h]$;
3     $\omega(u, t) \leftarrow freq[h]$;
4     $\Delta(s, t) \leftarrow r(s, u) \cdot \frac{\omega(u,t)}{\omega(u)}$;
5     atomicAdd($\hat{\pi}(s, t), \Delta(s, t)$);
```

We then design a structure that efficiently computes the PPR scores of all the nodes in H, matching the characteristics of GPUs, *e.g.*, memory coalescing and atomic operations. This structure trades space for time as explained below. We organize the nodes in H into a structure consisting of three arrays with the same length. The *start*-array concatenates all the start-lists of the nodes in H one by one. The *freq*-array concatenates the corresponding frequencies. Then the third array, *end*-array stores the respective nodes in H. The corresponding data of the high-irw nodes that are stored in IRW index are then removed, to avoid duplicated storage. The upper part of Figure 4 shows an example of the structure for $H = \{v_2, v_3\}$ in Figure 1. Without ambiguity, we denote the structure as H as well. H is accessed by array index h: $freq[h]$ is the number of random walks starting from $start[h]$ and stopping at $end[h]$. Algorithm 4 shows procedure ParallelBoundReduction that computes the PPR of all the nodes in H, which is invoked by Algorithm 3. Consecutive elements of the arrays in H are handled by consecutive threads and each thread handles one update. Lines 2 to 5 in Algorithm 4 are based on Equation 4.

Note that *end*-array stores same ending nodes at consecutive positions (the upper part of Figure 4). This is problematic in Algorithm 4. At Line 5, consecutive threads handling same ending node t will atomically update the same PPR $\hat{\pi}(s, t)$ almost at the same time, leading to sequential addition that is slow. We propose to group the three arrays in H based on the nodes in *start*-array to tackle the issue (the bottom half of Figure 4). The performance gain are two-fold: first, the atomicAdd operations to the same $\hat{\pi}(s, t)$ become scattered and consequently the parallelism is increased; second, the memory accesses of both $r(s, u)$ and $\omega(u)$ among neighboring threads are coalesced (Line 4 in Algorithm 4).

3.4.4 Choosing Initial Settings

If δ_{init} is set too large, Algorithm 3 will have many useless iterations before the top-k nodes are found; if δ_{init} is too small, it computes too much for top-k. We propose to choose a proper δ_{init} by estimating the k-th largest PPR. The assumption that the PPR values of all the nodes in real graphs follow power-law distribution is widely accepted in the literature [13, 36, 54]: $\pi_i \propto i^{-\alpha}$, where π_i is the i-th largest PPR and $\alpha \in (1/2, 1)$. For easy calculation, we set



start-array	v_0	v_1	v_3	v_4	v_1	v_4	v_0
freq-array	3	2	3	1	4	3	1
end-array	v_2	v_2	v_2	v_2	v_3	v_3	v_3

start-array	v_0	v_0	v_1	v_1	v_3	v_4	v_4
freq-array	3	1	2	4	3	1	3
end-array	v_2	v_3	v_2	v_3	v_2	v_2	v_3

Figure 4: Index structure $H = \{v_2, v_3\}$ example

$\alpha = 1$. Then given parameter k, we have $\pi_k = \kappa \cdot k^{-1}$, where κ is a normalization constant that $\kappa \sum_{i=1}^{n} i^{-1} = 1$. Consequently, we can get

$$\pi_k = \frac{1}{k(1 + 1/2 + 1/3 + \cdots)} \simeq \frac{1}{k \log n}.$$

This means that the k-th largest PPR value is roughly at the level of $\frac{1}{k \log n}$. To generate enough candidates, we choose to set $\delta_{init} = \pi_k/10$, *i.e.*, the $10k$-th PPR value. Regarding to L, the number of iterations allowed in Algorithm 3, since δ_{init} has already been set based on above, we set $L = 10$, which is large enough to get top-k result in our experiments.

3.4.5 Correctness and Complexities

Here we prove the correctness of Algorithm 3 and provide time complexity analysis (all the proofs are in Appendix).

THEOREM 2 ([15]). *Let $X_1, \cdots, X_\omega$ be independent random variables with $\Pr[X_j = 1] = p_j$ and $\Pr[X_j = 0] = 1-p_j$. Let $X = \sum_{j=1}^{\omega} a_j X_j$ with $a_j > 0$, and $\nu = \sum_{j=1}^{\omega} a_j^2 p_j$. Then, $\Pr[|X - \mathbb{E}[X]| \geq \lambda] \leq 2 \cdot \exp\left(-\frac{\lambda^2}{2\nu + 2a\lambda/3}\right)$, where $a = \max\{a_1, \cdots, a_\omega\}$.* □

Based on Theorem 2, we can derive Lemma 1, where δ' is a variable in $(0, 1)$ and it controls error bound ϵ'. How to set it is explained after Corollary 1 is introduced shortly and we will find that when δ' is set as $\hat{\pi}(s,t)$, we can derive the tightest upper and lower bounds.

LEMMA 1. *Let ϵ' satisfies $\frac{2\epsilon'/3+2}{\epsilon'^2 \cdot \delta'} = \frac{2\epsilon/3+2}{\epsilon^2 \cdot \sqrt{\delta \cdot \delta_{\min}}}$. For any node t with $\pi(s,t) > \delta'$ and any iterations, Algorithm 3 gets an approximated PPR $\hat{\pi}(s,t)$ that satisfies $\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon' \pi(s,t)]$ with at least $1 - p'_f$ probability.*

For any node t with $\pi(s,v) \leq \delta'$ and any iterations, Algorithm 3 gets an approximated PPR $\hat{\pi}(s,v)$ that satisfies $\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon' \delta']$ with at least $1 - p'_f$ probability.

When δ' and δ are both set to be $\delta_{\min}$, we have $\epsilon' = \epsilon$. Thus we get the following corollary.

COROLLARY 1. *For any node t with $\pi(s,t) > \delta_{\min}$ and query $\delta = \delta_{\min}$, Algorithm 3 gets an approximated PPR $\hat{\pi}(s,t)$ that satisfies $\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \pi(s,t)]$ with at least $1 - p'_f$ probability.*

Based on Lemma 1, we can derive the upper bound and lower bound of $\pi(s,t)$. In the ℓ-th iteration, we define

$$\begin{aligned} UB_\ell(t) &= \max\{(1+\epsilon') \cdot \hat{\pi}_j(s,t), \hat{\pi}_j(s,t) + \epsilon'\delta'\} \\ LB_\ell(t) &= \min\{(1-\epsilon') \cdot \hat{\pi}_j(s,t), \hat{\pi}_j(s,t) - \epsilon'\delta'\} \end{aligned} \tag{13}$$

To minimize $UB_\ell(t)$ and maximize $LB_\ell(t)$, for $UB_\ell(t)$, we set $(1+\epsilon') \cdot \hat{\pi}_j(s,t) = \hat{\pi}_j(s,t) + \epsilon'\delta'$ and, for $LB_\ell(t)$, we set $(1-\epsilon') \cdot \hat{\pi}_j(s,t) = \hat{\pi}_j(s,t) - \epsilon'\delta'$. We can derive that when δ' is $\hat{\pi}_j(s,t)$, the two bounds can be optimized under our setting. Therefore, we have

$$\begin{aligned} UB_\ell(t) =& (1+\epsilon') \cdot \hat{\pi}_j(s,t) \\ LB_\ell(t) =& (1-\epsilon') \cdot \hat{\pi}_j(s,t) \end{aligned} \tag{14}$$

where ϵ' satisfies $\frac{2\epsilon'/3+2}{\epsilon'^2 \cdot \hat{\pi}_j(s,t)} = \frac{2\epsilon/3+2}{\epsilon^2 \cdot \sqrt{\delta \cdot \delta_{\min}}}$.

Then Theorems 3 and 4 show the correctness and the time complexity of our algorithm respectively.

THEOREM 3. *Let $v'_1, \cdots, v'_k$ be the top-k node returned by Algorithm 3, then with probability at least $1-p_f$, the returned k nodes satisfy both Equation 1 and Equation 2.*

THEOREM 4. *Let T_{FP} be the time cost of forward push and T_{RW} be the time cost of random walk. We have $O(T_{FP}) = O\left(\frac{1}{\epsilon}\sqrt{\frac{m \log(1/p'_f)}{\delta_{\min}}}\right)$, and $O(T_{RW}) = O\left(n + \frac{1}{\epsilon}\sqrt{\frac{m \log(1/p'_f)}{\delta_{\min}}}\right)$. The total time complexity is $O(T) = O\left(n + \frac{1}{\epsilon}\sqrt{\frac{m \log(1/p'_f)}{\delta_{\min}}}\right)$. When $\delta_{\min} = O(1/n)$, $\log p'_f = O(\log n)$, and $m/n = O(\log n)$, the complexity becomes $O(T) = O\left(\frac{n \cdot \log n}{\epsilon}\right)$.*

4. PARALLEL OPTIMIZATIONS

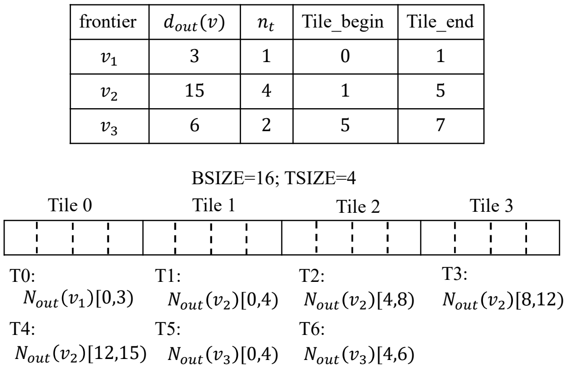
We develop several parallel optimizations for PPR processing. We identify that load-imbalance is common in the top-k query processing, due to the skewed distributions of both out-degrees and random walks, and proposed *tile-based* load balancing techniques to tackle the issue. Moreover, we explain how to generate frontiers in an efficient way.

4.1 Load-Balanced Parallel Residue Push

For different queries as well as different iterations within a query, the nodes who can become frontiers are always changing. Further, the out-degrees of frontiers vary a lot. Consequently, load-imbalance occurs frequently and it is impossible to allocate the exact number of threads needed on the fly, due to the dynamic nature of frontier generation. We propose a technique called *tile-based* residue push, to perform residue push in a more balanced manner, especially for the nodes with medium and small out-degrees (*e.g.*, < 32), which take a large portion of all the nodes. For the nodes with extremely large out-degrees, we handle them by assigning large block per frontier, following existing work [33, 34].

Given the grid-block-warp thread hierarchy on GPUs (see Section 2.2), we further partition the threads into finer subgroups, called *tiles*, using cooperative groups, a technique that allows kernels to dynamically organize groups of threads within a block. For a block, its block size BSIZE is a power of 2 and is larger than warp size WSIZE that is always 32. Denote tile size as TSIZE that is also a power of 2 and WSIZE ≤ 32. During residue push, we partition the out-neighbors of frontiers into TSIZE-sized groups, called *out-neighbor tiles*, and then allocate thread tiles to handle the corresponding out-neighbor tiles, using block-wide prefix sum.

Figure 5 exhibits an example of tile-based residue push with BSIZE 16 and TSIZE 4. The middle of the figure shows a block containing 4 tiles. There are three frontiers v_1, v_2, and v_3, with out-degrees 3, 15, 6 respectively. The out-degree of v_2 is relatively large. If each node is allocated to a block of 16 threads, for v_1 and v_3, most of the threads



frontier	$d_{out}(v)$	n_t	Tile_begin	Tile_end
v_1	3	1	0	1
v_2	15	4	1	5
v_3	6	2	5	7

Figure 5: Workload balance of tile-based residue push over three frontiers in a block with 16 threads.

would be idle; if using smaller blocks, *e.g.*, BSIZE=8, the time required by the block handling v_2 is twice as long as that of the blocks handling v_1 and v_3, which is inefficient. Therefore, we first dynamically break the out-neighbors of these nodes into out-neighbor tiles with size 4. The number of tiles that each frontier needs, $n_t(v) = \left\lceil \frac{d_{out}(v)}{\text{TSIZE}} \right\rceil$, is 1, 4, 2 respectively. Then, after a block-wide prefix sum , we can get the tile positions of each frontier (the top of the figure) and assign the out-neighbor tiles to thread tiles accordingly (lower part of the figure). The block processes out-neighbor tiles t_0 to t_3 first, and then t_4 to t_6. As we can see, although v_2 has larger out-degree than v_1 and v_3, the allocation of out-neighbor tiles are roughly balanced. This technique works well especially when there are many frontiers, since the amortized load per tile will be more balanced.

Algorithm 5 displays the pseudo code of ParallelResiduePush. Frontiers in F are assigned to blocks based on the number of tiles per block (*i.e.*, $\frac{\text{BSIZE}}{\text{TSIZE}}$). In other words, each block handles $\frac{\text{BSIZE}}{\text{TSIZE}}$ frontiers. Only the first thread of a tile calculates the number of tiles required by a frontier $n_t(v)$ (Lines 4-5). Then a block-wide prefix sum is applied to get the tile positions of the out-neighbor tiles and the total tiles required (Lines 6-8). From Lines 10 to 13, each thread tile updates the residue of all the out-neighbors in the tile, with the help of shared memory. Lines 14 to 16 find the frontiers for next iteration; there is a flag per node to ensure no frontiers are added into F_{new} more than once. atomicAdd is used at Line 13 to ensure that the residues are correctly updated in parallel. To avoid atomic operations, one can create a list of Key-Value pairs consisting of all the out-neighbor nodes and their corresponding residue increments, and then run a parallel sort and reduce to sum up the increments of the same node together, and then add the total increment to the node without atomicAdd. However, this way is inefficient [22,53,56], since (i) creating the KV list is an expensive streaming compaction with space overhead, and (ii) sorting and aggregating millions of such pairs is slow.

4.2 Load-balanced Parallel PPR Computing

Similar load-imbalance happens when computing the PPR of candidates since the start-list lengths for the nodes in IRW are also skewed. The tile-based technique introduced above can be applied as well. Algorithm 6 presents the details of ParallelComputePPR that takes as input a set of candidates whose $\hat{\pi}(s,v)$ needs to be estimated (called at Line 13 of Algorithm 3). Each block handles $\frac{\text{BSIZE}}{\text{TSIZE}}$ candidates. Then

Algorithm 5: ParallelResiduePush

```
   Input: $G$, $s$, $\alpha$, $F$, $r_{max}(v)$ for all $v \in V$
   Output: New frontiers $F_{new}$, $r(s,v)$ for all $v \in V$
1  $n_t(v) \leftarrow 0$;
2  $F_{new} \leftarrow \{\}$; $flag(u) \leftarrow 0$ for all $u \in V$;
3  parallel for each frontier $v$ assigned to block do
4      if $thread.id \,\%\, TSIZE == 0$ then
5          $n_t(v) = \left\lceil \frac{d_{out}(v)}{\text{TSIZE}} \right\rceil$;
6      synchronize;
7      [tile_begin($v$), tile_end($v$)] = PrefixSum($n_t(v)$);
8      synchronize;
9      parallel for each out-neighbor tile do
10         broadcast the out-neighbors $u$ to the thread tile via
             shared memory;
11         synchronize;
12         parallel for each $u$ in a tile do
13             atomicAdd($r(s,u), (1-\alpha) \cdot \frac{r(s,v)}{d_{out}(v)}$);
14             if $\frac{r(s,u)}{d_{out}(u)} > r_{max}(v)$ and $flag(u) = 0$ then
15                 atomically update $flag(u)$ to 1;
16                 atomically add $u$ into $F_{new}$;
17 return $F_{new}$;
```

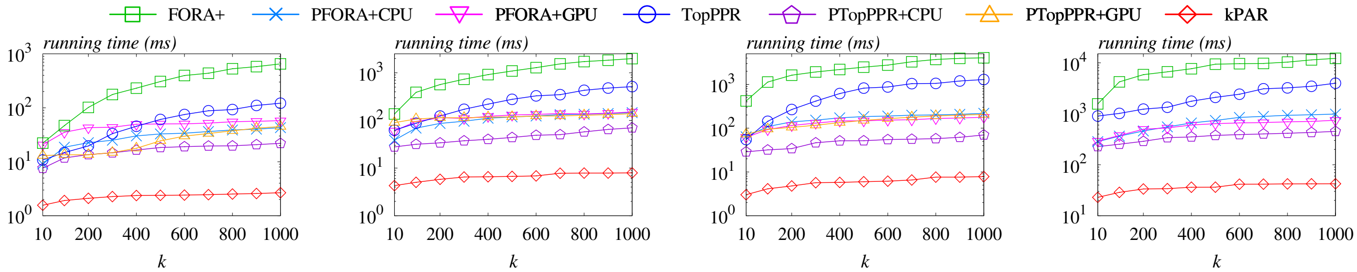
Algorithm 6: ParallelComputePPR

```
   Input: Candidate set $C_i$, $r(s,v)$ for all $v \in V$
   Output: $\hat{\pi}(s,t)$ for all $t \in C_i$
1  $n_t(t) \leftarrow 0$;
2  parallel for each candidate $t$ assigned to the block do
3      if $thread.id \,\%\, TSIZE == 0$ then
4          $n_t(t) = \left\lceil \frac{\text{start-list length of } t}{\text{TSIZE}} \right\rceil$;
5      synchronize;
6      [tile_begin($t$), tile_end($t$)] = PrefixSum($n_t(t)$);
7      synchronize;
8      parallel for each start-list tile do
9          broadcast the start-list tile to the corresponding
             thread tile via shared memory;
10         synchronize;
11         parallel for each $v$ in the start-list tile do
12             atomically add $r(s,v) \cdot \frac{\omega(v,t)}{\omega(v)}$ to a shared
                 memory variable $smem(t)$;
13         synchronize;
14         atomically add $smem(t)$ to $\hat{\pi}(s,t)$;
```

we partition the start-list of node v in IRW into tiles on the fly. Only the first thread of a tile calculates the number of tiles required by a candidate v (Lines 3-4). A block-wide prefix sum follows to get the tile positions of the start-list tiles (Lines 5-7). From Lines 8 to 14, each thread tile updates its start-list tile atomically, with the help of shared memory.

4.3 Frontier Generation

During the top-k PPR query processing, there are two places requiring frontier generation: (i) when current δ fails to find top-k and is reduced by half for next iteration (procedure ParallelGenFrontiers at Line 26 in Algorithm 3), and (ii) when new frontiers are collected for next iteration within procedure ParallelAFP (Algorithm 2). For the first case of ParallelGenFrontiers, we initialize a flag 0 per node, and then scan all the nodes in parallel to set the flag to 1 if the node is a frontier under the new parameters. Then a parallel select



(a) DBLP (b) LiveJournal (c) Orkut (d) Twitter

Figure 6: Query time for top-k PPR queries

Table 2: Datasets.

Name	n	m	Type
DBLP	613,586	1,990,159	undirected
LiveJournal	4,846,609	68,475,391	directed
Orkut	3,072,441	117,185,083	undirected
Twitter	41,652,230	1,468,365,182	directed

is performed to get all the new frontiers. For the second case, we provide two ways to get frontiers, based on the number of frontiers. During the iterations of ParallelAFP, if there are many frontiers to push or it is the first few iterations (when the number of frontiers will increase abruptly), we apply the same method as the first case. For other iterations with few frontiers, the new frontiers are collected atomically in Algorithm 5 (Lines 14-16), as explained. The reason is that, when many frontiers exist, the cost of scanning all nodes in parallel is affordable, compared with many atomic operations; otherwise, it is better to do atomic frontier collection.

5. EXPERIMENTS

We evaluate our method against states of the art, and their parallel versions on CPUs and GPUs. All experiments are conducted on a Linux machine with 80 threads powered by two 20-core Intel Xeon(R) E5-2698 v4@2.20GHz CPUs, 500GB memory, and a Tesla P100-SXM2-16GB GPU. According to their retail prices when launched in 2016, the two CPUs cost US$ 7398.00, while the GPU costs US$ 9428.00.

5.1 Experimental Settings

Methods. We compare kPAR against FORA+ and its parallel versions on CPUs and GPUs, dubbed as PFORA+CPU and PFORA+GPU. TopPPR and its parallel versions on CPUs and GPUs, PTopPPR+CPU and PTopPPR+GPU, are also evaluated. We obtain the single-threaded CPU codes of FORA+ from [2] and TopPPR from [3]. For CPU-parallelized competitors (PFORA+CPU and PTopPPR+CPU), following their single-threaded code design, we use parallel programming libraries Cilk Plus [46] and OpenMP [16] to parallelize all possible logic (e.g., while and for clauses) and computations of their whole process. Further, we use concurrent structures to replace single-threaded ones whenever possible. PFORA+GPU and PTopPPR+GPU are implemented using Nvidia CUDA. Specifically, we use arrays to replace the structures that are inefficient on GPUs, including sets, maps, and queues; we further translate all of their computations to fit the GPU-based parallel environment. We implement these GPU-based competitors with our best effort, which are all significantly faster than their respective single-threaded CPU versions. Our method kPAR and the GPU-based competitors are implemented and compiled using Nvidia CUDA 10.0 with -O3 flag. All the CPU-based methods are implemented in C++ and compiled using GCC 7.3 with -O3 flag.

Parameter Settings. Following previous work [51,52], we set α to 0.2, p_f to $1/n$, and ϵ to 0.5. Since top-k queries are only interested in the top-k nodes with large PPR and k is usually in hundreds ($k \ll n$), we set $\delta_{\min}$ to $16/n$ for FORA+ competitors and our method. For TopPPR-based methods, we follow its default settings [54] and set precision parameter $p = 0.99$. We set $\delta_{\text{init}} = 1/(10k \log n)$, $L = 10$, $\tau = 0.5\delta$. The extra space budget for IRW index is 20% since space budget larger than 20% does not improve speed much. We set block size = 512 for all GPU methods.

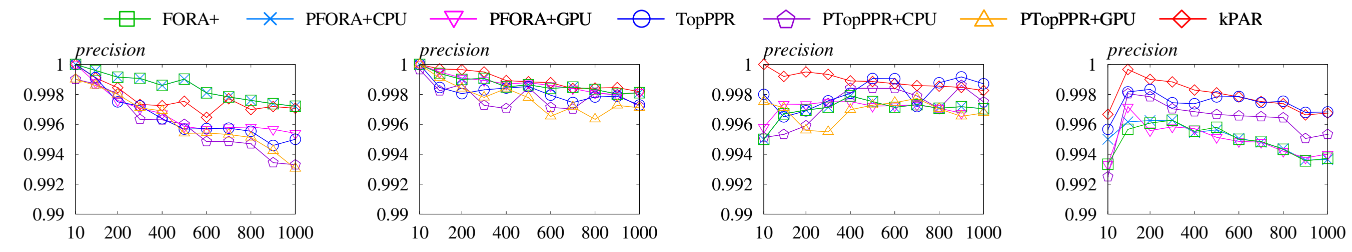
Datasets and Query Sets. We use 4 benchmark datasets that are used in previous work [37,51,52,54] (see Table 2). The datasets are from public sources [4,5]. *LiveJournal*, *Orkut* and *Twitter* are social networks, whereas *DBLP* is a collaboration network. All graphs are stored in Compressed Sparse Row (CSR) format on GPUs [22,34]. For each dataset, we randomly generate 100 query nodes. For the query nodes, we apply *Power Method* [45] with 100 iterations to compute the ground-truth PPR values of all the nodes with respect to each query node. The ground-truth PPR values have at most 10^{-10} absolute error. For each query node, we use the k nodes with the highest ground-truth PPR values as the ground truth for the top-k PPR query. Following [52], we set $k = 10, 100, 200, \cdots, 1000$.

5.2 Query Efficiency

We first evaluate the query time of all methods, and then present the performance gain brought by each of our proposed techniques.

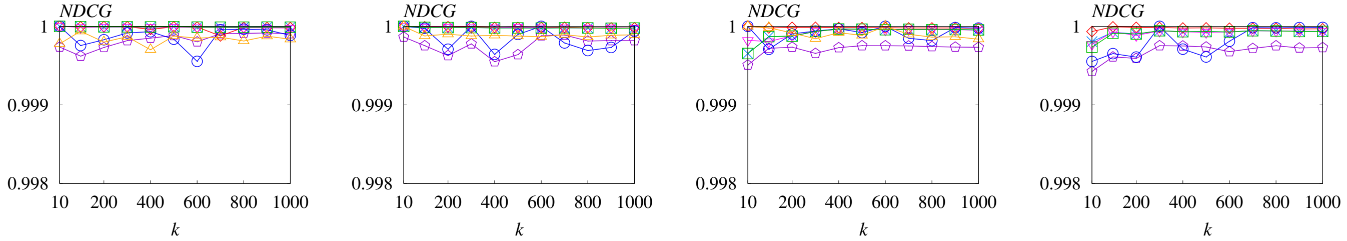
5.2.1 Query Time

Figure 6 reports the average query time of each method on all datasets. The y-axis is in log-scale and in milliseconds (ms). kPAR outperforms the competitors by 1 to 3 orders of magnitude over all datasets. In Figure 6(d) of Twitter, kPAR can answer a top-1000 query in $42.4ms$ on average, while the fastest competitor PTopPPR+CPU needs $451ms$. PTopPPR+GPU on Twitter is not reported since it requires 24GB memory, which exceeds the amount of video memory available on our GPU, i.e., 16GB. Note that the CPU-parallelized PTopPPR+CPU and PFORA+CPU run on a powerful machine with 80 threads and are at least 10 times faster than their single-threaded versions, indicating that the comparison is fair. PFORA+GPU is no faster than PFORA+CPU and similar situation holds for PTopPPR+GPU



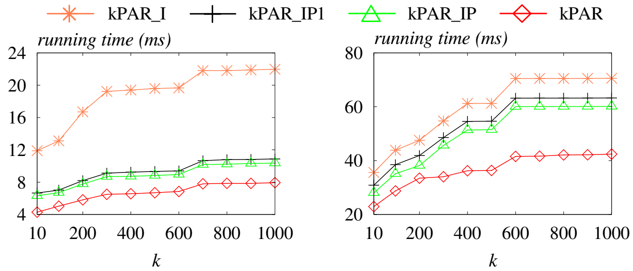
(a) DBLP (b) LiveJournal (c) Orkut (d) Twitter

Figure 7: Precision for top-k PPR queries



(a) DBLP (b) LiveJournal (c) Orkut (d) Twitter

Figure 8: NDCG for top-k PPR queries



(a) LiveJournal (b) Twitter

Figure 9: Query time of the proposed techniques

and PTopPPR+CPU, indicating that FORA+ and TopPPR are designed for CPUs, and translating them to GPUs is inefficient. For larger k, the competitors require significantly more time, but the performance of kPAR is quite stable, indicating that kPAR exposes good parallelism.

5.2.2 *Query Time Breakdown*

We evaluate the performance gain brought by each of our proposed techniques, including AFP (Section 3.2), IRW (Section 3.3) and parallel optimizations (Section 4). Since IRW is the core of our algorithm kPAR in Section 3.4, we start with kPAR_I, the algorithm with *only* IRW and its related techniques in Section 3.4 enabled. The techniques in Sections 3.4.1-3.4.3 are included in kPAR_I and cannot be evaluated separately. Then we have kPAR_IP1 that is kPAR_I augmented with the first parallel optimization in Section 4.1, and kPAR_IP that further enables the second optimization in Section 4.2. Finally, kPAR is kPAR_IP augmented with AFP (i.e., all techniques enabled).

As shown in Figure 9, each of our techniques reduces query time, on LiveJournal and Twitter datasets (See technical report [6] for other datasets) . In Figure 9(b), kPAR_I can already use only 70.56ms to answer a top-1000 query on Twitter, about 6 times faster than the fastest competitor PTopPPR+CPU that needs 451ms (see Figure 6). Then kPAR_IP1 with the parallel optimization in Section 4.1 improves the top-1000 query time to 63.3ms, and kPAR_IP with the parallel optimization in Section 4.2 further improves it to 60.1ms. The gain of Section 4.2 is large on large datasets, while modest on small datasets. Finally, with AFP enabled, kPAR takes only 42.4ms to answer a top-1000 query on Twitter. All techniques offer speedup, especially on large graphs.

5.3 Query Accuracy

We measure the accuracy of each method using two classic metrics for evaluating ranking results: *precision* and *Normalized Discounted Cumulative Gain* (*NDCG*) [27], which are used in previous work [54]. Given source node s, let $V_k = \{v_1, \ldots, v_k\}$ be the set of its ground-truth top-k nodes, and $V'_k = \{v'_1, \ldots, v'_k\}$ be the set of its top-k nodes reported by the method to be evaluated. The precision of V'_k is defined as $|V_k \cap V'_k|/k$, *i.e.*, the fraction of ground-truth top-k nodes in V'_k. The NDCG of V'_k evaluates if the returned top-k nodes having the correct order, w.r.t., the ground truth. Specifically, $\text{NDCG}(V'_k) = (\sum_{i=1}^{k} \frac{2^{\pi(s,v'_i)}-1}{\log(i+1)})/(\sum_{i=1}^{k} \frac{2^{\pi(s,v_i)}-1}{\log(i+1)})$.

Figures 7 and 8 show the precision and NDCG values of all methods on all datasets, except PTopPPR+GPU on Twitter, which runs out of GPU memory as explained before. Generally, all the methods achieve high precision and NDCG. In particular, kPAR has the best precision under almost all the k settings over all the datasets, except DBLP. kPAR has quite stable NDCGs that are nearly 1 under all settings, indicating that kPAR can correctly order the top-k nodes. The NDCGs of TopPPR and its paralleled versions are slightly unstable when varying k because TopPPR does not provide guarantee about the order of the returned nodes. When transferring FORA+ and TopPPR from single-threaded to parallel (and from CPUs to GPUs), the guarantees of these methods are unchanged, but the actual accuracy scores vary a little, due to the multi-threading environments and new hardware characteristics [55].

5.4 Preprocessing Time and Space Overhead

All the precomputation are done using single thread, for the ease of comparison. Table 3 shows the preprocessing time and space consumption of the index-based methods,

Table 3: Preprocessing costs and space overheads.

Datasets	Preprocessing time (sec)		Space overhead	
	FORA+	kPAR	FORA+	kPAR
DBLP	1.23	2.16	30.0MB	38.5MB
LiveJournal	34.51	56.69	348.2MB	421.6MB
Orkut	65.23	111.28	469.7MB	532.2MB
Twitter	919.3	1753.78	4.75GB	6.92GB

FORA+ and kPAR. Compared to FORA+, kPAR uses moderately more space and time for pre-processing, which is a cost worth paying for considering the latter's significant sppedup during query time. Specifically, preprocessing in kPAR includes IRW construction and global PageRank computation, required by AFP; meanwhile, the techniques in Section 3.4 also require some memory space. Overall, the preprocessing cost of kPAR is still within affordable range.

6. RELATED WORK

There are many studies on various types of PPR queries [11–13, 17–24, 28, 29, 35–37, 39, 40, 47, 48, 51, 52, 57–59]. The studies on approximate top-k PPR queries [35–37, 51, 52] are most relevant to ours. BiPPR [36] combines Backward Search [8] and Monte-Carlo together to answer point-to-point PPR queries and top-k queries. HubPPR [51] designs advanced indexing scheme for BiPPR. Wang *et al.* [52] propose FORA+ for top-k PPR queries with accuracy guarantees over PPR scores. FORA+ shows superior performance among all these methods. We provide the same guarantees as FORA+. TopPPR [54] guarantees the precision of top-k PPR results, which is different from ours. All these top-k PPR methods are designed for CPUs, and extending them to GPUs is non-trivial, as discussed in Section 2.3.

There are also methods [14, 29, 40, 48, 52, 59] for single-source PPR queries (SSPPR), which ask for the PPR value of every node $v \in V$ with respect to a query node s. As shown in the experiments of [52], there is a huge query time gap between answering the SSPPR query and the top-k PPR query of a node s, since usually $k \ll |V|$. Specifically, on Twitter, a top-500 PPR query can be answered in 7.7 seconds, but its corresponding SSPPR query costs 103.1 seconds. We focus on top-k PPR queries due to its widespread adoption in real applications [24, 32].

Distributed PPR algorithms [12, 21, 31] have also been extensively studied. These studies are orthogonal to our work, because (i) we aim to answer a top-k PPR query on a single commodity GPU machine, rather than in a computing cluster; (ii) the architecture of a distributed cluster is clearly different from that of a single GPU; (iii) these methods are all based on CPUs, which are non-trivial to extend to GPUs.

There also exist PPR-related studies on GPUs. Guo *et al.* [22] focus on dynamic PPR value updates, rather than answering top-k PPR queries. Techniques for sparse matrix vector multiplication on GPUs [10, 56] can be used by power iteration [45] for global PageRank computation [7, 53], which could potentially be adapted to SSPPR queries. However, it is unclear how to extend these methods to top-k PPR queries with $k \ll |V|$; further, matrix-based solutions tend to incur significant space and/or time costs on large graphs [54].

Many other graph algorithms have been developed on GPUs. Harish *et al.* [25] propose GPU-based algorithms for breadth-first search (BFS), single-source and all-pair shortest path. Following this, many GPU-based BFS algorithms are proposed [33, 34, 38, 41]. Enterprise [33] classifies BFS frontiers based on out-degrees, for better load-balance; iBFS [34] handles a batch of BFSs by sharing computation. These work are orthogonal to the problem studied in this paper.

7. CONCLUSION

This paper presents kPAR, an efficient algorithm for approximate top-k PPR queries on GPUs. Our contributions include both algorithmic designs on GPUs (AFP and IRW in Section 3) and system engineering on GPUs (parallel optimizations in Section 4). Extensive experiments demonstrate that kPAR outperforms existing solutions by a large margin, and answers top-k PPR queries on billion-edge graphs using just tens of milliseconds. We plan to consider PPR queries on multi-GPUs, as well as dynamic index updates.

8. ACKNOWLEDGMENTS

This work is supported by the National University of Singapore under SUG grant R-252-000-686-133. This publication was made possible by NPRP grant #NPRP10-0208-170408 from the Qatar National Research Fund (a member of Qatar Foundation). The findings herein reflect the work, and are solely the responsibility, of the authors.

APPENDIX

Proof of Lemma 1. Let $b_i = \frac{r(s,v_i)}{w(v_i)}$ for any $v_i \in V$. We exclude all the b_i that are 0, *i.e.*, $r(s, v_i) = 0$. Let $a_j = b_i$ if the j-th random walk starts from v_i. Among all the ω random walks, we exclude all those starting from nodes with zero residue, and get ω' walks remaining. Let X_j be a random variable that equals to 1 if the j-th random walk terminates at t, and 0 otherwise. Define $Y' = \sum_{j=1}^{\omega'} a_j X_j$, and $\nu = \sum_{j=1}^{\omega'} a_j^2 p_j$. Let $a = \max\{a_1, \cdots, a_{\omega'}\}$. Given $\rho = \sqrt{\delta \cdot \delta_{\min}}/\psi$, $a \le \rho$. Then $\sum_{j=1}^{w'} (a_j^2 p_j) \le a \sum_{j=1}^{w'} (a_j \cdot p_j) \le \rho \sum_j (a_j \cdot p_j) \le \rho \cdot \pi(s,t)$. Applying Theorem 2 and $\pi(s,t) \ge \delta'$, we have $\Pr[|Y' - \mathbb{E}[Y']| \ge \lambda] \le 2\exp\left(-\frac{\lambda^2}{2(\rho\pi(s,t))+2\rho\lambda/3}\right)$. Since $\mathbb{E}[Y'] = \sum_{v_i} r(s, v_i)\pi(v_i, t)$, $|Y' - \mathbb{E}[Y']| = |\pi(s,t) - \hat{\pi}(s,t)|$. Let $\lambda = \epsilon'\pi(s,t)$. We have,

$$\begin{aligned}
&\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon'\pi(s,t)] \\
&\le 2 \cdot \exp\left(-\frac{\epsilon'^2 \pi(s,t)^2}{2(\rho\pi(s,t)) + 2\rho\epsilon'\pi(s,t)/3}\right) \\
&\le 2 \cdot \exp\left(-\frac{\epsilon'^2 \cdot \delta'}{\rho(2 + 2\epsilon'/3)}\right) \\
&\le 2 \cdot \exp\left(-\frac{\epsilon'^2 \cdot \delta'}{2 + 2\epsilon'/3} \cdot \frac{(2\epsilon/3 + 2) \cdot \log(2/p'_f)}{\epsilon^2 \cdot \sqrt{\delta \cdot \delta_{\min}}}\right) \le p'_f.
\end{aligned}$$

For second part, $\sum_{j=1}^{w'} a_j^2 p_j \le \rho\pi(s,t) \le \delta'\rho$. By Theorem 2, we have $\Pr[|Y' - \mathbb{E}[Y']| \ge \lambda] \le 2\exp\left(-\frac{\lambda^2}{2(\rho\delta')+2\rho\lambda/3}\right)$. Then,

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon'\delta'] \le 2 \cdot \exp\left(-\frac{\epsilon'^2 \delta'^2}{2(\rho\delta') + 2\rho\epsilon'\delta'/3}\right)$$

$\le 2 \cdot \exp\left(-\frac{\epsilon'^2 \cdot \delta'}{2+2\epsilon'/3} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p'_f)}{\epsilon^2 \cdot \sqrt{\delta \cdot \delta_{\min}}}\right) \le p'_f$. $\square$

Proofs of Theorems 3 and 4. Due to space constraints, the proofs are presented in our technical report [6].

9. REFERENCES

[1] https://www.nngroup.com/articles/response-times-3-important-limits/.

[2] https://github.com/wangsibovictor/fora.

[3] https://github.com/wzskytop/TopPPR.

[4] http://snap.stanford.edu/data.

[5] http://law.di.unimi.it/datasets.php.

[6] https://sites.google.com/view/kpar-tr.

[7] https://developer.nvidia.com/nvgraph.

[8] R. Andersen, C. Borgs, J. T. Chayes, J. E. Hopcroft, V. S. Mirrokni, and S. Teng. Local computation of pagerank contributions. In *WAW*, pages 150–165, 2007.

[9] R. Andersen, F. R. K. Chung, and K. J. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.

[10] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *SC*, pages 781–792, 2014.

[11] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *WSDM*, pages 635–644, 2011.

[12] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, pages 973–984, 2011.

[13] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *PVLDB*, 4(3):173–184, 2010.

[14] S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007.

[15] F. R. K. Chung and L. Lu. Survey: Concentration inequalities and martingale inequalities: A survey. *Internet Mathematics*, 3(1):79–127, 2006.

[16] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *CiSE*, pages 46–55, 1998.

[17] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[18] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, T. Mishima, and M. Onizuka. Efficient ad-hoc search for personalized pagerank. In *SIGMOD*, pages 445–456, 2013.

[19] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, T. Mishima, and M. Onizuka. Fast and exact top-k algorithm for pagerank. In *AAAI*, 2013.

[20] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *KDD*, pages 15–23, 2012.

[21] T. Guo, X. Cao, G. Cong, J. Lu, and X. Lin. Distributed algorithms on exact personalized pagerank. In *SIGMOD*, pages 479–494, 2017.

[22] W. Guo, Y. Li, M. Sha, and K.-L. Tan. Parallel personalized pagerank on dynamic graphs. *PVLDB*, 11(1):93–106, 2017.

[23] M. S. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for topk personalized pagerank queries. In *WWW*, pages 1225–1226, 2008.

[24] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *WWW*, pages 505–514, 2013.

[25] P. Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[26] G. Iván and V. Grolmusz. When the web meets the cell: using personalized pagerank for analyzing protein interaction networks. *Bioinformatics*, pages 405–407, 2011.

[27] K. Järvelin and J. Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *SIGIR*, pages 41–48, 2000.

[28] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

[29] J. Jung, N. Park, L. Sael, and U. Kang. Bepi: Fast and memory-efficient method for billion-scale random walk with restart. In *SIGMOD*, pages 789–804, 2017.

[30] J. Klicpera, A. Bojchevski, and S. Gnnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *ICLR*, 2019.

[31] W. Lin. Distributed algorithms for fully personalized pagerank on large graphs. In *WWW*, 2019.

[32] D. C. Liu, S. Rogers, R. Shiau, D. Kislyuk, K. C. Ma, Z. Zhong, J. Liu, and Y. Jing. Related pins at pinterest: The evolution of a real-world recommender system. In *WWW Companion*, pages 583–592, 2017.

[33] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpus. In *SC*, pages 1–12, 2015.

[34] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In *SIGMOD*, pages 403–416, 2016.

[35] P. Lofgren, S. Banerjee, and A. Goel. Bidirectional pagerank estimation: From average-case to worst-case. In *WAW 2015*, pages 164–176, 2015.

[36] P. Lofgren, S. Banerjee, and A. Goel. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*, pages 163–172, 2016.

[37] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*, pages 1436–1445, 2014.

[38] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *DAC*, pages 52–55, 2010.

[39] S. Luo, X. Xiao, W. Lin, and B. Kao. Efficient batch one-hop personalized pageranks. In *ICDE*, pages 245–256, 2019.

[40] T. Maehara, T. Akiba, Y. Iwata, and K.-i. Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *PVLDB*, 7(12):1023–1034, 2014.

[41] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *PPOPP*, pages 117–128, 2012.

[42] D. Merrill, M. Garland, and A. S. Grimshaw. High-performance and scalable GPU graph traversal. *TOPC*, pages 14:1–14:30, 2015.

[43] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SIGOPS*, pages 456–471, 2013.

[44] J. Nielsen. Usability engineering. In *The Computer Science and Engineering Handbook*, pages 1440–1460. 1997.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[46] A. D. Robison. Composable parallel patterns with intel cilk plus. *CiSE*, page 66, 2013.

[47] A. D. Sarma, A. R. Molla, G. Pandurangan, and E. Upfal. Fast distributed pagerank computation. In *ICDCN*, pages 11–26, 2013.

[48] K. Shin, J. Jung, L. Sael, and U. Kang. BEAR: block elimination approach for random walk with restart on large graphs. In *SIGMOD*, pages 1571–1585, 2015.

[49] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[50] A. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, pages 127 – 128, 1974.

[51] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li. Hubppr: Effective indexing for approximate personalized pagerank. *PVLDB*, 10(3):205–216, 2016.

[52] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang. Fora: Simple and effective approximate single-source personalized pagerank. In *KDD*, pages 505–514, 2017.

[53] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP*, pages 11:1–11:12, 2016.

[54] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J.-R. Wen. Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In *SIGMOD*, pages 441–456, 2018.

[55] N. Whitehead and A. Fit-florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus, 2011.

[56] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *PVLDB*, 4(4):231–242, 2011.

[57] W. Yu and X. Lin. IRWR: incremental random walk with restart. In *SIGIR*, pages 1017–1020, 2013.

[58] H. Zhang, P. Lofgren, and A. Goel. Approximate personalized pagerank on dynamic graphs. In *KDD*, pages 1315–1324, 2016.

[59] F. Zhu, Y. Fang, K. C. Chang, and J. Ying. Incremental and accuracy-aware personalized pagerank through scheduled approximation. *PVLDB*, 6(6):481–492, 2013.