SimGQ: Simultaneously Evaluating Iterative Graph Queries *

Chengshuo Xu
CSE Dept., UC Riverside
cxu009@ucr.edu

Abbas Mazloumi
CSE Dept., UC Riverside
amazl001@ucr.edu

Xiaolin Jiang
CSE Dept., UC Riverside
xjian049@ucr.edu

Rajiv Gupta
CSE Dept., UC Riverside
gupta@cs.ucr.edu

***Abstract*—Graph processing frameworks are typically designed to optimize the evaluation of a single graph query. However, in practice, we often need to respond to multiple graph queries, either from different users or from a single user performing a complex analytics task. Therefore in this paper we develop SimGQ, a system that optimizes simultaneous evaluation of a group of *vertex queries* that originate at different source vertices (e.g., multiple shortest path queries originating at different source vertices) and delivers substantial speedups over a conventional framework that evaluates and responds to queries one by one. The performance benefits are achieved via *batching* and *sharing*. *Batching* fully utilizes system resources to evaluate a batch of queries and amortizes runtime overheads incurred due to fetching vertices and edge lists, synchronizing threads, and maintaining computation frontiers. *Sharing* dynamically identifies *shared queries* that substantially represent subcomputations in the evaluation of different queries in a batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all queries in the batch. With four input power-law graphs and four graph algorithms SimGQ achieves speedups of up to 45.67× with batch sizes of up to 512 queries over the baseline implementation that evaluates the queries one by one using the state of the art Ligra system. Moreover, both batching and sharing contribute substantially to the speedups.**

***Index Terms*—batch of queries, sharing computation, amortizing overhead, power-law graphs, shared-memory parallelism**

I. Introduction

Graph analytics is employed in many domains (e.g., social networks, web graphs, internet topology, brain networks etc.) to uncover insights by analyzing high volumes of connected data. An iterative algorithm updates vertex property values of active vertices in each iteration driving them towards their final stable solution. When the solution values of all vertices become stable, the algorithm terminates. It has been seen that real world graphs are often large with millions of vertices and billions of edges. Moreover, iterative graph analytics requires repeated passes over the graph till the algorithm converges to a stable solution. As a result, in practice, iterative graph analytics workloads are data-intensive and often compute-intensive. Therefore, there has been a great deal of interest in developing scalable graph analytics systems like Pregel [13], GraphLab [12], GraphIt [28], PowerGraph [5], Galois [17], GraphChi [10], Ligra [21], ASPIRE [25].

* This work is supported in part by National Science Foundation grants CCF-2002554, CCF-1813173, and CCF-2028714 to the University of California Riverside, California, USA

While the performance of graph analytics has improved greatly due to advances introduced by the above systems, much of this research has focussed on developing highly parallel algorithms for solving a single iterative graph analytic query (e.g., SSSP(s) query computes shortest paths from a single source s to all other vertices in the graph) on different computing platforms including shared-memory systems, distributed clusters, and systems with accelerators like GPUs. However, in practice the following two scenarios involve multiple queries: (a) Single-User scenario as in Quegel [26], where the authors developed an analyzer for online shopping platform that frequently computes shortest-paths between some important shoppers in a large network extracted from online shopping data; and (b) Multi-User scenarios as in Congra [16] and [23] where the same data set is queried by many users. In both scenarios, machine resources can be fully utilized delivering higher throughput by simultaneously evaluating multiple queries on a modern server with many cores and substantial memory resources.

In this paper we develop SimGQ, a graph analytics system aimed at evaluating a batch of *vertex queries* received from users for different source vertices of a large graph. For example, for SSSP algorithm, we may be faced with the following batch of queries: $SSSP(s_1)$, $SSSP(s_2)$, $\cdots\cdots$ $SSSP(s_n)$. Many other important algorithms belong to this category [6], [7], [11] etc. Our overall approach is as follows. Given an input graph and batch of vertex queries, we synergistically perform simultaneous evaluation of all queries in a batch to deliver results of all queries in a greatly reduced time. Essentially the synergy in evaluation of queries, that exists due to the substantial overlap between computations and graph traversals for different queries, is exploited to amortize the runtime overhead and computation costs across the simultaneously evaluated queries. Two techniques, *batching* and *online sharing*, are employed to simultaneously and efficiently evaluate a set of queries.

(a) Batching for Resource Utilization and Amortizing Overheads. Batching takes a group of queries, forming the batch, and simultaneously processes these queries to achieve higher throughput by fully utilizing system resources and amortizing runtime overhead (e.g., synchronization) costs across queries. Some prior works [23], [26] process multiple queries simultaneously. In [23] authors process multiple

978-0-7381-1035-6/20/$31.00 ©2020 IEEE
DOI 10.1109/HiPC50609.2020.00014

queries but the solution is optimized specifically for BFS queries. Quegel [26] pipelines execution of a few queries delivering limited performance enhancement as shown in [14]. MultiLyra [14] performs batching on distributed systems and thus mainly derives benefits from amortizing cost of communication between machines. In contrast, SimGQ is capable of evaluating large batches (up to 512 queries) of a general class of queries on a shared-memory system for high throughputs.

(b) Online Sharing. To amortize computation costs, we develop a novel strategy that dynamically identifies *shared queries* whose computations substantially overlap with the computations performed by multiple queries in the batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all the queries in the batch. The shared subcomputations are essentially query evaluations for a small set of high degree source vertices, different from the source vertices of queries in the batch, such that they can be used to accelerate the evaluation of all queries in the batch.

Online sharing has multiple advantages over classical global indexing methods for optimizing evaluation of queries. First, indexing entails heavy weight precomputation used to build a large index that can be used to accelerate all future queries (e.g., Quegel [26] uses Hub-Accelerator based indexing). Second, as soon as the graph changes, precomputed indexing/profiling information becomes invalid. The *online sharing* as performed by SimGQ involves no precomputation and thus eliminates its high cost while also accommodating changes to the graph between different batches of queries. Thus, our approach applies to streaming/evolving graphs.

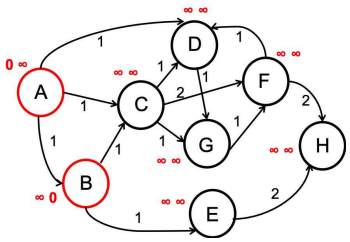
In SimGQ, the evaluation of the batch of queries is carried out as follows. We partially evaluate the queries in the batch for a few iterations till some high degree vertices enter the frontiers of the queries in the batch. We *pause* the evaluation of the batch queries and select a small set of high degree vertices encountered. Treating selected vertices as source vertices of queries, we construct a batch of *shared queries* and then evaluate this batch. The results of shared queries are then used to *quickly update* the solutions of all queries in the original batch and hence accelerate their advance towards the final stable solution. Finally, we *resume* the paused evaluation of original batch till their stable solutions have been found. By simultaneously evaluating queries we also amortize the runtime overheads incurred, such as costs of accessing graph vertices and edges, synchronization costs, and maintaining computation frontiers as multiple queries traverse the same regions of the graph.

We implemented SimGQ by modifying the state-of-the-art Ligra [21] system. Our experiments with multiple input power-law graphs and multiple graph algorithms demonstrate best speedups ranging from 1.53× to 45.67× with batch sizes ranging up to 512 queries over the a baseline implementation that evaluates the queries one by one using the state of the art Ligra system. Moreover, we show that both batching and sharing techniques contribute substantially to the speedups.

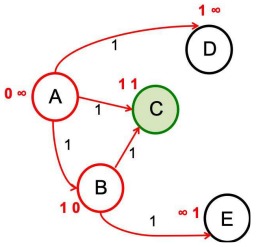
The remainder of the paper is organized as follows. In section 2 we first provide an overview of SimGQ and then present our algorithms in detail. In section 3 we evaluate SimGQ. In section 4 we discuss related work. Finally, we conclude in section 5.

II. SimGQ: Evaluating a Batch of Queries

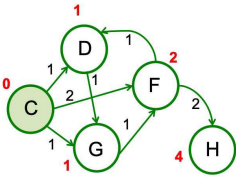
When a group of iterative graph queries are evaluated as a *batch*, following opportunities for speeding up their evaluation arise that are ignored when evaluating the queries one by one.



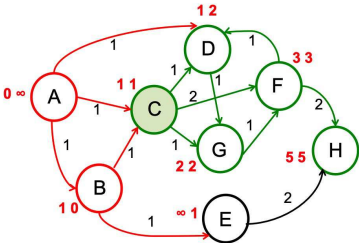
(a) *Initialization* for Batch(A,B) of SSSP Queries.



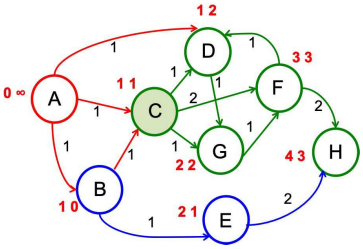
(b) *Phase I.* Evaluate Batch(A,B), Pause and Identify SSET.



(c) *Phase II.* Evaluate SSET.



(d) *Phase II Contd.* Update Batch(A,B) Results Using SSET Results.



(e) *Phase III.* Resume Batch(A,B) Evaluation till Termination.

Fig. 1: **Overview of Sharing Among a Batch of Queries**

First, it is easy to see that during batched evaluation, we can share the iteration overhead across the queries. This overhead includes the cost of iterating over the loop, synchronizing threads at the barrier, as well as fetching vertex values and edge lists of active vertices to update vertex values and the computation frontier. Second, synergy or overlap between computations performed by the queries can be exploited to reduce the overall computation performed. In particular, we can identify and evaluate *shared queries* whose results can be used to accelerate the evaluation of all the queries in the batch. The computation performed by the shared queries substantially represent subcomputations that are performed by many queries in the batch. This is because different queries typically traverse the majority of the graph and consequently present an opportunity to share a subcomputation across multiple queries. By evaluating the shared queries once, we can speedup the evaluation of the entire batch of queries. Note that the shared queries must be identified dynamically because they may vary from one batch to another.

A. Overview of SimGQ

Next we provide an overview of SimGQ via an illustrating example. Figure 1 shows how a batch of two queries can be synergistically evaluated by identifying and evaluating shared queries first. While in the example we use a directed graph, our approach works equally well for undirected graphs with a minor adjustment. As in other works, each undirected edge is represented by a pair of directed edges with equal weight.

Initialization Step. Since all queries in a given batch are to be evaluated simultaneously, each vertex is assigned a vector to hold data values for all queries in the batch – each position in the vector corresponds to a specific query in the batch. In Figure 1a we aim to solve a batch of two SSSP queries for source vertices A and B marked in red. Each node is annotated with a pair of initial values for the two queries, A first and then B. Initial value 0 is assigned to source vertices and value ∞ to all non-source vertices for each of the SSSP queries.

Phase I: Identifying Shared Queries. Simultaneously starting from the source vertices, we start traversing the graph updating the shortest path lengths for the processed vertices along the frontier as shown in red in Figure 1b. The evaluation of the batch continues and once good candidate vertices for *shared queries* SSET are found, the evaluation of the batch is *paused*. Let us assume that after one iteration we identify SSSP(C) (C marked in green) as a good shared query candidate for the two queries in the batch in our example. Thus, we *pause* the evaluation of the batch queries and proceed to the next step to process the identified shared queries.

Phase II. Accelerating Batch Queries Using Shared Queries. In this step we evaluate the shared queries first, that is we evaluate them till their stable results have been computed. For example, in Figure 1c we evaluate the shared query SSSP(C). Once the shared queries have been evaluated, their results are used to rapidly update the partial results of all the original batch queries as shown in Figure 1d. Note that at this point the results of all vertices except B and E have already reached their final stable values. That is, the evaluation of batch queries has greatly advanced or *accelerated*.

Phase III. Completing the Evaluation of Batch Queries. In this final step we *resume* the evaluation of batch queries from the frontier at which the evaluation was paused earlier. In our example, the resumption of evaluation takes place at vertices B and E and finally the algorithm terminates after updating the results at vertices E and H. Note that if the acceleration performed in Phase II is effective, the combined cost of Phase I and Phase III would be significantly less than the cost of evaluating the batch without employing *sharing* affected via Phase II.

While the above example provides an overview of our approach, many algorithm details and heuristic criteria need to be developed. For example, there are different ways to select shared queries (queries on vertices with high centrality or high degree, queries on vertices that are reachable by most source vertices in the batch etc.). Since our work focuses on power-law graphs that have small diameter and skewed degree distribution, *high degree vertices* are the best candidates for global queries that in general traverse nearly the entire graph. Our algorithm first marks a set of high degree vertices as *potential shared vertices*. At runtime, a heuristic is used to select a small subset of *shared vertices* that are not only marked, but also have been encountered more frequently during partial evaluation of batch queries. After evaluating the *shared queries*, we use the results to quickly update the results of all batch queries. In subsequent subsections we present a push-style evaluation of a batch of queries assisted by our idea of using shared queries.

B. Push-Style Batch Evaluation With Sharing

Now we present a detailed algorithm that evaluates a batch of vertex queries, employing both batching and sharing, using Push model (a similar algorithm can be easily designed for the Pull model). In Algorithm 1, function EVALUATEBATCH (line 6) simultaneously evaluates a batch of vertex queries for source vertices $s_1, s_2, ..., s_k$, over a directed graph $G\ (V, E)$. The algorithm uses $M \subset V$ as a set of marked high degree vertices from which a small number of vertices are selected to form *shared queries*; different batches of queries yield different shared queries. In our experiments $|M|$ is set to 100 to provide choices that suit different batches, while up to 5 shared queries are selected to limit the overhead of sharing (i.e., SSET size is 5). The algorithm maintains an ACTIVE vertex set, the combined frontier for all queries in the batch. Although ACTIVE tells which vertices are active, it cannot tell which queries are associated with each active vertex. Therefore, in addition to ACTIVE, our algorithm maintains two fine-grained active lists, CURRTRACK and NEXTTRACK, to indicate for each active vertex all the queries whose frontier the active vertex belongs to. While CURRTRACK is the information for active set being processed, NEXTTRACK is the corresponding information for the active set being formed for the next super

Algorithm 1 Batched Evaluation With Sharing

```
1: Given: Directed graph G(V, E);
2:         High Degree Set M ⊂ V of Marked Vertices
3: Goal: Evaluate a Batch of Queries
4:         QUERYBATCH ← { Q_1(s_1), Q_2(s_2), ..., Q_k(s_k) }
5:
6: function EVALUATEBATCH( QUERYBATCH )
7:     [Initialization Step]
8:     INITIALIZE RESULTTABLE for QUERYBATCH
9:     ACTIVE ← { s_1, s_2, ..., s_k }
10:    CURRTRACK ← { (s_i, Q_i) : Q_i(s_i) ∈ QUERYBATCH }
11:    NEXTTRACK ← φ
12:    ITERATION ← 0
13:    ▷ Iterate till Convergence
14:    while ACTIVE ≠ φ do
15:        [Phase I: Iteration ≤ p] [Phase III: Iteration > p]
16:        ▷ Process Active Vertices
17:        ACTIVE ← PROCESSBATCH ( ACTIVE, ITERATION,
                    CurrTrack, NEXTTRACK, RESULTTABLE )
18:        if ITERATION = p then
19:            [Phase II]
20:            ▷ Identify #SSET as the Most Frequently Visited
                  Vertices from M as the source of Shared Queries
21:            SSET ← SELECTSHAREDQS (M, Visits, #SSET)
22:            ▷ Evaluate Shared Queries with Sources in SSET
23:            SHAREDTABLE ← EVALUATEBATCH (SSET)
24:            ▷ Update RESULTTABLE using SHAREDTABLE
25:            SHAREUPDATEBATCH ( SSET,
                              SHAREDTABLE, RESULTTABLE )
26:        end if
27:        CURRTRACK ← NEXTTRACK
28:        NEXTTRACK ← φ
29:    end while
30:    return RESULTTABLE
31: end function
32:
33: function PROCESSBATCH ( ACTIVE, ITERATION, CURRTRACK,
    NEXTTRACK, RESULTTABLE )
34:    NEWACTIVE ← φ
35:    for all v ∈ ACTIVE in parallel do
36:        for all e ∈ G.outEdges(v) in parallel do
37:            ▷ Apply conventional Update on e.dest
38:            changed ← EDGEFUNCBATCH ( e,
                    CURRTRACK, NEXTTRACK, RESULTTABLE )
39:            if ( ITERATION ≤ p ) and ( e.dest ∈ M ) then
40:                Visits[e.dest]++
41:            end if
42:            ▷ Update Active Vertex Set for next Iteration
43:            if changed then
44:                NEWACTIVE ← NEWACTIVE ∪ {e.dest}
45:            end if
46:        end forall
47:    end forall
48:    return NEWACTIVE
49: end function
50:
```

Algorithm 2 Batched Edge Update Function

```
1: function EDGEFUNCBATCH (e, CURRTRACK, NEXTTRACK,
   RESULTTABLE)
2:     ▷ Initialize RETVALUE to false.
3:     ▷ Set to true if value of e.dest is changed.
4:     RETVALUE ← false
5:     for all Q_i(s_i) ∈ QueryBatch do
6:         ▷ Only Attemp Update for Queries activated e.source
7:         if (e.source, Q_i) ∈ CURRTRACK then
8:             ▷ Perform Update via e
9:             if UPDATEFUNC(e, Q_i, RESULTTABLE) == true then
10:                ▷ Schedule e.dest for next Iteration
11:                RETVALUE ← true
12:                NEXTTRACK ← NEXTTRACK ∪ {(e.dest, Q_i)}
13:            end if
14:        end if
15:    end for
16:    return RETVALUE
17: end function
```

Algorithm 3 Identify Shared Queries from M

```
1: Given: High Degree Set M ⊂ V of Marked Vertices
2:         Vector Visits: Number of Visits of All Vertices ∈ M
3:         Constant #SSET: # of Shared Vertices Selected
4: Goal: Select #SSET most frequently visited Vertices in M
5:
6: function SELECTSHAREDQS (M, Visits, #SSET)
7:     ▷ Init: Set of Source Vertices for Shared Queries
8:     SSET ← φ
9:     ▷ Init: Set of (vertex, vertex visits number) pairs
10:    VERTVISITSPAIRS ← φ
11:    for all v ∈ M do
12:        VERTVISITSPAIRS←VERTVISITSPAIRS∪{v,Visits[v]}
13:    end for
14:    ▷ Sort Vertices subject to Number of Visits
15:    Sort(VERTVISITSPAIRS, moreVisits())
16:    ▷ Select most frequently visited Marked Vertices
17:    for #SSET top {v, Visits[v]} ∈ VERTVISITSPAIRS do
18:        SSET ← SSET ∪ {v}
19:    end for
20:    return SSET
21: end function
```

step of the algorithm. The RESULTTABLE maintains the results of all the queries for each vertex, and at termination the results of all queries can be found in it.

Following the initialization step (lines 7-12), in each super iteration (lines 14-29), the vertices in ACTIVE vertex set are processed in parallel by calling function PROCESSBATCH (lines 33-49) . This function updates the value of out-neighbors of active vertices in Push style fashion and generates NEWACTIVE containing the active vertices for next iteration which it returns to EVALUATEBATCH at the end. The work performed by the loop at line 14 executes the three phases of our algorithm. The first p iterations form Phase I, following which, next in Phase II first shared queries SSET are identified by calling SELECTSHAREDQS (line 21) and then the queries in SSET are evaluated (line 23). Finally, the evaluation of original batch of queries is accelerated by updating their results in RESULTTABLE using the results of SSET queries in SHAREDTABLE (line 25). Finally in Phase III the computation of batch queries is resumed and completed in remaining iterations of the while loop. During Phase I the algorithm maintains a count of number of visits to each vertex in M (line 40). These counts are used for selecting vertices to form SSET, more visits implies greater relevance to queries in the original batch and hence higher priority for inclusion in SSET. Following the call to PROCESSBATCH in the p^{th}

iteration (1^{st} in our experiments), we enter Phase II at which point SSET is built. The details of SSET construction are presented in Algorithm 3.

Function PROCESSBATCH loops over each outedge e of every active vertex, and calls function EDGEFUNCBATCH (Algorithm 2) to attempt update of $e.dest$ by relaxing edge e using conventional edge update function UPDATEFUNC. If the relaxation is successful, i.e. the value of $e.dest$ is changed, $e.dest$ becomes an active vertex for next iteration. Note that function EDGEFUNCBATCH does not blindly relax e for all queries. Instead it looks up CURRTRACK to check which queries activated $e.source$ in the previous iteration, and only attempts update of value of $e.dest$ for corresponding queries.

Note that if lines 18-26 are eliminated, the algorithm will not perform sharing and thus its execution will revert to simple batched evaluation. We present conventional edge update function UPDATEFUNC for five algorithms in Table I. Here CASMIN(a, b) sets $a = b$ if $b < a$ atomically using compare-and-swap; and CASMAX(a, b) sets $a = b$ if $b > a$ atomically using compare-and-swap.

Algorithm 4 Accelerate Batch Queries Using Results of Shared Queries From SSET

```
1: function SHAREUPDATEBATCH (SSET, SHAREDTABLE, RESULTTABLE)
2:     for all Q_i(s_i) ∈ QUERYBATCH do
3:         for r ∈ SSET do
4:             ▷ Update using r only if r is reachable from s_i
5:             if RESULTTABLE[s_i][r] ≠ −1 then
6:                 for d ∈ ALLVERTICES do
7:                     ▷ Attempt Update if d is reachable from r
8:                     if SHAREDTABLE[r][d] ≠ −1 then
9:                         ▷ Update d for Query i using r
10:                        SHAREUPDATEFUNC ( d, r, Q_i,
11:                            SHAREDTABLE, RESULTTABLE )
12:                    end if
13:                end for
14:            end if
15:        end for
16:    end for
17: end function
```

Finally, Algorithm 4 shows how we accelerate the convergence of the solution of the original batch of queries in RESULTTABLE using the results of the shared queries in SHAREDTABLE. Since the cost for looping over all vertices and applying share updates is significant, we limit the number of shared vertices with which each query is used to speed up convergence of property values by choosing a small SSET size. Let us see how the result of a shared query with source vertex r can benefit a batch query suppose the reachability is known to be true. Given a vertex d, its value in query Q_i can take advantage of the shared result of subquery on vertex r in SHAREDTABLE as follows: SHAREUPDATEFUNC(d, r, Q_i, SHAREDTABLE, RESULTTABLE). The above function for five benchmarks is given in Table II. For example, for SSSP,

$$\text{RESULTTABLE}[s_i][r] + \text{SHAREDTABLE}[r][d]$$

is a safe approximation of the shortest path value from source vertex of q_i to d via r, and we can use the estimation to accelerate the convergence of the value of d.

For undirected graphs, when applying update using result of shared queries, we can benefit from a more accurate measurement of the property value from source vertex to shared vertex. Take SSSP as an example. Given an undirected graph, SHAREDTABLE$[r][s_i]$ can be used as the accurate measurement of the distance from s_i to r. Compared with RESULTTABLE$[s_i][r]$ used in Table II, which is an approximation value, SHAREDTABLE$[r][s_i]$ can be used to compute a better estimation of the distance between s_i and d and therefore give better acceleration on the evaluation of original batch queries.

C. Applicability

Our sharing algorithm can be applied to batched iterative graph algorithms where each query in the batch begins at single source vertex and the property values from these sources to all other vertices are computed. Sharing of results of subqueries is effective because they represent overlapping subcomputations. Graph problems with dynamic programming solutions have the opportunity to benefit from our sharing algorithm because of the optimal substructure property of dynamic programming. Examples include monotonic computations like **SSWP**, **Viterbi**, **TopkSSSP**, and **BFS** used in our evaluation as well as other non-monotonic algorithms like Personalized Page Rank (**PPR**) [6] used by recommender services like twitter and Single-Source SimRank (**SimRank**) [7] queries that are evaluated to compute similarities of graph nodes. It does not apply to algorithms with a global solution, i.e. not originating at source-vertex (e.g., Connected Components). Sharing will work less effectively for local queries like 2-Hop queries due to low overlap between them; however, local queries are inexpensive and can be processed efficiently with batching alone. Sharing works well on power-law graphs as they contain high centrality nodes but it is less effective for high-diameter graphs like road-networks. Only when source vertices are in proximity of each other can there be significant reuse in high-diameter graphs.

III. EXPERIMENTAL EVALUATION

A. Experimental Setup

For evaluation we implemented our SimGQ framework using Ligra [21] which uses the Bulk Synchronous Model [24] and provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. We evaluate our techniques for evaluation of batches of queries using four benchmark applications (**SSWP** – Single Source Widest Path, **Viterbi** [11], **BFS** – Breadth First Search, and **TopkSSSP** – Top k Single Source Shortest Paths). We used four real world power-law graphs shown in Table III in these experiments – **TT** [3] and **TTW** [9] are large graphs with 2.0 and 1.5 billion edges respectively; and **LJ** [2] and **PK** [22] are smaller graphs with 69 and 31 million edges respectively. Benchmarks are implemented using the PUSH model on a machine with 32 cores (2 sockets, each with 16 cores) with

TABLE I: Conventional Updates for Five Algorithms.

ALG	RESULTTABLE[s_i][$e.dest$] ← UPDATEFUNC (e, Q_i, RESULTTABLE)
SSWP	CASMAX(RESULTTABLE[s_i][$e.dest$], min(RESULTTABLE[s_i][$e.src$], $e.w$)))
Viterbi	CASMAX(RESULTTABLE[s_i][$e.dest$], RESULTTABLE[s_i][$e.src$] / $e.w$)
BFS	CASMIN(RESULTTABLE[s_i][$e.dest$], RESULTTABLE[s_i][$e.src$] + 1)
SSSP	CASMIN(RESULTTABLE[s_i][$e.dest$], RESULTTABLE[s_i][$e.src$] + $e.w$)
TopkSSSP	KSMALLEST($\{$RESULTTABLE[s_i][$e.dest$]$\}$ $\cup$ $\{$RESULTTABLE[s_i][$e.src$] + $e.w\}$)

TABLE II: Directed Graphs: SHAREUPDATEFUNC for Five Algorithms.

ALG	RESULTTABLE[s_i][d] ← SHAREUPDATEFUNC(d,r,Q_i, SHAREDTABLE,RESULTTABLE)
SSWP	CASMAX(RESULTTABLE[s_i][d], min(RESULTTABLE[s_i][r], SHAREDTABLE[r][d]))
Viterbi	CASMAX(RESULTTABLE[s_i][d], RESULTTABLE[s_i][r] * SHAREDTABLE[r][d])
BFS	CASMIN(RESULTTABLE[s_i][d], RESULTTABLE[s_i][r] + SHAREDTABLE[r][d])
SSSP	CASMIN(RESULTTABLE[s_i][d], RESULTTABLE[s_i][r] + SHAREDTABLE[r][d])
TopkSSSP	KSMALLEST($\{$RESULTTABLE[s_i][d]$\}$ $\cup$ $\{$RESULTTABLE[s_i][r] + SHAREDTABLE[r][d]$\}$)

TABLE III: Input graphs used in experiments.

Graphs	#Edges	#Vertices
Twitter (TT) [3]	2.0B	52.6M
Twitter (TTW) [9]	1.5B	41.7M
LiveJournal (LJ) [2]	69M	4.8M
PokeC (PK) [22]	31M	1.6M

TABLE IV: BASELINE – Total Execution Times for Evaluating Randomly Selected Queries One by One in `Seconds` on the Ligra [21] System. For first 3 benchmarks 512 queries are used and for TopkSSSP we use 64 queries.

Graph	SSWP	Viterbi	BFS	Top 2 & 1 SSSP	
TTW	2,989s	3,737s	2,574s	4,073s	2,337s
TT	3,949s	4,902s	3,538s	2,768s	1,574s
LJ	134s	258s	102s	389s	226s
PK	63s	116s	55s	232s	123s

Intel Xeon Processor E5-2683 v4 processors, 512 GB memory, and running CentOS Linux 7.

For each combination of benchmark application and input graph, we used 512 randomly generated queries to carry out the evaluation, expect for **TopkSSSP** for which we use 64 queries because of runtime cost. The *baseline* total execution times when the queries are evaluated one by one is given in Table IV. Because **TTW** and **TT** are far bigger in size than **LJ** and **PK** , the execution times for **TT** and **TTW** are higher.

B. Benefits of Sharing and Batching

In this section we present the results of our algorithm, we refer to them as **Batch+Share**. In addition, we also collect execution times of algorithm that uses batching but no sharing, we refer to this algorithm as **Batch**. Since the batch size is an important parameter in this evaluation, we vary batch sizes from **4** queries (the smallest) to a very large number of **512** queries. For **TTW** and **TT** the maximum batch size was limited to 256 because our 512 GB machine did not have sufficient memory to run 512 queries for very large graphs. For **TopkSSSP** maximum batch size of 64 was used due to its high runtimes.

The results of running the above algorithms are presented in Table V and Figure 2. While Table V presents the total execution times for 512 queries for batch sizes (number in parentheses) that yielded the highest speedup for each of the algorithms, Figure 2 presents average per query execution times for all batch sizes for **TT** the largest graph.

The data in this Table V shows that our algorithms yield speedups of up to **45.67**× over the baseline that executes the queries one by one using the state of the art Ligra system. For the first two benchmarks of **SSWP** and **Viterbi** the **Batch+Share** algorithm delivers speedups ranging from **22.11**× to **45.67**×. In contrast, for the last two benchmarks of **BFS** and **TopkSSSP** the highest speedups observed range from **1.53**× to **6.63**×.

The sharing algorithm is more profitable if the result values of queries fall in a narrow range and hence often overlap. Like the result of **SSWP** query is usually an integer between 17 and 25, and the answer of Viterbi is between 0 and 1. In these cases, sharing produces lots of stable values and reduces the number of iterations because vertices made stable by sharing will never be activated again. Sharing is also effective when the vertex update function is expensive even if it produces few stable values – **TopkSSSP** is a representative graph algorithm from this category. Here sharing reduces the number of updates by 34% but produces few stable values. **BFS** does not fall into any of these two categories and thus, as expected, does not benefit much from sharing.

Let us consider results in Figures 2 that present the *average per query execution times* for varying batch sizes. The trends for the first three benchmarks show that performance continues to improve with with increasing batch sizes. For **Batch** the improvement is due to greater amortization of runtime overheads while for **Batch+Share** the improvement is greater due to additional benefits of sharing. Further, we observe that on our machine, once we cross the batch size of **64**, the improvements in performance are relatively small although the best performances reported in Table V are for batch sizes of **256** and **512** for majority of the cases (i.e., different graphs and benchmarks). Based upon the trends observed in Figure 2, for a larger machine with more memory and number of cores, performance can be expected to scale further for larger batch sizes. For **TopkSSSP** while there is less variation with batch size the difference between **Batch** and **Batch+Share** remains substantial.

TABLE V: Best Sharing+Batching Execution Times in `Seconds` for all Queries and Corresponding (Batch Sizes) and Speedup Over No-Batching Baseline (i.e., times in Table IV).

Alg.	SSWP (512 Queries)			Viterbi (512 Queries)			BFS (512 Queries)			Top 2 & 1 SSSP (64 Queries)		
						TTW						
Batch+Share	71	(256)	42.37×	86	(256)	43.42×	440	(128)	5.84×	2671 1260	(32) (32)	1.53× 1.86×
Batch	629	(256)	4.75×	729	(256)	5.13×	388	(256)	6.63×	3652 1876	(32) (8)	1.12× 1.25×
						TT						
Batch+Share	90	(256)	43.96×	107	(256)	45.67×	723	(128)	4.90×	1605 858	(8) (8)	1.73× 1.84×
Batch	1034	(64)	3.82×	1274	(64)	3.85×	692	(128)	5.12×	2768 1574	(1) (1)	1.00× 1.00×
						LJ						
Batch+Share	6	(512)	22.11×	12	(128)	22.27×	23	(256)	4.36×	237 135	(64) (64)	1.64× 1.67×
Batch	37	(256)	3.63×	59	(256)	4.34×	18	(256)	5.63×	375 190	(32) (32)	1.04× 1.19×
						PK						
Batch+Share	2	(512)	28.38×	4	(128)	28.97×	11	(512)	5.01×	119 58	(64) (64)	1.95× 2.13×
Batch	20	(512)	3.24×	30	(256)	3.89×	9	(512)	6.40×	196 98	(16) (16)	1.19× 1.26×

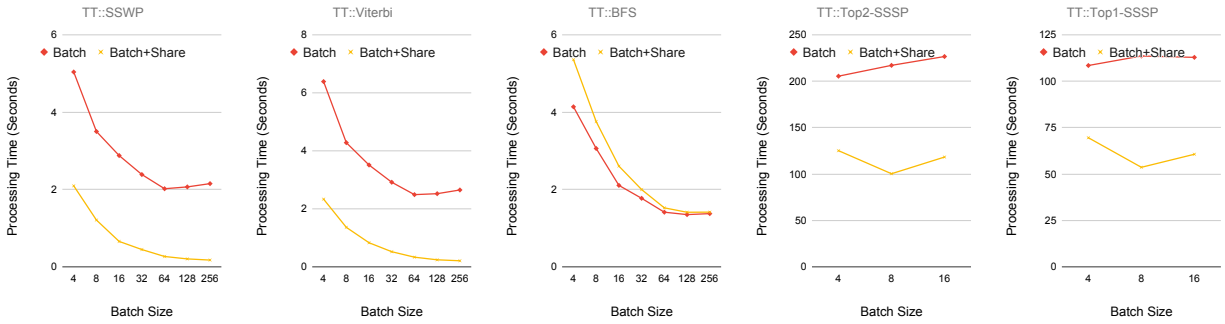


Fig. 2: Average Per Query Execution Times of Batch vs. Batch+Share.

TABLE VI: Batch+Share Over Batch Alone: Cost of Phase II, Benefit of Phase II, Speedup Due to Batch+Share Over Batch Alone. Speedups computed for best Batch+Share configurations for all Queries.

SSWP (512 queries)			Viterbi (512 Queries)			BFS (512 Queries)			Top 2 & 1 SSSP (64 Queries)		
					TTW						
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.08	0.97	8.92×	0.09	0.97	8.47×	0.15	0.07	0.93×	0.04 0.04	0.31 0.41	1.37× 1.60×
					TT						
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.06	0.98	12.26×	0.06	0.98	12.64×	0.11	0.07	0.96×	0.09 0.09	0.63 0.62	2.16× 2.12×
					LJ						
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.12	0.96	6.43×	0.12	0.92	5.14×	0.26	-0.03	0.77×	0.02 0.02	0.43 0.39	1.69× 1.58×
					PK						
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.09	0.98	8.76×	0.08	0.96	8.22×	0.20	-0.08	0.78×	0.02 0.02	0.50 0.50	1.94× 1.94×

C. Contributions of Sharing vs. Batching

We observed that for **SSWP** and **Viterbi** both sharing and batching are responsible for delivering high performance while for **TopkSSSP** batching does not provide benefit, and for **BFS** sharing does not deliver additional performance improvement. We analyze the cost and benefit of sharing to show that for first three benchmarks the benefit far outweighs the cost while for **BFS** the benefit is smaller than the cost incurred thus leading to lower speedup with sharing.

Using the execution times of **Batch** as baseline, Table VI presents the speedups achieved by **Batch+Share**. As we can see from the results, for benchmarks of **SSWP** and **Viterbi**, the speedups range from **5.14**× to **12.64**× demonstrating that sharing delivers substantial additional speedups over batching alone for these benchmarks. For benchmark of **TopkSSSP**, the benefit from sharing is less, but there are still descent speedups of up to **2.16**× due to sharing. On the other hand, for benchmark of **BFS** there is even some slowdown.

The Cost and Benefit of sharing are also shown explaining the above results. The Cost is the time spent in Phase II while Benefit is reduction in total time spent on Phase I + Phase III due to sharing based updates performed by Phase II. Both the Cost and Benefit are presented as fraction of execution times of corresponding **Batch** algorithms. Thus, the Speedups are related to the Cost and Benefit as follows: $Speedup = 1/(1 + Cost - Benefit)$. For **SSWP**, **Viterbi**,

TABLE VII: Factoring Speedups: Batching × Sharing = Total Speedup.

SSWP	Viterbi
TTW	
4.75 × 8.92 = 42.37×	5.13 × 8.47 = 43.42×
TT	
3.58 × 12.26 = 43.96×	3.61 × 12.64 = 45.67×
LJ	
3.44 × 6.43 = 22.11×	4.33 × 5.14 = 22.27×
PK	
3.24 × 8.76 = 28.38×	3.52 × 8.22 = 28.97×

and **TopkSSSP**, the Benefit far exceeds the Cost while for **BFS**, the Cost exceeds the Benefit hence the observed speedup results. Finally, Table VII summarizes how the overall speedups achieved for **SSWP** and **Viterbi** can be *factored* between batching and sharing showing the importance of employing both batching and sharing techniques.

The cost of sharing is reasonable because overheads of sharing come from three sources and all of them are low. First, we need to maintain a counter of the number of visits for each marked high degree vertex in Phase I. This overhead is negligible because we only mark a very small amount of high degree vertices (e.g., 100 out of millions in the current setting) and Phase I is very short (e.g., 1 iteration) and thus has relatively small frontier sizes. Second, we need to solve the shared queries in Phase II. Given that it only computes a small number of shared queries (e.g., only 5 from 100) while the batch size for original queries can be much larger (up to 512), the cost is amortized well across all queries in a batch and thus it has little impact on each individual query. Third, we introduce extra computation cost when applying the result of shared queries to accelerate the convergence of original query. Since this step is a linear scan of the array, it leads to better cache performance due to spatial locality compared with the usual updates for a query which can be randomly scattered across the value array in Ligra. Besides, our sharing algorithm only allows each query to reuse the result of one shared query and only once, keeping the reuse cost low.

To better understand the effectiveness of sharing, we also collected the *stable value* percentages – this is the percentage of vertices reachable from the source vertex whose vertex values converge as a result of performing share updates. We collected this data for the **Batch+Share** configuration. Since we pause the original computation only after the first iteration (i.e., $p = 1$), the percentage of vertices that are stable prior to sharing updates is negligible (less than 0.01%). The percentages of values that are stable following sharing updates are presented in Table VIII. As shown in the table, sharing greatly benefits **SSWP** and **Viterbi** as it causes nearly all the values ($> 99\%$) to converge. To explain the phenomenon that **Top2SSSP** and **Top1SSSP** has lower stable percentage than **BFS** but sharing delivers much more speedups for the former than the latter, we collected the reduction in number of vertex updates resulting from sharing. It turns out that the reduction for **Top2SSSP** and **Top1SSSP** (34%) is much higher than the reduction for **BFS** (7%).

TABLE VIII: Percentage of Vertex Values that become **Stable** due to Sharing Updates.

G	Batch Sizes	SSWP	Viterbi	BFS	Top 2 & 1 SSSP
TTW	4	99.99	99.99	28.95	6.93 - 6.93
	8	99.99	99.99	25.14	7.38 - 7.41
	16	99.99	99.99	22.07	5.21 - 5.25
TT	4	99.99	99.99	23.71	16.65 - 16.78
	8	99.99	99.99	20.57	19.85 - 20.03
	16	99.99	99.99	18.14	13.61 - 13.82
LJ	4	99.99	99.64	7.64	2.21 - 1.03
	8	99.99	99.64	6.56	2.01 - 1.20
	16	99.99	99.64	5.61	2.53 - 1.40
PK	4	99.99	99.63	6.26	0.88 - 1.92
	8	99.99	99.63	6.11	1.01 - 2.03
	16	99.99	99.63	5.38	3.24 - 4.35
Average		**99.99**	**99.80**	**12.87**	**6.10 - 6.18**

D. Sensitivity of Performance to *p* Value

All our preceding experiments were performed for **p** value of **1**, i.e. Phase I lasted one iteration following which Phase II was performed and then the updates from Phase II results optimized the remainder of time spent in Phase III till convergence. We varied the **p** value from 1 to 3 and compared the speedups that were obtained by sharing over batching alone. The results in Table IX show that **p** value 1 delivers best overall speedups and the trend is that speedup falls as **p** value is increased. The only exceptions are LJ::Viterbi and PK::Viterbi where **p** value of 2 slightly outperforms **p** value of 1 (5.48× v.s. 5.14×, 8.57× v.s. 8.22×). There is a performance tradeoff in selecting **p** value. A smaller **p** enables an earlier reuse which leads to earlier convergence of queries. However, if **p** is small, limited number of marked high degree vertices may be visited and considered as candidates for sharing. We conclude the following from this experiment. First, executing Phase I for one iteration is sufficient as high quality SSET nodes have already been encountered. Second,

TABLE IX: Sensitivity to **p** Value: Cost of Phase II, Benefit of Phase II, Speedup of Sharing Over Batching Alone on 256 Queries.

p	SSWP			Viterbi		
	TTW					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.08	0.97	8.92×	0.09	0.97	8.47×
2	0.08	0.81	3.74×	0.07	0.84	4.21×
3	0.08	0.49	1.70×	0.08	0.50	1.72×
	TT					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.06	0.98	12.26×	0.06	0.98	12.64×
2	0.06	0.88	5.61×	0.05	0.89	6.39×
3	0.06	0.57	2.05×	0.06	0.59	2.17×
	LJ					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.12	0.96	6.43×	0.12	0.92	5.14×
2	0.13	0.94	5.26×	0.09	0.91	5.48×
3	0.12	0.88	4.19×	0.10	0.85	3.89×
	PK					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.09	0.98	8.76×	0.08	0.96	8.22×
2	0.09	0.95	7.25×	0.07	0.95	8.57×
3	0.09	0.83	3.91×	0.08	0.87	4.68×

executing Phase II early has the added benefit that greater fraction of overall iterations is optimized by the updates performed from the results of Phase II.

We observe that **p** value of 1 causes sharing to deliver much higher speedups than **p** value of 2 for **SSWP** and **Viterbi** on large graphs than small graphs. For example, for the **TT** graph on **Viterbi** benchmark, the speedup over batching alone for **p** value of 1 is 12.64× while for the second best **p** value of 2, is much smaller 6.36×.

E. Dynamic Selection of SSET

One of the key characteristics of our algorithm is that the vertices in SSET are selected dynamically during the evaluation of a batch of queries. This has two main advantages. First, the selection of SSET vertices is customized to the batch of queries being evaluated. This is important that different batches may contain queries that are close to, in terms of number of hops, different high degree vertices and selection of closer high degree vertices offers greater opportunities of sharing. Second, our technique can be used to speedup the evaluation even when only a single batch of queries is to be evaluated. Note that alternative techniques can be devised to profile executions of batches to identify SSET vertices and then use them to implement sharing in future batches. However, such an approach would lose both of the advantages of our approach mentioned above.

We next confirm that dynamic custom selection of SSET vertices for each batch does indeed lead to selection of different high degree vertices which deliver better speedups. We performed an experiment in which we split 256 queries for the two large graphs TTW and TT into four batches of 64 queries each. We identified the SSET vertices using the first batch and used it to perform sharing in the other three batches. Table X presents batch running time as follows: time using a single dynamically selected SSET vertex for the batch → time using a single dynamically selected SSET vertex in the first batch. The results show that for **TTW::SSWP**, **TTW::Viterbi**, and **TT::SSWP** custom/dynamic selection of SSET vertices for the last three batches delivers better performance (i.e., lower execution times) than the speedups that result from using SSET vertices identified using the first batch. For **TTW::Viterbi** batches 1 and 4 selected the same vertex and hence there is no change in execution time. For **TT::Viterbi** the nodes selected give nearly the same performance.

TABLE X: Changes in Batch Execution Time (seconds): Dynamically Selected → From Other Batch

Graph::Alg.	Batch 2	Batch 3	Batch 4
TTW::SSWP	14.1 → 14.4	12.9 → 14.1	12.3 → 13.3
TTW::Viterbi	15.1 → 16.4	13.4 → 14.5	14.4 → 14.4
TT::SSWP	17.5 → 18.6	16.2 → 17.5	16.2 → 17.3
TT::Viterbi	18.6 → 18.5	17.5 → 17.6	17.1 → 17.3

TABLE XI: Number of Unique Shared Vertices Selected Over Four Batches: Min < Actual < Max

Graph::Alg.	$\|SSET\| = 1$	$\|SSET\| = 3$	$\|SSET\| = 5$
TTW::SSWP	1 <3 <4	3 <7 <12	5 <9 <20
TTW::Viterbi	1 <3 <4	3 <7 <12	5 <9 <20
TT::SSWP	1 <2 <4	3 <8 <12	5 <9 <20
TT::Viterbi	1 <2 <4	3 <8 <12	5 <9 <20

Finally, we examined the identities of selected SSET vertices for various batches to study the diversity of SSET vertices. In Table XI we present actual number of distinct vertices included in SSETs versus the minimum number (size of SSET) and maximum number (number of batches × the size of SSET) of distinct vertices that can be observed. We found that the number of distinct SSET vertices selected are well above the minimum, i.e. during evaluation of different batches often different vertices are selected as SSET vertices.

IV. Related Works

Multi Query Frameworks. Recently, MultiLyra [14] and its extensions in BEAD [15] were developed to simultaneously evaluate a batch of iterative graph queries. There are important differences between the algorithms developed in this paper and MultiLyra/BEAD. First, MultiLyra and BEAD are frameworks for distributed systems and hence its emphasis is on amortizing *communication costs* between machines of a cluster while in this paper we show how batching can be deployed on a single multicore shared-memory machine to amortize overhead costs. Second, we show how to *dynamically* identify shared queries and exploit them to amortize computation costs of queries in a *single batch*. MultiLyra presents a limited algorithm that profiles multiple batches to find *fixed* shared queries that it uses to help speedup future batches. Thus, it cannot be used to speedup a single batch of queries and it cannot select shared queries that are customized to the batch being evaluated. Also in [23] authors show that a batch of BFS queries starting from different source vertices can be simultaneously evaluated efficiently. In [8] authors group vertices into multiple batches to reduce message passing and remote memory access in computing pruned landmark labels. However, they do not exploit sharing. Moreover, both works are aimed at solving a specific application while we present a general system.

Congra [16] schedules a group of concurrent queries to fully utilize the memory bandwidth while preventing contention between different queries. It relies upon offline profiling with different number of threads to determine the scalability and memory bandwidth consumption of different graph algorithms on different input graphs. Multiple queries are processed by creating different processes for different queries where each process has suitable number of threads. This approach thus exploits available system resources fully. In contrast, SimGQ does not require offline profiling but is entirely online, lightweight, and enjoys additional benefits from sharing and batching because it does not use multiple processes. Unlike our sharing of computation across queries, Congra does not exploit shared computations across multiple queries in a batch and thus it does not reduce the amount of computation in terms of number of updates or active vertices scheduled. As for batching, we group the updates from different queries on the same vertex together to achieve better cache performance,

while Congra cannot do so as execution of each query is decoupled from other queries.

The two other recent works that address the problem of evaluating multiple graph queries are Quegel [26] and PnP [27]. However, both these works are aimed at evaluation of point-to-point queries, i.e. queries that compute a property such as shortest path limited to a single source and destination vertex pair. Quegel achieves higher throughput by simultaneous evaluating multiple queries in a pipelined fashion on a distributed system. Essentially a batch of queries is simultaneously evaluated by efficiently sharing memory and computing resources among the queries. PnP [27] is similar to other graph frameworks in that it speedups the evaluation of a single iterative query, using dynamic techniques, independent of other queries evaluated earlier. SimGQ is different from above systems in two ways. It evaluates general queries and not point-to-point. It takes advantages of results computed for a small number of shared queries to speedup all queries. Wonderland [29] supports both point-to-point and general queries, however it does not support sharing.

Graph Databases and Query Systems. There has been a great deal of work on graph based query languages (e.g., Gremlin [20]) and query support in graph databases (e.g., Neo4J and DEX [1], [4]) that enable graph traversals and joins via lower-level graph primitives (e.g., vertices, edges, etc.). However, they are not efficient for iterative graph algorithms over large graphs. For example, although Neo4J supports shortest path queries, as shown in [26], Neo4J runs out of memory for large graphs (e.g., TT used in this paper) and although it can handle small graphs (e.g., LJ used in this paper) it runs extremely slowly taking tens of thousands of seconds in comparison to just few seconds required by our system. Their strength lies in their ability to program wide range of queries especially neighborhood queries [18], [19]. In [30] authors present SPath, an indexing method which leverages decomposed shortest paths around neighborhood of each vertex as basic indexing unit, to accelerate subgraph matching queries. SPath performs very large amounts of precomputation (to enable the optimization) before it can begin to answer queries. In fact the overhead is substantial – comparable to solving a very large number of queries. SimGQ requires no precomputation, rather it identifies shared computation for a batch of queries such that performing the shared computation once leads to net reduction in execution time.

V. Conclusions

We developed techniques for simultaneous evaluation of large batches of iterative graph queries. By employing batching, the overhead costs of query evaluation are amortized across the queries. By employing sharing the cost of computations involving shared queries are amortized across the original batch of queries. Our experiments based upon the state of the art Ligra system yielded speedups ranging from 1.53× to 45.67× across four input graphs and four benchmarks. Both batching and sharing contribute to the substantial speedups.

References

[1] A.B. Ammar. Query Optimization Techniques in Graph Databases. In *Int. J. of Database Management Systems*, Vol.8, No.4, August 2016.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD*, pages 44-54, 2006.

[3] M. Cha, H. Haddadi, F. Benevenuto, and P.K. Gummadi. Measuring user influence in twitter: The million follower fallacy. *Intl. AAAI Conference on Web and Social Media* (ICWSM), 10(10-17):30, 2010.

[4] DEVELOPERS. Neo4J. *Graph NoSQL Database*, 2012.

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

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

[7] G. He, H. Feng, C. Li, and H. Chen. Parallel simrank computation on large graphs with iterative aggregation. In *KDD*, pages 543-552, 2010.

[8] R. Jin, Z. Peng, W. Wu, F. Dragan, G. Agrawal, and B. Ren. Parallelizing pruned landmark labeling: dealing with dependencies in graph algorithms. In *ACM ICS*, Article 11, 1-13, 2020.

[9] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591-600, 2010.

[10] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi : Large-scale graph computation on just a PC. In *USENIX OSDI*, pages 31-46, 2012.

[11] J. Lember, D. Gasbarra, A. Koloydenko, and K. Kuljus. Estimation of Viterbi Path in Bayesian Hidden Markov Models. arXiv:1802.01630, pages 1-27, Feb. 2018.

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endowment 5*, 8 (2012).

[13] G. Malewicz, M.H. Austern, A.J.C Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD Int. Conf. on Management of Data*, 2010.

[14] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable Distributed Evaluation of batches of Iterative Graph Queries. In *IEEE International Conference on Big Data*, pages 349-358, Dec. 2019.

[15] A. Mazloumi, C. Xu, Z. Zhao, and R. Gupta. BEAD: Batched Evaluation of Iterative GraphQueries with Evolving Analytics Demands. In *IEEE International Conference on Big Data*, Dec. 2020.

[16] P. Pan and C. Li. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *IEEE ICCD*, 2017.

[17] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM SOSP*, pages 456-471, 2013.

[18] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. k-Nearest Neighbors in Uncertain Graphs. In *Proc. of the VLDB Endowment*, 2010.

[19] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Analytics on Large Graphs. In *VLDB*, 7(13):1673-1676, 2014.

[20] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Symp. on Database Prog. Languages*, pages 1-10, 2015.

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

[22] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pages 1-6, 2012.

[23] 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. In *Proc. VLDB Endowment*, 2015.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM* (CACM), 33(8):103-111, 1990.

[25] K. Vora, S-C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM. In *SIGPLAN OOPSLA*, pages 861-878, October 2014.

[26] D. Yan, J. Cheng, M.T. Ozsu, F. Yang, Y. Lu, J.C.S. Lui, Q. Zheng and W. Ng. A General-Purpose Query-Centric Framework for Querying Big Graphs. In *Proc. VLDB Endowment*, 9(7):564-575, 2016.

[27] C. Xu, K. Vora, and R. Gupta. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *ACM ASPLOS*, 2019.

[28] Y. Zhang, M.Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. GraphIt: a high-performance graph DSL. In *PACM 2, OOPSLA*, 2018.

[29] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ACM ASPLOS*, pages 608-621, 2018.

[30] P. Zhao and J. Han. On Graph Query Optimization in Large Networks In *Proc. VLDB*, VLDB Endowment, Vol. 3, 1-2, pages 340-351, 2010.