Expressway: Prioritizing Edges for Distributed Evaluation of Graph Queries

Abbas Mazloumi
Computer Science and Engineering
University of California, Riverside
Riverside, USA
amazl001@ucr.edu

Mahbod Afarin
Computer Science and Engineering
University of California, Riverside
Riverside, USA
mafar001@ucr.edu

Rajiv Gupta
Computer Science and Engineering
University of California, Riverside
Riverside, USA
rajivg@ucr.edu

Abstract—Distributed Graph analytics is being widely used in various domains for analyzing large real-world graphs. There have been numerous efforts to build distributed frameworks for graph analytics aimed at improving scalability. These frameworks enable the processing of huge graphs that do not fit in the memory of a single machine by imposing message-passing overhead among a cluster of multiple machines, underutilizing the available computing resources. To mitigate this, we present *Expressway*, a technique to identify important edges, i.e., *Highways*, that play a key role in delivering the results for their boundary vertices. *Expressway* first runs the queries using only *Highways*, reducing the number of edges that needed to be processed during the execution of a graph query significantly. Thus, it can be accomplished in each machine separately in the cluster, avoiding the message-passing overheads. Then *Expressway* takes the results from running the query on *Highways* and initializes the vertices to these values, enabling faster convergence of graph algorithms. Our experiments show applying *Expressway* on the state-of-the-art frameworks results in up to 4.08× speedup over the single-query framework and up to 4.04× speedup over the framework to run a batch of concurrent graph queries.

Index Terms—Distributed Graph Processing, Iterative Queries, Faster Convergence, Prioritized Edges.

I. INTRODUCTION

Graph analytics has been focused on in both academia and industry due to its ability to extract valuable insights from high volumes of connected data by iteratively traversing large real-world graphs. Various domains such as social networks [9], web graphs, etc., benefit from graph analytics algorithms. These iterative graph analytics require repetitive traversals of the graph until the algorithm converges to a stable solution demanding a significant amount of computational resources. In addition, the size and irregularity of real-world graphs, such as those seen in social networks and web graphs, provide difficulties for graph analytics workloads.

Therefore, this has led to a great deal of interest in developing efficient graph analytics systems for shared memory (e.g., Galois [12], Ligra [13]), GPUs, and custom accelerators [30] [31] [32] as well as platforms in the distributed environment (e.g., Pregel [11], GraphLab [10], GraphX [6], PowerGraph [1], PowerLyra [2], ASPIRE [24]). Among these, systems that are aimed at distributed computing platforms are the most scalable. In addition, there have been also some recent works focusing on improving the throughput of these systems by evaluating multiple simultaneous queries at once and amortizing the existing overheads across multiple queries both in shared and distributed environments (e.g. Quegel [27], MultiLyra [16], [17], SimGQ [19], [20]).

While most of the existing works are focused on making the platform itself efficient and scalable, one can focus on the input graph and the running algorithm looking for opportunities to enhance the computation load. We have observed that when running graph queries using a specific algorithm, the contribution of certain edges is crucial for achieving convergence in their boundary vertices. These edges play a vital role in delivering the converged results to their connected vertices.

In this paper, we present *Expressway*, a technique to further improve the efficiency of distributed graph frameworks by prioritizing important edges of an input graph. First, we begin by identifying the most important edges in the graph, which we refer to as "highways". *Highways* contribute to the accurate calculation of property values of a significant number of vertices. Therefore, by running the algorithm on the graph using only these *highways*, we can obtain precise property values for most of the vertices. After this initial run, we execute the algorithm on the graph using all the edges to obtain precise values for all the vertices. This technique offers a significant speedup. As our experiments show, the *highways* comprise only a small subset of the graph's edges. Running the graph initially with just these *highways* is much faster because it involves a smaller subset of edges. The second step is also so fast because most of the vertices already have precise values, allowing for rapid convergence. By employing the Expressway technique, we can achieve up to 4.08× speedup compared to a single-query framework and up to a 4.04× speedup compared to a framework designed for a batch of concurrent queries.

The key contributions of this paper are as follows:

- Our study demonstrates that we can obtain precise results for most of the vertices with ease by using only a small subset of the edges, i.e., *highways*.
- We introduce a novel algorithm for identifying *highways* in a given graph.
- We enhance the performance of distributed graph query evaluation through a two-step algorithm. The algorithm first runs on the graph using only the *highways*, and then it runs on the graph using all the edges.
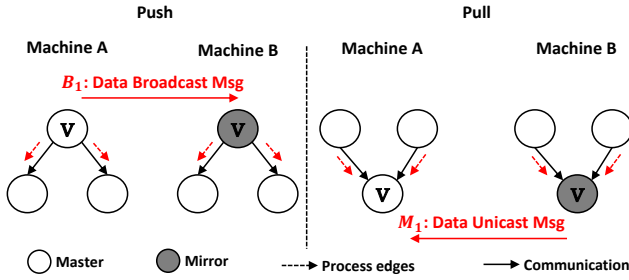
Fig. 1. Communication Pattern in Gemini for Push and Pull Modes. In Push, the Updated Value of Vertex v is Sent to All Machines No Matter Whether They Need It Or Not. Gemini Overlaps Communications with Computations Hiding Message-Passing Overhead.

TABLE I
RUNNING 10 QUERIES ON THE SINGLE-QUERY BASELINE FRAMEWORK (GEMINI) USING DIFFERENT MODES (I.E., PUSH-PULL, PUSH-ONLY, AND PULL-ONLY).

G	Algo.	Gemini: Time (seconds)		
		Push_Pull	Push_Only	Pull_Only
TTW	**SSSP**	28.71	22.42	83.96
	SSWP	21.21	14.39	74.97
	SSNP	22.68	14.50	67.16
	VT	30.34	20.49	73.68

In Section II we present the background of distributed graph processing and related work. Section III explains detailed design of our *Expressway* approach. Thorough evaluation of the design will be presented in Section IV. Finally, Section V offers the concluding remarks.

II. BACKGROUND & RELATED WORK

Now that real-world graphs are huge (e.g., *Friendster* [14] has 2 billion edges and 65.6 million vertices), they can not fit into the memory of a single machine. Also, out-of-core processing is not efficient enough. Hence, the input graph is partitioned among a cluster of multiple machines. Each machine is responsible for carrying out the updates of vertices that reside locally. The machines communicate through message-passing to exchange needed vertex values and synchronize between iterations before continuing to the next iteration. This whole system is known as distributed graph processing in which the combined memories of multiple machines are able to hold large graphs and the large number of cores made available by multiple machines enhances the degree of parallelism delivering scalability.

Distributed graph frameworks have been using different techniques to improve efficiency whether by accelerating the execution of a single query or maximizing the throughput by executing a batch of multiple queries at once. Next, we will discuss the state-of-the-art for each framework that later we use to implement and evaluate *Expressway*.

We select *Gemini* for the former, as it is the most efficient distributed platform to run a single graph query, thanks to its NUMA-aware design and its technique to overlap the communication and computation loads. On the other hand, for the latter, we choose *MultiLyra*, which achieves massive scalability and efficiency by amortizing the high communication and computation costs across multiple queries. Additionally, its ability to compress data messages by adopting fine-grained tracking methods to track the status of each query stands out.

A. Single Query: Gemini

Many frameworks have been developed to run a single graph query efficiently. The most relevant ones include *PowerGraph [1]*, *PowerLyra* [2], *Gemini* [3], and *Ligra* [13]. The latter is a shared-memory system on a single machine and

lacks scalability while the former ones are able to load large graphs into the combined memory of multiple machines delivering scalability. *PowerGraph* introduced the GAS model (i.e., Gather, Apply, and Scatter) and benefits the load balancing by dividing the edges evenly among multiple machines (vertex-cut) creating vertex replicas. One of the replicas is selected as the master and the rest become the mirrors. However, *PowerLyra* improves *PowerGraph* by adopting a hybrid-cut graph partitioning that differentiates the partitioning as well as the computation of the low-degree versus high-degree vertices aiming at reducing both computation and communication loads [18]. These systems mostly focus on minimizing inter-machine communication and computation load balancing without paying attention to intra-machine computation load balancing and locality. In contrast, *Gemini* tries to achieve scalability while maintaining the intra-machine efficiency, inspired by the shared-memory systems.

Gemini leverages its NUMA-aware design, keeping the required data (i.e., vertex values, graph edges) close to the corresponding compute cores in each machine of the cluster. Therefore, it not only delivers the scalability that any other distributed framework aims for but also cares about the intra-machine load balancing and improves the locality within each single machine. In addition, *Gemini* utilizes an overlapping technique to overlap inter-machine communications within the cluster with intra-machine computations. This makes *Gemini* the most efficient distributed framework, delivering up to 39× speedups over other single query systems [3]. *Gemini* employs the familiar push-pull modes seen in shared memory platforms and automatically switches between modes based on the computation load (i.e., number of active edges). In the following section, we discuss how *Gemini* operates in terms of computation and communication for each of these modes (see Figure 1).

– *Push:* In this mode, each master (i.e., a vertex that resides locally) added to the frontier list after being updated in the previous iteration will push its value along with its outgoing edges to their outgoing neighbors. The dashed arrow in Figure 1 shows the direction of vertex value propagation. When there are replicas requiring remote value propagation, a single broadcast message is sent to all other machines in the cluster. Machines with a vertex replica then push the value
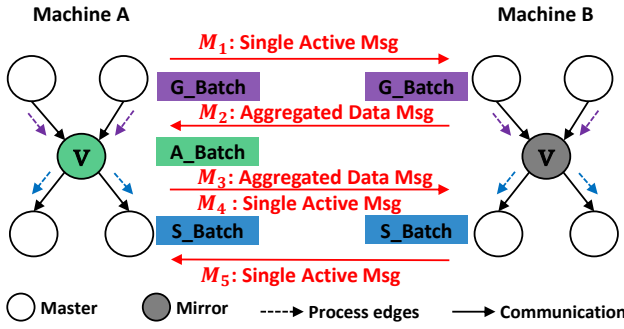
Fig. 2. Communication Pattern in MultiLyra GAS Model: Five Messages are Needed for Each Vertex Processing; Two of Them Carry Vertex Values and Three are Active Messages. MultiLyra Does Not Overlap Communications with Computations

to their respective outgoing remote neighbors, as depicted in Figure 1, left. Ultimately, destination vertices are updated with the aggregated result of all data values pushed toward them. The aggregating equation for each graph query can be found in Table III.

– *Pull:* In this mode, all vertices collect data from their incoming neighbors through their incoming edges. When a vertex serves as a mirror (i.e., it's a replica of a remote vertex residing on another machine), it sends the aggregated result of the collected data to the machine hosting the master vertex, as shown in Figure 1, right. Finally, destination vertices are updated with aggregated results from both locally and remotely collected data.

We add a switch in *Gemini* to control these modes to create the three modes of *Push_only* which always use the Push mode, *Pull_only* uses Pull mode all the time, and *Push_Pull* which is the default version and switches between Pull and Push automatically in each iteration based on the computation load. Table I shows the total execution time of running 10 random queries one by one on *Gemini* for different graph algorithms on a large graph, i.e., TTW (see Table IV for information about input graphs). *Push_only* delivers the best execution time among all modes. Therefore, for the rest of the experiments in this work, we use mode *Push_only*.

B. Multiple Queries: MultiLyra

The most relevant works for batching systems include *Quegel* [27], *MultiLyra* [16], *SimGQ* [19], *Glign* [33], and Krill [34]. *SimGQ*, *Glign* and *Krill* are limited in their scalability due to their shared-memory nature while *Quegel* and *MultiLyra* are distributed systems. Although *Quegel* was designed to handle a batch of queries, the expensive precomputations required to enable indexing impose more overhead on the system. In addition, *Quegel*'s applicability is limited to point-to-point queries [29] as opposed to the more general point-to-all queries evaluated by *MultiLyra* and *SimGQ*. Finally, *Quegel* can overlap the evaluation of only a few queries as it employs pipelined parallelism. [22] evaluates a batch of queries but it is specialized for BFS and [21] executes different queries in different processes making it inefficient. While other related works have considered simultaneous evaluation of multiple

G	Algo.	MultiLyra: Time (seconds)		
		Basic-batch	**FQT-batch**	**IQT-batch**
TTW	**SSSP**	350.70	411.93	430.56
	SSWP	266.25	296.90	313.97
	SSNP	229.38	259.10	299.6
	VT	390.21.	446.48	476.70

queries, they are limited in their scale. The only system that evaluates hundreds of queries simultaneously is *MultiLyra* which is built upon *PowerLyra* [2].

MultiLyra follows the GAS model of computing which divides the distributed computation of batches of concurrent graph queries into three main phases, i.e., Gather, Apply, and Scatter (*G_batch*, *A_batch*, and *S_batch* in Figure 2, respectively). These phases will be done in parallel for all active vertices in each machine and the message passing occurs in between phases without any overlapping between communication and computation loads. First, before *G_batch* begins, each active vertex sends a signal (i.e., active message) to their mirrors on other machines to inform them of being activated at least for one of the queries in the current iteration of the batch (*M1* in Figure 2), asking them to participate in the Gather phase. Then, in *G_batch*, each vertex whether master or mirror, goes through its incoming edges and collects data from its source neighbors. This process is executed for all the active queries associated with that vertex. Subsequently, the mirrors transmit their partially gathered data to the machine where their master is located. This allows the data to be aggregated and utilized in the Apply phase. This is being done by sending a compressed data message, based on one of the modes Basic_batch, FQT_batch, and IQT_batch (explained in the next paragraph), including the data for all the active queries (*M2* in Figure 2). In *A_batch* phase, all the masters are being updated with the aggregate result of the received partially collected remote data combined with their own locally collected one. Then, another compressed data message which includes the current values of each active query for the same vertex will be sent back to the mirrors for the purpose of coherency as well as a signal message to ask the mirrors to participate in the final Scatter phase (*M3* and *M4* in Figure 2). Finally, in the *S_batch* phase, all the vertices, whether masters or mirrors are going through their outgoing edges, and add their destination neighbor to the frontier list if at least for one of the queries in the batch it needs to be activated. Then, mirrors that get added to the frontier will send a signal to their master to assure that the master is aware of being activated for the next iteration (*M5* in Figure 2).

MultiLyra has three different modes (i.e., versions) regarding its level of query status tracking. *Basic_batch* does not

have any knowledge of whether a query is finished or activated for a vertex in the current iteration. Therefore, it computes all the phases for all the queries regardless of their status (no computation reduction). The data messages are not compressed in *Basic_batch*, and it sends all queries' data between mirrors and the master, even if the vertex value is not changed for some of the queries in the batch. The *Basic_batch* is better for small batch sizes with multiple queries, where the reduced computation and communication load cannot hide the overhead of the query status tracking systems. On the other hand, *FQT_batch* tracks the already finished queries and reduces the computation by not doing each phase for the finished queries. It compresses data messages by excluding data for the finished queries when communicating between the master and its mirrors, thereby improving communications. *FQT_batch* fits better for the midsize batches. Finally, *IQT_batch* leverages a fine-grained tracking system that, in addition to tracking the already finished queries, tracks the active queries in each current iteration for each vertex dynamically. This reduces both computations by only doing each phase for the active queries, and communication by omitting vertex values for the queries that are not active for that current iteration. Hence, it is suitable for large batch sizes, such as hundreds of queries. Table II shows that *Basic_batch* offers a better execution time when running a small batch of 10 random queries, as it avoids the overhead associated with the query tracking system for small batches. Thus, we use *Basic_batch* in our experiments.

III. EXPRESSWAY

In this section, we present *Expressway* by introducing how to identify *Highways* and develop an algorithm for it. Then, we analyze different *Expressway* policies led to different scenarios that are proposed aiming at utilizing the most benefit that *Highways* can offer. Finally we explain the *Expressway* setup in various frameworks with and example and algorithm.

A. Building Highways

Highways are the edges in the graph that most significantly contribute to the final value of many vertices. We have developed a heuristic algorithm to identify these crucial edges. Through our observations, we found that we can determine the most important edges for nearly all vertices when focusing on solving the problem for high-degree vertices. The Algorithm for Building *Highways* consists of four steps:

– **Identifying High-Degree Vertices:** The initial step involves identifying the high-degree vertices in the graph. High-degree vertices are those which have the most incoming and outgoing edges. As discussed, we only require 20 high-degree vertices for building the *highways*. To determine these 20 high-degree vertices, we can sort all vertices based on their degrees and select the top 20. As previously mentioned, having more than 20 high-degree vertices doesn't significantly improve the precision of the final reduced graph. Instead, it unnecessarily increases the number of *highways* in the reduced graph.

– **Forward Query Evaluation:** After identifying the 20 high-degree vertices, we apply our algorithm to these vertices on

---

Algorithm 1 Identifying Highways on a Given Graph.

1: **Input**: Graph $G(V, E)$
2: **Output**: $G(V, E_{highways})$; $E_{highways}$ contains $highways$
3:
4: ▷ Finding High-Degree vertices
5: $D[V]$: array for collecting degree of each vertex
6: H: high-degree vertex set
7: **for each** $v \in V$ **do**
8:     $D[v]$ = OutDegree(v) + InDegree(v)
9: **end for**
10: H = Index of 20 high values on array $D[V]$
11:
12: ▷ Forward Query Evaluation
13: **for each** $h \in H$ **do**
14:     $E_{forward}(h)$ = SOLVE ($G(V, E)$, DIRECTION f)
15:     $E_{highways} = E_{highways} \cup E_{forward}(h)$
16: **end for**
17:
18: ▷ Backward Query Evaluation
19: **for each** $h \in H$ **do**
20:     $E_{backward}(h)$ = SOLVE ($G(V, E)$, DIRECTION b)
21:     $E_{highways} = E_{highways} \cup E_{backward}(h)$
22: **end for**
23:
24: ▷ Check for Connectivity of the Highways
25: **for all** $v \in V$ **do**
26:     **if** (OutDegree(v)$\neq$ 0) $\wedge$ (OutEdges(v) $\cap$ $E_{highways}$) $= \phi$ **then**
27:         Add an out edge of v to $E_{highways}$
28:     **end if**
29: **end for**
30:
31: ▷ Solve Function
32: **function** SOLVE ($G(V, E)$, DIRECTION d)
33:     Evaluate Query $Q(s)$ on $G(V, E)$
34:     **for all** $e(u, v) \in E$ **do**
35:         **if** $Q(s)$ updates $Q(s).Val(u)$ **then**
36:             **if** $(Q(s).Val(u) \bigoplus w(u, v) = Q(s).Val(v))$ **then**
37:                 **if** $(d == f)$ **then**
38:                     $E_{highways}(h) = E_{highways}(h) \cup \{ e(u, v) \}$
39:                 **else** ▷ ($d == b$)
40:                     $E_{highways}(h) = E_{highways}(h) \cup \{ e(v, u) \}$
41:                 **end if**
42:             **end if**
43:         **end if**
44:     **end for**
45: **end function**

---

the graph in a forward direction. We then select the edges that contribute to the results for these 20 high-degree vertices. These selected edges become our *highways*.

– **Backward Query Evaluation:** This step mirrors the previous one but with a twist. Here, we perform a backward query evaluation for the 20 high-degree vertices and select the contributing edges, marking them as our *highways*.

– **Connectivity of the highways:** Once the *highways* are identified in the second and third steps, we must ensure the connectivity of the graph. We examine all the vertices, and if any vertex lacks an outgoing edge, we select one outgoing edge for that vertex and include it in our set of *highways*.

Upon completing the above four steps, we obtain a condensed graph. This graph retains the same vertices as the

original but has a significantly reduced number of edges, now termed *highways*. In algorithm 1, the procedure for building *highways* is detailed. As illustrated in the algorithm, its input is a graph in the form of $G(V, E)$, where V represents the number of vertices and E denotes the number of edges in the graph. The output is $G(V, E_{highways})$, a graph with the same vertex count but a reduced edge count ($E_{highways}$). Thus, the output graph only contains *highways*. Initially, the algorithm identifies the twenty highest degree vertices in the graph. To achieve this, we loop over the vertices, calculating the sum of in and out edges for each vertex. These degrees are stored in an array named $D[V]$. Subsequently, the twenty highest values in the $D[V]$ array are identified, and their indexes are stored in H. According to algorithm 1, after pinpointing the twenty high-degree vertices, a forward query evaluation is performed. For each vertex in our high-degree vertex set H, the $Solve$ function is invoked. This function identifies the edges contributing to the results for each high-degree vertex. The identified edges are then added to the $E_{highways}$ set. Following this, a backward query evaluation is conducted. Again, the $Solve$ function is called to identify edges in the backward direction, which are then added to $E_{highways}$. The final step ensures the connectivity of the vertices. We iterate over the graph's vertices. If a vertex is connected in the original graph but not through the edges in the $E_{highways}$ set, an outgoing edge is added to $E_{highways}$ to ensure connectivity using the $E_{highways}$ edges. The resulting graph contains all the *highways*, enabling accelerated distributed graph processing. The $Solve$ function identifies edges contributing to our query results in both forward and backward directions. It accepts the graph and direction as inputs, evaluates the query on the graph, and finds all answers. For each edge in the graph, if it contributes to a vertex's value, that edge is added to $E_{highways}$, considering both forward and backward directions.

Let us demonstrate Algorithm 1 using an example. As you can see in Figure 3(a), we have a full graph, and our goal is to find the *highways* on this graph for the single source shortest path (SSSP) algorithm. For this example, we only want to do that for one high-degree vertex, which is our highest degree node, a. As demonstrated in Figure 3(b), first, we should perform a forward query evaluation. We start from the high-degree node a, run the SSSP algorithm, and find the shortest path from vertex a to all other vertices in the forward direction. We identified the edges selected in this step using a blue color. Then, as depicted in Figure 3(c), we should evaluate in the backward direction. Therefore, we will find the shortest paths from every other vertex to our highest degree vertex, which is vertex a. We identified the edges selected in this step with a red color. The final step is to check connectivity. As shown in Figure 3(d), we should check each vertex, and if the vertex has at least one outgoing edge on the full graph, it should also have at least one outgoing edge on the reduced graph. Therefore, we will examine all the vertices and add two outgoing edges for the h and j vertices. Finally, in Figure 1(e), you can see the final graph with only *highways*.

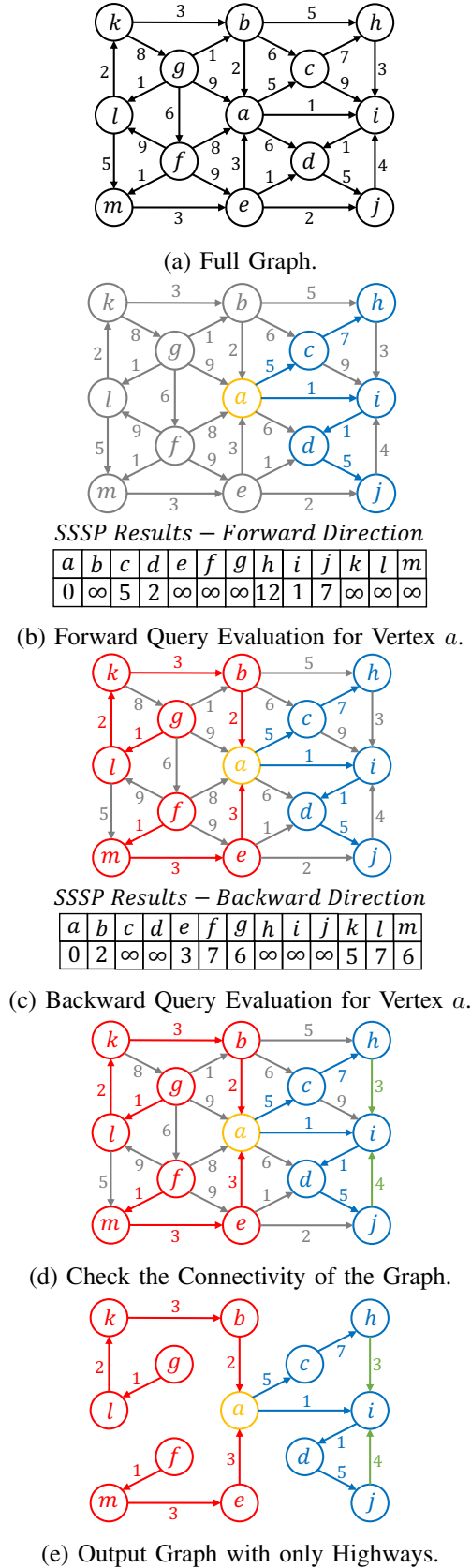After identifying *highways* on a graph, as you can see in

(a) Full Graph.

SSSP Results − Forward Direction

a	b	c	d	e	f	g	h	i	j	k	l	m
0	∞	5	2	∞	∞	∞	12	1	7	∞	∞	∞

(b) Forward Query Evaluation for Vertex a.

SSSP Results − Backward Direction

a	b	c	d	e	f	g	h	i	j	k	l	m
0	2	∞	∞	3	7	6	∞	∞	∞	5	7	6

(c) Backward Query Evaluation for Vertex a.

(d) Check the Connectivity of the Graph.

(e) Output Graph with only Highways.

Fig. 3. Example to Show the Steps for Identifying the Highways on a Graph for the Single Source Shortest Path Algorithm and High-degree Vertex a.
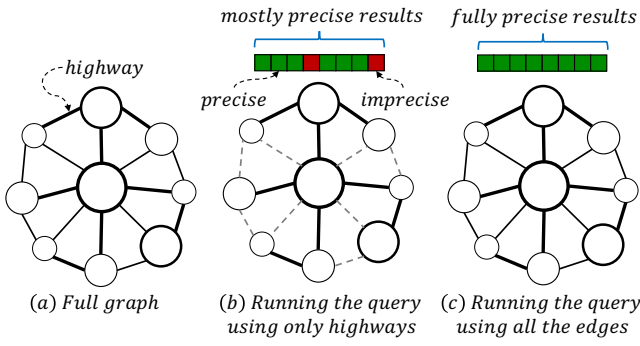
Fig. 4. Query Evaluation Using the Highways in a Graph.

the Figure 4, we should first run the query using only the *highways*, and then run the query using all the edges in the graph. Since the *highways* contribute to the final results for most of the vertices, running the query with just the *highways* yields correct results for the majority of the vertices. Given that the number of edges identified as *highways* is quite small, this step is executed quickly. By utilizing this swift step, we obtain accurate results for most of the vertices. To ensure correct results for all vertices, we should run the query using all the edges in the graph after the initial step. This subsequent step is also efficient, as most of the results are already stable, leading to rapid convergence.

B. Expressway Policies

We can have four types of policies for using the expressway in the graph, and we will explore each policy here.

– *ExWaySin: ExWaySin* stands for single expressway. In this technique, we run the graph with *highways* on a single machine instead of a distributed machine. If our input graph is small, the edges identified as *highways* will also be few. Therefore, we can run the *highways* on a single machine (i.e., on each machine in the cluster separately at the same time), eliminating the communication and barriers between machines.

– *ExWayDis: ExWayDis* stands for distributed expressway. In this approach, if the number of edges identified as *highways* is high, they should be run on a distributed machine. This approach is suitable for large graphs because as the size of the graph increases, the number of edges identified as *highways* also increases. If the graph with the *highways* becomes too large, it cannot be run on a single machine, necessitating the use of a distributed machine.

– *ExWayHalf:* In this technique, we don't execute the entire graph using the identified *highways*. Instead, if we can achieve predominantly accurate results with just half the execution of the *highways*, we adopt this approach and don't wait for all nodes to stabilize. We opt for this method because, in most graphs, the last iterations exhibit a long tail before all nodes stabilize. Notably, the first few iterations of the graph evaluation show a substantial update, but this update diminishes significantly in the final iteration. As a result,

---

Algorithm 2 *Expressway* employed in the Gather and Scatter phases of the GAS model. Similarly, it applies to the Pull and Push modes respectively for the single query platform assuming batch_size is 1.

```
 1: ▷ Expressway for gather/pull
 2: Input: active vertex v, Expressway_Enable
 3: Output: the aggregated collected data
 4:
 5: function G_BATCH ( v, Expressway_Enable)
 6:     edge_list = in_edges_of(v)
 7:     if Expressway_Enable then
 8:         edge_list = highways_of(v, out_edge=false)
 9:     end if
10:     agg_results[0:batch_size] = INITIAL_VALUE
11:     for q ∈ active_queries_for(v) do
12:         for e ∈ edge_list do
13:             agg_result[q] = agg(e.src().value[q], e.data())
14:         end for
15:     end for
16:     RETURN agg_result
17: end function
18:
19: ▷ Expressway for scatter/push
20: Input: active vertex v, Expressway_Enable
21: Output: makes the next frontier
22:
23: function S_BATCH ( v, Expressway_Enable)
24:     edge_list = out_edges_of(v_i)
25:     if Expressway_Enable then
26:         edge_list = highways_of(v_i, out_edge=true)
27:     end if
28:     for q ∈ active_queries_for(v_i) do
29:         for e ∈ edge_list do
30:             if (e.dst().value[q] ⨁ agg(v.value[q], e.data())) then
31:                 active_list ← e.dst().id
32:             end if
33:         end for
34:     end for
35: end function
36:
37: ▷ Main loop
38: Input: G, Expressway_Enable=true, ExHalf=false, i_threshold
39: Output: The final result for the running algorithm
40:
41: function RUN( G, Expressway_Enable, ExHalf)
42:     i ← 0
43:     while !active_list.empty() do
44:         if Highway.isDone() or (ExHalf and i=i_threshold) then
45:             Expressway_Enable = false
46:             active_list ← all v.ids visited
47:         end if
48:         ▷ run in parallel for each active v
49:         collected_data = G_batch(v, Expressway_Enable)
50:         A_batch(v, collected_data[v])
51:         S_batch(v, Expressway_Enable)
52:         i++;
53:     end while
54: end function
```

there's limited advantage in executing the concluding iterations of the graph evaluation. Moreover, as we'll discuss in the evaluation section, for all our algorithms and input graphs, we can secure highly accurate results (exceeding 97 percent) by solely utilizing the *highways* in the graph.

– *ExWayFull:* In this technique, we run the graph entirely with *highways* and skip the second step, which involves running the graph with all the edges. We can employ this method when our graph doesn't have a long tail and all the vertices stabilize quickly.

We create three Scenarios by combining the above policies. *ExFDis* combines *ExWayDis* with *ExWayFull* and runs the input graph entirely using *highways* in a distributed manner while *ExHDis* combines it with *ExWayHalf* and stops the execution of *highways* midway to avoid the communication cost for the iterations that converge fewer vertices. Similarly, Similarly, *ExFSin* combines *ExWaySin* and *ExWayFull*. Please note that running graphs fully on *highways* in a single machine is fast enough to not impose any communication cost. Therefore, combining *ExWaySin* with *ExWayHalf* is not feasible.

C. Expressway Setup

Algorithm 2 shows the *Expressway* setup for a batching system by being applied to the Gather and Scatter phases in the GAS model. To avoid repetition, we only discuss this algorithm while one can similarly apply *Expressway* to the Pull/Push single query systems since the gather function is similar to pull function, and the scatter function is similar to push function (i.e., when batch size is equal to 1).

Throughout the run time of a batch of graph queries, $Expressway\_Enable$ flag determines, in the current iteration i, for each active query q and for an active vertex v, whether the query runs on *highways* only or all connected edges (see Algorithm 2, lines 7-9 for Gather, and line 25-27 for Scatter). Particularly, when $Expressway\_Enable$ is set to $true$ during the Gather phase, the active vertex v for each query q that is active for v in the current iteration, will select the edges from *highways*, line 8, to loop over lines 12-14. It calculates the aggregated result by using data from the source vertex of the incoming edge e, as well as the edge data itself, based on the aggregating equation presented in Table III. Later, this collected data from the Gather phase will be used to update the vertex v value in the following Apply phase as seen in line 50. Scatter will also loop over the *highways* only when the $Expressway\_Enable$ flag is set to $true$, as shown in Algorithm 2, lines 25-27. For each vertex v that has been updated in the Apply phase and for each active query q, scatter will only propagate the data through *highways* by adding the destination of the outgoing edge e to the next active list, as indicated in line 31.

Finally in Algorithm 2, line 44 - 47 carries out the transition from running only using *highways* to the full graph. It manages the *ExWayFull* and *ExWayHalf* policies which is used to create *ExFDis* and *ExHDis* scenarios explained above. If no threshold to stop early for the *highways* is specified, (i.e., $XHalf$ is $false$), then the *highways* will be used until all the vertices converge to their pre-final values. This will be determined by $Highway.isDone()$. On the other hand, the *highway* run can be interrupted early in iteration equal to $i\_threshold$ when $XHalf$ is $true$. The transition will be complete by adding all the vertices that have been visited during the *highway* run

Algorithm	Aggregating Equation
SSSP	$v.vlaue = \text{Min}\ (v.value,\ u.value + e.data)$
SSWP	$v.vlaue = \text{Max}\ (v.value,\ \text{Min}\ (u.value,\ e.data))$
SSNP	$v.vlaue = \text{Min}\ (v.value,\ \text{Max}\ (u.value,\ e.data))$
VT	$v.vlaue = \text{Max}\ (v.value,\ u.value\ /\ e.data)$

Input Graph	#Edges	#Vertices
Twitter WWW (TTW) [7]	1.5 B	41.6 M
Twitter MPI (TT) [4]	2.0 B	52.6 M
Friendster (FS) [14]	2.6 B	68.3 M

time to the $active\_list$ before proceeding to the final run. This is necessary to ensure the correctness of the vertex values, making sure that the pre-final values of all vertices propagate via all edges.

IV. EXPERIMENTS

We implemented *Expressway* using *Gemini* [3] which advances the distributed graph processing via its NUMA-Aware design for a single query, and *MultiLyra* [16] which enables scalable and efficient evaluation of multiple concurrent queries. In our evaluation, we consider four algorithms - Single Source Shortest Path (SSSP), Single Source Widest Path (SSWP), Single Source Narrowest Path (SSNP), and Viterbi (VT) [8] (see Table III). We use three large input graphs listed in Table IV, with a billion edges named TTW, TT, and FS. For each input graph and for each algorithm, we ran 10 queries. The sources are unique and were selected randomly. All experiments were performed on a cluster of four identical machines. Each machine has 2 NUMA nodes of 32 Intel Xeon cores (i.e., total number of 64 threads), 256 GB memory, and runs Rocky Linux release 8.5.

A. Policies Analysis

In this section, we analyze our decision towards the policies that we propose. We applied *Expressway* on the baseline *Gemini* and ran 10 random SSSP queries on input graph TTW using a cluster of four machines specified above. Table V shows the reduction in execution time when we run the queries on the input graph using only *Highways* without transition to the full graph, versus when we ran the queries using all edges, which is our baseline. It is important to note that the vertex values obtained from running on *highways* are not yet finalized. However, we do obtain mostly precise values for the

G	Execution Time (seconds)		Speedup
	All-edges	Highways-only	
TTW	21.68	8.57	2.53×
TT	27.43	12.36	2.22×
FS	44.02	19.03	2.31×

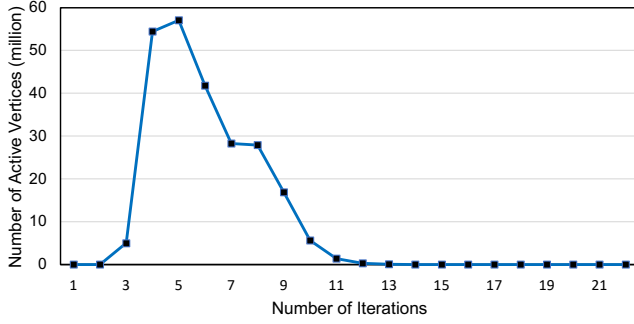| G	Algo.	highway/edge	ExWayHalf	ExWayFull
TTW	SSSP	7.55%	0.64	0.99
	SSWP	10.16%	0.55	0.99
	SSNP	10.30%	0.46	0.99
	VT	6.24%	0.42	0.99
TT	SSSP	9.36%	0.52	0.99
	SSWP	7.71%	0.64	0.99
	SSNP	7.71%	0.58	0.99
	VT	7.73%	0.43	0.99
FS	SSSP	13.77%	0.35	0.97
	SSWP	9.57%	0.44	0.99
	SSNP	9.57%	0.58	0.99
	VT	9.65%	0.75	0.99

Fig. 5. The Frontier Size Over the Iterations of a SSSP Query on TTW.

vertex when using only *highways*. The column "ExWayFull" on Table VI shows the ratio of vertex values that converges to their final results only by executing the *highways*. It shows that we can achieve the final results for more than 97% of the vertices by only running an average of 9.11% of the total number of edges for different algorithms on different graphs, see Table VI, column "Highway/Edge".

Therefore, the speedups displayed in Table V represent the ideal performance gains that *Expressway* policies aim to converge toward. Table VII shows the number of edges needed to be processed, i.e., the amount of computation, the number of updates to vertex values and finally the number of communication that took place for the above experiment. Table VII indicates that using only the *highways* significantly reduces the computation load by 28.35× reduction rate on TTW and reduces the number of vertex updates by 3.93× while the number of communications among the machines is reduced the least by 2.87×. Comparing these numbers with the ideal speedups from Table V shows that the communication load is the bottleneck to get closer to the ultimate speedups in a distributed environment.

This leads us to propose the *ExHDis* scenario in which we interrupt the *Expressway* while running on *highways* when it reaches the iteration at which the number of active vertices, i.e., the frontier size, starts to decrease. Column "ExWayHalf" in Table VI shows he ratio of vertices that have converged to their final value after interrupting the *Expressway* process midway. On average, more than 50% of vertices have already finalized. For example, Figure 5 shows the number of active vertices throughout the run time of a random SSSP query on

TTW. This indicates that in the initial iterations (e.g., before iteration 9), the majority of vertices are processed, and the remaining vertices are addressed in the subsequent iterations from 9 to 24. By stopping early, we eliminate the need to incur the synchronization overhead associated with the distributed platform for those final iterations. These would otherwise involve processing the smaller remaining set of vertices over a greater number of iterations.

Finally, as mentioned above, since the highway edges are only 9.11% of the total number of edges on average (see Table VI for detailed numbers for each graph and algorithm), they are small enough to be loaded on each machine in the cluster leading us to our next scenario *ExFSin*. Then, each machine can run the highways independently, avoiding the communication load before transitioning to using the full set of edges in a distributed manner.

B. Single Query: Gemini

We applied the three scenarios discussed above, i.e., *ExFDis*, *ExHDis*, and *ExFSin*, and implemented *Expressway* as discussed in Section III (C) on Gemini, which is the state-of-the-sate and the current most efficient distributed framework to run a single graph query. We ran 10 queries for each input graphs and for each algorithm, following the above scenarios. We compared their execution times with the baseline *Gemini* that does not utilize the *highways*. Table VIII shows the execution time in seconds for the baseline (*Gemini*) as well as the speedups achieved by the *Expressway* policies over the baseline. *ExFDis* delivers speedups ranging from 1.35× for SSNP on TTW to 2.46× for VT on the input graph FS while *ExHDis*, leveraging its early transition to the full graph, improves the speedups from 1.57× in SSSP for TTW to 2.01×. This behavior repeats for all algorithms on all input

G	# of	$\times 10^9$		
		All-edges	Hways-only	Reduction
TTW	edge comp.	56.99	2.01	28.35×
	updates	2.28	0.58	3.93×
	comm.	1.52	0.53	2.87×
TT	edge comp.	69.71	2.85	24.46×
	updates	3.02	0.86	3.51×
	comm.	1.97	0.81	2.43×
FS	edge comp.	127.20	5.52	23.04×
	updates	4.88	1.49	3.27×
	comm.	3.18	1.31	2.43×

G	Algo.	Time (s)	Expressway: Speedup		
		Gemini	ExFDis	ExHDis	ExFSin
TTW	SSSP	21.98	1.57×	2.01×	2.17×
	SSWP	16.72	1.35×	1.67×	1.88×
	SSNP	17.38	1.39×	1.74×	1.93×
	VT	25.86	1.86×	2.10×	3.03×
TT	SSSP	27.86	1.42×	1.90×	2.44×
	SSWP	22.40	1.44×	1.77×	2.22×
	SSNP	21.25	1.42×	1.71×	2.11×
	VT	32.94	2.01×	2.46×	3.26×
FS	SSSP	44.96	1.48×	1.85×	2.36×
	SSWP	30.51	1.52×	1.84×	2.23×
	SSNP	30.60	1.56×	1.94×	2.26×
	VT	60.75	2.46×	2.73×	4.08×

G	Algo.	Time (s)	Expressway: Speedup	
		MultiLyra	ExFDis	ExFSin
TTW	SSSP	377.66	2.32×	3.65×
	SSWP	274.75	1.72×	2.10×
	SSNP	251.70	1.89×	2.53×
	VT	391.84	2.53×	4.04×
TT	SSSP	214.69	1.55×	2.23×
	SSWP	219.00	1.84×	2.61×
	SSNP	218.82	1.93×	3.02×
	VT	280.09	2.15×	3.49×

graphs. It even improves VT on FS, which already showed up 2.46× speedup, further to 2.73×. As a further step to avoid the bottlenecks and overheads of distributed computation of highways, since there is a limited number of highways, we applied *ExFSin* scenario and runs the highways in a shared-memory manner on each machine before transitioning to distributed computation for final convergence. *ExFSin* obtains speedups ranging from 1.88× for SSWP on TTW up to 4.08× for VT on FS boosting the performance for all algorithms on all three input graphs. Note that among the four algorithms, VT is the most expensive one in terms of computation load due to its floating point operation. On the other hand, the larger the graph, the more computations need to be performed. Hence, any reduction ratio in amount of computation can be significant for expensive algorithms and larger graphs. Therefore, *Expressway* as shown in Table VIII, delivers better speedups for VT on FS.

C. Multiple Queries: MultiLyra

We integrated the *Expressway* approach into the *MultiLyra* system to assess its performance when executing a batch of simultaneous graph queries. For this evaluation, we selected the *ExFDis* and *ExFSin* scenarios. These choices allowed us to juxtapose both the minimum and maximum speedup delivery scenarios against a baseline, which involved running a batch of 10 concurrent graph queries. We applied various algorithms, as outlined in Table III, to the TTW and TT graphs. Table IX illustrates our results for concurrent run of 10 graph queries. *Expressway* delivers speedups ranging from 1.72× for SSWP to 4.04× for VT on TTW and ranging from 1.55× for SSSP to 3.49× for VT on TT over *MultiLyra* and follows the same direction as seen over the single query framework.

V. CONCLUSION

In this paper, we present *Expressway*, a technique to determine the important edges called *Highways* in a large graph, aiming at accelerating the distributed evaluation of iterative graph queries. *Highways* are selected from those edges that deliver the aggregated final value between their endpoints. We evaluated our technique using state-of-the-art distributed graph processing frameworks. The experiments for evaluating a single graph query as well as a batch of simultaneous graph queries show that our technique can be successfully applied to any framework and speed up their execution times. Expressway achieves up to 4.08× speedup over *Gemini*, a single-query framework, and obtains up to 4.04× speedup over *MultiLyra*, a scalable framework to evaluate a batch of concurrent graph queries.

REFERENCES

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 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. In *ACM Transactions on Parallel Computing* (TOPC), 5(3), 13, 2019.

[3] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 301-316, 2016.

[4] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *Proceedings of the 4th international AAAI conference on weblogs and social media*, 2010.

[5] 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*, pages 44-54, 2006.

[6] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 599-613, 2014.

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

[8] 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.

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

[10] 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.

[11] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135-146, 2010.

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

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

[14] J. Yang and J. Leskovec. Defining and Evaluating Network Communities based on Ground-truth. In *Proceedings of the IEEE 12th International Conference on Data Mining* (ICDM), pages 745-754, 2012.

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

[16] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable Distributed Evaluation of Batches of Iterative Graph Queries. In *Proceedings of the IEEE Big Data Conference*, pages 349-358, 2019.

[17] A. Mazloumi, C. Xu, Z. Zhao, and R. Gupta. BEAD: Batched Evaluation of Iterative Graph Queries with Evolving Analytics Demands. In *Proceedings of the IEEE Big Data Conference*, pages 461-468, 2020.

[18] A. Mazloumi and R. Gupta. Enabling Faster Convergence in Distributed Irregular Graph Processing. In *Proceedings of the IEEE Big Data Conference*, pages 6151-6153, 2019.

[19] C. Xu, A. Mazloumi, X. Jiang, and R. Gupta. SimGQ: Simultaneously Evaluating Iterative Graph Queries. In *Proceedings of the IEEE HiPC Conference*, pages 1-10, 2020.

[20] C. Xu, A. Mazloumi, X. Jiang, and R. Gupta. SimGQ+: Simultaneously evaluating iterative point-to-all and point-to-point graph queries. In *Journal of Parallel and Distributed Computing*, volume 164, pages 12-27, 2022.

[21] P. Pan and C. Li. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *Proceedings of the IEEE ICCD Conference*, pages 217-224, 2017.

[22] 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 *Proceedings of the VLDB Endowment*, 2015.

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

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

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

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

[27] 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.

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

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

[30] S Rahman, M Afarin, N Abu-Ghazaleh, R Gupta. JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator. In *MICRO-54: Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 1091–1105, 2021.

[31] M Afarin, C Gao, S Rahman, N Abu-Ghazaleh, R Gupta. CommonGraph: Graph Analytics on Evolving Data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS), pages 133–145, 2023.

[32] M Afarin, C Gao, S Rahman, N Abu-Ghazaleh, R Gupta. CommonGraph: Graph Analytics on Evolving Data (Abstract). In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing* (HOPC), pages 1–2, 2023.

[33] X. Yin, Z. Zhao, and R Gupta. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (ASPLOS), pages 78-92, 2023.

[34] H. Chen, M. Shen, N. Xiao, and Y. Lu. Krill: a compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC), Article 51, pages 1-16, Nov. 2021.