Parallel Personalized PageRank on Dynamic Graphs

Wentian Guo, Yuchen Li, Mo Sha, Kian-Lee Tan
School of Computing, National University of Singapore
{wentian, liyuchen, sham, tankl}@comp.nus.edu.sg

ABSTRACT

Personalized PageRank (PPR) is a well-known proximity measure in graphs. To meet the need for dynamic PPR maintenance, recent works have proposed a local update scheme to support incremental computation. Nevertheless, sequential execution of the scheme is still too slow for high-speed stream processing. Therefore, we are motivated to design a parallel approach for dynamic PPR computation. First, as updates always come in batches, we devise a batch processing method to reduce synchronization cost among every single update and enable more parallelism for iterative parallel execution. Our theoretical analysis shows that the parallel approach has the same asymptotic complexity as the sequential approach. Second, we devise novel optimization techniques to effectively reduce runtime overheads for parallel processes. Experimental evaluation shows that our parallel algorithm can achieve orders of magnitude speedups on GPUs and multi-core CPUs compared with the state-of-the-art sequential algorithm.

PVLDB Reference Format:
Wentian Guo, Yuchen Li, Mo Sha, Kian-Lee Tan. Parallel Personalized PageRank on Dynamic Graphs. *PVLDB*, 11(1): 93-106, 2017.
DOI: https://doi.org/10.14778/3136610.3136618

1. INTRODUCTION

Personalized PageRank is one of the most widely used proximity measure in graphs. While PageRank quantifies the importance of vertices overall, the PPR vector measures the importance of vertices w.r.t to a given source vertex s. For instance, a vertex v with a larger PPR value w.r.t. s implies that a random walk starting from s has a higher probability to reach v. PPR is used in applications of many fields, such as web search for Internet individuals [39, 22], user recommendation [8, 19], community detection [48] and graph partitioning [6, 5].

Most existing works [39, 22, 6, 5, 30, 31, 29, 11] focus on PPR computation on static graphs. However, the representative graphs often evolve rapidly in many applications. For example, network traffic data averages 10^9 packets/hour/router for large ISPs [17] and Twitter has more than 500 million tweets per day [3]. Since computation of PPR from scratch is prohibitively slow against high rate of graph updates, recent works [38, 49] focus on incremental PPR maintenance. The state-of-the-art dynamic PPR approaches are based on a local update scheme [6, 5, 11, 31, 29, 38, 49]. This scheme takes two steps against each single update: (1) restore *invariants*; (2) perform *local pushes*. More specifically, each vertex v keeps two values $P_s(v)$ and $R_s(v)$ where $P_s(v)$ is the current estimate to the PPR value of v w.r.t. the source vertex s and $R_s(v)$ is the upper bound on the estimation bias. We call $P_s(v)$ and $R_s(v)$ as the estimate and residual of v respectively. The invariant is maintained for all vertices to ensure the residual is valid. To handle any update, e.g., an edge insertion/deletion, the invariant is first restored which results in change in the residual. Subsequently, if the residual of a vertex v is over a user-specified threshold, i.e., ϵ, the residual is pushed to v's neighbors.

Although it has been shown in [49] that only a few operations are needed to handle any single edge update (insertion/deletion) under the common edge arrival models, it is practically inefficient. We have observed latency of up to $2-3$ minutes against an update batch consisting of 5000 edges using the state-of-the-art approach [49] in our experiments. This hardly matches the streaming rates of real-world dynamic graphs (e.g., 7000 tweets are generated per second in Twitter [3]). Meanwhile, the emerging hardware architectures provide the consumers with massive computation power. The two of the most popular hardwares, multi-core CPUs and GPUs have shown their great potentials in accelerating the graph applications [42, 36, 47, 34, 25, 35]. By utilizing the emerging hardwares, we are motivated to devise an efficient parallel approach for dynamic PPR computation under the state-of-the-art local update scheme to achieve high-speed stream processing.

Parallelizing dynamic PPR computation poses two new challenges. First, as the update often comes in batches, it remains unclear if the batch can be updated in parallel for the local update scheme. A naive solution is to synchronize on each single update to repair the invariant and then launch the local push. However, this solution causes significant synchronization overheads against batch updates. Second, parallelizing the local push is non-trivial as it incurs extra overheads compared with a sequential approach. We call the local push getting parallelized as the parallel push. The extra overheads of the parallel push come from

Proceedings of the VLDB Endowment, Vol. 11, No. 1

DOI: https://doi.org/10.14778/3136610.3136618

two sources: (a) the parallel push, can take more operations, as it reads stale values of the residual at the beginning of iterations and then propagates less probability mass from the frontier vertices (the frontier contains vertices that need to be pushed in current iteration); (b) the parallel push needs to iteratively maintain a vertex frontier and thus results in synchronization costs to merge duplicate vertices.

In this paper, we propose a novel parallel approach to tackle the aforementioned challenges. First, we devise a batch processing method to avoid synchronization costs within a batch as well as to generate enough workloads for parallelism. Our theoretical analysis proves that the parallel approach has the same asymptotic complexity as its sequential counterparts. Second, we propose novel optimization techniques to practically reduce the number of local push operations (*eager propagation*) and to efficiently maintain the vertex frontier in parallel by utilizing the monotonicity of the local push process (*local duplicate detection*).

We hereby summarize our contributions as follows:

- We introduce a parallel approach to support dynamic PPR computation over high-speed graph streams. One salient feature is that updates are processed in batches. Theoretical analysis shows that the parallel approach has the same asymptotic complexity with its sequential counterpart under the two common edge arrival models. To the best of our knowledge, this is the first work on parallelizing dynamic PPR processing.
- We propose several optimization techniques to improve the performance of the parallel push. Eager propagation helps to reduce the number of local push operations. Our novel frontier generation method efficiently maintains the vertex frontier by mitigating synchronization overheads to merge duplicate vertices.
- We implement our parallel approach on GPUs and multi-core CPUs. The experiments over real-world datasets have verified the effectiveness of our proposal over both architectures. The proposed approach can achieve orders of magnitude speedups on GPUs and multi-core CPUs compared to the state-of-the-art sequential approach.

The remaining part of this paper is organized as follows. Section 2 introduces preliminaries and the background. Section 3 presents the parallel local update approach, together with its theoretical analysis. We propose the optimizations for the parallel push in Section 4. Section 5 reports the experimental results. The related works are discussed in Section 6. Finally, we conclude the paper in Section 7.

2. PRELIMINARY

In this section, we briefly describe the PPR problem and review the state-of-the-art local update scheme for dynamic PPR maintenance.

2.1 Personalized PageRank (PPR)

Let $G(V, E)$ be a directed graph with the vertex set V and the edge set E. Intuitively, PPR can be interpreted as a random walk process on G that starts from a source vertex s and then iteratively jumps to an uniformly chosen out-neighbor or teleports to s with probability α. The PPR value $\pi_s(v)$ is the probability that such random walk starts from s and stops at v. We denote A as the adjacent matrix of G and D as the diagonal matrix, where the diagonal entries

Algorithm 1 RestoreInvariant

Input: (G,P_s,R_s,u,v,op)
Require: (u, v, op) is the updating edge with op being the insert/delete operation.
1: **procedure** Insert(u, v)
2: $\quad R_s(u)$ += $\frac{(1-\alpha)\cdot P_s(v)-P_s(u)-\alpha\cdot R_s(u)+\alpha\cdot \mathbf{1}_{u=s}}{d_{out}(u)} \cdot \frac{1}{\alpha}$
3: **procedure** Delete(u, v)
4: $\quad R_s(u)$ −= $\frac{(1-\alpha)\cdot P_s(v)-P_s(u)-\alpha\cdot R_s(u)+\alpha\cdot \mathbf{1}_{u=s}}{d_{out}(u)} \cdot \frac{1}{\alpha}$

are the out degrees of all vertices in G. The PPR vector π_s w.r.t. s is the solution of the following linear equation:

$$\pi_s = \alpha e_s + (1 - \alpha)W\pi_s \quad (1)$$

where α is the teleport probability (which is a constant and it is typically set to 0.15), e_s is a personalized unit vector with a single non-zero entry of 1 at s, and W is the transition matrix s.t. $W = A^T D^{-1}$.

Note that general PPR formulation does not restrict the personalized vector to be a unit vector. We omit the discussion for the general case as it can be reduced to the case with the unit vector scenario. In fact, many existing works study how to use the unit vector formulation as a building block to efficiently support the general case by maintaining multiple PPR vectors with different personalized unit vectors. Interested readers are referred to [49, 38, 10, 31, 6, 9, 29, 11]. Henceforth, this paper focuses on providing efficient parallel approach for the unit vector formalization.

2.2 Dynamic Graph Model

We introduce a typical dynamic graph model [10, 49] to articulate the dynamic PPR problem. The dynamic model consists of an unbounded sequence of updates ${}^{\Delta}E^t$, which indicate the set of edges arriving at time step t. Each element of ${}^{\Delta}E^t$ is (u, v, op) that means an edge $u \to v$ with op denoting the type (insertion/deletion). Note that in previous works [38, 49] only one single edge update arrives at time step t, in the following we will discuss them with the same notations but readers should notice $\left|{}^{\Delta}E^t\right| = 1$ for the case of single update.

Initially, the graph is $G^0 = (V^0, E^0)$ at time step 0. At time step $t > 0$, ${}^{\Delta}E^t$ is applied to G^{t-1} and produces a new graph as G^t [1]. Note that an edge insertion may introduce new vertices to V^t and the deletion of an edge may discard some vertices with zero degree from V^{t-1}. Given the model, the goal of dynamic PPR problem is to incrementally maintain the PPR vector for each update.

2.3 Sequential Local Update

The state-of-the-art approaches on dynamic PPR vector maintenance relies on a local update scheme [49, 6, 5]. The PPR value of a vertex returned by the local update scheme is an ϵ-approximation to the true PPR value, i.e., the absolute error is smaller than the error threshold ϵ. The scheme keeps two values for each vertex v: $P_s(v)$ and $R_s(v)$ where $P_s(v)$ is the current estimate to the PPR value of v w.r.t. the source vertex s, and $R_s(v)$ is the "residual" which bounds the estimation bias. Given a vertex $v \in V$ with its out neighbor

[1]To simplify the presentation, the time step subscript is dropped when the context is clear.

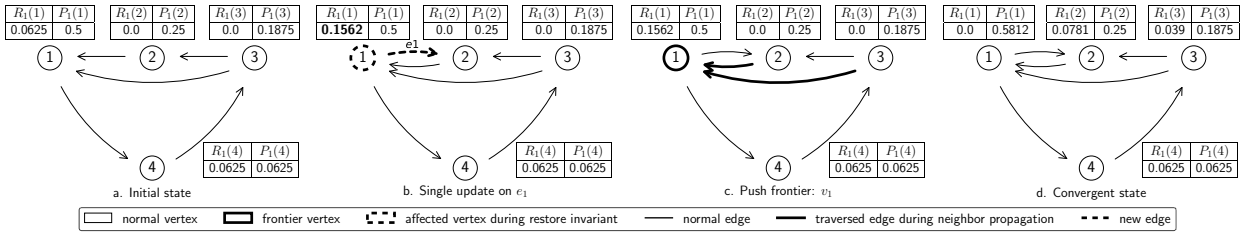

Figure 1: An example of the sequential local update. Assuming $\alpha = 0.5$ and $\epsilon = 0.1$. When a new edge e_1 arrive, invoke Algorithm 1 to restore the invariant for v_1 and then launch Algorithm 2 to proceed the local push.

Algorithm 2 SequentialLocalPush

Input: (G, P_s, R_s, ϵ)
1: **while** $\max_u R_s(u) > \epsilon$ **do**
2: SeqPush(u)
3: **while** $min_u R_s(u) < -\epsilon$ **do**
4: SeqPush(u)
5: **return** (P_s, R_s)

6: **procedure** SeqPush(u)
7: $P_s(u)$ += $\alpha \cdot R_s(u)$
8: **for all** $v \in N_{in}(u)$ **do**
9: $R_s(v)$ += $(1-\alpha) \cdot R_s(u)/d_{out}(v)$
10: $R_s(u) = 0$

set $N_{out}(v)$ and out degrees $d_{out}(v)$, an invariant [49, 6] is formulated as follows to ensure $P_s(v)$ is $R_s(v)$-approximate.

$$P_s(v) + \alpha R_s(v) = \sum_{x \in N_{out}(v)} (1-\alpha) \cdot \frac{P_s(x)}{d_{out}(v)} + \alpha \cdot \mathbf{1}_{v=s} \quad (2)$$

To handle an update in the dynamic graph model, the framework of the scheme takes a two-step procedure: (1) restore *invariant*; (2) perform *local pushes*. When an edge update (u, v, op) arrives, the scheme repairs the invariant specified in Equation 2, which may change the residual value of u and v. In case this ***activates*** any vertex, which means the residual value exceeds ϵ, the local push is invoked. It updates the estimate of this vertex and propagates the residual to its neighbors, which may activate more vertices. The process continues until no residual exceeds ϵ.

The step (1) and step (2) of the local update scheme are presented in Algorithm 1 (RestoreInvariant) and Algorithm 2 (SequentialLocalPush) respectively. RestoreInvariant describes how to maintain the invariant for all vertices for an update of directed edge $u \to v$. We present the case for insertion and the case for deletion is similar. Note that the only vertex where the invariant does not hold is u as $d_{out}(u)$ becomes larger with the extra out-neighbor v. In this case, it suffices to change only $R_s(u)$ without touching any other vertices. After invoking RestoreInvariant, if the residual value exceeds ϵ, SequentialLocalPush is launched to preserve the approximate guarantee.

There are two phases in SequentialLocalPush. The first one handles vertices with positive excessive residuals and the second one deals with negative residuals. Each phase iteratively invokes SeqPush (Lines 6-10) until no residual has an absolute value larger than ϵ. When this happens, it is guaranteed that $|\pi_s(u) - P_s(u)| \leq \epsilon$ for any $u \in V$. And we say the local push *converges* at this time. Lemma 1 ensures the invariant holds throughout the local push.

Lemma 1 ([5]). *For any vector p, the following transformation from P_s and R_s to P'_s and R'_s maintains the invariant shown by Equation 2.*

$$P'_s = P_s + \alpha p$$
$$R'_s = R_s - p + (1-\alpha) W^T p$$

Zhang et al. [49] prove the complexity required for the sequential local update to maintain the PPR vector under two common edge arrival models.

Theorem 1 ([49]). *Given a random source vertex, the number of operations required by the sequential local update to maintain a valid approximate solution on the dynamic graph is at most $O(K + K/(n\epsilon) + \bar{d}/\epsilon)$ for K updates ($\bar{d}$ is the average degree), under two edge arrival models: random edge permutation of a directed graph and arbitrary edge updates of an undirected graph.*

Example 1. *We show a running example in Figure 1 to illustrate the sequential local update. As a new edge $v_1 \to v_2$ arrives in Figure 1(b),* RestoreInvariant *is invoked to modify $R_1(1)$, which will activate v_1.* SeqPush *performs local push on v_1. It updates $R_1(2)$ and $R_1(3)$ but does not activate v_2 and v_3. Since there is no active vertex afterwards, convergence is achieved.*

Although there are many variants of the local update scheme [49, 6, 5, 38, 31, 11], they share the similar behavior (i.e., restore invariant against an update and then perform local pushes to reduce the residuals). We thus focus on the state-of-the-art approach proposed in [49] and omit other approaches as there is no essential difference in parallelizing them compared to parallelizing the approach in [49]. Before moving on to the technical contributions of this paper, we summarize the frequently used notations in Table 1.

3. DYNAMIC PPR IN PARALLEL

As previously mentioned in the Introduction, the sequential approach for dynamic PPR is practically inefficient to meet the requirement for real-world stream processing. We thus propose a novel parallel approach to boost the performance of dynamic PPR computation.

3.1 Parallel Local Update

A naive parallel approach is to synchronize on each single update to repair the invariant and then push all vertices in the *frontier* in parallel (the frontier consists of vertices which have larger residuals than ϵ). However, a single edge update is not expected to result in substantial change in the representative graph, and hence the residuals will not

Table 1: Frequently used notations in this paper.

Symbol	Descriptions
G^t, V^t, E^t	the graph, the vertex set and the edge set at time step t
$N_{in}(u), N_{out}(u)$	in- and out- neighbor set of u
$d_{out}(u)$	Out-degree of u
$\overline{d}$	the average degree
K	Total number of edge updates
s	the source vertex of PPR
α	Teleport probability of PPR
π_s	True PPR vector w.r.t s
e_s	Unit vector with 1 at s
ϵ	Error threshold
P_s	Estimate PPR vector w.r.t. s
R_s	Residual vector of P_s w.r.t. s

vary significantly as well. Thus, the single update scenario often causes a small frontier for any parallel approach to be effective. Moreover, as updates often come in batches, the naive approach causes significant synchronization overheads against batch updates.

In this paper, we propose a simple-to-implement yet effective approach to support efficient batch processing in parallel. Given a batch of k updates, we first restore the invariant by calling RestoreInvariant (Algorithm 1) k times to adapt to the graph changes for the batch and then parallelize the local push. The critical component of our approach is the parallel local push presented in Algorithm 3. Similar to SequentialLocalPush (Algorithm 2), there are two phases to handle positive and negative residuals respectively. In each phase, it first initializes a frontier queue FQ and then iteratively invokes the push procedure (ParallelPush) until no vertex in the frontier. ParallelPush consists of two parallel *sessions*. The first session takes out the residual of each frontier vertex, and increments the estimate by α portion of its residual. The second session updates the neighbors of the frontier vertex with the remaining $1-\alpha$ portion of the residual and generates a new frontier queue FQ' if necessary. To simplify the presentation, we refer to the first parallel session as ***self-update*** and the second session as ***neighbor-propagation***.

We note that in Line 21, atomic operations are utilized to ensure the residuals are correctly transferred from the frontier to their neighbors. An alternative method to avoid atomic operations is to run a parallel sort on the key-value pairs consisting of the neighbor vertex ids and the corresponding residuals. Then, it runs a parallel reduce operation to aggregate the residuals with the same keys and then transfer the residuals to the designated vertices. However, this sorting-and-aggregate method incurs significant overheads for large frontiers. In fact, most graph processing systems [42, 47, 16, 36, 23] and similar applications [13, 33, 44, 43, 35] adopt atomic operations over the sorting-and-aggregate method to update the neighbors. Thus, we use the atomic method in our implementation [2].

Example 2. *We continue the running example in Figure 2 for the parallel local update. Two new insertions $v_1 \to v_2$ and $v_4 \to v_1$ arrive in Figure 2(b),* RestoreInvariant *is invoked for both insertions and changes the residuals of v_1 and v_4. Then, v_1 and v_4 become the frontier as their residuals exceed ϵ and* ParallelLocalPush *is launched correspondingly. It achieves convergence in one iteration.*

[2]We will not show the experimental results for using the sorting-and-aggregate method as the performance is significantly worse than the atomic method according to our investigations and results.

Algorithm 3 ParallelLocalPush

```
Input: (P_s,R_s,G,ε)
 1: FQ = {u ∈ V | pushCond(R_s(u), POS)}
 2: while FQ ≠ ∅ do
 3:     FQ = ParallelPush(P_s, R_s, FQ, POS)
 4: FQ = {u ∈ V | pushCond(R_s(u), NEG)}
 5: while FQ ≠ ∅ do
 6:     FQ = ParallelPush(P_s, R_s, FQ, NEG)
 7: return (P_s, R_s)

 8: procedure pushCond(r, phase)
 9:     if phase = POS then return r > ε
10:     else return r < −ε
11: procedure ParallelPush(P_s, R_s, FQ, phase)
12:     S = ∅
13:     parallel for u ∈ FQ
14:         S = S ∪ (u, R_s(u))
15:         P_s(u) += α · R_s(u)
16:         R_s(u) = 0
17:     synchronize
18:     FQ' = ∅
19:     parallel for (u, w) ∈ S
20:         parallel for (v, u) ∈ N_in(u)
21:             atomicAdd(R_s(v), (1 − α) · w/d_out(v))
22:             if pushCond(v, phase) then
23:                 UniqueEnqueue(FQ', v)
24:     synchronize
25:     return FQ'
```

3.2 Theoretical Analysis

Although it is apparently simple to implement our parallel approach, we note the theoretical analysis is intricate. In this section, we verify the soundness and efficiency of our proposed approach. First, we need to examine if the parallel approach produces a valid approximation result. Second, we check the complexity of the parallel approach against the sequential approach in terms of number of operations performed. In the following, we prove the correctness of the parallel push in Theorem 2.

Theorem 2. *For a given ϵ, the parallel push (Algorithm 3) can produce a valid approximate result with the same error threshold as the sequential push (Algorithm 2).*

Proof. We borrow Lemma 1 to aid our proof. Lemma 1 suggests that, as long as the vector p used for the self-update and the neighbor-propagation is the same, the invariant always holds. Let χ be the vector s.t. $\chi(u) = 1$ if $u \in FQ$ and zero otherwise, where $u \in V$. We rewrite the parallel push with a transformation in a vector form:

$$P'_s = P_s + \alpha \cdot (\chi \circ R_s)$$
$$R'_s = R_s - (\chi \circ R_s) + (1-\alpha)W^T(\chi \circ R_s)$$

Let $p = \chi \circ R_s$, the above transformation maintains the invariant by Lemma 1. The rest of the proof naturally follows as when Algorithm 3 terminates, $|R_s(u)| < \epsilon$ for any $u \in V$, and the invariant still holds. □

We are left to examine the complexity of the parallel local update. Previous study on the sequential local update [49] bounds the complexity for dynamic PPR maintenance by tracking the residual change caused by each edge update,

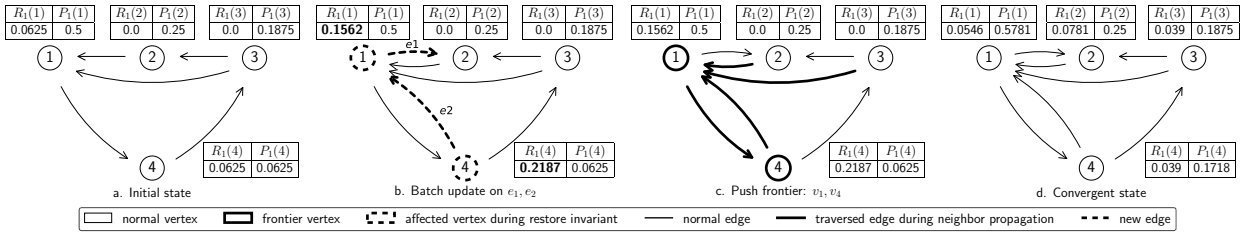

Figure 2: An example of the parallel local update. Assume $\alpha = 0.5$ and $\epsilon = 0.1$. When new edges e_1 and e_2 arrive, invoke Algorithm 1 to restore the invariant for both v_1 and v_4 and then launch Algorithm 3 to proceed the local push.

under the two common edge arrival models mentioned in Theorem 1. Following this method, we analyze the complexity for the parallel local update. The major difference is that we track the residual change caused by a batch update instead of a single update. Our subsequent analysis is structured as follows: Lemma 2 computes the number of operations required for the parallel local update as a function of the residual change. Lemma 3 thus deduces an upper bound on the residual change for an arbitrary batch update. By combining Lemmas 2 and 3, the overall complexity of the parallel local update is presented in Theorem 3.

LEMMA 2. *Given a number of t batches which collectively contains K updates and a random source vertex, the total number of operations required by the parallel local update to process the t batches is bounded by Ψ_d under random edge permutation of a directed graph.*

$$\Psi_d = \frac{\overline{d}}{\alpha\epsilon} + K \cdot \frac{\alpha+4}{n\alpha^2} + \frac{1}{n} \cdot \sum_{i=1}^{t} \sum_{u \in V} \frac{d^i_{out}(u) \cdot \sum_{s \in V} \Delta^i_s(u)}{\alpha\epsilon}$$

and it is bounded by Ψ_u under arbitrary edge updates of an undirected graph.

$$\Psi_u = \frac{\overline{d}}{\alpha\epsilon} + K \cdot \frac{2}{\alpha} + \frac{1}{n} \cdot \sum_{i=1}^{t} \sum_{u \in V} \frac{d^i_{out}(u) \cdot \sum_{s \in V} \Delta^i_s(u)}{\alpha\epsilon}$$

Note that $\Delta^i_s(u)$ is the residual change of a vertex u caused by RESTOREINVARIANT *at time step i.*

Since the proof process is similar to Theorem 1, we refer interested readers to our extended version [1]. Note that the expressions above are both associated with $\sum_{s \in V} \Delta^i_s(u)$, for which we derive an upper bound in the following lemma.

LEMMA 3. *At time step i, suppose the batch contains k directed edge updates that start from u, i.e. $(u \to v_j, op_j)$, for $1 \le j \le k$, we have:*

$$\sum_{s \in V} \Delta^i_s(u) \le \frac{2n\epsilon + 2}{\alpha d^i_{out}(u)} \cdot k$$

PROOF. To facilitate our proof, we define $op = 1$ for insertion and $op = -1$ for deletion. We further denote the residual and out-degrees of u after applying j^{th} update as $r_j(u)$ and $d_j(u)$. So $r_0(u)$ and $r_k(u)$ are the initial value and new value of $R_s(u)$ respectively after invoking RESTOREINVARIANT on these k updates. To bound $\sum_{s \in V} \Delta^i_s(u)$, we first compute $\Delta^i_s(u)$. Define A_j, B_j and λ_j as follows:

$$A_j \triangleq \frac{op_j}{\alpha d_j(u)} \cdot (1-\alpha) \cdot P_s(v_j)$$
$$B_j \triangleq \frac{op_j}{\alpha d_j(u)} \cdot \big[\alpha \cdot \mathbf{1}_{u=s} - P_s(u)\big]$$
$$\lambda_j \triangleq 1 - \frac{op_j}{d_j(u)}$$

By Algorithm 1, we have the recursive formula for $r_j(u)$:

$$r_j(u) = \lambda_j \cdot r_{j-1}(u) + A_j + B_j$$

Based on the above equation, we can start from $r_k(u)$ and recursively substitute $r_j(u)$ with $r_{j-1}(u)$, and derive the relationship between $r_k(u)$ and $r_0(u)$.

$$r_k(u) = \big(\prod_{j=1}^{k} \lambda_j\big) r_0(u) + \sum_{j=1}^{k} \big(\prod_{x=j+1}^{k} \lambda_x\big)(A_j + B_j) \qquad (3)$$

Note that λ_j is the ratio of u's out-degree before and after j^{th} edge update, i.e. $\frac{d_{j-1}(u)}{d_j(u)}$, so we have the following proposition for λ_j.

$$\prod_{j=a+1}^{b} \lambda_j = \prod_{j=a+1}^{b} \frac{d_{j-1}(u)}{d_j(u)} = \frac{d_a(u)}{d_b(u)}$$

Replace the multiplication sequence of λ_j in Equation 3 with the above proposition, which will cancel $d_j(u)$ in A_j and B_j. Using the fact that $\sum_{j=1}^{k} op_j = d_k(u) - d_0(u)$, we can further reorganize Equation 3 as:

$$\Delta^i_s = r_k(u) - r_0(u) = \frac{\sum_{j=1}^{k} op_j \cdot U(u, v_j)}{\alpha d_k(u)}$$

where $U(u, v_j) = (1-\alpha) \cdot P_s(v_j) - P_s(u) - \alpha \cdot r_0(u) + \alpha \cdot \mathbf{1}_{u=s}$.

To calculate the upper bound of $\Delta^i_s(u)$, take out the absolute value of each term in $op_j \cdot U(u, v_j)$ and have the following expression. We use the fact that $P_s(v_j) \le \pi_s(v_j) + \epsilon$ and $r_0(u) \le \epsilon$.

$$(1-\alpha) \cdot P_s(v_j) + P_s(u) + \alpha \cdot r_0(u) + \alpha \cdot \mathbf{1}_{u=s}$$
$$\le (1-\alpha) \cdot (\pi_s(v_j) + \epsilon) + (\pi_s(u) + \epsilon) + \alpha\epsilon + \alpha \cdot \mathbf{1}_{u=s}$$

Now the numerator of $\Delta^i_s(u)$ is at most k times the above expression. Notice that $\sum_{s \in V} \pi_s(v_j) = 1$ (similarly for u), if we take the summation of $\Delta^i_s(u)$ over $s \in V$, $\pi_s(v_j)$ and $\pi_s(u)$ will be canceled. In this way, we can complete the proof for the lemma. □

THEOREM 3. *The number of operations required by the parallel local update to maintain a valid approximate solution is asymptotically the same as that by the sequential local update, under the two edge arrival models (see Theorem 1).*

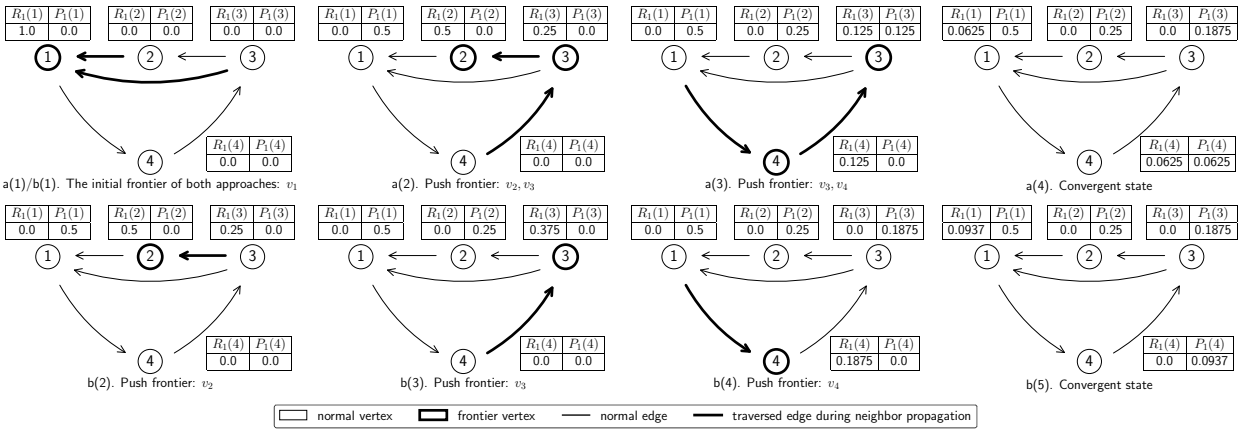

Figure 3: The comparison of the parallel push (Algorithm 3) a(1)-a(4) and the sequential push (Algorithm 2) b(1)-b(5). Assume $\alpha = 0.5$ and $\epsilon = 0.1$. Starting from the same initial state, the parallel push pushes $\{v_1, v_2, v_3, v_3, v_4\}$, while the sequential push pushes $\{v_1, v_2, v_3, v_4\}$. The parallel push performs one more push operation on v_3.

PROOF. Suppose at time step i there are $k^i(u)$ directed edge updates that start from u. By Lemma 3, we have the following inequality for the third term in Ψ_d and Ψ_u (Lemma 2).

$$\sum_{i=1}^{t}\sum_{u\in V}\frac{d^i_{out}(u)\cdot\sum_{s\in V}\Delta^i_s(u)}{\alpha\epsilon}$$
$$\leq \sum_{i=1}^{t}\sum_{u\in V}\frac{d^i_{out}(u)}{\alpha\epsilon}\cdot\frac{2n\epsilon+2}{\alpha d^i_{out}(u)}\cdot k^i(u)$$
$$\leq \sum_{i=1}^{t}\sum_{u\in V}k^i(u)\cdot\frac{2n\epsilon+2}{\alpha^2\epsilon}$$

In the following, we consider different edge arrival models.

In a directed graph with random edge permutation, assuming the total number of edge updates over t time steps is K, we have $\sum_{i=1}^{t}\sum_{u\in V}k^i(u) = K$. Thus, we can obtain the upper bound of Ψ_d as:

$$\Psi_d \leq \frac{\overline{d}}{\alpha\epsilon} + K\cdot\frac{\alpha+4}{n\alpha^2} + K\cdot(\frac{2}{\alpha^2}+\frac{2}{\alpha^2 n\epsilon}) \quad (4)$$

In an undirected graph with arbitrary edge updates, we need to apply RESTOREINVARIANT twice as an undirected edge update is treated as two directed updates, and thus the total number of directed edge updates is $2K$, which means $\sum_{i=1}^{t}\sum_{u\in V}k^i(u) = 2K$. Thus, we obtain the upper bound of Ψ_u.

$$\Psi_u \leq \frac{\overline{d}}{\alpha\epsilon} + K\cdot\frac{2}{\alpha} + K\cdot(\frac{4}{\alpha^2}+\frac{4}{\alpha^2 n\epsilon}) \quad (5)$$

The asymptotic bound of Ψ_d and Ψ_u is $O(K+K/(n\epsilon)+\overline{d}/\epsilon)$, which is the same as the sequential local update specified in Theorem 1. Thus we conclude the proof. □

4. OPTIMIZATION

As repairing the invariant only takes a constant time, the parallel push dominates the running performance of the parallel local update approach. In this section, we describe our novel optimization techniques, i.e., *eager propagation* and *local duplicate detection*, to improve the performance of the parallel push.

4.1 Eager Propagation

Parallel execution accelerates the local push, but the improvement does not come for free. Compared with the sequential push, we find the parallel push can take more operations in many cases. This is caused by a phenomenon we call ***parallel loss***. To solve this problem, we propose a technique called ***eager propagation*** to bridge the gap between the parallel and sequential push.

4.1.1 Parallel Loss

Consider an example shown in Figure 3. To achieve convergence (i.e., all vertices have smaller residual than ϵ), both the parallel and sequential push perform push operations on the same set of vertices $\{v_1, v_2, v_3, v_4\}$, but the parallel push costs one more operation because v_3 is pushed twice. For the first time to push v_3 in a(2) of Figure 3, the frontier vertices are v_2 and v_3. Since v_3 is the in-neighbor of v_2, v_2 propagates some residual to v_3. This amount is large enough to activate v_3 in the next iteration. Thus in a(3), v_3 needs to be pushed once more. The sequential push, however, does not need such effort. In b(2), it pushes v_2 and v_2 propagates its residual to v_3. In b(3), v_3 has a large amount of residual, 0.375, consisting of the amount it has in b(2) 0.25 and the propagation from v_2 0.125. The sequential push can resolve this residual with only one push operation. In b(2), if pushing both v_2 and v_3 in parallel, we will lose the opportunity to process a larger residual. In the end, it may take more operations to converge. This phenomenon, caused by concurrent push operations on multiple frontier vertices in one iteration, is called *parallel loss*.

Parallel loss cannot be avoided as long as we parallelize the local push. Although we prove the worst-case complexity of the parallel approach matches that of the sequential approach (Section 3), the former can take more operations at runtime. We introduce Lemma 4 to support this argument. It shows that when ϵ is sufficiently small, the overall residual of the parallel push is always larger than that of the sequential push because of parallel loss. With larger overall residual, the parallel push takes more operations than the sequential push under the same convergence condition.

LEMMA 4. *Let R_x^p and R_x^q be the residuals for the parallel and sequential push in iteration x for any fixed source vertex. Given any iteration x and a corresponding frontier, the sequential push handles all vertices in its frontier one by one in serial, whereas the parallel push handles the frontier vertices all at once. Given the same initial residual distribution, i.e. $R_0^p = R_0^q$, if $\epsilon \to 0$, the sum of absolute residual values for the parallel push is always larger than that of the sequential push. That is $\|R_s^p(x)\|_1 \geq \|R_s^q(x)\|_1$, for any iteration x.*

PROOF. First of all, when $\epsilon \to 0$, the frontiers of every iteration are the same for the parallel and sequential push since the initial residual distribution is the same. This can be proved by induction. The base case is trivial. Suppose in iteration x, the frontiers for both local pushes are the same, then any in-neighbor v of the frontier vertex u will become frontier in iteration $x+1$ as the propagated residual from u always activates v, i.e., $(1-\alpha)\cdot R_s(x,u)/d_{out}(v) \gg \epsilon$. Thus the frontiers for both local pushes at iteration $x+1$ are the same. To facilitate our proof, we denote FQ^x as the common frontier for both local pushes at iteration x.

Next, we show that if the sum of absolute residuals of the parallel push are larger than or equal to that of the sequential push in iteration x, i.e., $\|R_x^p\|_1 \geq \|R_x^q\|_1$, then $\|R_{x+1}^p\|_1 \geq \|R_{x+1}^q\|_1$. Suppose there are k frontier vertices in iteration x, $v_i \in FQ^x, 1 \leq i \leq k$, we denote $b(v_1), b(v_2), \ldots, b(v_k)$ as the residual values *pushed* at iteration x for the sequential push. Assuming the order of pushing the frontier vertices by the sequential push is v_1,v_2,..., we have

$$b(v_i) = R_x^q(v_i) + \sum_{j=1}^{i-1}(1-\alpha)\cdot b(v_j) \,/\, d_{out}(v_i)\cdot \mathbf{1}_{v_i \in N_{in}(v_j)}$$

Since the residuals of frontier vertices are either all positive or all negative, we have $|b(v_i)| \geq |R_x^q(v_i)|$ from the above equation. By Equation 2, the change of the residual sum of the sequential push $\|R_{x+1}^q\|_1 - \|R_x^q\|_1$ is

$$(1-\alpha)\sum_{i=1}^{k} b(v_i)\cdot \sum_{x\in N_{in}(v_i)} \frac{1}{d_{out}(x)} - \sum_{i=1}^{k} b(v_i)$$

$$= -\alpha\sum_{i=1}^{k} b(v_i) + \sum_{i=1}^{k} b(v_i)\cdot(1-\alpha)\cdot(\sum_{x\in N_{in}(v_i)} \frac{1}{d_{out}(x)} - 1)$$

For the parallel push, $\|R_{x+1}^p\|_1 - \|R_x^p\|_1$ can lead to the same result as the above equation except that $b(v_i)$ is replaced with $R_x^p(v_i)$. Let $c(v_i)$ be $(1-\alpha)\cdot(\sum_{x\in N_{in}(v_i)} \frac{1}{d_{out}(x)} - 1)$, we have the following inequality.

$$\|R_{x+1}^q\|_1 = \|R_x^q\|_1 - \alpha\sum_{i=1}^{k} b(v_i) + \sum_{i=1}^{k} b(v_i)\cdot c(v_i)$$

$$\leq \|R_x^q\|_1 - \alpha\sum_{i=1}^{k} |R_x^q(v_i)| + \sum_{i=1}^{k} b(v_i)\cdot c(v_i)$$

$$\leq (1-\alpha)\|R_x^q\|_1 + \alpha S_q + \sum_{i=1}^{k} b(v_i)\cdot c(v_i)$$

$$\leq (1-\alpha)\|R_x^p\|_1 + \alpha S_q + \sum_{i=1}^{k} b(v_i)\cdot c(v_i)$$

$$= (\|R_x^p\|_1 - \alpha\sum_{i=1}^{k} |R_x^p(v_i)|) + \alpha(S_q - S_p) + \sum_{i=1}^{k} b(v_i)\cdot c(v_i)$$

$$= (\|R_{x+1}^p\|_1 - (1-\alpha)\cdot\sum_{i=1}^{k} |R_x^p(v_i)|\cdot c(v_i)) + O(\epsilon) + \sum_{i=1}^{k}$$

$$b(v_i)\cdot c(v_i) = \|R_{x+1}^p\|_1 + O(\epsilon) + \sum_{i=1}^{k}(b(v_i) - |R_x^p(v_i)|)\cdot c(v_i)$$

where $S_q = \sum_{u\in V\setminus FQ^x} |R_x^q(u)|$ and $S_p = \sum_{u\in V\setminus FQ^x} |R_x^p(u)|$. When $\epsilon \to 0$, $\|R_{x+1}^q\|_1 \leq \|R_{x+1}^p\|_1 + \sum_{i=1}^{k}(b(v_i) - |R_x^p(v_i)|)\cdot c(v_i)$. By the friendship paradox [15], the mean number of friends of an individual is bounded by that of his friends, i.e. $\mathbb{E}\big[d_{out}(v_i)\big]/\mathbb{E}\big[d_{out}(x)|x \in N_{in}(v_i)\big] \leq 1$, so $\mathbb{E}\big[\sum_{x\in d_{out}(v_i)} \frac{1}{d_{out}(x)}\big] \leq 1$. This leads to $c(v_i) \leq 0$. Moreover, $b(v_i) \geq |R_x^q(v_i)| \geq |\,|R_x^p(v_i)|$, so $\sum_{i=1}^{k}(b(v_i) - |R_x^p(v_i)|)\cdot c(v_i) \leq 0$. With this condition, $\|R_{x+1}^q\|_1 \leq \|R_{x+1}^p\|_1$. Thus, we prove the lemma by induction as the base case always holds. □

4.1.2 Mitigate Loss

The major issue of parallel loss is due to the fact that the parallel push reads stale residual values and thus results in pushing smaller amount of residual. More specifically, as a vertex v is pushing its residual to its incoming neighbors, an out-going neighbor of v is pushing some amount of residual to v and v is not aware of the additional residual pushed to itself. Motivated by this issue, we propose *eager propagation* that reads the recent residuals of frontier vertices to mitigate parallel loss.

In the parallel push, Algorithm 3 reads the residuals of frontier vertices before the neighbor-propagation, without being aware of the potential increase of the residuals. Instead, eager propagation is anxious for more residuals to push and proactively reads the recent residuals of frontier vertices, which may get increased by the neighbors. To implement eager propagation, we redesign the parallel push: (1) We alter the order of the two parallel sessions, and (2) ensure the residuals of frontier vertices that are used in both sessions to be consistent.

Algorithm 4 shows the optimized parallel push that implements eager propagation. Compared with Algorithm 3, there are several major differences due to eager propagation. First, the local push executes the two parallel sessions in the reverse order, i.e. neighbor-propagation first and self-update afterwards. Second, the accesses to the residual values of frontier vertices are different. Note that at Line 10 we read the up-to-date residual of u as r_u, which is the key step for eager propagation. Since this read is in the loop of neighbor-propagation, $R_s(u)$ can be increased by concurrent propagation from u's neighbors and the value r_u is potentially larger than the value of $R_s(u)$ before the neighbor-propagation session starts. To ensure the consistent residual value used in both sessions, r_u is recorded in E at Line 11, which will be passed down to later session. Then $R_s(u)$ is subtracted with the consistent r_u at Line 21, instead of being zeroed as in Algorithm 3.

There are also some other differences in Algorithm 4 compared with Algorithm 3. First, there is an additional round of frontier generation in the self-update at Lines 22-23. Note that the absolute value of $R_s(u)$ could be larger than ϵ after the subtraction at Line 21, making u as the potential frontier for the next iteration, since $R_s(u)$ can still be increased by u's neighbors after u pushes out its residual. Compared with the other frontier generation process in the neighbor-propagation, this one is to take care of those vertices that are frontier in both the current and next iteration,

i.e. $u \in FQ \wedge u \in FQ'$, while the other is for those vertices that are frontier in the next iteration but not in the current iteration, i.e. $u \in FQ' \wedge u \notin FQ$. Second, the frontier generation in the neighbor-propagation of Algorithm 4 is also different from that in Algorithm 3. Both of these differences are related to our novel optimization technique for the frontier generation, which we will discuss in the next subsection.

4.2 Frontier Generation

Algorithm 4 OPTPARALLELPUSH

```
1: procedure PUSHCONDLOCAL(r_pre, r_cur, phase)
2:     if !PUSHCOND(r_pre, phase) then
3:         if PUSHCOND(r_cur, phase) then
4:             return true
5:     return false
6: procedure OPTPARALLELPUSH(P_s, R_s, FQ, phase)
7:     FQ' = ∅
8:     E = ∅
9:     parallel for u ∈ FQ
10:        r_u ← R_s(u)
11:        E = E ∪ (u, r_u)
12:        parallel for v ∈ N_in(u)
13:            inc = (1 − α) · r_u / d_out(v)
14:            r_pre = ATOMICADD(R_s(v), inc)
15:            r_cur = r_pre + inc
16:            if PUSHCONDLOCAL(r_pre, r_cur, phase) then
17:                ENQUEUE(FQ', v)
18:    synchronize
19:    parallel for (u, r_u) ∈ E
20:        P_s(u) += α · r_u
21:        R_s(u) −= r_u
22:        if PUSHCOND(R_s(u), phase) then
23:            ENQUEUE(FQ', u)
24:    synchronize
25:    return FQ'
```

In general, there are two methods for frontier generation [21, 34, 35, 25, 26, 13], and both of them cannot give satisfactory performance for the parallel push. The first one is a topology-driven method, which inspects all vertices at the start of each iteration to construct the frontier. This method is not work-efficient especially when the frontier size is small. The second one is a data-driven method that takes in a frontier queue and generates a frontier queue for the next iteration. However, to avoid duplicate items during frontier generation, such a method requires atomic operations, which incurs expensive overheads. The unoptimized parallel push, i.e., Algorithm 3, adopts the second method. The duplicate detection in UNIQUEENQUEUE of Algorithm 3 can cause significant synchronization overheads.

In fact, this overhead can be avoided by exploring application properties: ***monotonicity***. Note that in the work flow of the parallel push, the positive residuals are pushed out in the first phase and then the negative residuals in the second phase. Thus, during neighbor-propagation, the residuals are changing monotonically, increasing in the first phase and decreasing in the second phase. Take the first phase for example, for a vertex v, its residual absorbs incoming propagation from v's out-neighbors. There is one and only one out-neighbor, u, that turns v from the state when $R_s(v) < \epsilon$ to the other state when $R_s(v) > \epsilon$. Therefore, we can designate u as the one responsible to put v into new frontier queue, and others can avoid the attempt to enqueue v. But this raises a question: how can u know the value of $R_s(v)$ during frontier generation?

Note that in most parallel architectures like GPUs and multi-core CPUs we can implement an atomic operation that performs the addition to a 32/64 bit address atomically and returns the before-value after finishing this operation. For some architectures, such an atomic addition can be directly supported by the hardware intrinsics. For the remaining architectures where it cannot be directly supported, observe that the compare-and-swap intrinsics are always supported [20, 12], and we can use such intrinsics to implement the functionality of the atomic addition. ATOMICADD at Line 14 is the implementation of such an operation. The before-value r_u is the by-product of updating $R_s(u)$. With the before-value, the after-value can be computed locally. To perform duplicate detection on v, u enqueues v as the frontier if the before-value of $R_s(v)$ does not meet the push condition, but the after-value of $R_s(v)$ does. In this way, threads can identify duplicates locally without synchronizing globally on shared data structures. We call this efficient technique as ***local duplicate detection***.

We implement local duplicate detection at Lines 14-17 of Algorithm 4. At Line 14, the residual of v is atomically updated and the before-value of $R_s(v)$ is returned as r_{pre}. At Line 15, the after-value is calculated as r_{cur}. If a vertex v can pass the test at Line 16, it means that v passes the push condition and putting v to FQ' will not cause duplicates. At Line 17, we enqueue those qualified vertices with ENQUEUE. The difference between ENQUEUE here and UNIQUEENQUEUE is that ENQUEUE removes the duplicate detection process.

Note that Algorithm 4 has one more frontier generation process at Lines 22-23. This is because the first frontier generation process cannot handle the case when a vertex is in the frontier for both the current and the next iteration. Given a frontier vertex u, i.e. $|R_s(u)| > \epsilon$, a neighbor of u updates $R_s(u)$ and gets the before-value of $R_s(u)$ as r_{pre} ($|r_{pre}| > \epsilon$). Under such circumstances, the after-value $r_{cur} = r_{pre} + inc$ also satisfies $|r_{cur}| > \epsilon$ due to monotonicity. Thus, u is never enqueued, even if u can meet the push condition after self-update. To remedy this, we add the second frontier generation to handle the frontier vertices. As the parallel-for loop at Lines 19-23 of Algorithm 4 does not have shared vertices, the frontier can be generated efficiently without duplicate detection.

5. EXPERIMENTS

In this section, we will first introduce the experimental setups and then report the results of the experimental study.

5.1 Experimental Setup

Implementation. We implement the proposed parallel local update approach on both the CPU and GPU architectures. For comparison, we also implement various baselines. The details are shown as follows.

- *CPU Sequential Push with Single Update* (`CPU-Base`): We implement the state-of-the-art sequential algorithm for dynamic PPR computation [49]. For each edge update, it repairs the invariant and then invokes the sequential push.
- *CPU Sequential Push with Batch Update* (`CPU-Seq`): We add the batch update optimization for `CPU-Base`. To handle a batch of size k, it repairs the invariant for k updates and then launches the sequential push.
- *CPU Parallel Push with Batch Update* (`CPU-MT`): We implement our parallel local update using the multi-thread framework CilkPlus [24] to manage parallel contexts and resources on CPUs.

- *GPU Parallel Push with Batch Update* (GPU): We implement our parallel local update on GPUs using CUDA [37].
- *Incremental Monte-Carlo on CPU* (Monte-Carlo): We implement the state-of-the-art incremental Monte-Carlo approach [10]. To improve the performance, we parallelize the maintenance of random walks with the multi-thread framework CilkPlus [24].
- *Vertex-centric implementation in Ligra* (Ligra): We implement our approach, i.e. parallel push with batch update, on top of Ligra [42], the state-of-the-art in-memory graph processing system on multi-core architectures.

Datasets. Our experiments are conducted on the following real-world graph data sets [3].

- **Pokec** has 1.6M vertices and 30.6M edges, generated by the most popular on-line social network in Slovakia.
- **LiveJournal** has 4.8M vertices and 68.9M edges, generated by an online community that allows users to declare friendships with others.
- **Youtube** has 1.1M vertices and 2.9M edges, extracted from the users' friendships of Youtube.
- **Orkut** has 3.0M vertices and 117.1M edges, extracted from users' friendships of an online social network.
- **Twitter** has 41.6M vertices and 1.4B edges, generated by Twitter network of *followed-by* relationships from a sample in 2010.

Graph Stream. For all data sets, they do not possess a timestamp for each edge. Following previous works [10, 38, 49], we simulate the random edge arrival model by randomly setting the timestamps for all edges. Then, the graph stream of each dataset receives the edges with increasing timestamps. To evaluate both insertions and deletions on the arrival of graph streams, the sliding window model is adopted in the experiments. For initialization, the first 10% edges in the stream are used to construct the sliding window before updates start. As the window slides for a batch size of k, k edges are inserted and the same number of edges are deleted according to their timestamps. The experimental results are the average of 10 slides for Twitter and the average of 100 slides for the other data sets.

Parameter setting. We summarize the parameters used in the experiments together with their highlighted default values in Table 2. For a source vertex, No. of random walk samples maintained by Monte-Carlo is $w \geq \frac{3log(2/p_f)}{\epsilon_r^2 \delta}$ [46], where δ, p_f, ϵ_r are the result threshold, failure probability and relative error. Normally, δ, p_f and ϵ_r are set to $1/|V|$, $1/|V|$ and 0.5 respectively [46, 29, 31]. This makes w huge for large graphs like Twitter and leads to poor performance. In our experiment, we favor Monte-Carlo and set w to a smaller value, i.e. $6|V|$ to improve the performance by trading accuracies (with $\delta = 1/|V|$, $p_f = 2/e$, $\epsilon_r = 0.71$).

Experimental Environment. We conduct all the experiments on two machines.

- For CPU-based implementations (i.e. CPU-Base, CPU-Seq, CPU-MT, Monte-Carlo and Ligra), the experiments are performed on a machine running Ubuntu 16.04 with *four* 10-core Intel Xeon E7-4820 processors clocked at 1.9 GHz and 128GB main memory. Each core owns a private 32 KB L1 cache and a private 256 KB L2 cache. Every 10 cores share a 25 MB L3 cache and a 64 GB local DRAM. The programs are compiled with GCC 5.4.0 using $-O3$ flag. For multi-threaded execution, the program is also compiled with CilkPlus from GCC 5.4.0.
- For GPU-based implementation (GPU), we run the experiments on a CentOS server that has 64GB main memory and one Intel(R) Core(TM) i7-3820 4-core processor with the frequency of 3.60 GHz, and GeForce GTX TITAN X GPU with 12GB device memory that is connected to PCIe v2.0 ×16. The programs are compiled with CUDA-7.0 and GCC 4.4.7 with $-O3$ flag.

[3]https://snap.stanford.edu/data/

Table 2: All parameters used in the experiments. The default values are highlighted.

Parameter	Values
α	**0.15**
ϵ	$10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, \mathbf{10^{-9}}, 10^{-10}$
source vertex s	randomly chosen vertices with **Top-10**, Top-1K and Top-1M out-degrees
batch size	**1%**, 0.1% and 0.01% of sliding window size
No. of random walk samples	$\mathbf{6|V|}$ ($|V|$ is No. of vertices)

Table 3: Variants of the parallel push

	Opt	*Eager*	*DupDetect*	*Vanilla*
Eager Propagation	✓	✓	×	×
Local Duplicate Detection	✓	×	✓	×

5.2 Effects of Optimizations

In this subsection, we evaluate the effectiveness of the optimization techniques, including eager propagation and local duplicate detection. For comparison, we implement different variants of the parallel push with these techniques enabled or disabled, as shown in Table 3.

We run these variants of the parallel local update and report the average latency. As shown in Figure 4, the fully optimized algorithm can achieve about 2.5 times speedups compared with the unoptimized version for GPUs and multi-cores. Each of the techniques provides significant performance improvement over the unoptimized version. For eager propagation, the speedup comes from the mitigation of parallel loss. The number of required operations is reduced. For local duplicate detection, it achieves speedups by reducing synchronization efforts at runtime. Notice that both techniques show more significant improvements in larger data sets. This is because large data sets lead to large frontier sizes, which introduce more severe parallel loss and incur higher frontier generation costs.

5.3 Stream Throughtput

In this subsection, we compare the stream throughput between the parallel local update and various baselines. We report the number of edges consumed per second after running for 5 minutes. Meanwhile, we vary the batch size to evaluate its effect over the stream throughput.

In Figure 5, we can see the parallel local update achieves significant speedups over the sequential local update. Compared with CPU-Base, which is the state-of-the-art approach [49], GPU and CPU-MT achieve the speedups of more than 100-7000 and 30-4000 times respectively with the batch size

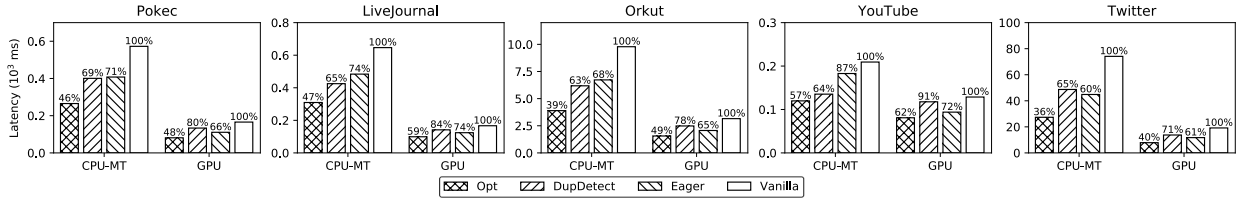

Figure 4: Effect of optimizations for the parallel local update.

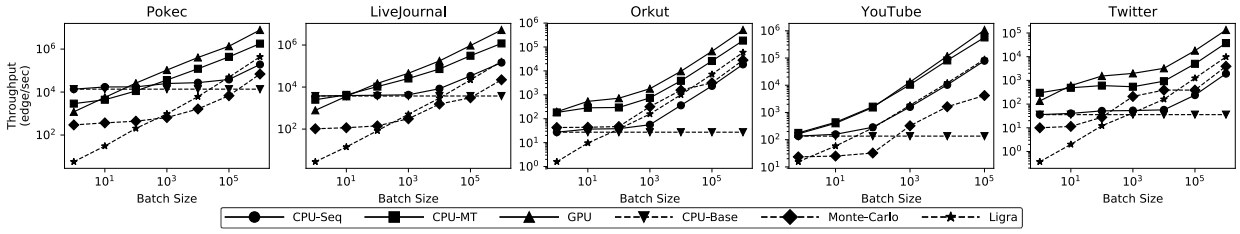

Figure 5: The comparison of streaming throughput.

of 10^5. Compared with `CPU-Seq`, `GPU` and `CPU-MT` achieve the speedups of more than $12-74$ and $6-20$ times respectively. The performance improvement of our approach comes from two dimensions. One is the parallel push and the other is the batch update. First, `GPU` and `CPU-MT` achieve speedups over `CPU-Seq`, which shows that the parallel push effectively accelerates the local push. Second, as the batch size increases, the throughputs of `GPU` and `CPU-MT` get larger, because the batch update raises the degree of parallelism. With a large batch size, there are more workloads in each iteration, so parallel threads can process more frontier vertices all at once. As `CPU-Base` is significantly slower than the other approaches,we will omit the results for `CPU-Base` in the rest of the experiments.

Figure 5 shows our parallel local update has superior performance over the incremental Monte-Carlo approach. Compared with `Monte-Carlo`, `GPU` and `CPU-MT` achieve the speedups of 18 - 254 and 9 - 135 times respectively with the batch size of 10^5. We attribute the unsatisfactory performance of Monte-Carlo to the huge overheads of incremental maintenance of random walk samples. The overheads come from two sources. First, the number of random walk samples w is huge especially for large graphs like Twitter (although we set w to a value smaller than existing works [46, 29, 31]). Second, the incremental maintenance of random walk samples needs to track some auxiliary data structures. The Monte-Carlo approach on static graphs only needs to maintain the number of visits made by random walks for each vertex. However, the incremental approach should also keep track of the traces of the random walk samples, i.e. the vertices visited, and the inverted index that records the set of random walks passing through each vertex. When new edges arrive, we should update the traces of these samples and the inverted index by regenerating random walks on the new graph. This updating process requires frequent atomic accesses to the shared memory. Thus, these auxiliary data structures are large and the maintenance incurs a huge cost.

As shown in Figure 5, our specialized implementations (`GPU` and `CPU-MT`) are better than the implementation in the general graph processing system with vertex-centric abstraction (`Ligra`). This is because those systems lack application knowledge to perform specific optimizations such as eager propagation and local duplicate detection.

5.4 Effects of Parameters

In this subsection, we study how the parameter setups affect the performance of the parallel local update.

Varying ϵ. The choice of ϵ determines the accuracy of the solution and a smaller ϵ requires more computation. We vary ϵ in the range from 10^{-4} to 10^{-10} and show the latency of window slides in Figure 6. As ϵ decreases, the latency for all approaches increases dramatically, because of more computation. The speedups of the parallel approach over the sequential approach become larger for smaller ϵ. This is because smaller ϵ creates larger frontier for the parallel approach to be effective. With the parallel speedups, our approach can achieve fairly low latency with high accuracy.

Varying source vertex selection. The degree of the source vertex can affect the performance of the local push. Starting from a source vertex s with low degrees, the vertices that have high PageRank values w.r.t. s are mostly in the local cluster of s. Hence, if the edge updates do not affect the structure of the local cluster of s, then the local push only needs few operations for the incremental computation. On the contrary, if s has high degree, a small number of edge updates can severely change the current estimate and the residual vector, since s has influence over a wide range of vertices. In Figure 7, we choose source vertices with various degrees (top-10, top-1K and top-1M out-degree) and validate that as the degrees increase, the latencies of all approaches become larger. We also find that the improvement of our parallel approach is not significant when the source vertex has a small degree. The phenomenon can be understood as the parallel workload is reduced for small-degree vertices compared with large-degree vertices. In practice, one is often interested in PPR of large-degree source vertices and our parallel approach is superior under such cases.

Varying batch sizes. To study the effect of different batch sizes, we vary it as different ratios of the sliding window size, i.e., 1%, 0.1% and 0.01%. We show the latency of all approaches in Figure 8. As the batch size of the window slide decreases, the latencies of all approaches get smaller for fewer updates are required. But the parallel approaches `GPU` and `CPU-MT` still keep the speedups over `CPU-Seq` for smaller batches, which shows that our parallelization is effective and robust to different batch sizes.

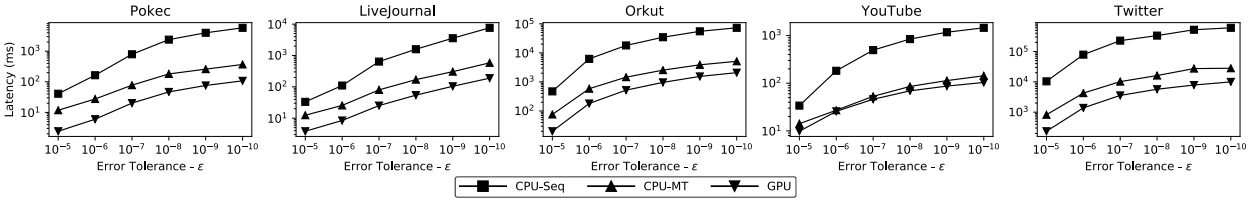

Figure 6: Effect of ϵ for parallel push.

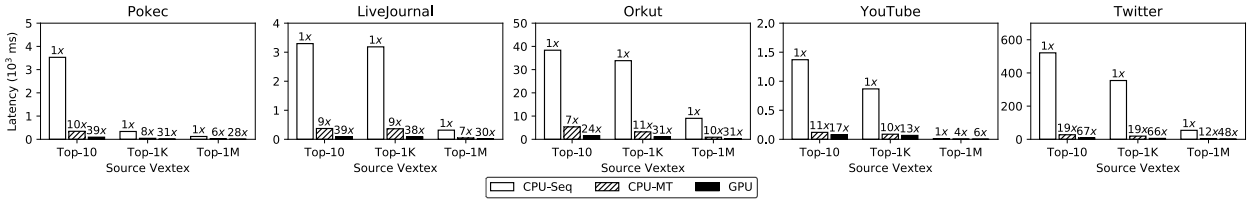

Figure 7: Effect of the source vertex. It is chosen from those vertices having top-10, top-1K and top-1M out-degree.

Table 4: Profiling metrics for resource consumption

Approach	Metrics
GPU	Achieved warp occupancy (WO)
	Global load efficiency (GLD)
CPU-MT	L2 data cache miss rate (L2DCM)
	L3 cache miss rate (L3CM)
	ratio of cycles stalled on resource (STL)

5.5 Resource Consumption

We study the effects of different optimizations and parameters on the resource consumption of the hardwares, including the memory efficiency and thread utilization. To profile the execution of our approaches, we use *nvprof* in CUDA toolkit [37] for GPU and *PAPI* [2] for CPU-MT. Table 4 lists the profiling metrics used in the experiments. Note that for GPU, WO is the ratio of the average warps per active cycle, while GLD is the ratio of the requested global memory load throughput to the maximum load throughput.

Figure 9 shows the profiling results with varying batch size. For GPU, we can observe that the achieved warp occupancy gets larger as the batch size increases. This is because a large batch size provides more workloads for the parallel push, and more warps can be utilized for the execution of GPU kernels. Meanwhile, the global store efficiency decreases with the increase of the batch size. As traversing the graph structure leads to many random memory accesses, when there are more workloads, the parallel push performs more random memory accesses, which degenerate the global memory throughput a bit. For the same reason, L2 data cache miss rate and L3 cache miss rate of CPU-MT increase a bit as the batch size gets large. For CPU-MT, we can also see more CPU cycles are spent waiting for resource as the batch size increases, since there are more memory accesses. We also profile the execution with varying optimization techniques, the error tolerance and the source vertex, the results of which are presented in our extended version [1].

5.6 Scalability

We also evaluate the scalability of our parallel approach on multi-core architectures. We vary the number of cores used for parallelization and report the throughputs for CPU-MT. The experiment is run for 5 minutes with a batch size of 10^5. As shown in Figure 10, we can see that the performance scales as the core number increases.

6. RELATED WORKS

Personalized PageRank. PPR is extensively explored in the literature [49, 38, 10, 31, 6, 5, 9, 29, 11]. They can be categorized into three lines of schemes.

The first scheme is the power iteration [39, 48, 32]. It is slow [38] since it requires $\Omega(m)$ for updating the PPR vector once, where m is the number of edges in the graph.

The second scheme is based on Monte-Carlo [7, 10, 9, 27]. It simulates w random walk samples from each vertex, and estimates the PPR value as the number of visits to a vertex made by these samples. On the arrival of any graph update, say $u \rightarrow v$, the samples that pass through u are found, and then redo the random walks on the new graph. The analysis in [10] shows that the expected number of random walk samples to be updated is only $O(w \cdot \log(k))$ assuming k edges randomly arrive for an arbitrary directed graph [10]. However, w is required to be at least $\frac{3log(2/p_f)}{\epsilon_r^2 \delta}$ [46], where δ, p_f, ϵ_r are the result threshold, failure probability and relative error ratio respectively such that when $\pi_s(x) > \delta$, $|P_s(x) - \pi_s(x)| \leq \epsilon_r \pi_s(x)$ with probability at least $1 - p_f$. Since δ and p_f are set to $1/|V|$ normally [46, 31, 29], w is $O(\frac{nlog(n)}{\epsilon_r^2})$, which implies the overheads for storage and computation can be significant especially for large graphs.

The third scheme is the local update [6, 5, 11, 38, 49, 31, 29, 28, 30], which is the one we parallelize in this work. Many previous works [6, 5, 11] propose variants of the local update scheme for different applications such as graph partitioning and web search, but the PPR is computed in static graphs. Recent works [38, 49] propose incremental approaches for dynamic PPR computation, which can outperform the Monte-Carlo-based approaches. Amortized analysis in [49] shows that dynamic maintenance of PPR vector is efficient. In a directed graph with random edge permutation or undirected graph with arbitrary edge updates, the cost to maintain a reverse PPR vector for a random target vertex is only $O(k + \overline{d}/\epsilon)$, where k is the number of edge updates and $\overline{d}$ is the average degree; given an undirected graph with arbitrary edge updates, the cost to maintain a forward PPR vector for a random start vertex is $O(k + 1/\epsilon)$. These complexities suggest that the maintenance cost is only $O(1)$ for each edge update plus the cost of computing the PPR vector once for the static graph. The variants of the local update scheme slightly differ in the local push, such as convergence

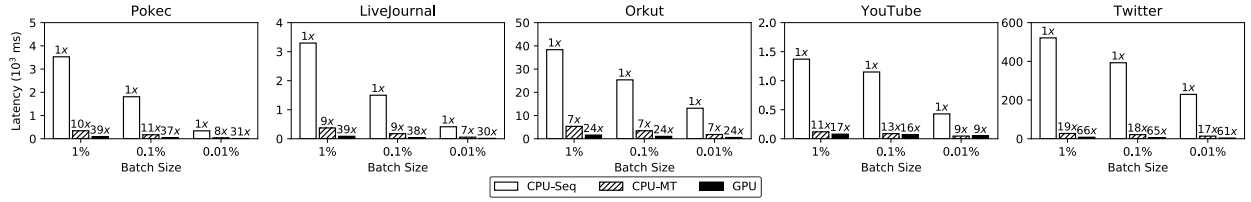

Figure 8: Effect of batch size

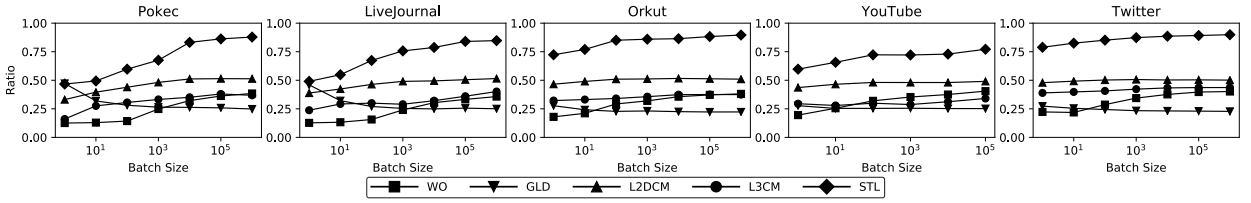

Figure 9: Resource consumption with varying batch size.

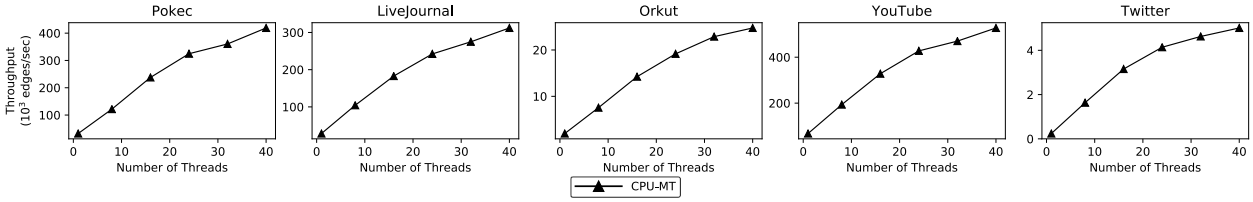

Figure 10: Scalability on Multi-cores.

criteria[6, 49], the direction of graph traversal [6, 38, 49] and the amount of residuals used for the local push [6, 5]. The variant of the local update we parallelize in this work is from [5, 49], but our parallel approach can be naturally extended to other variants of the local update, since they share many similarities. On the arrival of graph updates, they repair the estimates and residuals of the affected vertices to restore the invariant; the local push iteratively performs push operations to maintain the estimate and residual vector until the convergence condition is satisfied.

Recent works [27, 18, 46] rely on effective indexing methods to accelerate the PPR computation. PowerWalk [27] simulates the random walk samples based on the memory budget during offline index construction. For online queries, it reuses these samples and further invokes more iterative computation to meet the accuracy requirement. However, this Monte-Carlo based approach can incur large overheads to maintain the stored random walk samples on dynamic graphs. Guo et al. [18] designs a distributed local update approach that makes use of pre-computed PPR vectors of the selected hub vertices. Wang et al. [46] develops an elastic indexing approach that tradeoffs accuracy, query time and memory budget. It also exploits the pre-computed PPR vectors of the hub vertices to aid for the local update. Our approach is helpful for both these two works [18, 46] to maintain the indexed PPR vectors on dynamic graphs.

To the best of our knowledge, there are only two works on parallelizing the local update for PPR computation. Shun et al. [44] implements various local update algorithms for graph clustering on multi-core architectures; Perozzi et al. [40] studies PageRank-Nibble algorithm [6] in the distributed setting. However, these two works only discuss the computation on static graphs.

Parallel Graph Processing. There are many graph processing systems [36, 23, 47, 42, 16] proposed to accelerate graph applications by utilizing GPUs and multi-core CPUs. They provide the general vertex-centric abstraction that can implement most graph algorithms, but lack the abilities to support application-specific optimizations. For example, eager propagation requires active vertices to absorb incoming messages so as to use the up-to-date residuals, which cannot be supported in bulk synchronous processing model; local duplicate detection exploits the monotonicity to reduce synchronization overheads, but the graph systems cannot leverage this knowledge for the frontier generation.

While most existing graph systems only support static graphs, the recent frameworks [14, 41] can efficiently handle the dynamic scenario by the architecture-optimized graph storage schemes. How to integrate our work with these frameworks could be an interesting future work.

7. CONCLUSION

We propose the parallel local update approach for dynamic PPR computation over high-speed graph streams. Our approach adopts a batch processing method for incremental updates that can reduce synchronization costs within a batch as well as to generate more workload for parallelism. We propose optimization techniques to improve the runtime performance of the parallel push: eager propagation can practically reduce the number of push operations and local duplicate detection avoids synchronization overheads to merge duplicate vertices during frontier generation. The experimental evaluation shows that our parallel approach on GPUs architectures and multicore CPUs achieves significant speedups over the state-of-the-art sequential approach.

Acknowledgments. The project is supported by a grant, MOE2017-T2-1-141, from the Singapore Ministry of Education.

8. REFERENCES

[1] Extended version. http://www.comp.nus.edu.sg/~wentian/paper/tr2017-ppr.pdf.

[2] PAPI. http://icl.utk.edu/papi/.

[3] Twitter usage statistics. http://www.internetlivestats.com/twitter-statistics/. accessed 18-April-2017.

[4] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[5] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S. Mirrokni, and S.-H. Teng. Local computation of pagerank contributions. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 150–165. Springer, 2007.

[6] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE, 2006.

[7] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 45(2):890–904, 2007.

[8] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644. ACM, 2011.

[9] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 973–984. ACM, 2011.

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

[11] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

[12] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[13] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359. IEEE, 2014.

[14] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[15] S. L. Feld. Why your friends have more friends than you do. *American Journal of Sociology*, 96(6):1464–1477, 1991.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[17] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. *PVLDB*, 5(12):2030–2031, 2012.

[18] T. Guo, X. Cao, G. Cong, J. Lu, and X. Lin. Distributed algorithms on exact personalized pagerank. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 479–494. ACM, 2017.

[19] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. ACM, 2013.

[20] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[21] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[22] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, pages 271–279. ACM, 2003.

[23] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

[24] C. E. Leiserson. The cilk++ concurrency platform. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 522–527. IEEE, 2009.

[25] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2015.

[26] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416. ACM, 2016.

[27] Q. Liu, Z. Li, J. Lui, and J. Cheng. Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 195–204. ACM, 2016.

[28] P. Lofgren. Efficient algorithms for personalized pagerank. *arXiv preprint arXiv:1512.04633*, 2015.

[29] P. Lofgren, S. Banerjee, and A. Goel. Personalized pagerank estimation and search: A bidirectional approach. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 163–172. ACM, 2016.

[30] P. Lofgren and A. Goel. Personalized pagerank to a target node. *arXiv preprint arXiv:1304.4658*, 2013.

[31] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Fast-ppr: scaling personalized pagerank estimation for large graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge*

discovery and data mining, pages 1436–1445. ACM, 2014.

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

[33] A. McLaughlin and D. A. Bader. Scalable and high performance betweenness centrality on the gpu. In *Proceedings of the International Conference for High performance computing, networking, storage and analysis*, pages 572–583. IEEE Press, 2014.

[34] D. Merrill, M. Garland, and A. Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing*, 1(2):14, 2015.

[35] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474. IEEE, 2013.

[36] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[37] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[38] N. Ohsaka, T. Maehara, and K.-i. Kawarabayashi. Efficient pagerank tracking in evolving networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 875–884. ACM, 2015.

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

[40] B. Perozzi, C. McCubbin, and J. T. Halbert. Scalable graph clustering with parallel approximate pagerank. *Social Network Analysis and Mining*, 4(1):1–11, 2014.

[41] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *PVLDB*, 11(1), 2017.

[42] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[43] J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 143–153. ACM, 2014.

[44] J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney. Parallel local graph clustering. *PVLDB*, 9(12):1041–1052, 2016.

[45] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *PVLDB*, 8(4):449–460, 2014.

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

[47] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.

[48] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[49] H. Zhang, P. Lofgren, and A. Goel. Approximate personalized pagerank on dynamic graphs. *arXiv preprint arXiv:1603.07796*, 2016.