MultiLyra: Scalable Distributed Evaluation of Batches of Iterative Graph Queries

Abbas Mazloumi, Xiaolin Jiang and Rajiv Gupta
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521 USA
{amazl001,xjian049}@ucr.edu, and gupta@cs.ucr.edu

***Abstract*—Graph analytics is being increasingly used for analyzing large scale networks representing entities and relationships in many domains. Various distributed graph processing frameworks have been developed to deliver scalable performance for evaluation of individual iterative graph queries. In practice though, we may need to evaluate many queries. In this paper we develop *MultiLyra*, a distributed framework that efficiently evaluates a batch of graph queries. To deliver high performance, this system is designed to amortize the communication and synchronization costs of distributed query evaluation across multiple queries. Our experiments with *MultiLyra* for four iterative algorithms on a cluster of four 32-core machines show the following. Basic batching technique for amortizing communication and synchronization costs yield maximum speedups ranging from 3.08× to 5.55× across different batch sizes, algorithms and input graphs. After employing optimizations that improve scalability of expensive phases and perform reuse across the distributed computation, the improved maximum speedups range from 7.35× to 11.86×. *MultiLyra* also delivers superior scalabilty than the Quegel batch processing system.**

***Index Terms*—Distributed Graph Processing, Query Batching, Reuse, Redundancy Elimination.**

I. Introduction

Graph analytics is employed in many domains (e.g., social networks [8], web graphs, brain networks etc.) to uncover insights by analyzing high volumes of connected data. It has been observed that real world graphs are often large (Twitter - TT has 2 billion edges and 52.6 million vertices). Also, iterative graph analytics requires repeated passes over the graph till algorithm converges to a stable solution. As a result, in practice, iterative graph analytics workloads are highly data- and/or compute-intensive. Therefore, there has been a great deal of interest in developing efficient graph analytics systems for shared memory (e.g., Galois [11], Ligra [12]) and distributed platforms (e.g., Pregel [10], GraphLab [9], GraphX [5], PowerGraph [1], PowerLyra [2], ASPIRE [14], [15]). Of these, systems that are aimed at distributed computing platforms are the most scalable.

While the performance of graph analytics has improved greatly due to advances in aforementioned 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). However, in practice users can be expected to request solutions to multiple queries (e.g., multiple SSSP queries for different source vertices).

In this paper we address optimized evaluation of a *batch* of queries by amortizing the communication and synchronization costs of distributed evaluation. We present a general graph analytics framework, *MultiLyra*, aimed at evaluating a batch of *vertex queries* for different source vertices of a large graph. For example, for SSSP algorithm, we may be faced with the following batch of queries:

$$\{\mathsf{SSSP}(s_1), \mathsf{SSSP}(s_2), \cdots\cdots \mathsf{SSSP}(s_n)\}.$$

Our *Basic* algorithm maintains a unified list of active vertices and, when considering a single active vertex, it performs integrated processing of all queries in each phase (e.g., Gather, Apply, Scatter) of the distributed computation. Thus, when an active vertex is processed, all of its actions for all the queries in the batch are performed together. This approach leads to amortization of overhead costs across a batch of queries. We identify the least scalable phases of *Basic* and, to overcome their performance limitation, we develop two additional algorithms – *Finished Query Tracking* (*FQT*) and *Inactive Query Tracking* (*IQT*). These algorithms eliminate unnecessary processing associated with completed and inactive queries to improve upon *Basic*. Finally, we incorporate a *Reuse* optimization where results from earlier batches of queries are used to accelerate the execution of later batches of queries.

Experiments with power law graphs and multiple graph algorithms show that *MultiLyra* can accelerate evaluation of queries significantly. The *Basic* batching technique for amortizing communication and synchronization costs yields maximum speedups ranging from 3.08× to 5.55× across different algorithms and input graphs. After further employing optimizations focused at expensive and less scalable phases of the distributed implementation, the improved resulting maximum speedups range from 7.35× to 11.86×. The combination of *IQT* and *Reuse* yields the best overall performance.

Finally we compare the performance of *MultiLyra* with Quegel [17], the only other system that is capable of batched processing of iterative queries. Our results show that *MultiLyra* outperforms Quegel substantially due to its superior scalability.

In Section II we present the detailed design of *MultiLyra* including *Basic*, *FQT* and *IQT*, and *Reuse* algorithms. Section III carries out a detailed evaluation. Concluding remarks are given in Section IV.

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

II. MULTILYRA: BATCHED PROCESSING

During distributed graph processing the input graph is partitioned among the multiple machines and each machine is responsible for carrying out the updates of vertices that reside locally. The machines communicate to exchange needed vertex values and synchronize between iterations before continuing to the next iteration. The combined memories of multiple machines are able to hold large graphs and the large number of cores made available by multiple machines enhance the degree of parallelism delivering scalability.

Since the *PowerLyra* [2] system has the most sophisticated graph partitioning strategy, we build *MultiLyra* by generalizing *PowerLyra* to handle a batch of queries. While *PowerLyra* is based upon bulk-synchronous parallel [13] model of computation, our approach also applies to systems that employ the asynchronous computation model such as Grace [16], [18], Aspire [14], and Coral [15]. The graph partitioning technique guarantees that all incoming edges for the low-in-degree vertices are local to the same machine on which the vertex resides and hence computation is performed locally. For balancing the computation across machines for high-in-degree vertices, incoming edges are distributed across multiple machines. The vertices along the borders of graph partitions are replicated creating *masters* and *mirrors* where the former reside where the partition containing them resides and the latter reside on other machines to which subsets of edges have been distributed.

Since *PowerLyra*, and consequently *MultiLyra*, is based upon *PowerGraph* [1], it employs the GAS (**G**ather-**A**pply-**S**catter) model to divide the distributed computation into phases. The three conceptual phases, namely Gather, Apply, and Scatter, are executed during each iteration in the GAS model. Algorithm 1 shows the five steps in processing a batch of queries in each iteration of *MultiLyra* – we first describe the three main phases (Gather-Batch, Apply-Batch and Scatter-Batch) and then explain the responsibility of the two other phases (Exch-Batch and Recv-Batch). Next we present the details of the phases for our *Basic* batching algorithm that maintains a single *combined active list* such that a vertex is active if it is active for at least one of the queries in the batch being evaluated.

Gather-Batch **-** In the gather phase, all active vertices on each machine collect the required data from their predecessors, in parallel. More specifically, each active vertex goes through all incoming edges from its predecessors to collect vertex and/or edge data as required for all queries in the running batch of the graph algorithm (see Algorithm 2). In this phase, mirrors participate to carry out the task of collecting the remote data. Before data gathering starts, in the *Recv-Batch* of Algorithm 1, the active masters send activation messages to their mirrors and inform them to participate in the gather phase – we refer to this communication as *G-Active*. After both master and mirrors collect the data from their predecessors, mirrors send back their portion of collected data to their master for all the concurrent queries as one message in order to accumulate all the data at the host machine (i.e., the machine where the master resides) for each query in the running batch – we refer to this communication as *G-Data*. So, two messages per replica are needed in this phase for each active vertex.

Apply-Batch **-** The data collected by the gather phase is next used in the apply phase to compute the new vertex

Algorithm 1 The GAS model in MultiLyra

```
1:  function START(k , query_list)
2:      while !query_list.empty() do
3:          batch_qlist ← get next k queries from query_list
4:          unified_active_list ← all q ∈ batch_qlist
5:          while !unified_avtive_list.empty() do
6:              Exch-Batch()
7:              Recv-Batch()
8:              Gather-Batch()
9:              Apply-Batch()
10:             Scatter-Batch()
11:         end while
12:     end while
13: end function
```

Algorithm 2 Gather-Batch in Basic MaltiLyra

```
1:  function GATHER-BATCH
2:      for all master or mirror vertices v ∈ unified_active_list do
3:          for in_edge ∈ v.incoming_edges() do
4:              ▷ gathering data for all queries
5:              for all qid ∈ batch_qlist do
6:                  G-Data[] ← G-Data[] + PredData(qid, in_edge)
7:              end for
8:          end for
9:          gathered_data[v.id] ← G-Data[]
10:         ▷ sending gathered data for all queries
11:         if v is a mirror of a remote master then
12:             Send_G-Data_to_master(v, G-Data[])
13:         end if
14:     end for
15: end function
```

Algorithm 3 Apply-Batch in Basic MultiLyra

```
1:  function APPLY-BATCH
2:      for  master vertex v ∈ unified_active_list do
3:          changed ← false
4:          ▷ computing data for all queries
5:          for all qid ∈ batch_qlist do
6:              new_value ← Compute(gathered_data[v.id], qid)
7:              if Change(v.data[qid], new_value) is ture then
8:                  v.data[qid] ← new_value
9:                  changed ← true              ▷ at least by one query
10:             end if
11:         end for
12:         ▷ update the mirrors and activate them for scatter
13:         if !v.mirrors.empty() && changed then
14:             ▷ sending data for all queries
15:             for all qid ∈ batch_qlist do
16:                 message ← message ∪ v.data[qid]
17:             end for
18:             message ← message ∪ active
19:             Send_A-Mix_to_mirrors(v, message)
20:         end if
21:     end for
22: end function
```

Algorithm 4 Building the unified active list for Basic MaultiLyra

```
1: ▷ done for any master/mirror by Scatter-Batch
2: for any vertex v which got changed do
3:     while s ∈ v.succ() do
4:         s.mark ← true
5:     end while
6: end for
7: ▷ done for any marked mirror by Exch-Batch
8: for any mirror m which m.mark is true do
9:     ▷ mark the remote master
10:    Send_E-Active_to_master(m, active)
11: end for
12: ▷ done for any marked master by Recv-Batch
13: for any master v which v.mark is true do
14:    unified_active_list ← unified_active_list ∪ v
15:    if !v.mirrors.empty() then
16:        Send_G-Active_to_mirrors(v);
17:    end if
18: end for
```

data values for all queries in the executing batch using the *Compute* function for the graph algorithm (see Algorithm 3). To maintain consistency across the machines, when a vertex value is updated by at least one of the queries, the vertex values of all queries in the batch are sent to their mirrors in one aggregated message to affect update. Note that all values must be sent because the combined active list does not maintain the list of queries for which the vertex value was updated. In addition, the updated vertices inform their mirrors to further participate in the scatter phase. This is done by sending an active message along with the vertex data – we refer to this communication as *A-Mix*. In this phase, each active master sends one message for each of it's mirrors including the vertex data for all queries and an activation alert.

After *Apply-Batch*, it is time to generate the active list of the next iteration. Generating the next active list process begins by marking the vertices locally via *Scatter-Batch* in the current iteration, then continues by exchanging active messages in *Exch-Batch*, and finally terminates in *Recv-Batch* by adding the marked vertices into the active list for the next iteration (see Algorithm 4).

Scatter-Batch **-** Any updated vertices which have changed at least for one of the queries during the apply phase (Algorithm 3 – lines 7-10), mark their successors for processing in the next iteration. This is done by the scatter phase in which all the updated vertices go through their outgoing edges in parallel and mark their local successors that can be local masters or local mirrors of remote vertices (lines 1-6 of Algorithm 4). Actually, no communication happens in this phase. As mentioned earlier in the previous phase, updated masters send activation messages to their mirrors in order to inform them to participate in the scatter process (*A-Mix* in *Apply-Batch* contains such a message). This ensures that remote successors will be activated in the next iteration.

Exch-Batch **-** This phase is for actually sending the active messages between machines to build the current iteration's active list. During the scatter phase in previous iteration, all updated active vertices and their mirrors went through their outgoing edges and marked their local neighbors (which can be a master or a mirror) indicating that they must be active for the next iteration for at least one of the queries. Now, in this step, all the local mirrors that were marked during the previous scatter phase, send an activation message to their master which resides on a remote machine to mark and inform it of its selection for the current iteration active list – we refer to this communication as *E-Active* (see lines 7-11 of Algorithm 4). Thus, the active list in each machine for the current iteration is ready to be constructed in the next phase.

Recv-Batch **-** Now in this phase, all those masters which were informed as being marked to be active, whether by local vertices or through an *E-Active* message, are added to the unified active list (see lines 12-18 of Algorithm 4). Further in this phase, as mentioned in *Gather-Batch*, those masters that need their mirrors to participate in the next gather phase will send an activation message (*G-Active*) to their mirrors to activate them for the gather phase.

A. Amortization Effects in Basic in MultiLyra

Previous subsection showed our algorithm for *Basic MultiLyra*. Our *Basic* algorithm maintains a unified list of active vertices and, when considering a single active vertex, it performs integrated processing of all queries in each phase (e.g., Gather, Apply, Scatter) of the distributed computation. Thus, when an active vertex is processed, all of its actions for all the queries in the batch are performed together. This approach leads to amortization of overhead costs across the batch of queries.

In GAS model each iteration makes multiple passes over active vertices, one pass for each phase. Each iteration includes multiple communications per each active vertex which is the major source of overhead. At phase and iteration boundaries, the machines must also synchronize adding to the overhead. Finally, multiple threads running in parallel on each machine must engage in locking and unlocking operations when updating shared data structures. These overheads coming from the distribution and parallelism nature of the distributed frameworks are shared between multiple queries simultaneously by *MultiLyra* instead of executing queries one at a time.

Iteration-Sharing **-** In each iteration, for each of the five phases, each machine loops over all its active vertices. Going over all active vertices, and performing locking and unlocking shared data structures for each active vertex, leads to unavoidable overhead. Moreover, after each phase the machines need to be synchronized with barriers to guarantee that previous phase has completed on all machines in the cluster before continuing to the next phase. Likewise, at the end of each iteration machines need to communicate to synchronize before continuing to the next iteration. In *MultiLyra*, a batch of queries which are running concurrently share iterations together and in each phase when framework makes a pass over the vertices, the work for all concurrent queries is performed and by the end of an iteration all queries advance one iteration toward their final convergence. Hence, the number of passes over the vertices, the number of locking operations for shared

data on a machine for parallel updates of vertices, and the number of barriers between each phases are amortized across the queries in the batch.

Communication **-** For each active vertex ***v*** there are five communications in an iteration which are of two types. Three communications in *Active Category* (i.e. *G-Active*, one included in *A-Mix*, and *E-Active*) and two in the *Data Category* (i.e. *G-Data* and one included in *A-Mix*). The first active message (*G-Active*) is to enable mirrors of ***v*** to do gathering during the gather phase, second active message (included in *A-Mix*) is to enable the mirrors of ***v*** to do scattering during the scatter phase, and the third one (*E-Active*) is to activate the successors of ***v*** for the next iteration. The first message in *Data* category (*G-Data*) is for sending the remote gathered data by the mirrors of ***v*** to the master in order to use them in apply phase, and the second message in this category (included in *A-Mix*) is for updating the mirrors of ***v*** after updating its value in the apply phase. By running a batch of queries simultaneously in *MultiLyra*, communication cost in each category is amortized across the concurrent queries.

Assume *n* queries are running concurrently on *MultiLyra*. Let us consider vertex ***v*** that gets activated for the queries in iteration ***i***. In *Active Category*, each active message mentioned above is sent only once for all queries. No matter how many queries cause the vertex to be active, one single message is enough to activate the vertex. Thus, not only the number of communications but also the amount of communication is amortized across the queries. In *Data Category*, the data messages for all queries are merged together and a single aggregated message is sent. Thus, the cost of communication is amortized across the queries.

Table I shows the number of messages and their size in each category for an active vertex *v* in a specific iteration *i* for the following scenario. Assume vertex *v* is active for *k* queries when a batch of *n* queries (n $\geq$ k) is running on *Basic MultiLyra*, and let us compare it with the baseline *PowerLyra* that runs queries one at a time. As shown in the first two rows of Table I, since the size of communication in *Active category* remains the same, there is reduction in number of communications by a factor of *k*. Thus, *Basic MultiLyra* amortizes the amount of communication in the *Active Category*. For communication in *Data Category*, *MultiLyra* significantly reduces the number of communications but can increase the sizes of messages by up to a factor of *n*.

TABLE I
COMMUNICATIONS FOR AN ACTIVE VERTEX *v* IN A SPECIFIC ITERATION *i*. *n* IS THE BATCH SIZE. *k* IS # OF QUERIES IN WHICH *v* IS ACTIVE. AND *f* IS # OF QUERIES WHICH HAS FINISHED PRIOR TO THE ITERATION *i*.

	#Active	Size	#Data	Size
Baseline	k*3	size_of(Active)	k*2	size_of(Data)
Basic	3	size_of(Active)	2	n*size_of(Data)
FQT	3	size_of(Active)	2	(n-f)*size_of(Data)
IQT	3	size_of(Active)	2	k*size_of(Data)

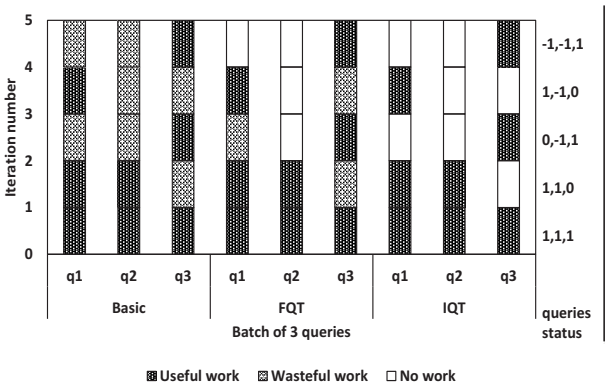

Fig. 1. Amount of wasteful work in five iterations of running a batch of three queries (q1, q2, q3) concurrently for a vertex *v* on MultiLyra versions. Query status 1 or 0 indicate whether vertex *v* is active or inactive for that query in the current iteration and query status -1 indicates that query is finished.

B. FQT and IQT

In *Basic MultiLyra*, each phase acts on behalf of all the queries in the batch for an active vertex. This is because a unified active list is maintained that does not maintain the list of queries for which the vertex has been activated. It also does not know whether some queries have already finished. To improve upon *Basic*, we develop two additional algorithms – *Finished Query Tracking* (*FQT*) and *Inactive Query Tracking* (*IQT*). These algorithms eliminate unnecessary work, both computation and communication, associated with completed queries and inactive vertices for queries respectively. The last two rows of Table I show that the number of communications in *FQT* and *IQT* remains the same when compared with *Basic* while the message size is reduced in *Data Category* where *f* indicates the number of queries in the batch which has finished prior to the iteration. Table II shows the work performed by *FQT* and *IQT* in terms of computation and communication and compares it with *Basic*. Next we explain how the unnecessary work is avoided by keeping track of status of each of the queries.

Assume a batch of three queries q_1, q_2, and q_3 are running on *MultiLyra* for five iterations. Figure 1 shows five running iterations of the three queries for an active vertex *v* indicating whether a query does useful, wasteful, or no work during each iteration for the vertex. Queries q_1 and q_2 finish at the end of fourth and second iterations respectively, and q_3 does not finish in these five iterations. The status of each query is shown in the figure for each iteration. Status 1 or 0 indicates that query is active or inactive for vertex *v* and status -1 indicates that the query has finished. Next, we present the detailed algorithms of *Apply-Batch* and how the active list is constructed for both *IQT* and *FQT*.

Algorithm 5 (a) and (b) show how FQT and IQT reduce the amount of wasteful work both in computation and communication. Unlike *Basic*, FQT has a loop only over the unfinished queries in the running batch to do the actions by using *unfinished_qlist* in lines 5 and 17. FQT keeps track of each unfinished query which has at least one changed vertex

TABLE II
DIFFERENT VERSIONS OF MULTILYRA IMPLEMENTATION. (Q: NUMBER OF SIMULTANEOUS QUERIES; V: NUMBER OF VERTICES)

Version	Description	Active List Size	Communication and Computation
Basic	No query related tracking performed	V	Both performed for All Queries
FQT	Performs tracking of Unfinished Queries	V+Q	Both performed only for Unfinished Queries
IQT	Performs tracking of Active Queries for each Active Vertex	V*Q	Both performed only for Active Queries of each Active Vertex

Algorithm 5 Modification in Apply-Batch algorithm for FQT (a) and IQT (b)

```
1: function APPLY-BATCH
2:     for master vertex v ∈ unified_active_list do
3:         changed ← false
4:         ▷ computing data only for unfinished queries
5:         for qid ∈ unfinished_qlist do
6:             new_value ← Compute(gathered_data[v.id], qid)
7:             if Change(v.data[qid], new_value) is ture then
8:                 v.data[qid] ← new_value
9:                 changed ← true ▷ at least by an unfinished query
10:                ▷ deposit qid for building next unfinished_qlist
11:                Deposit_Unfq(qid)
12:            end if
13:        end for
14:        ▷ update the mirrors and activate them for scatter
15:        if !v.mirrors.empty() && changed then
16:            ▷ sending data only for unfinished queries
17:            for all qid ∈ unfinished_qlist do
18:                message ← message ∪ v.data[qid]
19:            end for
20:            message ← message ∪ active
21:            Send_A-Mix_to_mirrors(v, message)
22:        end if
23:    end for
24: end function
25:                    (a)
```

```
1: function APPLY-BATCH
2:     for master vertex v ∈ unified_active_list do
3:         changed ← false
4:         ▷ computing only for those queries in which v is active
5:         for all qid ∈ active_qlist[v.id] do
6:             new_value ← Compute(gathered_data[v.id], qid)
7:             if Change(v.data[qid], new_value) is ture then
8:                 v.data[qid] ← new_value
9:                 changed ← true
10:                ▷ deposit qid for building next active_qlist[v.id]
11:                Deposit_vidqid(v.id, qid)
12:            end if
13:        end for
14:        ▷ update the mirrors and activate them for scatter
15:        if !v.mirrors.empty() && changed then
16:            ▷ sending only for those queries in which v is active
17:            for all qid ∈ active_qlist[v.id] do
18:                message ← message ∪ v.data[qid]
19:            end for
20:            message ← message ∪ active
21:            Send_A-Mix_to_mirrors(v, message)
22:        end if
23:    end for
24: end function
25:                    (b)
```

by depositing its id (*qid* in line 11) to use later for building the next *unfinished_qlist* for the next iteration. Note, FQT in *Gather-batch* also uses *unfinished_qlist* for collecting and sending data instead of all queries (using *unfinished_qlist* on line 5 of Algorithm 2 instead of *batch_qlist*). Algorithm 6 (a) on line 15 shows when *unfinished_qlist* is constructed. The following line makes sure that all machines in the cluster are aware of this list before continuing to the *Gather* phase. This requires exchange of only *n* bits as one single bit is set in each iteration while *n* is the number of queries in the batch. Each bit indicates whether the corresponding query has finished or not. IQT in Algorithm 5 (b) reduces the unnecessary work similarly; however, by using a list of active queries for each single vertex (*active_qlist[v.id]* at lines 5 and 17), it ensures that computation and communication is performed only for queries for which vertex *v* has been activated. Note, IQT uses *active_qlist[v.id]* also for collecting and sending data in *Gather_Batch* to ensure no wasteful work is done. In line 11 of Algorithm 5 (b), IQT keeps track of active queries for each vertex by depositing a bit-set of active queries for each vertex. Later in *Exch-Batch*, those local mirrors that need to send *E-Active* to their master to mark them for next iteration must also send the tracking information to the remote machine at which master resides. It requires addition of *n* bits to the end of *E-Active* message (see Algorithm 6 (b), line 11). Finally, now all the machines have the tracking information, the *active_qlist* is constructed (line 16 of Algorithm 6 (b)).

C. Reuse Optimization

This section describes the details of our online *Reuse* optimization on top of IQT. *Reuse* includes three steps. First, it extracts top high-centrality vertices during the run of the first batch. As the second step, it picks the top five to run a batch of five queries of the graph algorithm and maintains the results distributed on each machine (each machine maintains the results for it's local vertices). Then in the third step, we *Reuse* results of these five queries during the run of remaining batches to accelerate the convergence of each query. Hence, the remaining batches of queries will take advantage of the superior speedup offered by *Reuse*. To do this, we added a new phase named *Reuse-Batch* to the GAS model of *MultyLira* between *Apply-Batch* and *Scatter-Batch* phases to perform reuse for the remaining batches.

Algorithm 6 Modification in algorithm of building the active list for FQT (a) and IQT (b)

```
1: ▷ done for any master/mirror by Scatter-Batch
2: for any vertex v which got changed do
3:     while s ∈ v.succ() do
4:         s.mark ← true
5:     end while
6: end for
7: ▷ done for any marked mirror by Exch-Batch
8: for for any mirror m which m.mark is true do
9:     ▷ mark the remote master
10:    Send_E-Active_to_master(m, active)
11: end for
12: ▷ done for any marked master by Recv-Batch
13: for for any master v which v.mark is true do
14:    unified_active_list ← unified_active_list ∪ v
15:    unfinished_qlist ← Withdraw_qid()
16:    unfinished_qlist.sync()          ▷ exchange one single bitset
17:    if !v.mirrors.empty() then
18:        Send_G-Active_to_mirrors(v);
19:    end if
20: end for
21:                (a)
```

```
1: ▷ done for any master/mirror by Scatter-Batch
2: for any vertex v which got changed do
3:     while s ∈ v.succ() do
4:         s.mark ← true
5:     end while
6: end for
7: ▷ done for any marked mirror by Exch-Batch
8: for for any mirror m which m.mark is true do
9:     ▷ mark the remote master
10:    activeq_bitset ← Withdraw_vidqid(v.id)
11:    Send_E-Active_to_master(m, active + activeq_bitset)
12: end for
13: ▷ done for any marked master by Recv-Batch
14: for for any master v which v.mark is true do
15:    unified_active_list ← unified_active_list ∪ v
16:    active_qlist[v.id] ← activeq_bitset;
17:    if !v.mirrors.empty() then
18:        Send_G-Active_to_mirrors(v);
19:    end if
20: end for
21:                (b)
```

SSSP

$v.data[qid]$=Min($v.data[qid]$, $v.data_{hc}[v_{hc}.id]$+$v_{hc}.data[qid]$)

SSWP

$v.data[qid]$=Max($v.data[qid]$, Min($v.data_{hc}[v_{hc}.id]$,$v_{hc}.data[qid]$))

Viterbi

$v.data[qid]$=Max($v.data[qid]$, $v.data_{hc}[v_{hc}.id]$*$v_{hc}.data[qid]$)

Fig. 2. Reuse equations to update vertices in *Reuse-Batch*.

Reuse-Batch - When one of the extracted high-centrality vertices, v_{hc}, becomes active in an iteration then after *Apply-Batch* computes the intermediate results for v_{hc} on the host machine, m_h, the process of reuse in *Reuse-Batch* starts. In this phase, each machine iterates over all its local vertices and updates their current values towards faster convergence for those queries for which v_{hc} has been activate, *qid*. To do this two pieces of data are required, the final result of v_{hc} when it was the source vertex of the query that was computed and is being maintained on all machines, i.e. $v.data_{hc}[]$, and the current value of v_{hc}, i.e. $v_{hc}.data[]$. Since m_h has the current value, it is responsible for sending the value to other machines to ensure all the machines have required data for reuse process. Note, there is no need to send intermediate data to machines on which a mirror of v_{hc} exists since it has already been sent by *Apply-Batch*. Figure 2 shows the reuse equations for SSSP, SSWP and Viterbi [7].

III. EXPERIMENTS

TABLE III
ITERATIVE GRAPH ALGORITHMS.

Algorithm	Message Data Type
Single Source Shortest Path (SSSP)	Unsigned int
Single Source Widest Path (SSWP)	Unsigned int
Number of Paths (NP)	Unsigned int
Viterbi (VT) [7]	Float

TABLE IV
REAL WORLD INPUT GRAPHS.

Input Graph	#Edges	#Vertices	#Queries
Twitter (TT) [3], [6]	2.0B	52.6M	1K
LiveJournal (LJ) [4], [8]	69M	4.8M	1K

Experimental Setup - For this work we implemented our framework using *PowerLyra* [2] which improves upon *PowerGraph* [1] via its hybrid partitioning method. In our evaluation we consider four algorithms - Single Source Shortest Path (SSSP), Single Source Widest Path (SSWP), Number of Paths (NP), and Viterbi (VT) [7] (see Table III). We use two input graphs listed in Table IV - one is billion edge graph (TT) and one has tens of millions of edges (LJ). For each input graph and for each algorithm, we generated 1024 queries. The sources are unique and were selected randomly. All experiments were performed on a cluster of four identical machines. Each machine has 32 Intel Broadwell cores, 256 GB memory, and runs CentOS Linux release 7.4.1708.

We evaluate all the versions of batching discussed, namely *Basic*, *FQT*, and *IQT*. We also implemented our *Reuse* algorithm and performed its evaluation. Next we present our experimental results.

A. Basic Batching

We ran 1024 queries for each input graph and algorithm on the *Basic* version of *MultiLyra* for varying batch sizes and compared their execution times with those of the baseline *PowerLyra* framework that evaluates queries one at a time. Table V shows the execution time, and the percentage spent in each phase, for the baseline by running all the 1024 queries. We use these times to compute speedups obtained by our algorithms. In *Basic*, for each batch size k we ran 1024 queries by dividing them into multiple batches of size k.

TABLE V
RUNNING 1024 QUERIES ON THE NO-BATCHING BASELINE FRAMEWORK (POWERLYRA).

G	Algorithm	Exch-msg	Recv-msg	Gather	Apply	Scatter	Sync	Execution Time (s)
TT	SSSP	8.84%	8.57%	23.77%	36.42%	19.82%	2.57%	37,464
	SSWP	9.38%	9.41%	27.56%	31.37%	19.11%	3.16%	51,331
	NP	12.87%	9.87%	22.94%	31.57%	20.63%	2.11%	38,046
	VT	8.94%	8.67%	23.77%	36.15%	19.74%	2.73%	36246
LJ	SSSP	8.97%	11.24%	14.53%	40.47%	11.85%	12.94%	12,125
	SSWP	10.53%	12.09%	15.59%	35.95%	10.75%	15.10%	10,200
	NP	9.95%	11.62%	14.10%	38.62%	14.04%	11.67%	7,648
	VT	9.59%	11.60%	14.89%	38.40%	11.27%	14.25%	14287

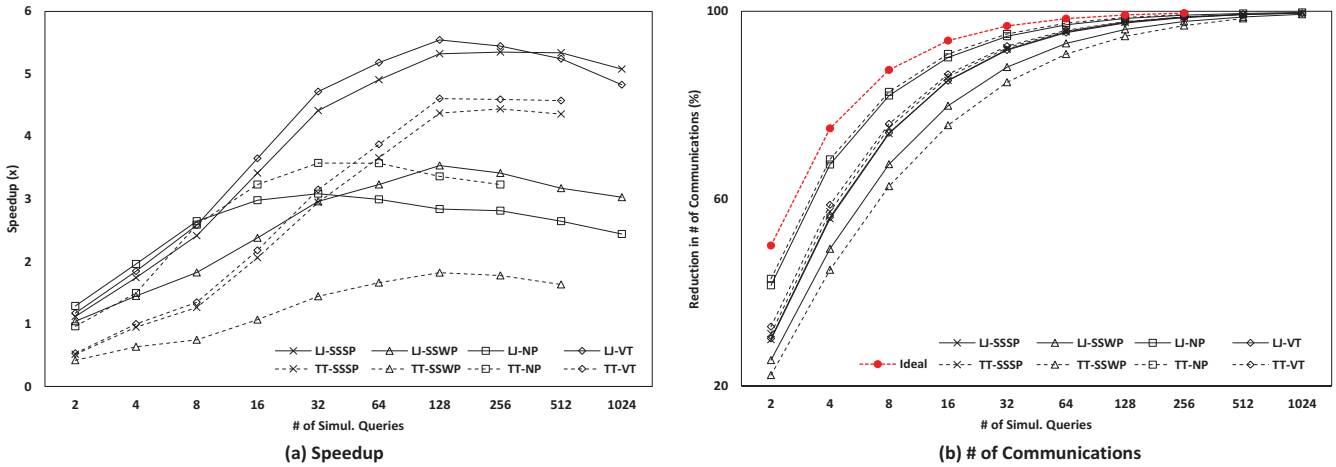

Fig. 3. Speedup (a) and the percentage of reduction in number of communications (b) when running 1K queries with different batch sizes for four algorithms(SSSP, SSWP, NP and VT) on two input graphs, Twitter (TT) and LiveJournal (LJ)

The speedups obtained by *Basic* version of *MultiLyra* are shown in Figure 3(a) as the batch size is varied from 2 to 1024. The *Basic* version delivers maximum speedups ranging from 1.82× for *SSWP* to 4.61× for *Viterbi* on Twitter and from 3.08× for *NP* to 5.55× for *Viterbi* on LiveJournal.

The most dominant performance obstacle in distributed graph processing is the number of communications needed by nodes in the cluster to exchange remote data for an iterative algorithm. Figure 3(b) shows the percentage of reduction in the number of communications when the batch size is varied for *Basic MultiLyra*. In this figure, the line in red shows the hypothetical ideal reduction achievable when the number of communications is reduced by a factor equal to the batch size. As we can see, reductions in number of communications achieved by *Basic* are not far from the ideal with most reduction observed in *NP* and the least reduction in *SSWP*.

From the data shown in Figure 3(a) we observe that initially the speedup increases with batch size because the number of communications between the machines in the cluster reduces. Then, at some point the speedup reaches its peak and then begins to slightly fall. To study this behavior, we collected the times spent in each of the five phases described earlier with the batch size that delivered the most speedup on average for each algorithm. Table VI shows the percentage of total execution time spent in each phase. Moreover, we collected the time spent in each step for every algorithm on every input graph when the number of simultaneous queries varies from 2 to 1024. A representative set of data is shown in Table VII for the *SSSP* algorithm on both input graphs. From this data we observe that the *Apply* phase takes more than half of the execution time, 54% on average and it increases as the batch size grows from 2 to 1024. Thus, *Apply* phase does not scale well with batch size. Moreover, although *Gather* and *Scatter* phases scale, they together still account for substantial part of the execution cost.

This limited scalability of these phases can be understood as follows. In *Basic* version of *MultiLyra* framework, all queries in the batch (say n queries) use only one active list. Hence, when a vertex becomes active, no matter due to which query, the gather function in *Gather* phase will collect the data needed for computation of all n queries and the update function in *Apply* phase will do the computation for all of these queries. The communications in *Data* Category also need to send the data for all the queries both in *Gather* and *Apply* phases.

Furthermore, let us consider Table VIII that shows the percentage of the total number of communications in *Data* category in each *Gather* and *Apply* phases. It shows that, on average, more than 82% of the total communications in *Data* category are communicated in *Apply* phase and also the last column shows that on average more than 75.9% of all communications are in *Data* category. Thus, only by improving communications in *Data* category can we help to improve the scalability of the *Apply* phase.

TABLE VI
PERCENTAGE OF TOTAL EXECUTION TIME SPENT IN EACH STEP FOR SELECTED BATCH SIZES USING BASIC MULTILYRA

G	Algorithm	#Batch	Exch-msg	Recv-msg	Gather	Apply	Scatter	Sync	Speedup
TT	SSSP	256	1.50%	0.41%	21.83%	53.75%	19.33%	3.17%	4.44×
	SSWP	128	1.31%	0.98%	38.00%	36.20%	20.03%	3.48%	1.82×
	NP	32	2.12%	1.44%	18.95%	54.28%	20.88%	2.32%	3.57×
	VT	128	1.34%	0.70%	23.67%	50.30%	20.63%	3.35%	4.61×
LJ	SSSP	256	1.24%	0.50%	9.72%	65.82%	5.06%	17.66%	5.35×
	SSWP	128	1.50%	1.16%	13.52%	51.03%	5.66%	27.13%	3.53×
	NP	32	3.00%	2.70%	9.26%	57.43%	5.61%	22.00%	3.08×
	VT	128	1.24%	0.86%	9.42%	63.16%	5.48%	19.84%	5.55×

TABLE VII
PERCENTAGE OF TOTAL EXECUTION TIME SPENT IN EACH STEP FOR SSSP ON THE INPUT GRAPHS WHEN THE BATCH SIZE VARIES.

G	#Batch	Exch-msg	Recv-msg	Gather	Apply	Scatter	Sync	Speedup
TT	2	3.18%	2.76%	27.50%	19.82%	45.97%	0.76%	0.51×
	4	3.72%	2.99%	30.87%	21.50%	39.96%	0.96%	0.95×
	8	2.75%	2.30%	27.69%	20.72%	45.56%	0.97%	1.26×
	16	2.63%	1.98%	28.32%	27.00%	38.68%	1.38%	2.06×
	32	2.18%	1.56%	25.87%	35.06%	33.50%	1.84%	2.94×
	64	1.30%	1.08%	24.65%	42.01%	28.58%	2.36%	3.66×
	128	1.25%	0.71%	24.27%	50.34%	20.50%	2.93%	4.37×
	256	1.50%	0.41%	21.83%	53.75%	19.33%	3.17%	4.44×
	512	1.39%	0.33%	20.66%	56.77%	17.84%	3.01%	4.36×
LJ	2	5.64%	7.19%	18.37%	37.56%	23.10%	8.14%	1.12×
	4	4.95%	6.27%	20.46%	37.81%	22.30%	8.20%	1.74×
	8	3.95%	4.75%	17.48%	40.88%	24.17%	8.77%	2.41×
	16	3.25%	3.62%	15.07%	48.32%	19.19%	10.55%	3.42×
	32	2.56%	2.43%	12.02%	54.89%	15.75%	12.34%	4.41×
	64	1.86%	1.37%	10.42%	59.67%	12.52%	14.16%	4.91×
	128	1.52%	0.82%	9.92%	66.79%	5.58%	15.37%	5.32×
	256	1.24%	0.50%	9.72%	65.82%	5.06%	17.66%	5.35×
	512	0.70%	0.36%	9.64%	66.79%	4.47%	18.04%	5.34×
	1024	0.61%	0.45%	9.35%	68.63%	4.29%	16.66%	5.08×

In summary, based upon the above experiments with *Basic*, we make two key observations. First, among all the phases the *Apply* phase is the least scalable phase and it is responsible for 82% of *Data* communications on average; hence, it consumes more than half of the execution time. Second, *Apply* phase does not scale due to extra communications and computations that can be removed by keeping track of status of the queries that are running concurrently. This leads us to implementions of the two versions of *MultiLyra* framework named *FQT* and *IQT* described next.

B. FQT and IQT

To improve the *Basic* version, we studied the status of the queries. To do this, we ran 1024 queries for each input graph for all four algorithms and collected the finish time for each query. Table IX shows the percentage of the total execution time during which some queries have finished and they are waiting for other queries to finish. The waiting time ranges from around 36% for SSWP to 2.3% for SSSP on average for the two input graphs. *Finished Query Tracking (FQT)* version of *MultiLyra* framework utilizes this opportunity.

We repeated our experiment, running 1024 queries with the selected batch sizes, to study the speedup of *FQT* and compared it with *Basic* version, shown in the related rows of Table X. As we expected from Table IX, *SSWP* takes advantage of this version since it has 36% waiting time on average for each finished query. But all other algorithms could not utilize *FQT* due to their very small waiting times. Note, in case of *NP*, removing less than 10% waiting time on average was not enough to overcome the overhead of *FQT*. As described in the previous section, *FQT* improves Basic by not performing the required actions (i.e. computation and communication) for already-finished queries. Since only *SSWP* among the four algorithms had the long waiting time to offer, it only could gain speedup from *FQT*.

FQT just knows whether a query is already finished or not. To further improve over *FQT*, we implemented *Inactive Query Tracking (IQT)* which kept track of the current active queries for each vertex in an iteration. We repeated the same experiment for *IQT*. Table X compares the speedup of *IQT* over the baseline with speedups of other versions, i.e. *FQT* and *Basic*. As we can see, *IQT* improves the speedup of the algorithms substantially since *IQT*, as described earlier, can remove the extra computations and communications not only for finished queries but also for the inactive unfinished queries per each active vertex in an iteration.

Among the four algorithms, *NP* is the only algorithm where *FQT* and *IQT* do not deliver speedups. This is due to the nature of *NP* where the vertex values always increase till they hit a set upper limit and converge. Consequently, most of the queries are active in each iteration for active vertices and few opportunities exist for *FQT* and *IQT* to exploit.

TABLE VIII
PERCENTAGE OF THE NUMBER OF COMMUNICATION IN DATA CATEGORY NEEDED IN GATHER AND APPLY PHASES.

G	Algo.	#Batch	Gather	Apply	#Data Comm. (% of Total)
TT	SSSP	256	19.70%	80.30%	2.34 $\times 10^9$ (73.26%)
	SSWP	128	31.55%	68.45%	12.04 $\times 10^9$ (66.08%)
	NP	32	22.10%	77.90%	11.71 $\times 10^9$ (70.25%)
	VT	128	19.16%	80.84%	4.02 $\times 10^9$ (73.52%)
LJ	SSSP	256	9.97%	90.03%	0.69 $\times 10^9$ (85.03%)
	SSWP	128	16.45%	83.55%	1.39 $\times 10^9$ (71.61%)
	NP	32	10.96%	89.04%	2.29 $\times 10^9$ (82.52%)
	VT	128	9.53%	90.47%	1.41 $\times 10^9$ (85.49%)

TABLE IX
THE PERCENTAGE OF TIME ON AVERAGE FOR WHICH A COMPLETED QUERY PERFORMS WASTEFUL PROCESSING.

G	Algo.	#Batch	Waiting Time (%)
TT	SSSP	256	0.66%
	SSWP	128	39.79%
	NP	32	0.89%
	VT	128	0.76%
LJ	SSSP	256	4.00%
	SSWP	128	32.10%
	NP	32	17.61%
	VT	128	5.99%

TABLE X
IQT AND FQT IN DETAILS FOR THE SELECTED BATCH SIZES AND THEIR SPEEDUP OVER THE BASELINE (TABLE V).

G	Algo.	#Batch	V	Exch-msg	Recv-msg	Gather	Apply	Scatter	Sync	Speedup	Basic
TT	SSSP	256	FQT	1.19%	0.39%	18.89%	51.53%	25.05%	2.95%	4.26×	4.44×
			IQT	1.52%	2.74%	32.66%	31.56%	27.38%	4.14%	6.45×	
	SSWP	128	FQT	1.65%	1.18%	35.92%	31.99%	25.00%	4.25%	2.30×	1.82×
			IQT	1.95%	3.20%	45.18%	16.94%	28.06%	4.67%	2.65×	
	NP	32	FQT	1.88%	1.28%	27.87%	48.33%	18.61%	2.03%	3.17×	3.57×
			IQT	2.37%	1.97%	30.18%	47.00%	16.62%	1.85%	2.90×	
	VT	128	FQT	1.23%	0.68%	21.98%	49.48%	23.25%	3.38%	4.52×	4.61×
			IQT	1.83%	2.41%	34.14%	31.94%	25.62%	4.05%	5.98×	
LJ	SSSP	256	FQT	1.13%	0.49%	8.84%	64.72%	8.23%	16.58%	5.25×	5.35×
			IQT	1.80%	1.97%	16.17%	43.14%	8.32%	28.60%	9.14×	
	SSWP	128	FQT	1.67%	1.24%	14.17%	47.81%	6.87%	28.23%	3.91×	3.53×
			IQT	1.90%	3.95%	19.91%	25.84%	8.26%	40.14%	5.39×	
	NP	32	FQT	2.99%	2.67%	10.41%	56.80%	5.40%	21.72%	3.03×	3.08×
			IQT	2.83%	3.49%	10.33%	57.28%	5.08%	20.97%	2.89×	
	VT	128	FQT	1.37%	0.86%	8.83%	63.23%	6.04%	19.68%	5.47×	5.55×
			IQT	1.58%	4.15%	15.09%	41.25%	8.22%	29.70%	8.92×	

C. IQT + Reuse

This subsection presents experimental results of our *Online Reuse* technique on top of *IQT* evaluated in the previous subsection. To enable reuse, when the first batch of queries, are executed using *IQT*, the top five high-centrality vertices are identified based upon the number of updates and scatters they experience. Prior to running the remaining batches of queries, we generated an extra small batch of queries using the five extracted vertices and ran it on *IQT* to store their result in order to reuse them during the run of remaining queries. Hence, the remaining batches of queries took advantage of the superior performance offered by *Reuse*.

Figure 4 compares *Reuse* speedup with *FQT* and *IQT*. Using *Reuse* optimization on top of *IQT* gives 8.04× and 11.06× speedups on average across the different graph algorithms on the input graphs Twitter (TT) and LiveJournal (LJ) respectively for running 1024+5 queries – more detailed data is given in Table XI. Please note that the data for *Reuse* is presented for three algorithms where *Reuse* optimization can be safely applied; the fourth algorithm is omitted because application of *Reuse* to *NP* is unsafe.

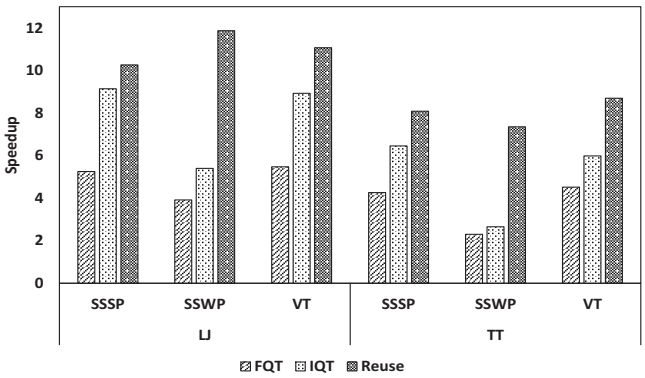

Fig. 4. Reuse vs. FQT and IQT

D. IQT Batching Vs. Quegel Batching

Quegel [17] is the only other graph processing system that has been designed to simultaneously evaluate a batch of iterative graph queries. By sharing computing and memory resources across multiple queries whose evaluation is overlapped via pipelining, *Quegel* optimizes the evaluation of a batch of queries. Since it does not integrate the data messages, the number of communications remain the same. While its focus is on evaluating point-to-point queries [19] (e.g., shortest path from vertex v to vertex w), it can be easily adapted to evaluate point-to-all queries (e.g., SSSP). We carried out this adaptation and then compared the performance of *Quegel* batching with our *IQT* batching.

The speedups obtained by *IQT* batching over *Quegel* batching are given in Table XII for different batch sizes. As we can see, the speedups of *IQT* over *Quegel* increase as batch

TABLE XI
PERCENTAGE OF TOTAL EXECUTION TIME SPENT IN EACH STEP FOR SELECTED BATCH SIZES USING IQT + REUSE AND COMPARING THE SPEEDUP WITH IQT ALONE.

G	Algorithm	#Batch	Exch-msg	Recv-msg	Gather	Apply	Reuse	Scatter	Sync	IQT+Reuse	IQT
TT	SSSP	256	1.84%	2.60%	37.59%	23.45%	3.26%	26.81%	5.26%	8.08×	6.45×
	SSWP	128	2.05%	3.09%	44.98%	11.61%	2.90%	25.92%	9.46%	7.35×	2.65×
	VT	128	1.90%	2.29%	37.64%	22.11%	3.40%	26.78%	6.31%	8.69×	5.98×
LJ	SSSP	256	1.56%	1.91%	17.97%	33.72%	2.95%	7.94%	33.94%	10.26×	9.14×
	SSWP	128	2.02%	3.33%	18.38%	16.92%	3.70%	7.54%	48.10%	11.86×	5.39×
	VT	128	1.71%	3.86%	17.03%	31.54%	2.99%	8.58%	34.29%	11.07×	8.92×

TABLE XII
SPEEDUPS OF IQT BATCHING OVER QUEGEL BATCHING ON A FOUR MACHINE CLUSTER.

G	Algo.	#Batch	IQT Speedup
LJ	SSSP	16	1.97×
		32	2.83×
		64	3.89×
	SSWP	16	1.61×
		32	2.75×
		64	4.34×
	NP	16	3.38×
		32	4.23×
		64	4.61×
	VT	16	2.29×
		32	3.86×
		64	5.33×

size is increased. This is expected, as Quegel's performance remains steady with batch size while *IQT* has been designed so phases of *MultiLyra* scale in performance with batch size giving improved performance.

IV. CONCLUSION

In this paper we presented the *MultiLyra* system, a generalization of the *PowerLyra* system to enable efficiently evaluation of a batch of iterative graph queries. *MultiLyra*'s query evaluation methodology (*Basic*) and added optimizations (*IQT* and *Reuse*) yield significant speedups. By amortizing the communication, synchronization and computation costs across multiple queries, MultiLyra delivers maximum speedups ranging from 7.35× to 11.86× across four iterative graph algorithms and multiple input graphs on a cluster of four 32-core machines. Finally, scalability of batching supported by *MultiLyra* is far superior to that of *Quegel*.

ACKNOWLEDGEMENTS

This work is supported by NSF grants CCF-1813173 and CCF-1524852 to the University of California Riverside.

REFERENCES

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," In Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12) (pp. 17-30), 2012.

[2] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," ACM Transactions on Parallel Computing (TOPC), 5(3), 13, 2019.

[3] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in twitter: The million follower fallacy," In fourth international AAAI conference on weblogs and social media, 2010.

[4] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 44-54, 2006.

[5] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *USENIX Symp. on Operating Systems Design and Implementation* (OSDI), pages 599-613, 2014.

[6] 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.

[7] 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.

[8] J. Leskovec. Stanford large network dataset collection. *http://snap.stanford.edu/data/index.html*, 2011.

[9] 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. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716-727.

[10] 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 International Conference on Management of Data*, pages 135-146, 2010.

[11] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles* (SOSP), pages 456-471, 2013.

[12] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 135-146, 2013.

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

[14] K. Vora, S-C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM. In *SIGPLAN International Conf. on Object Oriented Programming Systems, Languages and Applications* (OOPSLA), pages 861-878, October 2014.

[15] K. Vora, C. Tian, R. Gupta, and Z. Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 223-236, April 2017.

[16] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *Conference on Innovative Data Systems Research* (CIDR), pages 3-6, 2013.

[17] 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 *Proceedings of the VLDB Endowment*, Vol. 9, No. 7, pages 564-575, 2016.

[18] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 194-204, 2015.

[19] C. Xu, K. Vora, and R. Gupta. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *ACM 24nd International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 587-600, 2019.