The VLDB Journal (2019) 28:923–940
https://doi.org/10.1007/s00778-019-00576-7

REGULAR PAPER

Parallelizing approximate single-source personalized PageRank queries on shared memory

Runhui Wang[1] · Sibo Wang[2] · Xiaofang Zhou[1]

Received: 23 February 2019 / Revised: 20 August 2019 / Accepted: 21 September 2019 / Published online: 8 October 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract
Given a directed graph G, a source node s, and a target node t, the personalized PageRank (PPR) $\pi(s,t)$ measures the importance of node t with respect to node s. In this work, we study the single-source PPR query, which takes a source node s as input and outputs the PPR values of all nodes in G with respect to s. The single-source PPR query finds many important applications, e.g., community detection and recommendation. Deriving the exact answers for single-source PPR queries is prohibitive, so most existing work focuses on approximate solutions. Nevertheless, existing approximate solutions are still inefficient, and it is challenging to compute single-source PPR queries efficiently for online applications. This motivates us to devise efficient parallel algorithms running on shared-memory multi-core systems. In this work, we present how to efficiently parallelize the state-of-the-art index-based solution *FORA*, and theoretically analyze the complexity of the parallel algorithms. Theoretically, we prove that our proposed algorithm achieves a time complexity of $O(W/P + \log^2 n)$, where W is the time complexity of sequential *FORA* algorithm, P is the number of processors used, and n is the number of nodes in the graph. FORA includes a forward push phase and a random walk phase, and we present optimization techniques to both phases, including effective maintenance of active nodes, improving the efficiency of memory access, and cache-aware scheduling. Extensive experimental evaluation demonstrates that our solution achieves up to 37× speedup on 40 cores and 3.3× faster than alternatives on 40 cores. Moreover, the forward push alone can be used for local graph clustering, and our parallel algorithm for forward push is 4.8× faster than existing parallel alternatives.

Keywords Social networks · Personalized PageRank · Parallelism

1 Introduction

Given a directed graph G, a source node s, and a target node t, the personalized PageRank (PPR) of node t with respect to s, denoted as $\pi(s,t)$, is the probability that a random walk from s stops at node t, and indicates the importance of node t from the viewpoint of node s.

One important variant of PPR is the single-source PPR (SSPPR) query, which takes as input a source node s, and returns the PPR of each node with respect to s. The single-source PPR query finds many important applications, e.g., community detection [2,39,40] and recommendation [27,29]. Despite the importance of SSPPR queries, it is still challenging to process them efficiently for large graphs.

To derive the exact answer for an SSPPR query, it requires $O(n^{2.37})$ [28] computational cost, where n is the number of nodes in the graph and is prohibitive for large graphs. Meanwhile, it is expensive to pre-store all SSPPR query answers since it requires $O(n^2)$ space, which is infeasible for large graphs. Therefore, most existing solutions focus on approximate version of the SSPPR queries, which provides a trade-off between the running time and result quality. However, even under the approximate version, existing state-of-the-art solutions, e.g., FORA [36], are still inefficient to answer an SSPPR query for online applications. As shown by Twitter [19] and Pinterest [25], despite the fact that they use multiple machines to handle PPR queries, each machine will maintain a copy of the underlying graph for the processing of

✉ Runhui Wang
runhui.wang@uq.edu.au

Sibo Wang
swang@se.cuhk.edu.hk

Xiaofang Zhou
zxf@itee.uq.edu.au

1 University of Queensland, Brisbane, Australia

2 The Chinese University of Hong Kong, Sha Tin, Hong Kong, China

PPR queries mainly due to the widely known efficiency issue of distributed computation of graph problems. To explain, distributed algorithms usually need to communicate with other machines, which degrades the performance, and the network bandwidth becomes the bottleneck for efficiency. In contrast, the in-memory algorithms only need to access the shared main memory and are far more efficient than the distributed counterparts. Fortunately, in recent years, the fast development of multi-core CPU architectures brings the performance of single CPU chip to a new level. These motivate us to devise efficient algorithms for SSPPR queries by exploring multi-core parallelization with shared memory to boost the performance.

To the best of our knowledge, most research works of PPR computation on shared memory, e.g., [1,2,26,34,36], focus on sequential algorithms and do not consider the computing power of multi-core systems. There are few research works [17,33] on parallelizing PPR computation. Guo et al. [17] propose a parallel solution for PPR computing on dynamic graphs, focusing on updating PPRs when new edges are added into the graph. However, their solution assumes that all the forward push [2] results are available, and is impractical to support approximate SSPPR query answering. For instance, if they pre-store all the forward push results for the FORA algorithm, the space consumption is $O(n^2)$ as shown in [36], which is impractical for large graphs. What's more, since they need all the forward push results as the input, our parallel algorithm can be further used to help them reduce the prohibitive computational cost to derive all the forward push results. Shun et al. [33] extend their Ligra [31] framework to parallelize the PPR computation. However, their solution does not provide a theoretically linear speedup, and as shown in their experiment, the scalability of their proposed solution is still unsatisfactory and leaves much room for improvement.

In this paper, we present how to efficiently parallelize *FORA* for SSPPR queries and theoretically analyze the complexity of our proposed parallel algorithms. FORA consists of two phases: the *forward push* phase and the *random walk* phase [36]. The forward push phase traverses the graph from the source, iteratively proceed subsets of the vertices visited, and explore the out-neighbors of these vertices until the certain termination condition is satisfied. Next, the random walk phase of FORA samples the random walks on different nodes according to the outputs in the forward push phase and then finally outputs the estimated PPR values with approximation guarantees. However, it is non-trivial to provide theoretical linear speedup while providing superior performance to parallelize FORA.

To explain, in the first phase, it is difficult to bound the number of iterations of the forward push algorithm, and therefore challenging to provide linear speedup for this phase. Besides, we need to maintain the active nodes, i.e., the nodes to be processed in every iteration, and it is challenging to present an efficient data structure to support such an operation. Several data structures, e.g., Bag [23] and sparse set [32], are proposed to support the maintenance of the active nodes in parallel. Nevertheless, such implementations have poor cache localities since they include many random accesses. For the second phase, memory accesses are randomly issued and it is challenging to present an effective approach to reducing such random accesses and memory contentions caused by concurrent read/write. We present an efficient framework for parallel FORA, named *PAFO*, and make the following contributions:

– For the forward push phase, we present a hybrid approach to effectively maintain the active nodes and also reduce the memory access costs. Then, we present a cache-aware scheduling to further improve query performance and scalability.
– For the random walk phase, we propose an integer-counting-based method to reduce the memory access overhead and present techniques to reduce data contention based on the integer-counting-based method.
– Theoretically, we show how to bound the depth of parallel forward push and prove that PAFO achieves asymptotically linear speedup on scale-free graphs.
– Extensive experimental results show that PAFO achieves up to 37× speedup on 40 cores and 3.3× faster than alternatives on 40 cores.

2 Preliminary

2.1 Personalized PageRank

Given a directed graph $G = (V, E)$, a source node $s \in V$, and a stop probability α, a random walk from s is a traversal on the graph that starts from s and at each step it either stops at the current node with probability α, or proceeds to a randomly chosen outgoing neighbor of the current node. The personalized PageRank (PPR) of node t with respect to s, denoted as $\pi(s, t)$, is the probability that a random walk from s stops at node t, indicating the importance of t with respect to s. The single-source PPR (SSPPR) query takes a source node s as input and returns the PPR value of each node with respect to s. Solving the SSPPR query exactly is rather expensive [28] and requires $O(n^{2.37})$ computing cost. This motivates a line of research work [26,34,36] to study approximate SSPPR query, which is defined as follows.

Definition 1 (*Approximate SSPPR*) Given a source node s, a threshold δ, a relative error bound ϵ with $0 < \epsilon < 1$, and a failure probability p_f, an approximate single-source PPR (SSPPR) query returns an estimated PPR $\hat{\pi}(s, t)$ for each vertex $t \in V$, such that for any $\pi(s, t) > \delta$,

$$|\pi(s,t) - \hat{\pi}(s,t)| \le \epsilon \cdot \pi(s,t) \quad (1)$$

holds with at least $1 - p_f$ probability.

Typically, δ and p_f are set to be $O(1/n)$ [26,36]. Also, we assume that ϵ is no smaller than $1/n^2$, which is small enough to provide almost exact results for PPR scores [30].

Besides, in most applications of personalized PageRank, the underlying graphs are typically social graphs or web graphs, which are generally *scale-free*. In particular, on scale-free graphs, for any $k \ge 1$, the fraction $f(k)$ of nodes in G that have k edges satisfies that:

$$f(k) \propto k^{-\beta},$$

where β is a parameter whose value is in the range $2 < \beta < 3$. On scale-free graphs, the average degree $m/n = O(\log n)$. We will explore the property of scale-free graphs to prove the parallel time complexity of forward push phase.

2.2 Basics in parallel computing

Modeling parallel computation A popular parallel model for shared-memory parallel algorithms is the *work-depth model*, where the cost of an algorithm is determined by the total number of operations that it performs and the dependencies among these operations. Denote W as the total number of operations of an algorithm, namely the workload, and D as the longest chain of dependencies among its operations, namely the depth. The (rephrased) Brent's theorem [6] shows that the running time of a parallel algorithm can be bounded by W, D, and the number of processors P.

Theorem 1 (Brent's theorem) *For a computational task with workload W and depth D, on P processors, a greedy scheduler, which steals works from other processors when it becomes idle, achieves running time* $O(W/P + D)$.

A parallel algorithm is *work efficient* if it requires at most a constant factor more work than its sequential version. Another important factor is the depth of the parallel algorithm. When designing a parallel algorithm, the *goal* is to:

- Design work-efficient parallel algorithm.
- The depth of the algorithm should not be too large, typically within poly-logarithmic of the input size.

In this paper, we will use the work-depth model to analyze the parallel algorithms.

Atomic operations In parallel computing, concurrent reads and writes are allowed, and existing modern multi-core machines typically support atomic operations, which either successfully change the data, or have no effect at all, leaving no intermediate state. The most widely used atomic operation supported by modern CPUs is the **Compare-and-Swap** operation, which takes three arguments: a memory address, an expected value, and a new value. It compares the content of the input memory address with the expected value and, only if they are the same, modifies the contents of that memory location to the new value. If the update succeeds, it returns true, and otherwise returns false. Other atomic operations, like **Atomic-Add**, can be easily implemented with the **Compare-and-Swap** operation. In what follows, we will use **Atomic-X** to indicate that operation **X** is atomic.

2.3 State of the art

2.3.1 Sequential algorithm

The state-of-the-art sequential algorithm for approximate SSPPR is the FORA algorithm proposed in [36]. FORA processes an SSPPR query with two phases, a forward push phase and a random walk phase, on the input graph. Next, we explain the two phases of FORA and how to combine the results of the two phases.

Forward push phase The forward push phase simulates the random walk in a deterministic approach using the forward push algorithm proposed in [2]. It starts from the source s and simulates the message passing using a unit mass. It maintains two values for each node $v \in V$: a *residue* $r(s,v)$ and a *reserve* $\pi^\circ(s,v)$. The reserve $\pi^\circ(s,v)$ indicates the amount of mass that stopped at node v, and $r(s,v)$ indicates the amount of the mass that currently stays at node v. Initially, $r(s,s)$ is 1 and all the other values are zero. Then, at each step, the forward push algorithm selects a node v and does a *push* operation to the message as follows: (i) it first converts α portion of the residue $r(s,v)$ to its reserve; (ii) then it propagates the remaining message evenly to its neighbors. If we continue this process until all residues are zero, then the reserve values are exactly the PPR values. However, this incurs enormous computational costs and in [2], they propose to use a threshold $r_{\max}$ to control the computational cost. In particular, a node v is selected to do a push operation only if its residue satisfies that:

$$r(s,v)/|N^{\text{out}}(v)| > r_{\max},$$

where $N^{\text{out}}(v)$ is the set of out-neighbors of node v. With this strategy, the time complexity of the forward push algorithm can be bounded with $O\left(\frac{1}{r_{\max}}\right)$.

After any number of push operations, the following invariant holds for an arbitrary target node t:

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

It is difficult to bound $\sum_{v\in V} r(s,v)\cdot\pi(v,t)$, so FORA includes another random walk phase, which allows us to provide error bound for $\sum_{v\in V} r(s,v)\cdot\pi(v,t)$ with significantly smaller computational costs.

Random walk phase In the random walk phase, FORA mainly samples random walks according to the residue of each node. In particular, let

$$\omega = \frac{3\log(2/p_f)}{\epsilon^2\cdot\delta}, \tag{2}$$

where δ, ϵ, and p_f are the threshold, the relative error bound, and the failure probability as defined in Definition 1. Then, for each node v, it samples $\lceil r(s,v)\cdot\omega\rceil$ random walks from node v. Let X be a random walk from node v, and u be the ending node of random walk X, then we add $\frac{r(s,v)}{\lceil r(s,v)\cdot\omega\rceil}$ to the reserve of node u, i.e., $\pi^\circ(s,u)$ for this random walk X. Then, FORA repeats this process for all random walks and returns $\pi^\circ(s,v)$ for each v as the answer. The pseudo-code of FORA is shown in Algorithm 1.

Algorithm 1: Sequential FORA

```
  Input: Input graph G, source vertex s, probability α
  Output: the estimated PPR π̂(s, v) for each v ∈ V
1 Let ω = 3 log(2/p_f) / (ε^2 · δ);
2 Run forward push with input graph G, source s, probability α,
  and r_max = sqrt(1/(m·ω));
3 Let π°(s, v), r(s, v) be the output of the reserve and residue of
  each node v ∈ V;
4 for each node v ∈ V with r(s, v) ≠ 0 do
5     Let ω_i = ⌈ω · r(s, v)⌉;
6     for i = 1 to ω_i do
7         Pick an unused destination from the index stored for v,
          and let u be the destination.;
8         π°(s, u) += r(s, v)/ω_i;
9 Return π°(s, v) as the estimated PPR for each node;
```

FORA analysis Recall that the forward push phase has a time complexity of $O(\frac{1}{r_{\max}})$. To consider the complexity of the random walk phase, note that the residue of each node v is at most $\lceil|N^{\text{out}}(v)|\cdot r_{\max}\rceil$. Therefore, the total number of random walks can be bounded by:

$$\sum_{v\in V}\lceil|N^{\text{out}}(v)|\cdot r_{\max}\cdot\omega\rceil \le n + m\cdot r_{\max}\cdot\omega.$$

We assume that $m \ge n$, which typically holds for real-world graphs, then the time complexity of FORA is:

$$O\left(\frac{1}{r_{\max}} + m\cdot r_{\max}\cdot\omega\right).$$

By setting $r_{\max} = \sqrt{\frac{1}{m\cdot\omega}}$, FORA achieves the best time complexity, which is $\sqrt{m\cdot\omega}$. For each node v, FORA pre-stores the destinations of the maximum number of random walks required, i.e., $\lceil|N^{\text{out}}(v)|\cdot r_{\max}\cdot\omega\rceil$, and the total space consumption is $O(\sqrt{m\cdot\omega})$. When generating random walks from v, it directly uses the first unused destinations in the index and avoid the expensive online random walks. Since $p_f = O(1/n)$, and on scale-free graphs $m = O(n\cdot\log n)$, the time complexity of FORA is $O\left(\frac{n\cdot\log n}{\epsilon}\right)$.

2.3.2 Parallel algorithm

The state-of-the-art parallel algorithm for approximate SSPPR is a combination of the state-of-the-art parallel algorithm for forward push by Shun et al. [33] and a direct parallel solution for the random walk phase, since the random walk phase can be naturally parallelized with a parallel for loop supported by many multi-thread frameworks, e.g., CilkPlus [20], OpenMP [11]. Next, we explain the solution in [33] and show how to parallelize the random walk phase.

Parallel forward push [33]. Shun et al. extend their Ligra framework [31], a shared-memory parallel graph processing framework, to parallelize the forward push algorithm. The main intuition of Ligra is that in many graph traversal algorithms, e.g., BFS, forward push can be implemented in iterations. In each iteration, they process a subset of the vertices and update along their out-neighbors, which can be processed in parallel. In Ligra, there are two interfaces:

- *VertexMap* This function takes as input a vertex set F and an update function UF, which updates the data associated with each node in parallel.
- *EdgeMap* This function takes as an input graph G, a vertex set F, and an update function UF, which applies to the out-neighbors of the nodes in F and updates the data associated with these out-neighbors in parallel.

These two interfaces are sufficient to parallelize the forward push since in each iteration, it proceeds a set F of the vertices and updates their own residue and reserve values, which can be handled with VertexMap interface. Besides, recall that these nodes in F will propagate the masses to their out-neighbors with the push operation, which can be handled with the EdgeMap interface. Algorithm 2 shows the Ligra implementation for the forward push algorithm. In the VertexMap function, it proceeds the nodes in F in parallel, and for each node in F, it invokes the UpdateSelf procedure, which adds α portion of the residue to its reserve and reset its residue to zero (Algorithm 2 Lines 4–6). Then, it invokes

the EdgeMap function and handles the nodes in F in parallel. In particular, for each out-neighbor of $v \in F$, it invokes the UpdateNeighbor procedure, which propagates $\frac{(1-\alpha)}{|N^{\text{out}}(v)|}$ portion of $r(s, v)$ to each of its out-neighbor (Algorithm 2 Lines 1–2). Note that during the parallel processing of EdgeMap, different nodes in F may write on the residue value of the same node simultaneously. Therefore, an *Atomic-Add* is used to avoid unexpected behaviors.

Algorithm 2: Parallel forward push with Ligra

Input: Graph G, source vertex s, probability α, threshold $r_{\max}$
Output: $\pi^o(s, v), r(s, v)$ for all $v \in V$

```
1  Procedure UpdateNeighbor(v, u);
2      Atomic-Add(r'(s, u), (1−α)·r(s,v)/|N^out(v)|);
3
4  Procedure UpdateSelf(v);
5      π°(s, v) = π°(s, v) + α · r(s, v);
6      r'(s, v) = 0;
7
8  r(s, s) ← 1; r(s, v) ← 0 for all v ≠ s;
9  π^o(s, v) ← 0 for all v ∈ V ;
10 Let F = {s};
11 while F is not empty do
12     r'(s, v) = 0 for all v ∈ V;
13     VertexMap(F, UpdateSelf);
14     EdgeMap(G, F, UpdateNeighbor);
15     r(s, v) = r'(s, v) for all v ∈ V;
16     F = {v|r(s, v)/N^out(v) > r_max};
```

Parallel random walk As we can observe from Algorithm 1 Lines 4–7, the execution of the random walk of each node has no dependency on the random walks of other nodes. Therefore, the random walk phase can be naturally be parallelized by replacing Line 4 of Algorithm 1 with a parallel for and Line 8 of Algorithm 1 with an atomic operation.

Parallel cost analysis For the parallel forward push as proposed by Shun et al. [33], the total workload can be bounded by $O\left(\frac{1}{r_{\max}}\right)$, which is the same as the sequential algorithm. However, it is difficult to bound the depth D of the parallel forward push algorithm, and to our knowledge, it is still an open problem if the parallel forward push algorithm can finish in a poly-logarithmic depth of n. Therefore, theoretically, the parallel forward push algorithm can be quite inefficient due to the large depth D. For parallel random walks, it is easy to verify that the task can be finished with a workload the same as the sequential algorithm. For the depth of parallel random walks, the depth of the parallel for loop can be bounded by $O(\log n)$ depth. Meanwhile, for each node, the maximum number of random walks may reach polynomial of n. Therefore, by parallelizing random walks on such nodes, the depth of such nodes can also reach $\log n$. Combining them together, the depth can be bounded by $O(\log^2 n)$.

3 PAFO framework

In this section, we will present the details of our PAFO framework. In Sect. 3.1, we present the details of our parallel solutions to the forward push phase, followed by the parallel random walk phase in Sect. 3.2.

3.1 Parallel forward push phase

Our parallel forward push phase includes several techniques to improve the practical performance and bound the parallel time complexity. In this section, we present the optimizations and postpone the theoretical analysis to Sect. 4. Firstly, we demonstrate how to effectively maintain the active nodes in Sect. 3.1.1. Then, we demonstrate how to process push operations in a cache-friendly manner through a cache-aware scheduling in Sect. 3.1.2. At the end of Sect. 3.1.2, we further discuss under what scenarios our cache-aware scheduling can be potentially applied.

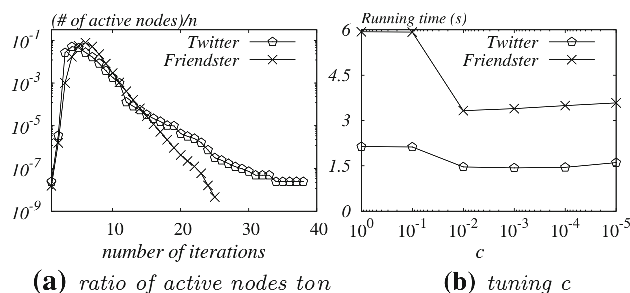
3.1.1 Hybrid approach

We first explain our hybrid approach to improve the memory efficiency when accessing active nodes in parallel. Our parallel algorithm shares the similar spirit as Algorithm 2 in that it also processes nodes by iterations. In each iteration, our forward push algorithm also proceeds the nodes

$$F = \{v|r(s, v)/N^{\text{out}}(v) > r_{\max}\},$$

in parallel and repeats this process until F becomes empty. We denote the nodes in F as *active nodes*.

In our parallel algorithm, we include two different models to maintain the active nodes. This is motivated by the observations as shown in Fig. 1a. In particular, in the first few iterations of the forward push algorithm, the number of nodes to be pushed, or simply say active nodes, is relatively small, but the size is growing sharply. When the forward push continues, e.g., when the number of iterations reaches 5, the number of active nodes in an iteration will be very large and reach $O(n)$. Then, with the further process of iterations,



(a) *ratio of active nodes to n* **(b)** *tuning c*

Fig. 1 Motivation for hybrid method (average on 20 sampled nodes)

the number of active nodes decreases sharply and size again becomes very small, and this process repeats until the size becomes zero. Moreover, as we can observe, the majority of the workload is processed in middle iterations when the number of active nodes is in the order of $O(n)$.

Main idea Motivated by the observation, we use two different maintaining mechanisms to the active nodes for the case when the number of active nodes reaches $O(n)$, denoted as *heavy workload iterations*, and when the number of active nodes is small, denoted as *light workload iterations*. In the light workload iterations, we use the *bag* structure, which will be explained in details shortly. However, the major deficiency of the bag structure is that it needs to synchronize the bag structure frequently, and the memory access in the bag structures is not very efficient since it maintains the active nodes in a relatively random fashion. Therefore, this motivates us to explore a direct scan of the entire node set when the workload shifts from the light workload to the heavy workload. With this strategy, we avoid the expensive cost to maintain the bag structure and make the memory access cache-friendly. Figure 1b shows how the parameter affects the performance, and we can see that when we choose $n/100$ as the boundary, it achieves the best practical performance. Therefore, we set $c = 1/100$ as the default value in our experiment. Also, the major computational cost of this hybrid forward push algorithm actually mainly comes from the heavy workload iterations, and by splitting these two cases, we can focus the optimization to the heavy workload iterations, as will be explained in Sect. 3.1.2.

Light workload iterations For the light workload iterations, we use the *bag* structure, which is initially proposed in [23] for parallel BFS and has the following property:

Property 1 (Bag) *The bag data structure provides two efficient operations to support high parallelism.*

- *Insert The insert operation takes $O(1)$ amortized time.*
- *Split The split operation, which divides elements in the bag into two bags with roughly equal size, takes $O(\log x)$ time, where x is the number of elements in the bag.*

Next, we explain how we use the bag structure in our hybrid approach. As shown in Algorithm 3, initially, the source node is added into the bag structure B. Then, when the number of active nodes is smaller than $c \cdot |V|$, where c is a small constant to split the heavy workload and light workload, we use the bag structure to handle the nodes in parallel. In particular, it first recursively divides the Bag B into smaller bags of which the sizes are under the specified threshold *grainsize*[1] (Algorithm 3 Line 4), and then process the bags in parallel

Algorithm 3: PAFO hybrid forward push

```
Input: Graph G, source vertex s, probability α, threshold r_max
Output: π°(v, t), r(v, t) for all v ∈ V
1  r(s, v), π°(s, v) ← 0 for all v ∈ V; r(s, s) ← 1;
2  Bag-create(B, {s}); Bag-create(B′, ∅); F ← |B|;
3  while F < c · |V| do
4      recursively split B into small bags with constant size; denote
       𝓑 be the set of the small bags;
5      Parallel for each X ∈ 𝓑 do
6          for each v ∈ X do
7              r_v = r(s, v); π°(s, v) ← π°(s, v) + r_v · α;
8              for each i ∈ N^out(v) do
9                  r_i ← r(s, i);
10                 Atomic-Add(r(s, i), (1−α)·r_v / |N^out(V)|);
11                 if r(s, i) ≥ r_max |N^out(i)| > r_i then
12                     Bag-insert(B′, i);
13             Atomic-Add(r(s, v), −r_v);
14     B ← B′, B′ ← ∅; F ← |B|;
15 while F ≥ c · |V| do
16     F ← 0 ;
17     Parallel for each v ∈ V do
18         if r(s, v) ≥ r_max · |N^out(v)| then
19             repeat Lines 7–10 and 13;
20             F ← F + 1;
21 B ← ∅;
22 Parallel for each v ∈ V do
23     if r(s, v) ≥ r_max · |N^out(v)| then
24         Bag-insert(B, v);
25 Repeat Lines 3–24 until F becomes zero;
26 return π°(s, v), r(s, v) for each v ∈ V;
```

(Lines 5–13). This helps to balance the workload on different processors and guarantee that each processor will have sufficient work to do. As we can see, to use the bag structure, it needs to insert the active nodes in the next iteration into the bag structure. Also, since a node may be touched by multiple active nodes in this iteration, duplicates may be inserted into the bag structure, which should be avoided. To handle this case, a condition checking is added to identify whether the node should be added into the bag or not (Algorithm 3 Lines 11–12). To explain, since residues are added with atomic operations, it behaves like sequential operations, and in a sequence of additions to $r(s, i)$, only one addition will make $r(s, i)$ satisfy the condition in Line 11. As we can see, to maintain the bag structure, we need to do synchronization on the bag structure, which brings additional costs. This cost will be unnecessarily high when the number of nodes in the bag reaches $O(n)$. Therefore, for heavy workload iterations, we use a mechanism with far less synchronization cost, which is more effective without increasing the time complexity.

Heavy workload iterations In the heavy workload iterations, we directly scan the entire node set in the graph to find the active nodes (Algorithm 3 Lines 15–20). Since the number of

[1] The *grainsize* is set to 128 according to Leiserson et al. [23].

active nodes is $O(n)$, the scan cost does not increase the time complexity. In the heavy workload iterations, all the nodes are processed in parallel in a way that if this node is active, a push operation is processed on this node, and otherwise ignored. Notice that to verify if a node v is active or not, we can directly check if $r(s, v)/N^{\text{out}}(v) > r_{\max}$ or not (Algorithm 3 Lines 18). This direct parallel scan avoids the synchronization cost to maintain the active nodes and also processes the nodes in a cache-friendly manner since the nodes are processed sequentially in parallel. Also, instead of counting the number of active nodes in the next iteration, we count the number of active nodes in the current iteration (Algorithm 3 Line 20), which helps avoid the checking cost as shown in Algorithm 3 Lines 11–12. With this strategy, we may do one light workload iteration using the parallel scan. However, this cost will be still $O(n)$ and can be bounded by previous scanning iterations. Finally, when the workload becomes light workload, it rebuilds the bag structure as shown in Algorithm 3 Lines 22–24 and repeats the above two phases until the algorithm stops. Also, we note that for Algorithm 3 Line 20, a local copy of F is maintained in each thread and aggregated when the parallel for loop finishes. This avoids the contention on F and at the same time preserves the correctness.

In terms of the choice of c, we set c to be $1/100$ according to the experimental results as shown in Fig. 1. Besides, according to our evaluation, around 90% of the time is spent on heavy workload iterations in the forward push phase. Therefore, we propose several optimizations for the heavy workload iterations, which will be explained shortly.

Remark Beamer et al. [5] also propose a hybrid solution for breadth-first search (BFS) on scale-free graphs, which can reduce the number of edges examined by a large scale. However, our hybrid solution differs from the ones in [5] in two aspects. First, our solution works on the forward push algorithm, whose time complexity does not depend on the number of edges. Therefore, the hybrid approach proposed in [5] will not help on the forward push algorithm. Second, our hybrid solution does not maintain the active nodes in heavy workload iterations in order to improve memory efficiency, while the hybrid solution proposed in [5] maintains the active nodes all the time.

3.1.2 Cache-aware scheduling

Rationale In our heavy workload iterations, it applies a direct scan of all the nodes and proceeds a push operation to a node if it is active. Such a process is handled in parallel, and which core will handle which part of the task entirely depends on the default schedule, which may not be cache-friendly at all. For instance, assume that a processor p_x is processing a node v_i, then all the out-neighbors of v_i and the residue array of the out-neighbors are also loaded into the L_1 or L_2 cache of core p_x. Then, suppose after the processing of v_i, another node v_j is immediately dispatched to p_x to process. In this case, if v_j shares no common out-neighbors with v_i at all, then the cache stored on p_x becomes utterly useless and another load process is required. However, if v_j shares most of the out-neighbors of v_j, then most of the content will be already in the L_1 or L_2 cache of p_x, and the processing can benefit a lot from the existing cache content. Besides, when two processors are concurrently updating the residue of the nodes on the same cache line, cache contention happens, resulting in L1 cache stalls, which incurs additional costs. This motivates us to propose a cache-aware scheduling for heavy workload iterations.

Quantify cache misses and cache contentions In most scheduling, e.g., CilkPlus default scheduling, tasks are divided into smaller tasks of small grain sizes. We follow this paradigm and group g nodes with consecutive IDs into a task. Then, the goal of the schedule is to provide a schedule such that some grain-size tasks $g_{1,1}, g_{1,2}, \ldots$ are assigned to core 1, some grain-size tasks $g_{2,1}, g_{2,2}, \ldots$ are assigned to core 2, …, and some grain-size tasks $g_{P,1}, g_{P,2}, \ldots$ are assigned to core P that are aware of the cache contention and cache misses. Let c_{m} be the penalty of a cache miss and c_{c} be the penalty of cache contention. Let x be the total number of cache misses and y be the total number of cache contentions during the processing of all these grain-size tasks. Then, the goal of the scheduling is to minimize $c_{\text{m}} \cdot x + c_{\text{c}} \cdot y$. However, it is rather challenging to track the execution time of each task since it highly depends on the source node, and it is actually difficult to quantify the cache contention and cache misses. Next, we first explain how to quantify the cache contention and cache misses between two grain-size tasks.

Denote B as the number of update data that can be fitted into a cache line. For instance, in forward push, the update will touch the residue array of the out-neighbors. Therefore, the size is 8 bytes and assume that a cache line is 64 bytes, we can put 8 residue data into a cache line and B is therefore 8. We map the nodes in V to numbers from 0 to $|V| - 1$ and then divide the residue array into $\lceil n/B \rceil$ disjoint sets: $R_1, R_2, \ldots, R_{\lceil n/B \rceil}$, such that each set includes nodes with ids in $[B \cdot i, B(i + 1))$. We denote each such R_i as a cache line base, using i for the ease of exposition. Then, every time to load part of the residue array into the cache line, we load some set R_i $0 \leq i < \lceil n/B \rceil$. Then, for each task g_i, we can get the set $C(g_i)$ of cache line bases that will be loaded into cache, which is:

$$C(g_i) = \{R_i | N^{\text{out}}(v_j) \in R_i \text{ where } v_j \text{ is in task } g_i\}.$$

Then, the number of cache lines that g_i will occupy is $|C(g_i)|$. For two tasks g_i and g_j, we define the cache overlap score of two tasks g_i and g_j as

$$O(g_i, g_j) = |C(g_i) \cap C(g_j)|.$$

The cache overlap score is then a good indicator of the cache contention of the tasks among different cores, and the cache misses of the tasks in the same core. In particular, if two tasks g_i and g_j are processed in parallel, then the smaller the cache overlap score it is, the less cache contention will be caused by these two tasks. In the meantime, if g_i and g_j are processed consecutively in the same core, the higher the cache overlap score is, the more cache line can be reused, and the less number of cache misses it will cause.

Now assume that each grain-size task takes the same amount time to finish, and each core i has $h = \lceil \frac{n}{g \cdot P} \rceil$ grain-size tasks denoted as $g_{i,1}, g_{i,2}, \ldots, g_{i,h}$. Then, we denote the cache miss score of core i as:

$$CM(i) = \sum_{\beta=2}^{h} (|C(g_{i,\beta})| - O(g_{i,\beta}, g_{i,\beta-1})).$$

We further define the contention score of the jth parallel tasks $g_{1,j}, g_{2,j} \ldots, g_{P,j}$ as:

$$CC(i, j) = \sum_{1 \le \beta < i \le P} O(g_{\beta,j}, g_{i,j}).$$

Then, we formalize our cache-aware scheduling as the following optimization problem.

Definition 2 (*Cache-aware scheduling*) The cache-aware scheduling aims to find a schedule that minimizes

$$c_{\mathrm{m}} \cdot \sum_{i=1}^{P} CM(i) + c_{\mathrm{c}} \sum_{i=1}^{P} \sum_{j=1}^{h} CC(i, j),$$

i.e., the total penalty of the cache misses and cache contentions during the task processing.

However, the number of possible scheduling options is exponential, which will incur prohibitive processing time to derive the optimal solution. Therefore, we present a greedy approach, which aims to minimize the contention and maximize the cache locality in each iteration. In particular, we first assign a task $g_{1,j}$ to core 1 such that its cache miss penalty over the previous task of core 1 is minimum. Then, we assign a task $g_{2,j}$ for the second core such that

$$c_{\mathrm{m}} \cdot (C(g_{1,j}) - O(g_{2,j-1}, g_{2,j})) + c_{\mathrm{c}} \cdot CC(2, j)$$

is minimized among all possible tasks. Note that the first part $c_{\mathrm{m}} \cdot (C(g_{1,j}) - O(g_{2,j-1}, g_{2,j}))$ is the total cache miss penalty and the second part $c_{\mathrm{c}} \cdot CC(2, j)$ is the total cache contention penalty. For the ith core, we assign a task that:

$$c_{\mathrm{m}} \cdot (C(g_{i,j}) - O(g_{i,j-1}, g_{i,j})) + c_{\mathrm{c}} \cdot C(i, j)$$

is minimized. After assigning a grain-size task to each core, we start from core 1 again and repeat this process until all tasks are assigned. In the above greedy approach, one expensive process is to calculate the cache overlap score $O(g_i, g_j)$ for two different grain-size tasks g_i and g_j, since we need to examine the out-neighbors of all nodes in g_i and g_j, which can be quite huge. We next present an efficient k-min sketch-based solution to approximate the cache overlap scores and significantly reduce the cache overlap score calculation cost to $O(k)$, where k is the input of k-min sketch.

Cache overlap score computation We use the k-min sketch [8] to improve the efficiency of computing cache overlap scores. Recall that we divide the whole residue array into $\{R_1, R_2, \ldots, R_{\lceil n/B \rceil}\}$. For each R_i, we generate k independent random variable $l_{i,1}, l_{i,2}, \ldots, l_{i,k} \in [0, 1]$ in uniform. Then, for each task g_i, let $l_{\min}^{i,j} = \min_{R_x \in C(g_i)} l_{x,j}$. According to [8], we have that:

$$\beta(g_i) = \frac{k}{\sum_{j=1}^{k} l_{\min}^{i,j}} - 1$$

is an unbiased estimation of $|C(g_i)|$.

Also, given any two tasks g_i and g_w, we can also use the k-min sketch to provide an unbiased estimation $\gamma(g_i, g_w)$ of $|C(g_i) \cup C(g_w)|$, which is:

$$\gamma(g_i, g_w) = \frac{k}{\sum_{j=1}^{k} \min\{l_{\min}^{i,j}, l_{\min}^{w,j}\}} - 1.$$

Recall that the cache overlap score of g_i and g_j is defined as $O(g_i, g_j) = |C(g_i) \cap C(g_j)|$ and satisfies that:

$$O(g_i, g_j) = |C_i| + |C_j| - |C_i \cup C_j|.$$

Therefore, we use

$$\beta(g_i) + \beta(g_j) - \gamma(g_i, g_j)$$

as the estimation of the cache overlap score. Since $\beta(g_i)$, $\beta(g_j)$, and $\gamma(g_i, g_j)$ all can be computed with $O(k)$ time, we can calculate the approximation of the cache overlap score with $O(k)$ cost.

Total schedule cost analysis Now, we consider the cost of the greedy approach for cache-aware scheduling. We have $\lceil n/g \rceil$ tasks to do the schedule, and in the greedy approach, we need to choose from the remaining node whose penalty is minimal. Such a greedy strategy takes quadratic time in terms of the number of tasks. Also, to compute $CC(i, j)$ it takes $O(P)$ time. Therefore, the total cost is $O(c^2 \cdot k \cdot P \cdot n^2/g^2)$. In our scheduling, we set c_{c} and c_{m} to be the same. Also,

Algorithm 4: PAFO random walk phase

Input: Graph G, source vertex s, probability α
Output: the estimated PPR $\hat{\pi}(s, v)$ for each $v \in V$

1 Let $\omega = \frac{3\log(2/p_f)}{\epsilon^2 \cdot \delta}$;
2 Let $\pi^\circ(s, v)$, $r(s, v)$ be the reserve and residue of each node $v \in V$ after the forward push phase;
3 Let $c(v) = 0$ for all $v \in V$;
4 **Parallel for** *each node* $v \in V$ *with* $r(s, v) \neq 0$ **do**
5   Let $\omega_v = \lfloor \omega \cdot r(s, v) \rfloor$;
6   **for** $i = 1$ *to* ω_v **do**
7   Pick an unused destination from the index stored for v, and let u be the destination;
8   Atomic-Add($c(u)$, 1);
9 **Parallel for** *each node* $v \in V$ **do**
10     $\pi^\circ(s, v) \leftarrow \pi^\circ(s, v) + c(v)/\omega$;
11 **Parallel for** *each node* $v \in V$ **do**
12   **if** $r(s, v) \cdot \omega > \omega_v$ **then**
13   Pick an unused destination from the index stored for v, and let u be the destination;
14   Atomic-Add($\pi^\circ(s, u)$, $r(s, v) - \omega_v/\omega$);
15 Return $\pi^\circ(s, v)$ as the estimated PPR for each node;

when the tasks scheduled to a core i are all finished, core i will steal tasks from other cores to make each core busy and balance the workload.

Discussion on cache-aware scheduling on more graph algorithms Intuitively, our cache-aware scheduling can be generalized to more graph algorithms instead of just the forward push algorithm. However, it should be observed that the proposed cache-aware scheduling would only be effective when there are heavy iterations that are frequently visiting a large portion of the nodes to traverse its out-neighbors or in-neighbors; also, in different iterations, the set of visited nodes has quite large overlaps. Therefore, some graph algorithms may not be well fitted for the cache-aware scheduling. For instance, the BFS or DFS traversal algorithms will not visit the same node twice due to its traversal natural. The power iteration [28] method to calculate PageRank scores tends to benefit from our scheduling algorithm since on the one hand it will involve several heavy iterations; on the other hand, the set of visited nodes in heavy iterations typically overlaps by a large margin.

3.2 Parallel random walk phase

Next, we elaborate in detail on how to parallelize the random walk phase. In the random walk phase, given a node v, it samples $\lceil r(s, v) \cdot \omega \rceil$ random walks. Let u be the destination of a random walk, it then adds $\frac{r(s, v)}{\lceil \omega \cdot r(s, v) \rceil}$ to $\pi^\circ(s, u)$. It is expected that the update to π° will be very cache-unfriendly since the destination accessed is typically not following any order. To alleviate such a situation, one possible way is to try to put as much of the data to be updated into the cache as possible. Therefore, it is important to reduce the size of the update data array. Besides, since multiple processors are updating the data array, by maintaining multiple copies of the update data array, chances are that we can alleviate the data contention and improve the performance. Motivated by these intuitions, we first present an integer-based random walk counting approach, which stands as the backbone for reducing the size of the update array. Then, we present our techniques to alleviate the data contentions in random walk phase by maintaining multiple copies of the data array. Notably, with our technique to reduce the array size, even if we maintain multiple copies, the total size of the update array is still no more than the reserve array. Next, we first explain the integer-based random walk counting method.

3.2.1 Integer-based random walk counting

In a Monte Carlo approach, we can simply count the number $c(v)$ of random walks that stopped at v and then use $c(v)/\omega$ as the estimation of $\pi(s, v)$, where ω is the number of random walks sampled with s as the source node. Therefore, most of the calculations can be handled by integer additions. However, in FORA, as we can see from Algorithm 1 Line 8, when we sample a random walk from a node v, if it terminates at node t, let $a_i = \frac{r(s, v)}{\lceil \omega \cdot r(s, v) \rceil}$, then we add a_i to $\pi^\circ(s, t)$. Since a_i is different for different node v, we cannot simply record the number of random walks that stopped at t and divide them by the total number of random samples as the estimation. Thus, it needs to use float-point values instead of integer values, which brings additional memory overhead. To overcome the deficiency, we propose an integer-based random walk solution, which updates on integer arrays instead on float-point arrays and thus is more likely to reduce cache misses.

Recall from Sect. 2.3.1 that, after the forward push from s, for an arbitrary node v, it samples $\omega_v = \lceil r(s, v) \cdot \omega \rceil$ random walks from v. For a sampled random walk from v, let X_j be a Bernoulli variable depending on the random walk from v such that if t is the destination, $X_j = 1$, and otherwise $X_j = 0$. Then, in expectation, given ω_v random walks, the total number of random walks to terminate at node t is:

$$\mathbb{E}\left[\sum_{j=1}^{\omega_v} \frac{r(s, v)}{\omega_v} X_j\right] = r(s, v) \cdot \pi(v, t). \qquad (3)$$

Here, for any X_j ($0 < j < \omega_v$), it multiplies the same coefficient $a_i = \frac{r(s,v)}{\omega_v}$ in FORA, which is used to guarantee that its expectation is $r(s, v) \cdot \pi(v, t)$. Then, by adding all the random walks from different nodes, the sum of the expectation of the random walks is exactly

$$\sum_{v \in V} r(s, v)\pi(v, t),$$

and then concentration bound can be applied according to [36] to provide an approximation guarantee.

Denote $a_{v,j}$ as the coefficient of X_j regarding node v as the source of random walks, and replace it within Eq. 4. We have that:

$$\mathbb{E}\left[\sum_{j=1}^{\omega_v} a_{v,j} \cdot X_j\right] = \sum_{j=1}^{\omega_v} a_{v,j} \cdot \pi(v, t).$$

That is to say, we need to guarantee the following:

$$\sum_{j=1}^{\omega_v} a_{v,j} = r(s, v). \tag{4}$$

to provide approximation guarantees.

Clearly, when we set $a_{v,j} = \dfrac{r(s, v)}{\omega_v}$, the above equation holds. However, in such a setting $a_{v,j}$ will be highly dependent on $r(s, v)$, and may differ when the node changes. Therefore, we aim to find an assignment for $a_{v,j}$ that depends on v_i as less as possible. To achieve this, our settings are as follows:

$$a_{v,j} = \begin{cases} 1/\omega & \text{if } j \leq \lfloor r(s, v) \cdot \omega \rfloor \\ r(s, v) - \lfloor r(s, v) \cdot \omega \rfloor / \omega & \text{otherwise} \end{cases}.$$

As we can observe, for all $j \leq \lfloor r(s, v) \cdot \omega \rfloor$, the coefficient does not depend on $r(s, v)$ and, at most one case, i.e., $j = \lceil r(s, v) \cdot \omega \rceil$, will depend on $r(s, v)$ and this happens only if $r(s, v) \cdot \omega > \lfloor r(s, v) \cdot \omega \rfloor$. Also,

$$\sum_{j=1}^{\omega_v} a_{v,j} = r(s, v),$$

which means Eq. 4 holds for such an assignment. Therefore, we can still provide an approximation guarantee for the answers. Algorithm 4 shows our parallel random walk phase with the integer-counting-based method. In particular, when a random walk is sampled with v as the source, most likely, we will increment the counter $c(u)$ of the destination u (Algorithm 4 Lines 5–8). Therefore, most of the update will access the count array $c(v)$, and this further motivates us to control the size of the count array to improve memory access efficiency in a multi-core setting.

3.2.2 Improving parallel memory access efficiency

Recall from the last section that, in the random walk phase, the major overhead now is to count the number $c(v)$ of random walks that terminates at node v. In our implementation, we observe that for the most of the indexed random walk destinations, the number $c(v)$ will be less than $2^8 - 1 = 255$, and will always be smaller than $2^{16} - 1$. Therefore, for the count array, instead of using 4 bytes for each node, we choose the size of each entry according to the statistics in the indexed random walks.

Given a significantly reduced size of update arrays, we can afford to make multiple copies of the count array and alleviate the update costs. Firstly, we reorder the nodes in the count array such that the nodes with 1 byte are stored sequentially and then come to the nodes with 2 bytes. Then, for each node, we make M multiple copies of the count array. When a processor updates the count of the arrays, it uses the $(i+1)$th copy for the update if its id is in the range $[i \cdot P/M, (i+1) \cdot P/M)$ $(0 \leq i < M)$. As we will see in Sect. 6, when 4 copies of the count array are maintained, the random walk phase achieves the best query efficiency and improves over the alternative solution by up to $2\times$.

4 Analysis of PAFO

4.1 Forward push phase

In the forward push phase, we first analyze the workload of the hybrid approach. Note that the memory-contention-aware scheduling only needs one pass preprocessing and can reuse the scheduling repeatedly. Then, the scheduling will not affect the complexity and therefore we omit its discussion in parallel time complexity analysis.

Workload Consider the workload in the light workload iterations. In each light workload iteration, the cost mainly includes two parts. The first part is the push operation to each node; the second part is the maintenance of the bag structure for the next iteration. We charge the cost of the bag maintenance to the next iteration for the ease of cost analysis. Let b be the number of active nodes in a light workload iteration. Then, the total cost to insert these nodes to the bag structure is $O(b)$ since it takes $O(1)$ amortized time to insert an element into the bag structure. Then, to divide the bag into many small constant size bag structures, we analyze as follows. Let $T(b)$ be the cost to divide b elements into two (roughly) equal size parts. Then, the total cost of the dividing part is:

$$T(b) = \log b + 2T(b/2),$$

According to [9], it is not difficult to find that the total cost $T(b)$ can be bounded by $O(b)$. Therefore, the total cost of the bag maintenance can be bounded by $O(b)$. Considering the push cost of each node, it does the same work as the

sequential version, and therefore the total workload in this iteration has the same complexity as sequential algorithms.

Next, consider the heavy workload iterations. Since the number of nodes is $O(n)$, the scanning cost is also $O(n)$. Therefore, the maintenance cost for the active nodes does not increase the complexity. The push cost is also the same as the sequential algorithms. Hence, the workload in the iteration has the same complexity as sequential ones.

Recall from Sect. 2.3.1 that the workload of sequential forward push is $O\left(\frac{1}{r_{\max}}\right)$; therefore, the workload of the hybrid approach is also $O\left(\frac{1}{r_{\max}}\right)$.

Depth As we mentioned in Sect. 2.3.2, it is still an open problem that whether the depth of the forward push algorithm proposed in [2] can be bounded in poly-logarithmic of n. Denote $r_{\text{sum}} = \sum_{v \in V} r(s, v)$, we notice that in our problem, it suffices to guarantee that $r_{\text{sum}} \leq m \cdot r_{\max}$ to preserve the time complexity of FORA in the random walk phase if the time complexity of the forward push phase can be still bounded by $O\left(\frac{1}{r_{\max}}\right)$, with $r_{\max} = \sqrt{\frac{1}{m \cdot \omega}}$. Our main observation is that due to the small choice of $r_{\max}$, the cost of the forward push is $O(\frac{n \cdot \log n}{\epsilon})$. On scale-free graphs, such cost is also $O(\frac{m}{\epsilon})$. Also, notice that by doing a batched forward push on each node in an iteration with a cost of $O(m)$, the sum of the residues after this iteration will be at most $(1-\alpha) \cdot r_{\text{sum}}$, where r_{sum} is the sum of residues in this iteration. Then, we can apply the following strategy to achieve bounded depth. We apply the hybrid approach with $O(\log n)$ rounds, and if the algorithm still does not stop. Let r_{sum}^h be the sum of the residues after the last iteration of the hybrid approach. Then, we do batched forward push with $O(\log_{1-\alpha}(\epsilon / r_{\text{sum}}^h))$ rounds if $r_{\text{sum}}^h > \epsilon$. We denote this solution as the batch-push strategy.

Lemma 1 *On scale-free graphs, with batch-push strategy, the depth of the parallel forward push can be bounded by $O(\log^2 n)$, and the workload is still the same as the sequential FORA.*

Proof We first note that

$$m \cdot r_{\max} = \sqrt{\frac{m}{\omega}} = \sqrt{\frac{\epsilon^2 m}{3n \log n}} = \epsilon \sqrt{\frac{m}{3n \log n}}.$$

On scale-free graphs, since $m/n = \log n$, when the sequential forward push terminates, it actually only requires that $r_{\text{sum}} = O(\epsilon)$ to preserve the workload of the random walk phase. With the batch-push strategy, we have that the sum of the residue after each iteration reduces by α portion. Therefore, after $\log_{1-\alpha}(\epsilon / r_{\text{sum}}^h)$ iterations, where r_{sum}^h is the sum of residues after the last iteration of the hybrid approach after $O(\log n)$ round, the sum of residue r_{sum} after the batch-push satisfies that:

$$r_{\text{sum}} \leq r_{\text{sum}}^h \cdot (1-\alpha)^{\log_{1-\alpha}(\epsilon / r_{\text{sum}}^h)} = \epsilon.$$

Hence, with $O(\log_{1-\alpha}(\epsilon / r_{\text{sum}}^h))$ iterations, we guarantee that $r_{\text{sum}} = O(\epsilon)$, which indicates the workload of the random walk phase is the same as sequential FORA. Also, notice that the number of iterations in batch-push is bounded by $O(\log \frac{1}{\epsilon})$. Recall that we assume $\epsilon \geq n^{-2}$. Therefore, the total number of iterations in forward push can be bounded by $O(\log n + \log \frac{1}{\epsilon}) = O(\log n)$ iterations. Consider the workload of batch-push. The workload is bounded by $O(m \log \frac{1}{\epsilon})$, while the workload of the forward push is bounded by $O(m/\epsilon)$. Hence, the cost can be still bounded by $O(m/\epsilon)$, which does not increase the workload of the forward push phase. Hence, the workload is still the same as sequential FORA.

Consider the depth of parallel forward push. Since the number of iterations can be bounded by $O(\log n)$. In each iteration, the maximum workload may reach polynomial of n, and therefore the depth in each iteration may reach $O(\log n)$. The depth can then be bounded by $O(\log^2 n)$. □

According to Lemma 1, we revise our hybrid approach such that when the total number of iterations exceeds $c_2 \cdot \log n$, we apply the batch-push strategy. Nevertheless, we observe that in practice, the number of iterations is typically small, e.g., as shown in Fig. 1. Also, when applying a batch-push, the overhead is actually quite large since it strictly does a push operation on each node with nonzero residue, while in our heavy workload iterations or light workload iterations, only out-degree of the nodes whose residue is above $r_{\max}$ times need to do push operations. Therefore, in our evaluation, we set $c_2 = 10$ to avoid unexpected large iterations in the hybrid parallel forward push algorithm, and in most cases, it will avoid the expensive batch-push costs.

4.2 Combining two phases

Next, we first analyze the workload and depth of the random walk phase. Then, we combine the two phases together and analyze the total cost of PAFO.

Lemma 2 *In the random walk phase of PAFO, the depth of Algorithm 4 can be bounded by $O(\log^2 n)$, and the workload is still the same as the sequential FORA.*

Proof Recall from the analysis of the forward push phase, when the algorithm terminates, it assures that the total number of random walks is still the same as the sequential FORA. In our PAFO, we maintain multiple copies of the update array, but each random walk will update exactly one copy of the array. In our parallel random walk algorithm, we have

an additional aggregation phase as shown in Algorithm 4 Lines 9–14. However, since we maintain a constant number of update arrays, such aggregation cost can be bounded by $O(n)$, while the update cost is $O(n \cdot \log n/\epsilon)$ on scale-free graphs. Therefore, the cost can be still bounded by $O\left(\frac{n \cdot \log n}{\epsilon}\right)$, which is the same as the sequential random walk phase. The depth of the random walk phase can also be bounded by $O(\log^2 n)$ since we process n nodes in parallel, and for each node, the number of random walks can reach polynomial of n. □

Combining Lemmas 1, 2 and Theorem 1, we have Theorem 2 on the parallel time complexity of our PAFO.

Theorem 2 *PAFO achieves a time complexity of $O(W/P + \log^2 n)$, where W is the workload of the sequential FORA and P is the number of used processors. When $W \gg \log^2 n$, PAFO achieves linear speedup with respect to the number P of used processors.*

In our problem, the time complexity of our sequential FORA is higher than $O(\log^2 n)$, and therefore, asymptotically PAFO achieves linear speedup.

5 Related work

In the literature, there exists a plethora of research work on PPR computation [1–4,10,12–18,21,22,26,28,30,33–35,38, 42]. Among them, considerable efforts [2–4,12,15–17,21,22, 28,30,33,36,42] have been made to improve the query efficiency of the single-source PPR queries. To provide exact or approximate answers with theoretical guarantees for single-source PPR queries, there exist mainly two categories of solutions.

The first category relies on matrix-based definition of PPR:

$$\boldsymbol{\pi}_s = (1-\alpha) \cdot A^{\mathrm{T}} D^{-1} \boldsymbol{\pi}_s + \alpha \cdot \boldsymbol{e}_s, \tag{5}$$

where $\boldsymbol{\pi}_s$ is the PPR vector where the ith entry stores the PPR $\pi(s,i)$ of the node with ith id with respect to s, $\boldsymbol{e}_s$ is a unit vector where only the sth entry is 1 and other entries are zero, α is the stopping probability as defined in Sect. 2.1, A is the adjacent matrix of the input graph, and D is a diagonal matrix where the ith entry is the out-degree of node i. The solutions in this category are mainly based on the *power method* as proposed in [28], which makes an initial guess to the PPR vector and then repeatedly uses the estimated PPR score as the input of the RHS of Eq. 5 and derives the new estimation of $\boldsymbol{\pi}_s$. The solutions in this category mainly explore how to accelerate the calculation of Eq. 5, and the state of the art in this category is the BePI algorithm proposed in [22]. However, the solutions in this category typically incur thousands of seconds to handle large graphs, e.g., on the Twitter, and is too slow to support real-world applications. There is another line of *local update*-based approach [1,2,17,41] which is also based on the matrix-based definition. However, such solutions either provide no approximation guarantee to the SSPPR queries or cannot be directly applied to answer SSPPR queries.

Another category relies on the random walk-based definition of personalized PageRank and explores Monte Carlo simulation to provide an approximation for the PPR values [3,4,12,26,34,36,38]. The state-of-the-art solution in this category is FORA. However, even under approximation, the state-of-the-art FORA still needs tens of seconds to finish an SSPPR query processing, which motivates us to devise parallel algorithms to reduce the SSPPR query time. Among existing work on PPR, there exist only two research works that focus on parallelizing PPR computation on shared-memory multi-core setting. The first is the state-of-the-art parallel solution for PPR as we mentioned in Sect. 2.3.2. The solution proposed by Guo et al. [17], however, provides no approximation guarantee to the SSPPR query, and the space consumption to pre-store all the forward push result is prohibitive for large graphs, e.g., Twitter.

Distributed systems are also considered for computing PPR in parallel. Bahmani et al. [3] utilize MapReduce to calculate single-source PPR queries on distributed computer systems, aiming to handle the graphs that are too large to fit in the memory on a single machine. They provide distributed algorithms for the Monte Carlo approach, which is orders of magnitude slower than FORA [36] when providing the same accuracy, not to mention our parallel algorithms. Guo et al. [16] propose algorithms to achieve bounded communication cost and balanced work load on each machine for answering exact PPR. Recently, Lin [24] propose several optimizations for the Monte Carlo approach to (i) alleviate the issue of large node for random walk sampling, (ii) pre-compute short random walks, and (iii) optimize the number of random walks to compute in each pipeline to reduce the overhead. However, these solutions trade scalability with efficiency by exploring different nodes to store the graph and requires prohibitive communication costs. These distributed algorithms are typically the preferred choice when graphs cannot be fitted into the memory, which is not the main focus of this work.

6 Experimental evaluations

In this section, we evaluate our proposed solutions against the states of the art. All experiments are conducted on a Linux machine with 4 CPUs, each with 10 cores clocked at 2.2GHz, and 1TB memory. All the implementations are written in C++ and compiled with full optimization. For all methods, we use the CilkPlus [20] as the framework for multi-thread pro-

Table 1 Datasets

Name	n	m	Type
LiveJournal (LJ)	4.8M	69.0M	Directed
Orkut	3.1M	117.2M	Undirected
Twitter	41.7M	1.4B	Directed
Friendster	65.6M	3.6B	Undirected
RMAT27	128M	8.6B	Directed
RMAT28	256M	20.8B	Directed

$M = 10^6$, $B = 10^9$

gramming, which is shown in [17,31] to be the most efficient framework on parallel graph processing.

Datasets and query sets To compare the performance and show the effectiveness of our proposed solution, we test on 4 public datasets: *Livejournal*, *Orkut*, *Twitter*, and *Friendster*. All these datasets are social networks that are widely used to evaluate PPR query efficiency [26,33,34,36,38]. The statistics are shown in Table 1. To evaluate the scalability of our proposed solutions, we further generate 2 synthetic large-scale datasets *RMAT27* and *RMAT28*, which include 8.6 billion edges and 20.8 billion edges, respectively, using the RMAT graph generator [7]. For these 6 datasets, we generate two query sets each with 100 queries where the source nodes are sampled uniformly at random in one query set and sampled with probability proportional to their out-degrees in the other query set. We denote the two datasets as UNIFORM and POWER-LAW, respectively. We use the average query time as the measure of query performance. For each experiment, we repeat 5 times and show the average performance.
Methods To compare the performance of parallel forward push algorithm, we include the state-of-the-art solution proposed in [33], denoted as *Ligra*. For our methods, we include two versions of the forward push algorithm. The first one is the hybrid approach, dubbed as *hybrid*, which does not include the cache-aware scheduling optimization as mentioned in Sect. 3.1.2. The second one is the solution that includes the cache-aware scheduling optimization, dubbed as *hybrid-cs*. For random walks, we use the straightforward solution (as mentioned in Sect. 2.3.2) as the baseline and compare with our optimized random walk algorithms. Finally, when comparing the total query performance, we use *Ligra* to denote the combination of the parallel forward push algorithm in [33] and the straightforward parallel random walk algorithm as mentioned in Sect. 2.3.2. Besides, since the Monte Carlo approach can be embarrassingly parallelized, we include the parallel version as our baseline and denote this approach as *Parallel MC* for the comparison of total query performance. To compare the scalability, we test all the solutions with different numbers of threads varying in $\{1, 4, 8, 12, 16, 20, \ldots 36, 40\}$.

Parameter setting Following previous work [26,34,36], we set $\delta = 1/n$, $p_f = 1/n$, and $\epsilon = 0.5$. For the cache scheduling, recall that we use k-min sketch to calculate the cache overlap score. In our preprocessing, we set $k = 32$ to derive the cache overlap scores and derive the scheduling. Besides, we tune the $r_{\max}$ for sequential FORA and find that when $r_{\max} = 3\epsilon \cdot \sqrt{\frac{1}{3m \cdot n \cdot \ln(2/n)}}$, it achieves the best trade-off between the query performance and space consumption. Therefore, we set $r_{\max}$ to this value in all experiments.

6.1 Scalability: overall performance

We first examine the overall performance of PAFO against Ligra. For PAFO, we include all the optimizations mentioned in Sect. 3 and postpone the evaluation of the effectiveness of each optimization to Sects. 6.2–6.3.

Figure 2 shows the experimental results that evaluate the scalability of PAFO and Ligra with different numbers of threads using the UNIFORM query set on 4 large datasets: Twitter, Friendster, RMAT27, and RMAT28. The results show that our PAFO achieves significantly better speedup on all the tested datasets. For instance, on Friendster, our PAFO achieves a speedup of 35 with 40 cores, while the speedup of Ligra is only around 17 with 40 cores. When the size of the graph increases, the speedup of PAFO generally increases. For instance, on the Twitter graph, our PAFO achieves a speedup of 23 with 40 cores, and on RMAT28, the speedup becomes 37 with 40 cores. In contrast, the speedup of Ligra with 40 cores is 15 and 16 on Twitter and RMAT28, respectively. Figure 3 shows the results of PAFO and Ligra with the POWER-LAW query set. The experimental result shows a similar trend for both PAFO and Ligra when the number of threads increases and the dataset size grows. In the remaining paper, we omit the result for POWER-LAW query set and use the UNIFORM query set to report the results since the results are similar.

Next, we examine the performance of PAFO and Ligra. We first compare the performance of PAFO and Ligra against the state-of-the-art sequential FORA algorithm. As we can see from Table 2, both parallel implementations incur considerable overhead over the sequential solution. However, since Ligra follows their previous design paradigm, which uses the VertexMap and EdgeMap interface, it brings additional overheads. For instance, it needs to maintain two copies of the residue array. In our solution, only one residue array is required. Besides, our solution includes optimizations that are tailored for the SSPPR query, which further reduces the overhead of our forward push algorithm. Comparing the performance of PAFO and Ligra with 40 cores, as shown in Table 3, our PAFO is up to 3.3 faster than Ligra and is always at least 2.4x faster than Ligra. Notice that it is different from

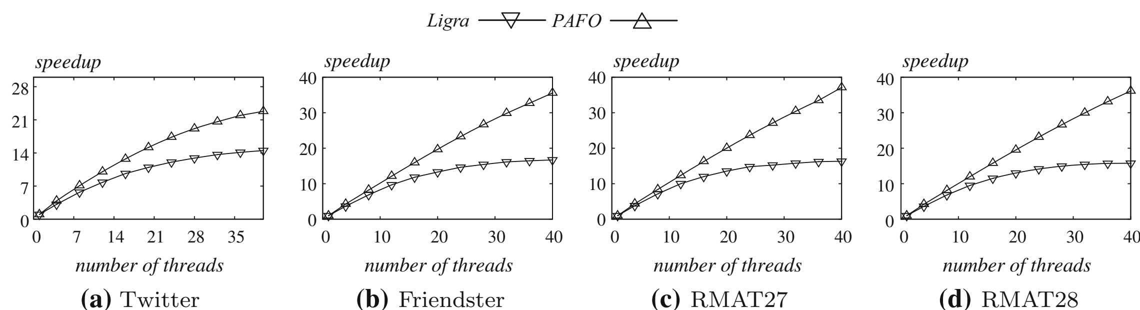


Fig. 2 Scalability: overall performance on UNIFORM query set

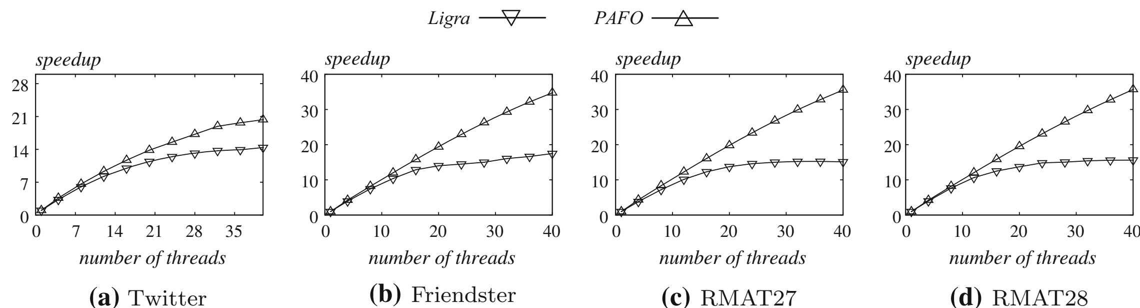


Fig. 3 Scalability: overall performance on POWER-LAW query set

Table 2 Overall performance on a single-core

Dataset	FORA (s)	Ligra (s)	PAFO (s)
LJ	1.8	7.1	2.9
Orkut	2.4	6.1	3.0
Twitter	35.3	109.4	63.6
Friendster	115.9	259.5	169.9
RMAT27	385	712	496
RMAT28	792	1364	1057

Table 3 Overall performance on 40 threads

Dataset	Parallel MC (s)	Ligra (s)	PAFO (s)
LJ	8.4	0.61	0.18
Orkut	5.8	0.57	0.23
Twitter	110	7.4	2.7
Friendster	216	17	5.6
RMAT27	472	44	13
RMAT28	1004	87	29

the numbers in Fig. 2 since the parallel implementation incurs additional overheads.

Besides, as we can observe, even under parallelization, the Monte Carlo approach is slower than the sequential version of FORA running on a single core. With 40 threads, our PAFO can finish an SSPPR query with 5.6 s on the 3.6 billion edge Friendster network, and 29 s on 20 billion edge RMAT28 network. Compared to the sequential FORA algorithm, our PAFO achieves up to 30× speedup (Table 2, 3).

In summary, our PAFO framework achieves superb scalability and efficient query processing and is the preferred choice when parallelizing the SSPPR queries.

6.2 Forward push phase

In this set of experiments, we evaluate the scalability of our parallel forward push algorithm and the effectiveness of the proposed optimization techniques mentioned in Sect. 3.1.

We first examine the scalability of our parallel forward push algorithm against Ligra on four large datasets: Twitter, Friendster, RMAT27, and RMAT28. The results are as shown in Fig. 4. In the forward push phase, our PAFO achieves 35× (resp. 39×) speedup over its single-core counterparts on 40 cores on Friendster (resp. RMAT27) dataset. The improved scalability is mainly due to the optimizations applied.

Next, we examine the benefit of each optimization. We first examine the effectiveness of the proposed hybrid strategy. As shown in Fig. 5, with our hybrid strategy, our solution (dubbed as hybrid) can be 3× faster than the Ligra baseline solution. When the number of threads increases, the speedup of the hybrid approach over Ligra actually decreases. However, with our cache-aware scheduling, our approach (dubbed as hybrid-cs) further gains improved speedup with the increase in the number of threads. This demonstrates the effectiveness of the scheduling method. We note that the scheduling-based method improves over the hybrid approach

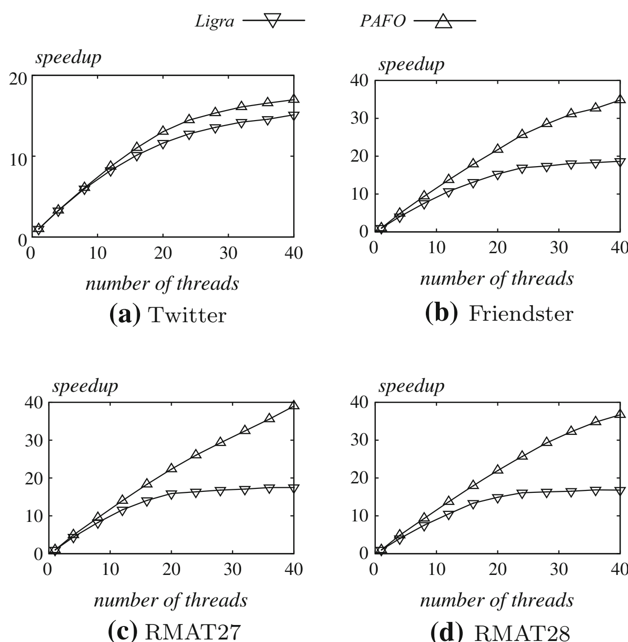


Fig. 4 Forward push scalability

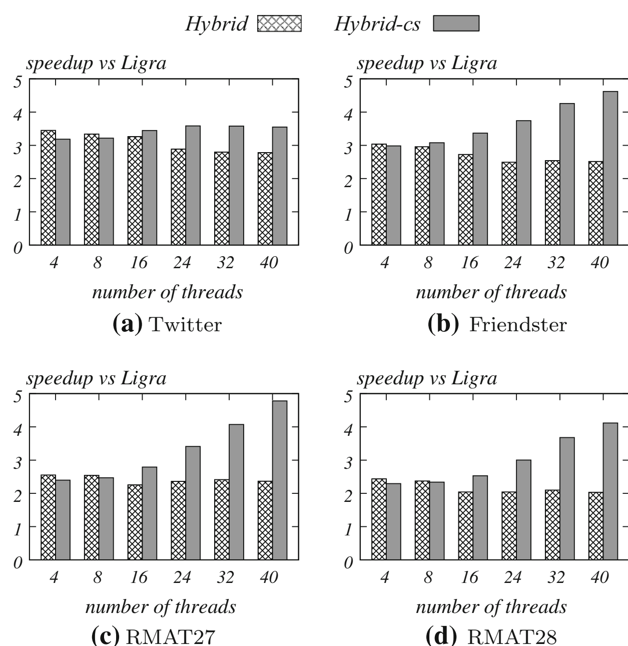


Fig. 5 Forward push speedup over Ligra

by up to 100% with 40 cores. Besides, our parallel forward push algorithm is up to 4.8× faster than Ligra.

Finally, we compare our cache-aware scheduling with the state-of-the-art cache-aware ordering method, Gorder [37], which orders the nodes so as to reduce the cache miss rate. We combine the Gorder method with our hybrid method to make it our competitor. Tables 4 and 5 report the query performance with a single thread and 40 threads, respectively.

Table 4 Forward push performance on a single thread

Dataset	Gorder (s)	PAFO (s)
Twitter	14.0	19.4
Friendster	46.4	63.1
RMAT27	98	164
RMAT28	177	284

Table 5 Forward push performance on 40 threads

Dataset	Gorder (s)	PAFO (s)
Twitter	1.51	1.14
Friendster	2.19	1.81
RMAT27	8.97	4.21
RMAT28	15.53	7.73

The experimental result on a single thread verifies the effectiveness of Gorder for reducing cache miss rate, and it is up to 67% faster than PAFO. However, when the number of threads increases to 40, PAFO is (up to 2×) faster than Gorder. The main reason is that Gorder only considers the cache miss but discards the cache contention in multi-core processing. In contrast, our scheduling method considers both cache miss and cache contention and therefore can achieve better practical performance than Gorder. Figure 6 further shows the scalability of the hybrid method with Gorder and the hybrid method with our cache-aware scheduling. The experimental result shows that PAFO achieves much better scalability than Gorder when increasing the number of threads.

As a summary, our proposed parallel forward push algorithm provides more efficient query processing and is more scalable with the growth of the number of threads than alternative methods. Since the forward push algorithm alone is also widely used in local graph clustering [2], our parallel forward push algorithm can be further used in a wider scope besides the approximate SSPPR queries.

6.3 Random walk phase

In this set of experiment, we examine the scalability of the random walk phase and effectiveness of the integer-counting-based optimization.

Firstly, we examine the impact of the number of update array copies to the efficiency of the random walk phase. The results are shown in Fig. 7. Our main observation is that when the number of update array copies increases, the efficiency can be improved since it alleviates the cache contention. When 4 copies are maintained, it actually achieves the best practical performance. However, when the number of copies further increases to 8, the query performance actually degrades compared to the performance when 4 copies

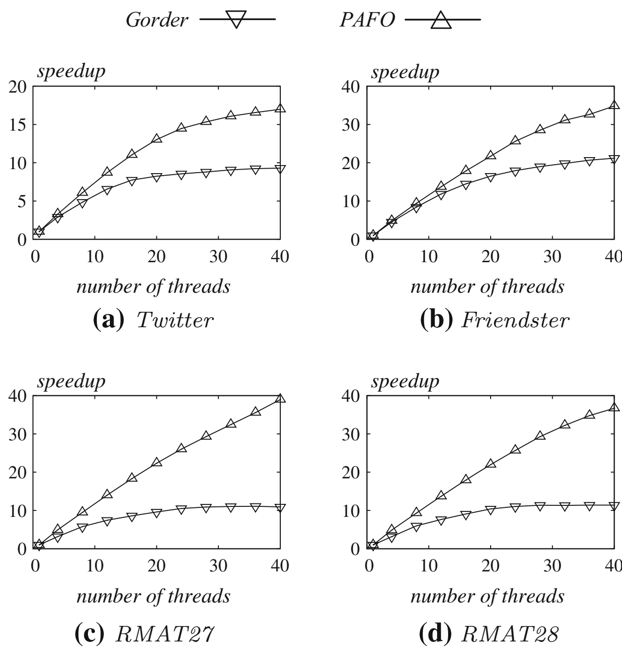


Fig. 6 Gorder versus PAFO

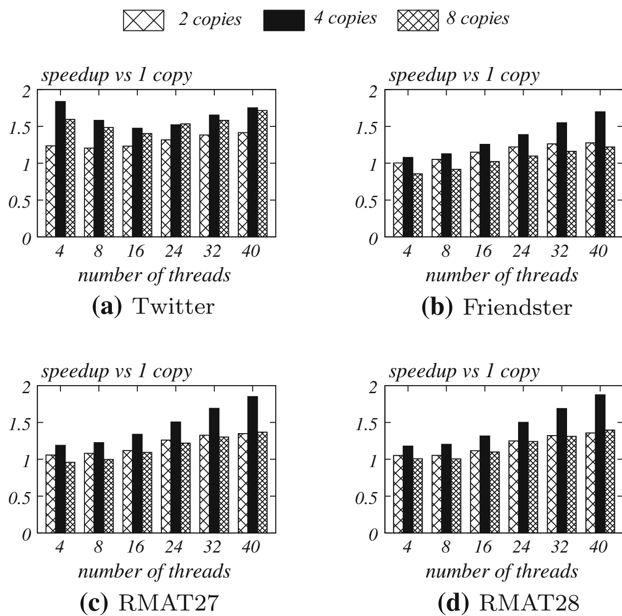


Fig. 7 Impact of the number of update array copies

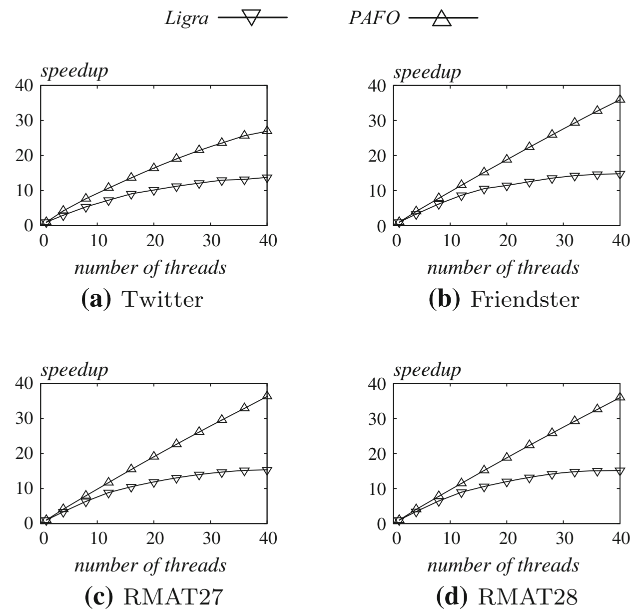


Fig. 8 Random walk scalability

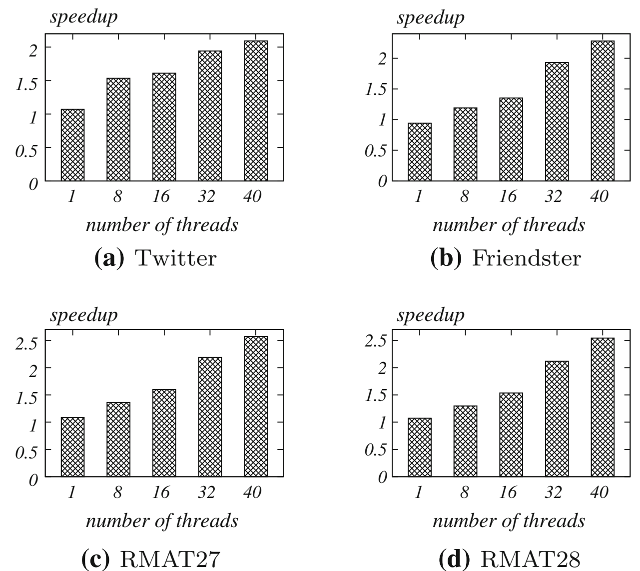


Fig. 9 Integer-based update speedup

are maintained. To explain, the increased number of copies will incur more space consumption, which further results in inferior cache performance. According to the result, in all other sets of experiment, we use 4 copies of the update array since it achieves the best practical query performance.

As shown in Fig. 8, the random walk phase for both PAFO and Ligra achieves better scalability than the forward push phase. This is expected since the forward push phase is more complicated, which includes multiple iterations and has more dependencies. In contrast, the random walk phase is naturally parallelizable with a straightforward parallel for loop (as we mentioned in Sect. 2.3.2). Nevertheless, if the straightforward approach is applied, the scalability is still unsatisfactory. As we mentioned, the random walk phase includes a lot of random memory accesses, which is not cache-friendly. With our integer-counting-based method, we can reduce the size of the update array to as small as 1/8 of the original update residue array, which is more cache-friendly. As shown in Fig. 9, with the increasing number of threads, the contention is significantly reduced by maintaining multiple copies of the update array. With our optimizations for the random walk phase, our

Table 6 Preprocessing time

Dataset	PAFO (s)	Gorder
Twitter	37.8 s ($k = 32$, $g = 2048$)	2 h
Friendster	158 s ($k = 32$, $g = 2048$)	10 h
RMAT27	275 s ($k = 32$, $g = 4096$)	5 days
RMAT28	840 s ($k = 32$, $g = 4096$)	10 days

parallel random walk is up to 2.5× faster than the straightforward parallel update algorithm, which demonstrates the effectiveness of our optimization techniques to the random walk phase.

6.4 Preprocessing cost

In this set of experiments, we examine the preprocessing cost of our PAFO and the competitor Gorder. Notice that we parallelize the greedy solution mentioned in Sect. 3.1.2 with full usage of the cores, while Gorder is preprocessed with a single core and it is unclear if it can be processed in parallel. As shown in Table 6, even on the largest real-world Friendster dataset with 65.6 million nodes, our off-line scheduling calculation can be finished in less than 160 s. For synthetic dataset with 20 billion edges, the calculation finishes in 840 s with $k = 32$ and $g = 4096$, which is about the time of a single query on FORA. As the scheduling can be calculated once and reused for the subsequent queries, the moderate preprocessing overhead is compensated by the query performance improvement with the cache-aware scheduling. The preprocessing cost of Gorder, on the other hand, is very prohibitive and requires 10 days to finish the processing. Even if it can be processed in parallel with 40 cores, it still needs more than 6 h and is still far more expensive than our PAFO in terms of preprocessing time.

6.5 Accuracy

Finally, we evaluate the accuracy of PAFO and FORA with two datasets: Livejournal and Orkut. We first calculate the ground truth of the PPR scores using power iteration by setting the number of iterations to be 100. For each query, we calculate the maximum absolute error of all the estimated PPRs compared to their ground truth. Then, we report the average of 100 queries. As shown in Table 7, both methods achieve highly accurate results, and the maximum absolute error of both PAFO and FORA falls into the same order of magnitude on both datasets. We further report the maximum relative error, where for each query, we calculate the maximum relative error of all the estimated PPRs compared to their ground truth and then report the average on 100 queries. Recall that we set $\epsilon = 0.5$, which means in the worst case, the relative error can reach 0.5. From Table 8, we can see that the maximum relative errors of both PAFO and FORA are actually far smaller than the worst-case guarantee. The conclusion is that by parallelizing the FORA algorithm, the impact to the accuracy is sufficiently small and the parallel algorithm can still provide highly accurate result. The results on Ligra show a similar trend, and we omit the results for simplicity.

Table 7 Accuracy: max absolute error

Dataset	FORA	PAFO
Livejournal	2.07×10^{-3}	5.59×10^{-3}
Orkut	5.73×10^{-7}	3.74×10^{-7}

Table 8 Accuracy: max relative error

Dataset	FORA (%)	PAFO (%)
Livejournal	1.09	3.06
Orkut	0.42	0.39

7 Conclusion

We present PAFO, an efficient parallel framework, which parallelizes the state-of-the-art index-based solution *FORA*, for approximate SSPPR query processing. Theoretically, we prove that our proposed PAFO achieves asymptotically linear speedup. For practical performance, we present several optimization techniques, including effective maintenance of active nodes in forward push phase, improving the efficiency of memory access and cache-aware scheduling. Extensive experimental evaluation on datasets with up to 20.6 billion edges shows that our solution achieves up to 37× speedup on 40 cores, is 3.4× faster than alternatives on 40 cores, and is scalable to super-large graphs with 20.6 billion edges. Moreover, our parallel forward push algorithm improves over the state of the art by 4.8×. Since the forward push algorithm has been extensively used for local graph clustering, our parallel forward push algorithm can be further used to improve the efficiency of these local graph clustering algorithms.

References

1. Andersen, R., Borgs, C., Chayes, J., Hopcraft, J., Mirrokni, V., Teng, S.-H.: Local computation of pagerank contributions. In: WAW, pp. 150–165 (2007)
2. Andersen, R., Chung, F.R.K., Lang, K.J.: Local graph partitioning using pagerank vectors. In: FOCS, pp. 475–486 (2006)
3. Bahmani, B., Chakrabarti, K., Xin, D.: Fast personalized pagerank on mapreduce. In: SIGMOD, pp. 973–984 (2011)

4. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. PVLDB **4**(3), 173–184 (2010)
5. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. Sci. Program. **21**(3–4), 137–148 (2013)
6. Brent, R.P.: The parallel evaluation of general arithmetic expressions. J. ACM **21**(2), 201–206 (1974)
7. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: a recursive model for graph mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining, pp. 442–446. SIAM (2004)
8. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. J. Comput. Syst. Sci. **55**(3), 441–453 (1997)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
10. Coskun, M., Grama, A., Koyutürk, M.: Efficient processing of network proximity queries via chebyshev acceleration. In: SIGKDD, pp. 1515–1524 (2016)
11. Dagum, L., Menon, R.: Openmp: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
12. Fogaras, D., Rácz, B., Csalogány, K., Sarlós, T.: Towards scaling fully personalized pagerank: algorithms, lower bounds, and experiments. Internet Math. **2**(3), 333–358 (2005)
13. Fujiwara, Y., Nakatsuji, M., Onizuka, M., Kitsuregawa, M.: Fast and exact top-k search for random walk with restart. PVLDB **5**(5), 442–453 (2012)
14. Fujiwara, Y., Nakatsuji, M., Shiokawa, H., Mishima, T., Onizuka, M.: Efficient ad-hoc search for personalized pagerank. In: SIGMOD, pp. 445–456 (2013)
15. Fujiwara, Y., Nakatsuji, M., Yamamuro, T., Shiokawa, H., Onizuka, M.: Efficient personalized pagerank with accuracy assurance. In: SIGKDD, pp. 15–23 (2012)
16. Guo, T., Cao, X., Cong, G., Lu, J., Lin, X.: Distributed algorithms on exact personalized pagerank. In: SIGMOD, pp. 479–494 (2017)
17. Guo, W., Li, Y., Sha, M., Tan, K.-L.: Parallel personalized pagerank on dynamic graphs. PVLDB **11**(1), 93–106 (2017)
18. Gupta, M., Pathak, A., Chakrabarti, S.: Fast algorithms for topk personalized pagerank queries. In: WWW, pp. 1225–1226 (2008)
19. Gupta, P., Goel, A., Lin, J.J., Sharma, A., Wang, D., Zadeh, R.: WTF: the who to follow service at twitter. In: WWW, pp. 505–514 (2013)
20. https://www.cilkplus.org/ (2018)
21. Jeh, G., Widom, J.: Scaling personalized web search. In: WWW, pp. 271–279 (2003)
22. Jung, J., Park, N., Sael, L., Kang, U.: Bepi: fast and memory-efficient method for billion-scale random walk with restart. In: SIGMOD, pp 789–804 (2017)
23. Leiserson, C.E., Schardl, T.B.: A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In: SPAA, pp. 303–314 (2010)
24. Lin, W.: Distributed algorithms for fully personalized pagerank on large graphs. In: WWW, pp. 1084–1094 (2019)
25. Liu, D.C., Rogers, S., Shiau, R., Kislyuk, D., Ma, K.C., Zhong, Z., Liu, J., Jing, Y.: Related pins at pinterest: the evolution of a real-world recommender system. In: WWW, pp. 583–592 (2017)
26. Lofgren, P., Banerjee, S., Goel, A.: Personalized pagerank estimation and search: a bidirectional approach. In: WSDM, pp. 163–172 (2016)
27. Nguyen, P., Tomeo, P., Noia, T.D., Sciascio, E.D.: An evaluation of simrank and personalized pagerank to build a recommender system for the web of data. In: WWW, pp. 1477–1482 (2015)
28. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab (1999)
29. Park, H., Jung, J., Kang, U.: A comparative study of matrix factorization and random walk with restart in recommender systems. In: BigData, pp. 756–765 (2017)
30. Shin, K., Jung, J., Sael, L., Kang, U.: BEAR: block elimination approach for random walk with restart on large graphs. In: SIGMOD, pp. 1571–1585 (2015)
31. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: PPoPP, pp. 135–146 (2013)
32. Shun, J., Blelloch, G.E.: Phase-concurrent hash tables for determinism. In: SPAA, pp. 96–107 (2014)
33. Shun, J., Roosta-Khorasani, F., Fountoulakis, K., Mahoney, M.W.: Parallel local graph clustering. PVLDB **9**(12), 1041–1052 (2016)
34. Wang, S., Tang, Y., Xiao, X., Yang, Y., Li, Z.: Hubppr: effective indexing for approximate personalized pagerank. Proc. VLDB Endow. **10**(3), 205–216 (2016)
35. Wang, S., Tao, Y.: Efficient algorithms for finding approximate heavy hitters in personalized pageranks. In: SIGMOD, pp. 1113–1127 (2018)
36. Wang, S., Yang, R., Xiao, X., Wei, Z., Yang, Y.: FORA: simple and effective approximate single-source personalized pagerank. In: SIGKDD, pp. 505–514 (2017)
37. Wei, H., Yu, J.X., Lu, C., Lin, X.: Speedup graph processing by graph ordering. In: SIGMOD, pp. 1813–1828 (2016)
38. Wei, Z., He, X., Xiao, X., Wang, S., Shang, S., Wen, J.-R.: Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In: SIGMOD, pp. 441–456 (2018)
39. Whang, J.J., Gleich, D.F., Dhillon, I.S.: Overlapping community detection using neighborhood-inflated seed expansion. IEEE Trans. Knowl. Data Eng. **28**(5), 1272–1284 (2016)
40. Yin, H., Benson, A.R., Leskovec, J., Gleich, D.F.: Local higher-order graph clustering. In: SIGKDD, pp. 555–564 (2017)
41. Zhang, H., Lofgren, P., Goel, A.: Approximate personalized pagerank on dynamic graphs. In: SIGKDD, pp. 1315–1324 (2016)
42. Zhu, F., Fang, Y., Chang, K.C., Ying, J.: Incremental and accuracy-aware personalized pagerank through scheduled approximation. PVLDB **6**(6), 481–492 (2013)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.