No More Leaky PageRank

Scott Sallinen  
University of British Columbia  
scotts@ece.ubc.ca

Matei Ripeanu  
University of British Columbia  
matei@ece.ubc.ca

**Abstract**—We have surveyed multiple PageRank implementations available with popular graph processing frameworks, and discovered that they treat sink vertices (i.e., vertices without outgoing edges) incorrectly. This leads to two issues: (i) incorrect PageRank scores, and (ii) flawed performance evaluations (as costly scatter operations are avoided). For synchronous PageRank implementations, a strategy to fix these issues exists (accumulating all values from sinks during an algorithmic superstep of a PageRank iteration), albeit with sizeable overhead. This solution, however, is not applicable in the context of asynchronous frameworks.

We present and evaluate a novel, low-cost algorithmic solution to address this issue. For asynchronous PageRank, our key target, our solution simply requires an inexpensive \(O(\text{Vertex})\) computation performed alongside the final normalization step. We also show that this strategy has advantages over prior work for synchronous PageRank, as it both avoids graph restructuring and reduces inline computation costs by performing a final score reassignment to vertices once at the end of processing.

I. INTRODUCTION

PageRank [15] is a popular graph analysis algorithm, and is often used as a primary benchmark for comparing different graph processing frameworks, algorithmic designs, and performance optimizations [12] [19]. We have investigated the PageRank implementations offered by many popular graph processing frameworks, including GraphLab [11], GraphChi [5], Galois [14] [20], TOTEM [8], and HavoqGT [16] [17], and found that the prototype PageRank implementations provided (and, presumably used for benchmarking) are incorrect. More specifically, they calculate incorrect rankings in the fairly common scenario where directed graphs include sink vertices.

While this issue has a fairly straightforward fix which was suggested as early as the original PageRank paper [15], this fix is costly (see Section IV). More efficient solutions have been suggested [6] [7] [8] (discussed in Section V), however, these solutions require graph restructuring and can only be used for synchronous algorithms that operate with algorithmic supersteps (barriers).

This situation raises a key issue: Can the conclusions of previous studies that compare the performance of different graph processing frameworks, algorithmic designs, or optimization choices for PageRank still be trusted? Indeed, one such case is in the comparison between synchronous versus asynchronous frameworks: while for the former there are solutions to deal with sink vertices, for the latter no clear solution exists. As such, this situation may call into question past work’s conclusions about the performance of processing frameworks, algorithmic designs, and various optimizations.

II. PAGE RANK BACKGROUND

One mental image of PageRank is the perspective of a web “surfer” who starts at a random page (vertex), and either: with (\(\alpha\)) probability, picks a random outbound link (edge) from that page, or with (1-\(\alpha\)) probability, finishes their current search on this page, and restarts on a random page. The PageRank score of a page can be thought of as the probability that the surfer landed on that page at a randomly chosen point in their surfing history.

Handling Sink Vertices. While the imaginary surfer continues until they decide to reset their search and jumps to a new random page, a sink vertex (i.e., a vertex with no outbound edges, sometimes called a dangling or leaf vertex) presents an issue: the surfer has no outbound path. To combat this issue, it is defined that the surfer will jump to a new (but different) page chosen at random. This is equivalent to redefining the graph such that any sink vertex now has outgoing edges to all other vertices. The challenge with directly modelling this strategy by adding edges is that, as many graphs we encounter in practice...

**Contributions.** In this paper, we:

- Expose the incorrect behaviour in many prototype PageRank implementations available with popular graph processing frameworks, and highlight that skipping a key algorithmic step significantly improves their observed performance.
- Introduce a novel solution to handle sink vertices (outlined in Section III). This has a number of key advantages, as it: (i) is generic in the sense that it can be used in the context of both synchronous and asynchronous PageRank strategies; (ii) is platform independent as it does not rely on platform features (e.g., it is not restricted to shared memory platforms); (iii) is low cost: only a single \(O(|G|)\) operation is performed to reconcile final results, and (iv) is compatible with the commonly-used vertex-centric graph algorithm philosophy, adopted by many distributed graph frameworks.
- Empirically evaluate the performance of the proposed solution, by amending the synchronous and asynchronous PageRank implementations available within the shared memory Galois framework, as well as the asynchronous HavoqGT framework targeting shared-nothing environments.
- Re-establish prior works’ results, by showing a negligible overhead to correct existing PageRank algorithms.
are sparse [2], modifying the graph such that each sink vertex is connected to all other vertices makes the graph far denser (expanding the number of edges by orders of magnitude, and leading to large overheads – see Table II in the evaluation section).

A way around this is to use an aggregation strategy which can be employed in the context of a synchronous PageRank implementation. In a given superstep iteration, once all vertices have received their respective incoming messages and are ready to distribute their new messages, a global barrier performs a side calculation that accumulates all sink vertex outbound messages, adds all values, divides the result by \( |G| - 1 \), and distributes this value to all vertices as an inbound message for the next step of the iteration (with some extra logic – for instance, a sink vertex that contributed to this global value would modulate by subtracting its own contribution). This strategy is a correct and effective solution, but locks into place the notion of global superstep iterations, whereby a strategy like Bulk Synchronous Parallel must be used.

For an asynchronous implementation of PageRank (whereby a vertex is not restricted in propagating messages, or advancing its iteration before a global barrier), the sink-aggregation strategy can no longer piggyback on the barrier between supersteps. For a vertex to make an independent decision on advancing its value in an asynchronous platform, continuously broadcasting to all other vertices has intractable overheads (far larger than the advantage offered by using an asynchronous algorithm), while ignoring the problem leads to incorrect results.

**Incorrect and “Leaky” PageRank.** If a solution has no special case for handling sink vertices, then the resulting PageRank scores will be incorrect in two ways:

- **Leaky.** Since the PageRank score is the landing probability of a random surfer on a specific page, then the following invariant should hold: the sum of all PageRank scores (landing probabilities) over all pages should be one. If a sink vertex does not distribute its inbound score outward, this score is “leaked” from the system and the invariant is violated.

- **Incorrect Ranking.** Consider the trivial case of a graph with three vertices and two edges, forming an \( A \rightarrow B \rightarrow C \) chain. Here, the correct PageRank result has \( B \) with a slightly higher score than \( C \) (e.g., 21%, 40%, 39%). This is expected, as although the combined flow ends in \( C \), \( C \) redistributes to both \( A \) and \( B \) with sink vertex handling (which leads to \( B \)’s higher score). However, without sink vertex handling, \( C \) consequentially becomes the highest scoring vertex (e.g., 5%, 9%, 13%). This distinction may lead to real-world consequences: expanding this trivial example by adding any number of vertices all pointing to \( B \) (clearly defining \( B \) as an important vertex), the result still always marks \( C \) as the most important if not handled correctly. In effect, a vertex would be able to game the score system by avoiding outgoing edges.

### Algorithm 1: Asynchronous PageRank Pseudocode

```go
func (g *Graph) Init() {
    E = 0.01 // Epsilon to determine significant change
    A = 0.85 // Alpha (1-probability to stay on page)
    for vidx := range g.Vertices {
        // Initialize by sending each vertex
        g.Vertices[vidx].inbox = []float64{1.0}
    }
}

func (vertex *Vertex) OnVisitVertex(g *Graph) bool {
    vertex.inbox = nil
    toDistrib := A * (vertex.Properties.residual)
    toAbsorb := (1.0 - A) * (vertex.Properties.residual)

    // Absorb and Distribute only if change exceeds E
    if math.Abs(toAbsorb / (vertex.Properties.mass - toAbsorb)) > E {
        vertex.Properties.residual = 0.0
        // Buffer incoming probability mass
        vertex.Properties.mass = toAbsorb
    }

    if len(vertex.OutEdges) > 0 {
        perE := toDistrib / float64(len(vertex.OutEdges))
        for eidx := range vertex.OutEdges {
            target := int(vertex.OutEdges[eidx].Target)
            g.Vertices[target].inbox = append(g.Vertices[target].inbox, perE)
        }
    } else {
        // Sink value... what to do?
        // for vidx := range g.Vertices | ... naive
        vertex.Properties.latent += toDistrib
    }
    return true // Did an update (useful for framework)
}

return false
```

### III. COMBATING SCORE LEAK

Previous work has identified improved solutions for synchronous PageRank, leveraging the observation that sink vertices have homogeneous behaviour, and thus enable message aggregation [7, 8]. While our solution does use this property, past work has focused on synchronous PageRank, and the proposed solutions do not work with asynchronous algorithms. We aim for a strategy that preserves the ability of a vertex to act independently – a key property that enables asynchronous algorithms.

**Key intuition behind our solution.** An attribute of asynchronous algorithms is their robustness to out-of-order behaviour – e.g., some calculations progress fast, and others slow, and many algorithms are immune to the consequences of message reordering. We can use this property to our advantage.

Assume that each sink vertex distributes to all other vertices, as intended and designed, but sending out these messages is delayed until PageRank scores have otherwise converged; yet each sink vertex has accumulated a backlog of messages it needs to propagate. Intelligently delaying all of these sink vertex computations until the very end exploits the property of irrelevant ordering, and we now have a final grouping of delayed or “latent” messages. This still contains a large amount of work that needs to be done, but it can be processed more efficiently, as we will show.
A. A Strategy for a Single Sink Vertex

We assume an implementation of PageRank that follows the notion of value update propagation \[13\]. We detail an brief example of this as Algorithm \[1\].

We define the term probability mass (often abbreviated as mass) as an intermediate update value that is propagated to and accumulated on a vertex by the PageRank algorithm. The current mass of a vertex \( v \in G \), noted by \( M(v) \), is what will determine the final vertex score (the landing probability is found at the end through normalization – dividing each vertex’s final mass by \( |G| \) if vertices have received one mass unit in the initialization step; line 6 in Algorithm \[1\]).

We outline our strategy as follows:

- We define what each sink vertex should do in preparation, during convergence: aggregate latent values, that is, aggregate the probability mass it intends to distribute.
- These values then need to be distributed to all vertices as if they were done so during the original convergence step.
- We identify cost-saving measures to distribute these values, by exploiting the homogeneity of the distribution process.
- Finally, we remove the need for true iterations by using a geometric series.

We start by presenting our solution from the perspective of a single sink, before moving to the more complex case of a graph with arbitrarily many sinks.

During computation, the sink will accumulate all messages that it should, but has not yet, distributed. This is \( L(s) \), the “latent” mass \( L \) the sink \( s \) has yet to distribute.

During the main execution of PageRank, e.g. Algorithm \[1\] upon receiving cumulative input probability mass \( m \) from neighbours, a vertex \( v \) absorbs into its held \( M(v) := M(v) + (1-\alpha) * (m) \) as usual – this is shown on line 23.

While a normal vertex would distribute \( \alpha * (m)/|nbrs(v)| \) to each of its neighbours (line 27), a sink node \( s \) will instead accumulate this into its latent value \( L(s) := L(s) + (\alpha) * (m) \) rather than distribute it to all vertices immediately, as shown on line 34.

When this phase has converged, we now let the latent values accumulated at sinks propagate. To optimize this process, we begin with a thought experiment: Suppose we were to insert new input mass of 1 to all vertices. We can determine that after convergence, as topology has not changed, the mass of all vertices would double – as the mass would simply flow through the system in the same way. We extend this observation to the mass accumulated at the sink vertex: the latent mass accumulated at the sink will be spread across all vertices, and some of it will end up again at the sink. We make two observations: First, the relative share of the mass each vertex receives is the same regardless of how much latent mass was distributed by the sink. Second, some mass would return to the sink, and the ratio between the final mass that ends up at the sink, and the the initial mass that is was distributed across the graph, stays constant regardless of the volume. We denote this ratio as \( \Phi \): the retained percentage at the sink.

Now we aim to determine \( \Phi \), the retained percentage at the sink, based on the values observed after inserting one unit of mass at each vertex, and letting the mass propagation converge while accumulating at the sink (as described earlier in this section).

This can be found as the latent mass accumulated strictly from incoming edges, divided by all other probability mass sources.

\[
\Phi := \frac{L(s) * \frac{1}{\alpha} - 1}{|G| - 1} \tag{1}
\]

We note the “gross up” of \( \frac{1}{\alpha} \), which is done to extract the pre-absorbed mass input from \( L(s) \), as \( L(s) \) is the dampened mass ready to be distributed. The sink was also initialized with the receipt of a mass of 1 unit – its own contribution to total mass – which we discount to compute inbound mass. Similarly, we discount one in the denominator of the ratio to consider only the non-sink vertices.

Once \( \Phi \), the retained percentage, has been computed, there is no need to do proper iterations over graph topology to distribute the latent mass accumulated at the sink. We can do “pseudo-iterations” of the sink distributing its latent mass to all other vertices; they are “pseudo” because we already know how much of what we distribute will come back to us in the next pseudo-iteration (this is the retained percentage). Further, we can already decide how much each other individual vertex

---

**Fig. 1:** A visual representation of an iteration of latent distribution, with the graph \( G \) separated from the sink \( s \). (1) \( s \) distributes \( L(s) \). (2) \( 1 - \Phi \) is allocated to the rest of the graph (each vertex receives a proportionate amount). (3) the amount returned to \( s \) is governed by \( \Phi \).

---

**TABLE I:** Summary of Notations and Terminology Used

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v \in G )</td>
<td>A page, aka vertex ( v ) in the graph ( G ), a set of vertices.</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>Probability a “surfer” continues rather than stays on a page.</td>
</tr>
<tr>
<td>mass</td>
<td>Raw aggregate “surfer” data; an un-normalized probability.</td>
</tr>
<tr>
<td>absorb</td>
<td>Amount of mass that stays on the page.</td>
</tr>
<tr>
<td>distribute</td>
<td>Amount of mass that leaves the page.</td>
</tr>
<tr>
<td>( M(v) )</td>
<td>Present absorbed mass of a vertex ( v ).</td>
</tr>
<tr>
<td>( M(v)' )</td>
<td>Final mass of ( v ). ( M(v)'/</td>
</tr>
<tr>
<td>( L(s) )</td>
<td>Present undistributed mass in a sink vertex ( s ).</td>
</tr>
<tr>
<td>( \Phi )</td>
<td>Retained percent; amount of new mass that would return to sink(s).</td>
</tr>
<tr>
<td>( \theta(v) )</td>
<td>Percentage a single sink ( v ) contributes to ( \Phi ).</td>
</tr>
</tbody>
</table>
will receive – it is relative to their current proportional mass, due to the homogeneous application.

During each pseudo-iteration (see Figure 1 for a visual aid):
- The sink $s$ distributes to all other vertices.
- All non-sink vertices absorb, in total: $(1 - \Phi) * L(s)$. This is the amount not retained by the sink. Each individual vertex would receive a relative value factored by its proportional mass.
- The sink absorbs its retained amount: $(1 - \alpha) * \Phi * L(s)$
- We set $L(s)$ to now be $\alpha * \Phi * L(s)$, and we repeat the pseudo-iteration.

The change in $L(s)$ between pseudo-iterations can be defined as:

$$L(s)_0 := L(s)$$
$$L(s)_{i+1} := \alpha * \Phi * L(s)_i$$  \hspace{1cm} (2)$$

The total absorption within the sink (i.e., to determine the final mass) is then defined as:

$$M(s)' := M(s) + L(s)_0 + (1 - \alpha) * \Phi + L(s)_1 + ... + L(s)_i + (1 - \alpha) * \Phi$$
$$= M(s) + \sum_{i=0}^{\infty} L(s)_i + (1 - \alpha) * \Phi$$  \hspace{1cm} (3)$$

Similarly, the total absorption within the non-sink graph can be defined. We note the total current mass as $M(G) = \sum_{v \in G} M(v)$. Each vertex will update its final mass as follows:

$$\forall v \in G \iff v \neq s :$$
$$M(v)' := M(v) + L(s)_0 + (1 - \Phi) * \frac{M(v)}{M(G) - M(s)} + L(s)_1 + ... + L(s)_{\infty} + (1 - \Phi) * \frac{M(v)}{M(G) - M(s)}$$
$$= M(v) + \sum_{i=0}^{\infty} L(s)_i + (1 - \Phi) * \frac{M(v)}{M(G) - M(s)}$$  \hspace{1cm} (4)$$

With the observation in equation (2) that $\alpha * \Phi \leq 1$, we can use the geometric series to convert $\sum_{i=0}^{\infty} L(s)_i$ to $\frac{L(s)_0}{1 - \alpha * \Phi}$. This removes all need for iterations in the previous formulas (as the other terms are constants).

B. Extension to Multiple Sink Vertices

With our derivation in place for a graph with only a single sink, we now show how to extend to many sinks. We imagine the graph as two partitions: the sinks $G_K$ and the rest of the graph $G_N$, where $|G_K|$ is the number of sinks, and $|G_N|$ is the number of vertices that are not sinks. We note $M(G_N) = \sum_{v \in G_N} M(v)$, i.e., the total probability mass currently absorbed by $G_N$.

Our retained percentage, which now reflects the total amount retained by $G_K$, is similar to the single sink case – however we now use $L(G_K) = \sum_{s \in G_K} L(s)$, the total summed latent mass among all sinks.

$$\Phi := \frac{L(G_K) * \frac{1}{\alpha} - |G_K|}{|G| - |G_K|}$$  \hspace{1cm} (5)$$

A single given sink vertex $s$ would have a relative contribution into this as follows:

$$\theta(s) := \frac{L(s) * \frac{1}{\alpha} - 1}{L(G_K) * \frac{1}{\alpha} - |G_K|}$$  \hspace{1cm} (6)$$

However, within a pseudo-iteration, each sink vertex’s latent mass would be distributed partially to $G_K$, but now also partially to $G_N$. The percentage going to $G_N$ is straightforward, as it is simply a ratio of the number of vertices in $G_N$ compared to the total distributed count, which is to all vertices in the graph except itself. We define this as the “normal quota”: $Q_N$. In each pseudo-iteration, each sink would also contribute to every other sink. Each would receive one portion, so we define this ratio as the “single quota”: $Q_K$.

$$Q_N := \frac{|G_N|}{(|G| - 1)}$$
$$Q_K := \frac{1}{(|G| - 1)}$$  \hspace{1cm} (7)$$

It can be observed that $Q_N + Q_K * (|G_K| - 1) = 1$, i.e., we are representing the total distribution of one sink within $G_K$ to all other vertices.

As in the case of a single sink, we again consider pseudo-iterations. In a single pseudo-iteration, each individual sink would distribute mass proportionally and receive a retained value. From the perspective of a single sink vertex $v$, in a single step it would contribute $Q_N * L(s)$ to the normal graph, and to each other sink, $Q_K * L(s)$. During this process, as each other sink also contributes in the same manner, our example vertex $s$ would also receive from every other sink, i.e., $Q_K * \sum_{v \in G_K \iff u \neq s} L(u)$, or in other words, $Q_K$ multiplied by the sum of all sinks’ latent mass, less itself. Thus, $s$ receives $Q_K * (L(G_K) - L(s))$. A visual representation of this iteration can be seen in Figure 2.

We can quickly identify the total absorption by the normal graph – as all sinks contribute proportionally, we can simply use $L(G_K)$.

$$\forall v \in G_N :$$
$$M(v)' := M(v) + Q_N * \sum_{i=0}^{\infty} L(G_K)_i * (1 - \Phi) * \frac{M(v)}{M(G_N)}$$  \hspace{1cm} (8)$$

The absorption by a given sink $s$ is as follows:

$$M(s)' := M(s) + \sum_{i=0}^{\infty} [(1 - \alpha) * Q_K * (L(G_K)_i - L(s)_i) + (1 - \alpha) * Q_N * (\theta(s) * \Phi) * L(G_K)_i]$$  \hspace{1cm} (9)$$

For a sink $s$, we can define its pseudo-iteration as follows:

$$L(s)_0 := L(s)$$
$$L(s)_{i+1} := \alpha * Q_K * (L(G_K)_i - L(s)_i) + \alpha * Q_N * (\theta(s) * \Phi) * L(G_K)_i$$  \hspace{1cm} (10)$$

It would follow that we could iterate $L$ for each sink, and sum them to give us a new $L(G_K)_i$ for each step of
the pseudo-iteration. However, we can determine $L(K_i)_i$ independently:

$$L(K_i)_{i+1} = \sum_{s \in G_K} L(s)_{i+1}$$

$$= \alpha \ast Q_K \ast (L(G_K)_{i} - L(s)_i) + \ldots \text{(term } \forall s)$$

$$+ \alpha \ast Q_N \ast (\theta(s) \ast G_K)_{i} + \ldots \text{(term } \forall s)$$

$$= \lambda$$

We can transform the two factors by using the observation that each $L(s)_i$ is subtracted once, and the sum of all $\theta(s)$ is one. Finally, we can once again use the geometric series to remove the infinite sum, defining this constant as $\lambda$.

Finally, we can use $\lambda$ and the same rationale of constants to conclude a single step equation for each non-sink $v$, reformulating equation $8$ as follows:

$$\forall v \in G_N :$$

$$M(v)' := M(v) + Q_N \ast \lambda \ast (1 - \Phi) \ast \frac{M(v)}{M(G_N)}$$

$\text{C. The Complete Algorithm}$

Using equations $13$ and $14$ each vertex can be independently adjusted to the final value. Indeed, this step can be performed after the conclusion of the traditional PageRank computation, and applied during a final step – where typically a normalization of probability is performed – so long as the latent mass values were accumulated during processing. We show how this completed algorithm can finalize the PageRank score result in $O(|G|)$ time in Algorithm $2$.

$\text{D. Correctness Argument}$

It has been shown previously that this style of asynchronous mass propagation converges, as, after each vertex propagates mass, the amount that is left to (re)distribute in the system strictly decreases and trends to zero as it is continuously dampened by $\alpha$ [13, 20]. As these propagated updates are commutative and associative [13, 18], re-ordering these messages (as we do for sink vertices) has no impact on the ability to converge. Similarly, the argument holds that the latent values from the sink vertices also converge, as they also trend to zero by damping with both $\Phi$ and $\alpha$ (which is also what allows us to simplify with the geometric series).

In terms of maintaining the invariant that the sum of mass over all vertices remains constant, we note that our solution is mathematically equivalent to the naive solution, with our algorithm being an algebraic simplification taking advantage of identified properties. In our algorithm, no mass (and therefore probability) is ever discarded or lost.
Algorithm 2: Final Normalization Psuedocode

```text
numSinks := 0
sumLatent := float64(0.0)
nonSinkSum := float64(0.0)

// Pass over all vertices to accumulate global totals.
for _, vertex := range g.Vertices {
    if len(vertex.OutEdges) == 0 {
        sumLatent += vertex.Properties.latent
        numSinks++
    } else {
        nonSinkSum += vertex.Properties.mass
    }
}

// See Equation 12.
normalQuota := float64(len(g.Vertices)-numSinks) / float64(len(g.Vertices))

// See Equation 13.
relativePct := vertex.Properties.latent*(1.0/A) - retainSumPct

toAbsorb = (1.0 - A) * (geometricSumLatent) * (singleQuota) * (1.0 - retainSumPct) * (vertex.Properties.mass /= float64(len(g.Vertices)))
value := toAbsorb

// See Equation 14.
toAbsorb = (1.0 - retainSumPct) * (vertex.Properties.mass / float64(numSinks))

// A Sink vertex: See Equation 15.
toAbsorb = (1.0 - retainSumPct) * (vertex.Properties.mass / float64(len(g.Vertices)))

// Can be amortized inside this O(|G|) operation.
g.Vertices[vidx].Properties.mass += toAbsorb
```

### IV. Evaluation

As we have shown a theoretically correct algorithm (and experimentally, verified its correctness on trial graphs), our evaluation primarily intends to confirm the claim of negligible overhead. To this end, we do not intend to dwell on advantages of asynchronous PageRank over a synchronous version (previous work has already demonstrated this advantage) nor to extract maximum performance or compare frameworks – instead, we are simply focused on the overhead for each correctness strategy.

Our evaluation also has a set of secondary objectives: (i) to show that our strategy is, by far, preferable to restructuring the graph or communication from sinks to all other vertices, and (ii) to show that the proposed strategy has advantages over prior work for synchronous PageRank, as it reduces inline computation costs by performing a final score reassignment to vertices only once at the end of processing.

### A. Experimental Setup

**Graph Processing Frameworks.** To evaluate performance, we implemented our solution in the Galois framework, fixing both its synchronous and asynchronous PageRank implementations. As Galois works on shared memory architectures, to evaluate our solution in a shared-nothing context, we have also constructed a simple PageRank algorithm in HavoqGT – an asynchronous, distributed, shared-nothing framework – based on Algorithm 1 and implemented our solution as well.

**Datasets.** We use two graphs from the Snap datasets, and the Twitter graph, with properties highlighted in Table II.

**Machines.** For Galois results, we use a single machine, with an 8-core Intel(R) Xeon(R) Gold 6244 CPU @ 3.60GHz, 256 GB memory, and a Samsung 970 EVO Plus 2TB NVME. For HavoqGT results, we use a cluster where each node is a dual 12-core Intel(R) Xeon(R) E5-2695v2 (2.4 GHz) processor, with 128 GB memory, and Intel 910 PCI-attached NAND Flash.

**Reporting runtime and other details.** When reporting runtime, we separate the time for graph conversion, graph loading, and initialization time; we focus on algorithm time, and average results over 10 executions. We use and modify Galois version 6.0, the cpu-push version of PageRank, with default tolerance ($\varepsilon$) as 0.001. For HavoqGT, we dynamically build the graph (with no a-priory knowledge of the data), then run a static PageRank on this constructed graph – we do this as it reinforces our approach not requiring any graph modifications or pre-computations. Due to the larger graph size and larger power law degree for Twitter, we keep execution times reasonable by changing $\varepsilon$ to 0.1. In all cases, the Fix/Norm stage is the final step of either only normalizing, or both using our solution and normalizing (amortized).

**Sensitivity to PageRank parameters.** We use $\alpha$ (damping factor) of 0.85, as used in the original experiments with PageRank and most of the following work. We note varying this factor (which influences variability across page scores) may impact convergence time, yet the relative convergence rate between the methods we compare is not impacted. The tolerance $\varepsilon$ is a stopping criteria (an alternative to stopping after some number of synchronous PageRank iterations), and represents a sensitivity to changes of the intermediate solution. It has been shown that asynchronous PageRank solutions

### TABLE II: Graph Properties

| Graph      | | G | | G | m(in) | m(out) | | E | | Ed |
|------------|---------------|----|----|----------|----------|----------|----------|------|------|
| webGoogle  | 0.9M          | 0.1M | 16% | 6k       | 0.5K      | 5M        | 0.1T     |
| LiveJournal| 4.8M          | 0.5M | 11% | 20K      | 14K       | 69M       | 2.6T     |
| Twitter    | 42M           | 1.5M | 4%  | 770K     | 3M        | 1.5B      | 64T      |
converge faster than synchronous ones. Based on this, and on the fact that our fix adds minimal overheads as this section demonstrates, we argue that our conclusions hold regardless of the choice of stopping criteria.

B. Galois Results

The results for the first two datasets are presented in Table [III]. The runtime of the original versions of Galois for both Sync and Async are shown, for reference of the performance of incorrect results (labeled Original in Table [III]). The asynchronous solution is fixed as described in Algorithm 2 (Async-EndFix).

We explore the performance of various alternatives.

- First, we explore the performance of the naive solution that adds edges from all sinks (Async-Naive). The runtime is huge as the graph effectively becomes much larger.

- Second, for the synchronous solution, we explore two types of fixes: a traditional per-iteration fix (Sync-PerIter), as well as using our Algorithm 2 (Sync-EndFix). The traditional fix works as follows: during an iteration, all sink propagation values are accumulated. At the conclusion of the iteration, this value is distributed to each vertex as required (sinks subtract their contribution when receiving). This result is correct, and does not require any extra finalization, but comes with a high performance penalty: as Galois operates with an optimized “work queue” of active vertices, this fix will re-activate vertices that the incorrect implementation would ignore. As such, the original incorrect version sees the work queue shrink quicker and stop earlier, though with leaked probability and thus incorrect values.

We now shift focus on the second part of Table [III] where the results on the LiveJournal dataset are presented. While for the smaller webGraph the normalization stage was too short to accurately compare, here we can highlight the low cost to the normalization stage itself that our solution generates. The naive async version (whereby each vertex would distribute to all other vertices) did not complete in a reasonable time on this larger graph.

We note here that (Sync-PerIter) and (Sync-EndFix) are also representative of prior work, the first being a simple correct synchronous version, and the second akin to the best achievable synchronous performance with intelligently handled sink vertices (e.g., partitioning them and modifying adjacency lists [7, 8]) – this is the case as it can be no faster than ignoring the sink vertices completely. Result correctness would be the same, however, this prior work would need to include further execution time in the graph conversion and load times, as they require additional restructuring and partitioning of the graph topology prior to execution.

C. HavoqGT Results

Table [IV] presents the performance of an asynchronous PageRank implementation (built on top of the HavoqGT framework) on a distributed shared-noting platform, and using a much larger graph, the Twitter dataset. We note there is no graph conversion time, as we stream the graph into the framework dynamically (as fast as it can handle, and measure this time), constructing the graph on-the-fly with no a-priori knowledge of structure. While this may reduce performance of the algorithm (due to an unoptimized layout), it showcases the flexibility of our solution. Like Galois, we see a negligible impact of performance in the final phase – indeed, the resolution phase even scales linearly with node count, as most of the work is done locally and in parallel, with only a few MPI all-reduce calls to compute global values.

D. Takeaways

The results highlight two key points: first, the overhead of our algorithmic resolution on an asynchronous implementation is negligible, with approximately a 0.1% performance impact. This is expected, as the correctness solution only requires an extra \(O(|G|)\) value accumulation before normalization. Second, even a synchronous implementation benefits from the proposed solution as it avoids an extra \(O(|G|)\) operation on each iteration (distributing the accumulated sink value to all vertices), while also reducing the required work by not re-activating vertices when this value is distributed. Due to this, the fixed synchronous mode’s performance effectively matches the performance of the original (and incorrect) solution, and is much faster than a per-iteration fix.

V. RELATED WORK

Asynchronous PageRank is well described by Sankaralingam et al., where they introduced the basic principle for how each vertex independently steps through the process of adjusting its vertex score from inbound messages, then propagating to outbound neighbours [18]. This design has become the basis for distributed and asynchronous PageRank implementations, yet unfortunately, their paper did not describe how to handle sink vertices (if at all).
Lee et al. provided the closest related work to ours, also targeting sink vertices, but in the synchronous PageRank domain [8]. Notably, their target is solely improved performance, and contains similar intuition to our algorithm. They note the homogeneity of how sinks operate, and use this advantage alongside the lumpability of Markov chains [4]. Their algorithm is a two stage process – first physically partitioning the sink vertices and computing PageRank on the remaining connected graph to compute two separate vectors, and second performing a tertiary iterative method with the two final vectors. Importantly, their result is both correct and typically offers a performance gain over a per-iteration fix. Expanding upon this design, Langville et al. [6, 7] provided a linear system formulation that also uniquely takes advantage of sink vertex homogeneity by reordering the graph’s adjacency list. We expect each of these solutions to offer performance identical to our algorithm’s advantage over the per-iteration fix, for synchronous PageRank, if we discount their graph restructuring time (which our solution does not require).

Our algorithm can be thought of as an asynchronous-focused variant leveraging prior observations (i.e., the homogeneity also enables our algorithm to operate). Their finding of improved performance by not requiring calculations with the sinks during main iterations is a key advantage that is shared, however our algorithm supersedes prior work by adding key advantages – primarily its applicability to asynchronous solutions – as highlighted in the introduction.

VI. CONCLUSION

We explored several synchronous and asynchronous graph frameworks, finding them to have “leaky” and incorrect PageRank results on directed graphs which have sink vertices. We proposed and evaluated a novel algorithm to solve these issues in $O(|G|)$ time. This algorithm is friendly to asynchronous PageRank, does not require shared memory, and induces negligible overhead (only a minor 0.1% overhead when implemented in Galois and HavoqGT). Furthermore, we anticipate this algorithm will be essential to support PageRank on dynamic or streaming graphs, which we aim to target in future work.

VII. ACKNOWLEDGMENTS

This project was sponsored in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by a gift fund from Huawei Toronto Heterogeneous Compiler Lab in Canada.

REFERENCES


