Updating PageRank for Streaming Graphs

Jason Riedy
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
jason.riedy@cc.gatech.edu

***Abstract*—Incremental graph algorithms can respond quickly to small changes in massive graphs by updating rather than recomputing analysis metrics. Here we use the linear system formulation of PageRank and ideas from iterative refinement to compute the update to a PageRank vector accurately and quickly. The core idea is to express the residual of the original solution with respect to the updated matrix representing the graph. The update to the residual is sparse. Solving for the solution update with a straight-forward iterative method spreads the change outward from the change locations but converges before traversing the entire graph. We achieve speed-ups of 2× to over 40× relative to a restarted, highly parallel PageRank iteration for small, low-latency batches of edge insertions. These cases traverse 2× to nearly 10 000× fewer edges than the restarted PageRank iteration. This provides an interesting test case for the ongoing GraphBLAS effort: Can the APIs support our incremental algorithms cleanly and efficiently?**

***Index Terms*—graph analysis; PageRank; streaming graphs**

I. Introduction

PageRank[1] is a common method for ranking graph vertices using only graph structural information. High PageRank implies that random walks through the graph tend to visit the highly ranked vertices. PageRank also is an archetypal linear algebra-based graph algorithm. Methods for efficiently computing PageRank inform methods for other, related analysis. Here we consider efficiently *updating* PageRank when the graph is altered as in analyzing streaming data. The incremental algorithm does *not* traverse the entire graph, unlike restarting the PageRank iteration.

There are many applications for PageRank beyond web seaches [2]. One of interest is using sparse initial probabilities to provide a *personalized* PageRank useful for local community detection [3]. Computing the PageRank update directly may assist with tracking conductance-minimizing communities analogous to work on tracking modularity-based communities [4].

Section II derives the updating algorithms by applying iterative refinement to the linear system formulation of PageRank in Equation (1). Section III details the algorithms and provides parallel implementation details. We present timing, edge traversal count, and accuracy results in Section IV that range from extreme speed-ups of over 40× on small changes in large graphs to only moderate slow-downs when updating large portions of tiny, cache-resident graphs when compared to restarting a PageRank iteration. There are a few accuracy and performance oddities that require additional investigation. Section V discusses requirements on a possible GraphBLAS API for efficient implementation. These requirements mostly deal with reducing overhead by combining operations. Minimizing overhead and optimizing little operations is necessary when there is no massive-scale graph traversal dominating running time. And Section VI compares with a sample of existing related work.

II. Updating PageRank via Iterative Refinement

A directed graph with vertex set $\mathcal{V}$ can be represented by a sparse, unsymmetric matrix A with $a_{ij} = 1$ where there is an edge $i \to j$. Here we ignore self edges and let the diagonal of A be zero. Define D to be the diagonal matrix of out-degrees, or $\operatorname{diag} D = A\mathbf{1}$ where $\mathbf{1}$ is a $|\mathcal{V}|$-long vector with unit entries. If a vertex i is the source of no edges, let $d_{ii} = 1$ so that $1/d_{ii} = 1$. The definitions and algorithms can generalize to graphs with arbitrary non-negative weights.

While classic PageRank defines an eigenvector problem, a little algebraic manipulation as in [5], [6] finds an equivalent linear system

$$(I - \alpha A^T D^{-1})x = (1 - \alpha)v, \tag{1}$$

where α is the "teleportation" constant, v is a personalization vector, and x is the PageRank vector. For the remainder of the paper, v is a vector with entries $1/|\mathcal{V}|$ denoting a uniformly random start, and $\|v\|_1 = 1$. We discuss the effect of adding and removing vertices in Section II-A. Solving this system with inexact arithmetic produces an approximate solution x. The backward error, or the distance to the nearest system solved exactly, is measured by the residual $r = (1 - \alpha)v - (I - \alpha A^T D^{-1})x$.

We incorporate edge insertions and removals by changing the graph to $A_\Delta = A + \Delta A$, also updating implicitly $D_\Delta = D + \Delta D$. In general, v_Δ denotes $v + \Delta v$. We later will use plain Δ to denote the source vertices in a batch of edge updates. The new residual r' provides a measure of how well the non-updated solution x satisfies the updated linear system with matrix $I - \alpha A_\Delta^T D_\Delta^{-1}$,

$$\begin{aligned} r' &= (1 - \alpha)v - x + \alpha A_\Delta^T D_\Delta^{-1} x \\ &= (1 - \alpha)v - x + \alpha A^T D^{-1} x - \alpha A^T D^{-1} x + \alpha A_\Delta^T D_\Delta^{-1} x \\ &= r + \alpha (A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x. \end{aligned}$$

The operator $(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})$ will occur frequently throughout and has an interesting structure. This operator has non-zero columns only where ΔA affects the graph, or columns associated with vertices in Δ. So the change in residual is *sparse* when ΔA is small relative to the graph. In that case, the updated PageRank $x_\Delta = x + \Delta x$ should be close to the previous x. In many low-latency applications, ΔA will involve only tens or hundreds of updates in graph with at least millions of vertices.

To correct the system and find the update Δx to the new solution, we apply iterative refinement. Iterative refinement is a terrifically efficient application of Newton's method for improving the solution of a linear system[7]. Here we solve

$$(I - A_\Delta^T D_\Delta^{-1})\Delta x = r' = r + \alpha(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x. \quad (2)$$

If solved exactly, this provides the exact new solution for *all* components of x_Δ. Solved with the same precision used for the initial system provides a good quality approximation, again for *all* components.

We want to use sparseness to minimize graph accesses while maintaining solution quality. We compute just enough of the solution to maintain global the convergence condition. Note that restarting an iterative method to solve PageRank always works to improve the quality of the *entire* solution. We are looking at *sparsely targetted* improvements.

While there are many, many techniques for solving Equation (2), here we use the simple iterative Jacobi method. Jacobi is equivalent to the power method often applied to the eigensystem formulation. For PageRank, we take equality satisfied by the true solution, $(I - \alpha A^T D^{-1})x = (1-\alpha)v$, and then split and rearrange terms to produce an iterative method

$$x^{(k+1)} = \alpha A^T D^{-1} x^{(k)} + (1+\alpha)v,$$

where $x^{(k)}$ denotes the solution at the kth iteration. The global convergence of PageRank is determined when the solution changes by less than a fixed threshold, $\|x^{(k+1)} - x^{(k)}\|_1 \leq \tau$. Examples below use $\tau = 10^{-4}$, sufficient accuracy for many cases.

The equation for Δx likewise can be converted to an iterative method,

$$\Delta x^{(k+1)} = \alpha A_\Delta^T D_\Delta^{-1} \Delta x^{(k)} + \alpha(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x + r.$$

As before, $A_\Delta^T D_\Delta^{-1} - A^T D^{-1}$ has non-zero rows only adjacent to the changes, so its contribution is sparse. The old residual r is dense, but refining only the changes requires only those entries in the pattern of the left-hand side. The change to the previous iteration is adding entries of r that appear in the new iterate,

$$\begin{aligned}\Delta x^{(k+1)} = \alpha A_\Delta^T D_\Delta^{-1} \Delta x^{(k)} + \alpha(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x \\ + r_{|\Delta x^{(k+1)}}.\end{aligned} \quad (3)$$

Here we introduce a restriction notation we will abuse vigorously. The notation $r_{|\Delta x^{(k+1)}}$ refers to a *sparse* vector constructed from the entries of r only on the pattern of the sparse vector $\Delta x^{(k+1)}$, or where entries of $\Delta x^{(k+1)}$ have explicit storage. We do not squeeze out entries that happen to be numerically zero; that occurs too rarely to bother. The notation will be used later on explicit sets. $x_{|\mathcal{H}}$ restricts the entries to those vertices in set $\mathcal{H}$.

The best way to *update* r to the new residual r_Δ is expanding one more step out and fully computing the actual new residual entries. However, this is only efficient with undirected graphs with typical row-major storage because it needs access to the *rows* of A_Δ^T or columns of A_Δ. And in small-diameter graphs of interest, each step out walks a much larger fraction of the graph. Instead, we incrementally update the residual. Consider expanding

$$\begin{aligned} r_\Delta &= kv - (I - \alpha A_\Delta^T D_\Delta^{-1})(x + \Delta x) \\ &= kv - (I - \alpha A_\Delta^T D_\Delta^{-1})x - (I - \alpha A_\Delta^T D_\Delta^{-1})\Delta x \\ &= r' - (I - \alpha A_\Delta^T D_\Delta^{-1})\Delta x \\ &= r + \alpha(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x - (I - \alpha A_\Delta^T D_\Delta^{-1})\Delta x. \end{aligned}$$

Then the change $\Delta r = r_\Delta - r$ is

$$\Delta r = \alpha(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x + (I - \alpha A_\Delta^T D_\Delta^{-1})\Delta x. \quad (4)$$

Updating r by explicitly computing Equation (4) will not suffice for high levels of accuracy like bitwise reproducible results, but those needs are rare. Hopefully cases that need extreme accuracy have bursty traffic and can cope with separate levels of accuracy. Here intermediate calculations are carried out in IEEE-754 binary double precision; round-off error is sufficiently small compared to the tolerance τ that it can be ignored.

Additionally, we can limit expansion of the pattern of $\Delta \hat{x}$ by not "pushing" small changes out through the $(A + \Delta A)^T (D + \Delta D)^{-1} \Delta \hat{x}^{(k)}$ product. Let $\mathcal{X}$ be the set of vertices selected to be expanded in a given step and $\mathcal{H}$ be the remaining, held vertices. Then we can treat the columns associated with $\mathcal{H}$ in the operator as the identity and approximate

$$\begin{aligned}\Delta x^{(k+1)} \approx \Delta \tilde{x}^{(k+1)} = \alpha A_\Delta^T D_\Delta^{-1} \Delta \tilde{x}_{\mathcal{X}}^{(k)} \\ + \alpha \Delta \tilde{x}_{\mathcal{H}}^{(k)} + \alpha w + r_{|\Delta \tilde{x}^{(k+1)}}.\end{aligned} \quad (5)$$

Note that the set of held vertices can change at each iteration, which complicates analysis. Simple algebraic iteration and upper bounds on $\Delta x^{(k+1)} - \Delta \tilde{x}^{(k+1)}$ produce unusable results much like interval analysis. In this work, we assume a simple per-vertex threshold controling the trade-off between accuracy and expansion size.

A. Adding and Removing Vertices

The assumptions used for adding and removing vertices are quite application dependent. Often there can be some reasonable bound on the total number of vertices just given available storage. With that assumption, you can assume a pool of vertices with appropriate starting probabilities in v. This assumption is unnecessary in applications like seed set expansion (local community detection) where v is sparse and limited to existing vertices. "Deletion" of vertices has repercussions beyond any simple kernel algorithm. Either all auxiliary information associated with a deleted vertex needs removed throughout a system along sufficient records to ensure the vertex is not reincarnated accidentally, or the vertex lives on although disconnected from the larger graph. The latter would depress an absolute PageRank value but not alter relative values.

To cope with new vertices, we assume an infinite pool of vertices with starting probabilities (entries in v) zero. Vertices are "added" by connecting them to the rest of the graph, but this does not update v. A similar analysis as above can compensate for a Δv term. The Δv would be dense to preserve the right-hand side's one-norm unless v already is sparse. If v is sparse, Δv is easily accommodated. For large graphs, however, the perturbation would be tiny, from $1/|\mathcal{V}|$ to $1/(1+|\mathcal{V}|)$ when adding a single vertex. This may affect very long-running kernels or kernels that tap into a live stream without a good estimate for the number of vertices. If in the long run the primary source of deleting vertices is aging out old information, then the small differences in initial probabilities can balance between the added and deleted vertices. We do not yet have experience to guide decisions, so we defer the larger issue to future work. Here are do not add vertices beyond the initial data set.

III. Algorithms and Implementations

The PageRank and incremental PageRank algorithms are implemented in our STINGER streaming graph analysis framework[8]. STINGER is a multi-process framework that maintains a graph snapshot in memory and ingests a graph update stream in a server that triggers separate analysis kernels sharing data via `mmap` on a RAM file system. Each process is multi-threaded through OpenMP[9] directives, falling back to standard C atomic operations for two operations in Figure 3.

STINGER's graph server gathers input graph changes into a batch, triggers a pre-update function in analysis kernels, updates the stored graph, and then triggers a post-update function in analysis kernels. The non-incremental PageRank routines use only the post-update hook and are expressed in Figure 1. These routines traverse the entire graph in every iteration and update the entire output vector. The incremental PageRank routines in Figure 2 return only the changes. The timing results in Section IV also apply these sparse updates to the dense vector. Computing the explicit change in the PageRank vector, useful for later analysis, is not included in the full algorithms. The operator preserves the one-norm in exact, so we need not re-normalize too often when τ is relatively larger than the computational precision.

The primary operation in the full-graph PageRank algorithms is applying the sparse operator $A^T D^{-1}$ to a dense vector x and accumulating into a dense vector y. STINGER's representation is row-major, so we iterate across the columns of $A^T D^{-1}$ and scatter the vector updates. We are using *unweighted* PageRank here, so each column i (applying a transpose) computes $t = x_i/D_{ii}$, walks the neighbors $j \neq i$ of i, and accumulates $y_j = y_j + t$. Our initial implementation is not highly optimized. We do not worry about the load imbalance from large degree vertices and use simple OpenMP atomic floating-point updates. The current STINGER data structure stores many other data items in each cache line that are unused here, so wasted bandwidth is a more important concern that affects all our implementations equivalently.

```
pr_core $(A, x^{(1)}, v)$
  Let $D =$ diagonal matrix of vertex out-degrees
  For $k = 1$ ... itmax
    $x^{(k+1)} = \alpha A^T D^{-1} x^{(k)} + (1-\alpha)v$
    Stop if $\|x^{(k+1)} - x^{(k)}\|_1 < \tau$
  Return $x^{(k+1)}$

pr $(A, v)$
  Return pr_core$(A, (1-\alpha)v, v)$
pr_restart $(A, x, v)$
  Return pr_core$(A, x, v)$


Implicit parameters:
  $\alpha$ : PageRank teleportation constant
  $\tau$ : Convergence tolerance
  itmax : Iteration maximum
```

Fig. 1. Algorithms for updating the entire PageRank vector either by completely recomputing (`pr`) or starting from the previous PageRank vector (`pr_restart`).

The core of the incremental algorithms is the same operation but applied to a *sparse* vector x to accumulate into a *sparse* output vector y. This operation involves fewer columns, only those associated with the pattern of x, but rather more overhead in tracking the pattern of y. Sparse vectors are stored in a packed format of two arrays, indices and values, "shrink-wrapped" to the length of the pattern. Before updating, the vector values of y are scattered into a $|\mathcal{V}|$-long dense vector, so the numerical updates are scattered similarly as the above. The pattern is copied into a $|\mathcal{V}|$-long scratch vector and each index's location is scattered into another $|\mathcal{V}|$-long dense vector. Each thread scatters the numerical updates from column i (transpose) and packs any new pattern entry into a thread-local buffer. Once the buffer is full or the thread finishes, the pattern is updated as sketched in Figure 3. After updating, y is re-packed into a short vector, re-setting the scratch vectors to the needed initial values only where changed. An alternative would be a clever method of only incrementing a threshold rather than clearing[10], but that complicates our atomic exchanges of known place-holder values.

IV. Results

The experimental results are intended to demonstrate that the incremental updating algorithms achieve acceptable error and traverse fewer graph edges. We also report timings, but specific timing results can be changed drastically by platform-specific optimizations as well as data structure reorganizations that are not our current focus. Throughout the rest, we use $\alpha = 0.85$ and an arbitrary convergence threshold of $\tau = 2^{-17} = 128\varepsilon_s$, were ε_s is the precision of the 32-bit binary IEEE-754 format. Lower convergence thresholds would benefit the restarted PageRank algorithm's running time most but risk delivering quite inaccurate results. We use $\gamma = 16\tau$ as an arbitrary expansion threshold in the restricted expansion algorithm `dpr_held`.

```
dpr_pre $(A, x, \Delta)$
  Let $D =$ diagonal matrix of vertex out-degrees
  $b = A^T D^{-1} x_{|\Delta}$
  Return $b$

dpr_core $(A, b, r, \Delta)$
  Let $D =$ diagonal matrix of vertex out-degrees
  $b = \alpha(A^T D^{-1} x_{|\Delta} - b)$
  $\Delta x^{(1)} = b + r_{|b}$
  For $k = 1$ ... itmax
    $\Delta x^{(k+1)} = \alpha A^T D^{-1} \Delta x^{(k)}_{|\geq\gamma} + \alpha \Delta x^{(k)}_{|<\gamma} + b$
    $\Delta x^{(k+1)} = \Delta x^{(k+1)} + r_{|\Delta x^{(k+1)}}$
    Stop if $\|x^{(k+1)} - x^{(k)}\|_1 < \tau$
  $\Delta r = b + \Delta x^{(k+1)} - \alpha A^T D^{-1} \Delta x^{(k+1)}$
  Return $\Delta x^{(k+1)}$ and $\Delta r$

dpr $(A, b, r, \Delta)$ : Compute only the updates
  $\gamma = 0$
  $\Delta x$ and $\Delta r =$ dpr_core $(A, b, r, \Delta)$
  Return $x + \Delta x$ and $r + \Delta r$
dpr_held $(A, b, r, \Delta)$ : Limit expansion
  $\gamma = \frac{1+\alpha}{1-\alpha}\tau$
  $\Delta x$ and $\Delta r =$ dpr_core $(A, b, r, \Delta)$
  Return $x + \Delta x$ and $r + \Delta r$
dpr_all $(A, b, r)$ : Measure overhead
  $\gamma = 0$
  $\Delta = V$
  $\Delta x$ and $\Delta r =$ dpr_core $(A, b, r, \Delta)$
  Return $x + \Delta x$ and $r + \Delta r$

Additional implicit parameter:
  $\gamma$ : Per-element threshold for limiting expansion
```

Fig. 2. Algorithms for computing PageRank changes directly. Routine dpr does not limit expansion, dpr_held applies a per-entry threshold to limit expansion, and dpr_all computes over the entire vector rather than the changes to measure overhead. Note that they each require the vector returned by dpr_pre *before* altering the graph.

```
Updated: Packed $y$ pattern in ypatt, degree ydeg,
         scattered pattern locations loc
Buffer: vertex indices in idx
$S = 0$
for $k$ indexing the buffer
  $j = \text{idx}_k$
  if atomic_compare_exchange($\text{loc}_j, -1, -2$)
    $S = S + 1$ (*successful*)
  else
    $\text{idx}_k = -1$
if $S > 0$
  $w =$ atomic_fetch_add(ydeg, $S$)
  for $k$ indexing the buffer
    $j = \text{idx}_k$
    if $j \geq 0$
      $\text{loc}_j = w$ *(release)*
      $\text{ypatt}_w = i$
      $w = w + 1$
clear the buffer
```

Fig. 3. Updating the scattered output vector's pattern from the thread pattern buffer. Threads *claim* new pattern locations by atomically swapping the default -1 value with another negative value. The compare and exchance needs only acquire-release ordering with a release on the successful store.

TABLE I
TEST GRAPHS. SIZE DENOTES THE DATA SIZE OF A TYPICAL PACKED UNWEIGHTED CSR FORMAT USING 64-BIT INTEGERS. THE STINGER REPRESENTATION IS LARGER.

Graph	$\|\mathcal{V}\|$	$\|E\|$	Avg. Degree	Size (MiB)
belgium.osm *road map*	1441295	1549970	1.08	22.82
caidaRouterLevel *networking*	192244	609066	3.17	5.38
coPapersCiteseer *citation*	434102	16036720	36.94	124.01
PGPgiantcompo *social network*	10680	24316	2.28	0.23
power *power grid*	4941	6594	1.33	0.07

Our test platform is an Oracle X4470 M2 server using two Intel Xeon E7-4820 processors with eight cores and two threads per core running at a peak speed of 2GHZ. Each processor has 18MiB of L3 cache and 256 MiB of 1066 MHZ DDR3 RAM. Here we bind the PageRank codes and all data to a single NUMA node and run the random edge generation and ingest on the other NUMA node. We also only run at most one thread per core; we are more interested in latency than thread scalability. The codes are compiled with gcc 5.3.1 and run on a system running Linux kernel 4.3.0 but without automatic transparent huge pages. Each experiment is run three times for every number of threads tested (1, 2, 4, 6, 8), and experiments are run in random order. For timings results, we select the median time for each number of threads to avoid occasional system noise.

The test graphs in Table I are selected arbitrarily from the Tenth DIMACS Challenge archive[11] to span a variety of sizes and application areas. The smaller graphs operate entirely within L3 cache, potentially penalizing the incremental algorithms' performance. These graphs begin are essentially undirected with every edge $i \rightarrow j$ having a corresponding reverse edge $j \rightarrow i$. We add directed edges, not reverse edge pairs, by selecting pairs of vertices uniformly at random. All runs use the same random seeds. We add edges in ten batches of 10, 100, and 1000 to examine possible low-latency performance.

Timing results here come with a large caveat: We have not gone through the excruciating work of tuning these operators through all the design points (atomic operations v. sorting and merging, *etc.*) and their parameters. Our implementations are reasonably readable and portable. This algorithm may be an excellent target for the GraphBLAS effort [12]. The times for the incremental algorithms also includes updating the dense PageRank vector. We do not include computing the incremental change from the dense vector in the restarted PageRank times.

That said, Table II shows the best update times achieved

TABLE II
BEST UPDATE TIME IN SECONDS FOR THE INCREMENTAL ALGORITHMS `DPR` AND `DPR_HELD` AS WELL AS RESTARTED PAGERANK `PR_RESTART`. THE TIMING EXPERIMENTS HAVE A FEW NON-REPRODUCIBLE OUTLIERS, SO USE THE MEDIAN TIME FOR EACH NUMBER OF THREADS. THE SPEED-UP OF INCREMENTAL TIME OVER RESTARTED PAGERANK TIME IS PROVIDED.

Graph	Batch	`dpr`		`dpr_held`		`pr_restart`
belgium.osm	10	.0118	42×	.0128	39×	.498
	100	.0127	39×	.0131	38×	.499
	1000	.0461	52×	.0171	140×	2.41
caidaRouterLevel	10	.00710	16×	.00199	57×	.112
	100	.0314	7.1×	.00477	47×	.224
	1000	1.30	.68×	.2290	3.9×	.889
coPapersCiteseer	10	.0729	27×	.0128	155×	1.98
	100	.8650	2.3×	.130	15×	1.98
	1000	2.97	1.3×	1.13	3.5×	3.95
PGPgiantcompo	10	.00211	4.3×	.00023	39×	.00916
	100	.0257	.81×	.00874	2.4×	.0207
	1000	.0372	.67×	.0341	.73×	.0249
power	10	.00304	1.8×	.00008	68×	.00535
	100	.0109	.79×	.00707	1.2×	.00860
	1000	.0126	.67×	.0124	.68×	.00843

TABLE III
THE FRACTION OF GRAPH EDGES TRAVERSED BY EACH ALGORITHM. INCREMENTAL ALGORITHMS ALSO SHOW THE IMPROVEMENT OF EDGES TRAVERSED OVER RESTARTED PAGERANK.

Graph	Batch	`dpr`		`dpr_held`		`pr_restart`
belgium.osm	10	0.00020	*9800×*	0.00007	*30000×*	2
	100	0.00203	*990×*	0.00066	*3000×*	2
	1000	0.120	*84×*	0.00660	*1500×*	10
caidaRouterLevel	10	0.0625	*32×*	0.00252	*790×*	2
	100	0.409	*9.8×*	0.0301	*130×*	4
	1000	15.2	*1.1×*	3.07	*5.2×*	16
coPapersCiteseer	10	0.0563	*36×*	0.00646	*310×*	2
	100	0.689	*2.9×*	0.0952	*21×*	2
	1000	2.32	*1.7×*	0.864	*4.6×*	4
PGPgiantcompo	10	1.58	*5.1×*	0.0453	*180×*	8
	100	21.3	*1.1×*	6.82	*3.6×*	24
	1000	31.2	*1.0×*	28.4	*1.1×*	32
power	10	7.54	*2.4×*	0.0229	*790×*	18
	100	29.2	*1.2×*	18.2	*1.9×*	34
	1000	38.3	*1.0×*	37.1	*1.0×*	39

regardless of the number of threads. The restarted PageRank implementation scales well; all its best times occur with eight threads. The incremental algorithms often achieve best performance with two or four threads. There is less work to perform. Table III shows the number of edges traversed normalized by the number in the updated graph. For the restarted PageRank algorithm `pr_restart` this is equivalent to the number of iterations needed to converge.

Updating a large fraction of the graph finds no real advantage to the incremental algorithms. Updating only a small portion of the graph, however, leads to drastic improvements in both unoptimized time and the number of edges traversed. The limited expansion algorithm `dpr_held` sometimes requires more iterations to converge for some graphs and at larger batch sizes. Further analysis is needed to determine an adaptive restriction threshold γ.

To check any additional error incurred by partial updates over restarting PageRank, we also compute a PageRank vector from scratch at every batch update and consider that a gold standard (`pr` in Figure 1). The from-scratch computation uses the same precision and convergence thresholds, so it is not necessarily a better approximation of the true solution. We then compute the one-norm difference between the solutions achieved by `dpr`, `dpr_held`, and `pr_restart` and the from-scratch PageRank. Figure 4 shows the ratio of these differences comparing each incremental algorithm to the restarted iteration. In general, the incremental algorithms achieve results relatively close to the from-scratch computation. There are a few bizarre exceptions still being investigated, like the massive divergence in belgium.osm for batches of 100 edge insertions and `dpr`. The difference growth for `dpr_held` is not unexpected given the present rudimentary thresholding. The ratios of the residuals' one-norms, the backward errors, behaves similarly, including the occasional odd behavior.

V. GUIDANCE FOR THE GRAPHBLAS

These incremental algorithms for a linear algebra problem on streaming graph data can be expressed using the kinds of operations to be supported in the GraphBLAS. However, the incremental algorithms rely heavily on sparse vectors. Their implementation can carry quite a bit of performance overhead, and that overhead can increase the latency of an update. Much of the GraphBLAS effort is focused on easily expressing some graph algorithms with the idea that the complete graph traversals will be the primary performance bottleneck. We are interested in small updates that do not have the time to traverse much of the graph and so have different requirements for reasonable performance.

Many API proposals would seem to separate the sparse vector operations in Equation (3). Producing another scaled sparse vector $D^{-1}\Delta x$ and then applying A^T could be a significant performance hit from copying the pattern and location look-up arrays. Separating Δx into different vectors $\Delta\tilde{x}_{\mathcal{H}}$ and $\Delta\tilde{x}_{\mathcal{X}}$ from Δx would almost certainly incur too much overhead compared to a threshold test within the matrix-vector product. Also, most APIs that expose sparse vectors as an opaque type lose quite a few simple, common optimizations. Our `dpr` and `dpr_held` implementations only extend the pattern of Δx between iterations. This does not require copying the pattern and location arrays but merely modifying them. Subtracting $\Delta x^{(k+1)} - \Delta x^{(k)}$ to test against τ needs subtract only over pattern of $\Delta x^{(k)}$, and that is the prefix of the pattern of $\Delta x^{(k+1)}$ when stored un-sorted. This can be a significant optimization in the termination test run every loop. Even worse, some APIs try to hide any difference between sparse and dense vectors, adding even more overhead in the common streaming cases.

To summarize, the algorithms `dpr` and `dpr_held` here need the following features for fast, low-latency operation:

- efficient sparse vector operations that can share the pattern and location arrays across multiple vectors / iterations,
- fused scaling and dynamic masking operations within the matrix-vector product loop, and
- matrix-vector products that support unweighted (implicitly unit weight) graphs and floating-point vectors.

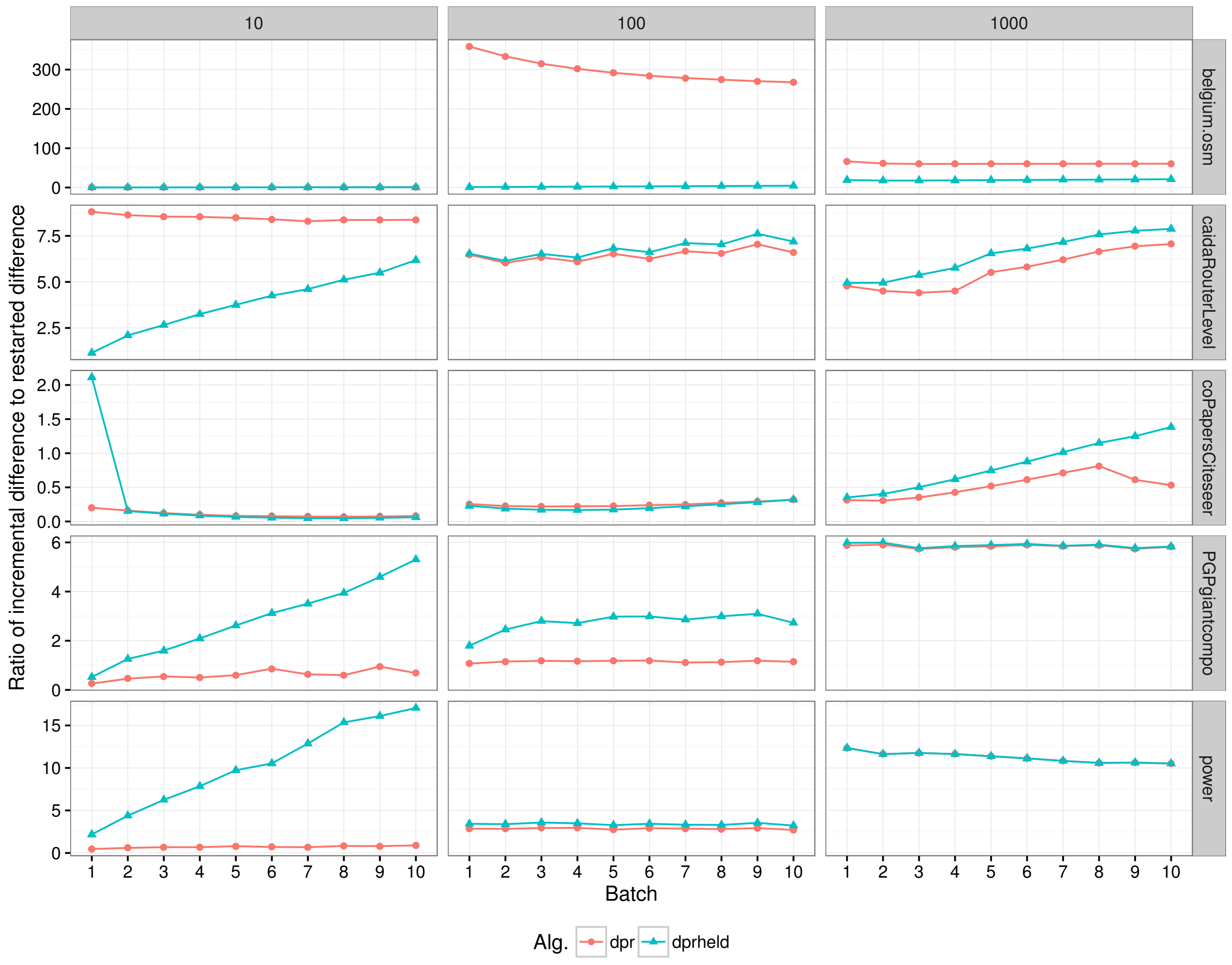

Fig. 4. Ratio of the one-norm differences in PageRank vectors compared to from-scratch recomputation. The incremental algorithms generally compute a result close to restarted PageRank with a few odd and unexpected results.

Note that our algorithms above never use the number of vertices $|\mathcal{V}|$ explicitly but only implicitly through v when computing a general, non-personalized PageRank. Section II-A discusses some options for continuing to ignore $|\mathcal{V}|$.

VI. Related Work

Trying to cover all the work related to computing PageRank is a massive unstructured data mining problem in itself. We compare only with work on *updating* PageRank, and likely this is only a subset. Every unpublished presentation of these algorithms so far have been accompanied by a plea for existing equivalent work to no response. That these algorithms have little value for updates *large* relative to the initial graph may be why they are not yet published. Or perhaps it's that no one will agree on how to capitalize PageRank.

Langville and Meyer [13] consider updating the eigenvalue formulation of PageRank and update the PageRank vector using the power method while summarizing unaffected regions through projection, reducing the matrix dimension to one more than the change size. This method requires re-projecting the matrix as the affected portion grows or starting with a larger region than just the changes. The projection requires accessing the entire graph or using clumped approximation. Convergence depends on the second largest (in magnitude) eigenvalue[14]. Langville and Meyer's method extends to general Markov chains[15].

Bahmani, Chowdhury, and Goel [16] incrementally update PageRank and other random walk metrics by maintaining a database of sampled paths. The number of paths necessary is inversely proportional to the estimation error. This method can be useful for finding the largest ranked entries, but tracking smaller entries on the fringe of an region defined by a very sparse seed or personalization vector will be indistinguishable. Large estimation errors may produce output that is not useful for later analysis like anomaly detection. Bahmani, *et al*. extend the method to updating PageRank without knowing the graph changes explicitly[17]. In their context, a distributed web crawl is being updated by some process that does not communicate changes to the PageRank computation. Randomly sampling paths from the graph periodically can achieve reasonably small errors on small graphs, but limited sampling may miss notable changes in a massive graph.

Ohsaka, *et al*. [18] present a nearly equivalent algorithm for updating PageRank coming from the statistical direction. The primary difference is that Ohsaka, *et al*.'s update equations focus on the residual and do not include the first term in Equation (3), $\alpha A_{\Delta}^{T} D_{\Delta}^{-1} \Delta x^{(k)}$, but rather directly add the residual to the changed component of x. This paper also contains intriguing data on time to converge using their incremental algorithm for initial computation that may reflect on tapping into a live data stream. Combining their analysis with our algorithm could prove very fruitful and may produce better criteria for limiting expansion in `dpr_held`.

Gleich and Polito [19] provide a static personalized PageRank algorithm that limits the amount of graph data used. Because our algorithm for computing the incremental update is a personalized PageRank problem, Gleich and Polito's boundary-restricted version may provide a method for further limiting the change area.

VII. Conclusion

Our algorithms in Section III essentially apply iterative refinement to maintain a PageRank vector through sparse refinement on a streaming graph. The incremental algorithms produce the update directly while traversing only a limited amount of the total graph, from traversing only slightly fewer edges with many scattered changes to thousands fewer when only ten edges are inserted. This can translate into hundred-fold speed ups over restarting an iterative PageRank solver. A few unexplained performance and accuracy oddities remain and are under investigation. The incremental algorithms in Figure 2 could benefit from an efficient implementation of the upcoming GraphBLAS API but needs methods to reduce overhead in sparse vector manipulation.

Iterative refinement is an application of Newton's method to linear systems $Ax = b$ and generalizes in theory to any numerical refinement procedure. Only linear systems so far provide the fast, non-dense refinement algorithms suitable for massive graph analysis. Spectral clustering notably may not benefit from similar sparse refinement to track streaming graphs. Techniques for eigenvalue problems use factored forms like the Schur decomposition [20] and are not suitable for massive graph analysis. Methods combining linear combinations of eigenvectors [21] with subspace tracking may be more fruitful.

Acknowledgment

The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." This work was also partially sponsored by NSF Grant ACI-1339745 (XScala). We thank Oracle for the hardware platform and reviewers for their careful attention to detail, relevant references, and useful suggestions.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford Digital Library Technologies Project, Stanford University, Stanford, CA, USA, Tech. Rep., Nov. 1998, p. 17.

[2] D. F. Gleich, "PageRank beyond the web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2015. DOI: 10.1137/140976649.

[3] J. J. Whang, D. F. Gleich, and I. S. Dhillon, "Overlapping community detection using seed set expansion," in *CIKM*, Q. He, A. Iyengar, W. Nejdl, J. Pei, and R. Rastogi, Eds., ACM, 2013, pp. 2099–2108, ISBN: 978-1-4503-2263-8.

[4] E. J. Riedy and D. A. Bader, "Multithreaded community monitoring for massive streaming graph data," in *7th Workshop on Multithreaded Architectures and Applications (MTAAP)*, Boston, MA, May 2013. DOI: 10.1109/IPDPSW.2013.229.

[5] D. Gleich, L. Zhukov, and P. Berkhin, "Fast parallel PageRank: A linear system approach," Yahoo! Research, Tech. Rep. YRL-2004-038, 2004, p. 22.

[6] G. M. Del Corso, A. Gull, and F. Romani, "Fast PageRank computation via a sparse linear system," *Internet Math.*, vol. 2, no. 3, pp. 251–273, 2005.

[7] N. J. Higham, "Iterative refinement for linear systems and LAPACK," *IMA J. Numer. Anal.*, vol. 17, no. 4, pp. 495–509, 1997.

[8] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *The IEEE High Performance Extreme Computing Conference (HPEC)*, Best paper award, Waltham, MA, Sep. 2012. DOI: 10.1109/HPEC.2012.6408680.

[9] OpenMP Architecture Review Board, *OpenMP application program interface version 4.0*, Jul. 2013.

[10] T. A. Davis, "Algorithm 832: UMFPACK V4.3 - an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw*, vol. 30, no. 2, pp. 196–199, 2004.

[11] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner, "Encyclopedia of social network analysis and mining," in. Springer, 2014, ch. Benchmarking for Graph Clustering and Partitioning, pp. 73–82.

[12] J. Kepner, D. A. Bader, A. Buluç, J. R. Gilbert, T. G. Mattson, and H. Meyerhenke, "Graphs, matrices, and the GraphBLAS: Seven good reasons," *CoRR*, vol. abs/1504.01039, 2015.

[13] A. N. Langville and C. D. Meyer, "Updating Pagerank with iterative aggregation," in *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, ser. WWW Alt. '04, New York, NY, USA: ACM, 2004, pp. 392–393, ISBN: 1-58113-912-8. DOI: 10.1145/1013367.1013491.

[14] I. C. F. Ipsen and S. Kirkland, "Convergence analysis of a PageRank updating algorithm by Langville and Meyer," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, no. 4, pp. 952–967, 2006. DOI: 10.1137/S0895479804439808.

[15] A. N. Langville and C. D. Meyer, "Updating Markov chains," in *Proceedings of the Markov Anniversary Meeting*, Boson Press, 2006.

[16] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized PageRank," *Proc. VLDB Endow.*, vol. 4, no. 3, pp. 173–184, Dec. 2010, ISSN: 2150-8097. DOI: 10.14778/1929861.1929864.

[17] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal, "PageRank on an evolving graph," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12, Beijing, China: ACM, 2012, pp. 24–32, ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339539.

[18] 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*, ser. KDD '15, Sydney, NSW, Australia: ACM, 2015, pp. 875–884, ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783297.

[19] D. Gleich and M. Polito, "Approximating personalized PageRank with minimal use of web graph data," *Internet Mathematics*, vol. 3, no. 3, pp. 257–294, 2006. DOI: 10.1080/15427951.2006.10129128.

[20] K. E. Prikopa and W. N. Gansterer, "On mixed precision iterative refinement for eigenvalue problems," in *ICCS*, H. D. Pfeiffer, D. I. Ignatov, J. Poelmans, and N. Gadiraju, Eds., ser. Lecture Notes in Computer Science, vol. 18, Elsevier, 2013, pp. 2647–2650, ISBN: 978-3-642-35785-5; 978-3-642-35786-2.

[21] J. P. Fairbanks, G. D. Sanders, and D. A. Bader, "Spectral partitioning with blends of eigenvectors," *CoRR*, vol. abs/1510.04658, 2015.