UNIVERSITY OF CALIFORNIA
RIVERSIDE

Distributed Evaluation of Batches of Iterative Graph Queries

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Abbas Mazloumi

December 2023

Acknowledgments

The work presented in this thesis would not have been possible without the inspiration, support, effectiveness, and impact that many individuals have had during this journey and throughout my life.

First, I would like to send my heartfelt respect to Prof. Rajiv Gupta for his exceptional intelligence, wisdom, innovative ideas, and constructive guidance. Prof. Rajiv Gupta has been not only a remarkable mentor in my Ph.D. and technical pursuits, but he has also demonstrated immense goodness as a person making him a great inspiration in both personal and professional life. *Thank you Prof. Gupta, without you this journey would not have been possible*.

I would also like to thank my dissertation committee members, Prof. Nael Abu-Ghazaleh, Prof. Zhijia Zhao and Prof. Daniel Wong, for their valuable feedback and support. I would like to express a special gratitude to Prof. Zhao for his collaboration and guidance in this research.

I also appreciate all supports from my previous advisor from University of Tehran, Prof. Mehdi Modarressi, whose guidance has formed my early steps into the research life more solid.

This thesis was supported in part by National Science Foundation Grants CCF-2002554, CCF-2028714, CCF-1813173, and CCF-1524852 to the University of California Riverside. I would like to thank the NSF for their support.

I would like to express my sincere gratitude to my awesome lab mates and colleagues for their contributions, collaborations, and most importantly their friendships. Chengshuo Xu, Xiaolin Jiang, and Mahbod Afarin also have been co-authors and I enjoyed working with them. I would like to thank Farzad Khorasani, Keval Vora, Hongbo Li, Zachary Benavides, Arash Alavi, Gurneet Kaur, Xiaofan Sun, Xizhe Yin, and Hadi Zamani.

I would like to thank my dear family without their support and sacrifices, my life would have been different. To my caring parents, Fatemeh and Alireza, for encouraging us to always go toward the right direction in life. To my only sister Azam for her kindness, love, and support to the family. To my brothers, Yusef that has been my life mentor and guided me through tough decisions throughout my life; Mohsen who has been willing to sacrifice his own to see his family's success and has always been there for me whenever I needed help and Ehsan that always has been my best friend to go to when I am down to cheer up. Last but not least to my parents-in-law for their constant support and love.

Finally, my deepest gratitude goes to my love Maryam, and our little one Mana with whom my life has tasted sweet and peaceful. To Maryam for her love, friendship, and being the best companion.

To Maryam, Mana, And My Whole Family.

ABSTRACT OF THE DISSERTATION

Distributed Evaluation of Batches
of Iterative Graph Queries

by

Abbas Mazloumi

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2023
Dr. Rajiv Gupta, Chairperson

Graph analytics has been widely used for analyzing large-scale networks using ever-growing graphs to represent the relationships and entities. The real-world graphs are often large and constantly evolving over time. Former requires parallel distributed processing across multiple multicore machines; latter requires repetitive processing over different snapshots of the graphs over time. 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. A significant drawback of distributed platforms is that they impose long latency operations due to their high communication latency between machines in the cluster; especially when the computation load of a graph query is small (e.g., finding the shortest path in a graph).

This research first introduces the MultiLyra *distributed batching* system that amortizes the communication and computation costs across multiple queries by simultaneously evaluating batches of hundreds of iterative graph queries improving the throughput while at the same time maintaining the scalability of distributed processing. Via use of a unified frontier for all queries in a batch, the overhead of distributed evaluation is amortized across queries. MultiLyra yields maximum speedups over the single query baseline ranging from 3.08× to 5.55× across different batch sizes, algorithms and input graphs which are then improved to speedups range from 7.35× to 11.86× by employing a fine-grained query tracking technique and value reuse optimization.

Second, it introduces the ExpressWay technique for *faster convergence* based on differential treatment of important edges in the graph to further improve the efficiency of the batching system. Each machine in the cluster loads a small portion of the edges from the graph, i.e., the important edges contributing the most in delivering the final results to the vertices, and run the graph queries independently on these edges avoiding the inter-machine communication cost before starting evaluation on the distributed full graph. ExpressWay benefits MultiLyra by giving additional speedups of up to 4.08× over the MultiLyra baseline for a batch size of 10 queries.

Finally, we present the BEAD system that expands the applicability of batching to evolving analytics demands when both the graph and the batch of queries are allowed to grow over time. To maintain the scalability in this case, an *incremental evaluation technique* will be used to take advantage of the existing result of the evaluation of a batch of queries on the previous version of the graph to accelerate the evaluation of the batch of queries on the current version of the graph. BEAD outperforms MultiLyra's batched evaluation by factors of up to 26.16× when the graph evolves and 5.66× when the batch of queries is updated.

Contents

List of Figures

List of Tables

Chapter 1

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 [10], 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 to the size, another important feature of real-world graphs is that they are constantly evolving over time, e.g., graphs representing the online shopping behavior of all customers of an online shopping system grow as the number of customers grows.

Therefore, this has led to a great deal of interest in developing efficient graph analytics systems for shared memory (e.g., Galois [13], Ligra [14]), GPUs, and custom accelerators [44] [45] [47] as well as platforms in the distributed environment (e.g., Pregel [12], GraphLab [11], GraphX [7], PowerGraph [1], PowerLyra [2], ASPIRE [26]). Among these, systems that are aimed at distributed computing platforms are the most scalable. The former assumes that the memory of a single machine is large enough to hold the entire graph and lacks scalability while the latter utilizes the combined memory of multiple machines across a cluster to be able to hold the very large real-world graphs. Thus, systems that are aimed at distributed computing platforms are the most scalable. Despite the above platforms that support fix graphs, there have been efforts to develop systems to support evolving graphs [52, 53, 54].

While the performance of graph analytics has improved greatly due to advances in aforementioned systems, much of this research has focused 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). For example, the following two scenarios involve multiple queries: (a) Single-User scenario in which a single user may conduct a complex analytics task requiring issuing of multiple queries; and (b) Multi-User scenarios as in [22] and [23] where the same data set is queried by many users. In both scenarios, machine resources can be fully utilized delivering higher throughput by simultaneously evaluating multiple queries on a modern server with many cores and substantial memory resources. None of the above works introduce a thorough optimization for evaluating multiple queries over fixed or evolving graphs unless multiple queries simply can be evaluated one after another in their system.

Therefore, this thesis first answers the need for a distributed batching system to improve the throughput of the distributed graph processing platforms while maintaining their scalability. After archiving throughput and scalability, it introduces various optimization techniques including a simple reuse technique, fine-grained query status tracking techniques, and various scenarios that enables faster convergence of running algorithms to further improve their efficiency. Finally, it explores the expansion of the batching system capabilities by enabling it to answer the needs when the input graph and the running queries evolve over time.

The rest of this chapter is as follows. Fist in Section 1.1 introduces the thesis overview follows by an introduction for each chapter of the thesis. Then, Section 1.2 presents the how this thesis is organized.

1.1 Dissertation Overview

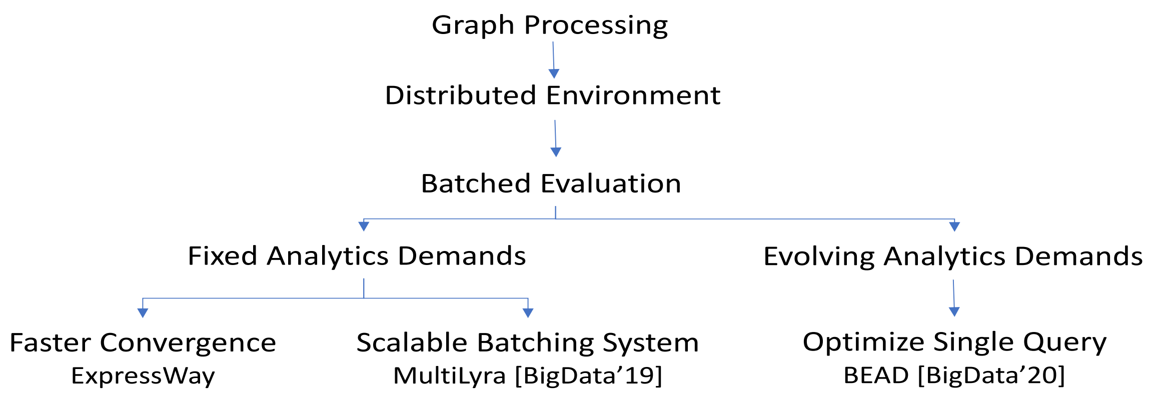


Figure 1.1: Dissertation Overview

This thesis focuses on developing scalable high-performance solutions for graph processing by employing resources available on a heterogeneous computing cluster. Figure 1.1 shows the high level overview of this thesis. We first develop the distributed MultiLyra [BigData'19] system whose scalability enables simultaneous evaluation of batches of hundreds of iterative graph queries. then, introduce ExpressWay technique in which we prioritize the edges on the graph and presents various policies to further enhance the distributed evaluation of MultiLyra. Finally, BEAD

[BigData'20] extends MultiLyra to consider scenarios in which a batch of queries needs to be continuously reevaluated due to changes to the graph (for growing graphs) as well as the scenarios that updates the running queries (for growing batches).

1.1.1 MultiLyra For Throughput

In this section, we address optimized evaluation of a *batch* of graph queries by amortizing the communication and synchronization costs of distributed evaluation. To do this, we presents a general graph analytics framework, *MultiLyra*, aimed at simultaneously evaluating a batch of hundreds of vertex queries for different source vertices of a large graph by using a scalable distributed batching technique. For example, for SSSP algorithm, we may be faced with the following batch of queries:

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

MultiLyra achieves high performance via utilizing a unified frontier, aggregated communication strategies and query status tracking policies, both of which amortize the communication and computation overhead across queries. The idea of batching is motivated by the practical scenario [38], where online shopping platforms frequently compute a set of interesting graph queries involving the most important shoppers over a large input graph that represents the online shopping behaviors of all the customers. To meet the needs, Yan and others proposed *Quegel* [38], which allows for overlapped execution of a small set of queries. By contrast, our proposed system *MultiLyra*, expands the batching capability to simultaneously evaluating hundreds of iterative queries.

1.1.2 ExpressWay For Efficiency

While most of the existing platforms 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.

Therefore, for further optimizing our system, we present *Expressway*, a technique to improve the efficiency of distributed graph frameworks by prioritizing important edges of an input graph for a specific type of the batch of the running queries. 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

1.1.3 BEAD For Evolving Demands

Despite the promises of batched graph query processing, *Quegel* and *MultiLyra* target a static scenario where the input graph is fixed and the queries of interests are pre-defined. However, in many real-world scenarios, both the graph and the batch of queries may evolve. For example, in the scenario of online shopping center, as the system is being used, new customers may join continuously and the graph representing the online shopping activities will also continue to grow. Consequently, there is a need to regularly reevaluate the batch of interesting queries as the graph grows in size and changes in its structure. In addition, as the set of customers increases, a need arises to also grow the set of interesting queries to account for newly identified important customers. In other words, in the real-world situation of continuous activity, the system is faced with Evolving Analytics Demands. While using the batching systems such as *MultiLyra* [17] or *Quegel* [38] for such evolving demands are possible by fully reevaluating the updated batch of queries on the updated graph, they may incur significant latency that keeps growing as the graph expands and more interesting queries are identified.

To meet these needs, we introduce *BEAD*, a system that greatly expands the *MultiLyra* batching techniques to support batching in presence of the evolving both graphs and the sets of queries. The key to the superior efficiency offered by *BEAD* lies in a series of incremental evaluation techniques that leverage the results of prior request to "fast-foward" the evaluation of the current request.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents MultiLyra, the system for distributed batched evaluation of iterative graph queries. Chapter 3 introduces various policies of faster convergence to further optimize MultiLyra. Chapter 4 presents BEAD that extends MultiLyra applicability in the presence of evolving analytics demands where both the input graph and the running batch of queries can evolve over time. Chapter 5 discusses various related works in the literature. Finally, Chapter 6 concludes the thesis and discusses directions for future work.

Chapter 2

MultiLyra: Distributed Batching Technique

For our proposed system, *MultiLyra*, we first introduce a *Basic* batching algorithm that maintains a unified list (i.e., frontier) 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 GAS model of 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. Further to improve *Basic*, 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. 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 the 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 on 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 [38], the only other system that is capable of distributed batched processing of iterative queries. Our results show that *MultiLyra* outperforms Quegel substantially due to its superior scalability.

The rest of this Chapter is organized as follows. In Section 2.1 we present the detailed design of *MultiLyra* including *Basic*, *FQT* and *IQT*, and *Reuse* algorithms. Section 2.2 carries out a detailed evaluation. Concluding remarks are given in Section 2.3.

2.1 Distributed 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 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. 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 [24] model of computation, our approach also applies to systems that employ the asynchronous computation model such as Grace [29, 42], Aspire [26], and Coral [28]. 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.

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 2.1 shows the GAS model in *MultiLyra* which is implemented by the five steps in processing a batch of queries in each iteration. Next we present the details of the phases for Basic batching algorithm of *MultiLyra* that maintains a single unified active list such that a vertex is active if it is active for at least one of the queries in the batch being evaluated.

2.1.1 Unified Frontier: Basic Batching

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. As illustrated by Algorithm 2.1, phases of GAS model

Algorithm 2.1 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
```

in *Basic MultiLyra* are as follows. 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).

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 2.2). 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 2.1, 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

Algorithm 2.2 Gather-Batch in Basic MaltiLyra

```
1: function GATHER-BATCH
2:     for all master or mirror vertices $v \in unified\_active\_list$ do
3:         for $in\_edge \in v.incoming\_edges()$ do
4:             ▷ gathering data for all queries
5:             for all $qid \in batch\_qlist$ do
6:                 $G\text{-}Data[\,] \leftarrow G\text{-}Data[\,]$ + PredData($qid, in\_edge$)
7:             end for
8:         end for
9:         $gathered\_data[v.id] \leftarrow G\text{-}Data[\,]$
10:        ▷ sending gathered data for all queries
11:        if $v$ is a mirror of a remote master then
12:            Send_G-Data_to_master($v$, $G\text{-}Data[\,]$)
13:        end if
14:    end for
15: end function
```

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 data values for all queries in the executing batch using the Compute

Algorithm 2.3 Apply-Batch in Basic MultiLyra

```
1:  function APPLY-BATCH
2:    for master vertex $v \in$ $unified\_active\_list$ do
3:      $changed \leftarrow$ false
4:      ▷ computing data for all queries
5:      for all $qid \in batch\_qlist$ do
6:        $new\_value \leftarrow$ Compute($gathered\_data[v.id]$, $qid$)
7:        if Change($v.data[qid], new\_value$) is ture then
8:          $v.data[qid] \leftarrow new\_value$
9:          $changed \leftarrow$ 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 $qid \in batch\_qlist$ do
16:         $message \leftarrow message \cup v.data[qid]$
17:       end for
18:       $message \leftarrow message \cup active$
19:       Send_A-Mix_to_mirrors($v$, $message$)
20:     end if
21:   end for
22: end function
```

function for the graph algorithm (see Algorithm 2.3). 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 2.4).

Scatter-Batch - Any updated vertices which have changed at least for one of the queries during the apply phase (Algorithm 2.3 – 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 2.4). 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

Algorithm 2.4 Building the unified active list for Basic MultiLyra

```
1:  ▷ done for any master/mirror by Scatter-Batch
2:  for any vertex $v$ which got changed do
3:      while $s \in v.succ()$ do
4:          $s.mark \leftarrow$ 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:     $unified\_active\_list \leftarrow unified\_active\_list \cup v$
15:     if !$v.mirrors.empty()$ then
16:         Send_G-Active_to_mirrors($v$);
17:     end if
18: end for
```

(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 2.4). 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 2.4). 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.

Above 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. Next we discuss the amortization effects obtained by the above batching system. the two important amortizations are as follows:

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

Table 2.1: 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)

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 2.1 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 2.1, 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, Basic MultiLyra significantly reduces the number of communications but can increase the sizes of messages by up to a factor of n.

2.1.2 Query Tracking: FQT & 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

Table 2.2: 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

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 2.1 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 2.2 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 2.1 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

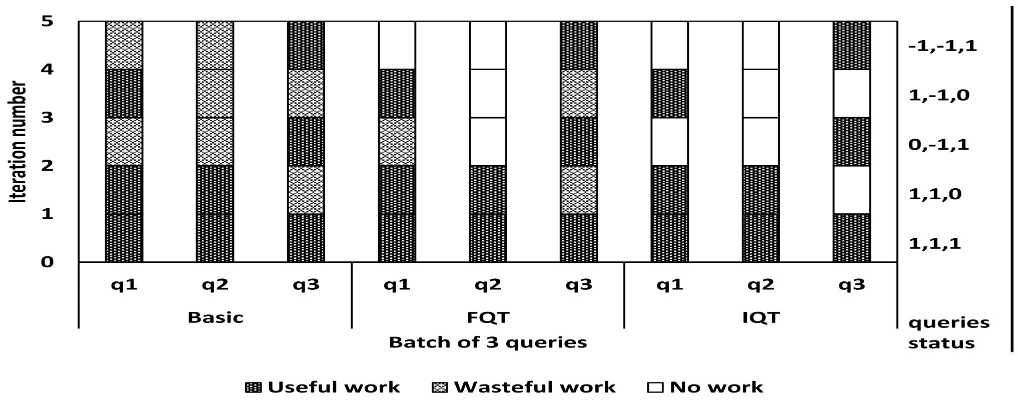


Figure 2.1: Amount of wasteful work in five iterations of running a batch of three queries (q1, q2, q3) concurrently for a vertex v on MultiLyra versions. Query status 1 or 0 indicate whether vertex v is active or inactive for that query in the current iteration and query status -1 indicates that query is finished.

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.

Algorithms 2.5 and 2.6 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 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 2.2 instead of batch_qlist). Algorithms 2.7 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

Algorithm 2.5 Modification in Apply-Batch algorithm for FQT

```
1: function APPLY-BATCH
2:     for master vertex $v \in$ *unified_active_list* do
3:         *changed* $\leftarrow$ false
4:         ▷ computing data only for unfinished queries
5:         for *qid* $\in$ *unfinished_qlist* do
6:             *new_value* $\leftarrow$ Compute(*gathered_data[v.id]*, *qid*)
7:             if Change(*v.data[qid]*, *new_value*) is ture then
8:                 *v.data[qid]* $\leftarrow$ *new_value*
9:                 *changed* $\leftarrow$ 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* $\in$ *unfinished_qlist* do
18:                *message* $\leftarrow$ *message* $\cup$ *v.data[qid]*
19:            end for
20:            *message* $\leftarrow$ *message* $\cup$ *active*
21:            Send_A-Mix_to_mirrors(*v*, *message*)
22:        end if
23:    end for
24: end function
```

Algorithm 2.6 Modification in Apply-Batch algorithm for IQT

```
 1: function APPLY-BATCH
 2:   for master vertex $v \in$ *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
```

Algorithm 2.7 Modification in algorithm of building the active list for FQT

```
1:  ▷ done for any master/mirror by Scatter-Batch
2:  for any vertex $v$ which got changed do
3:      while $s \in v.succ()$ do
4:          $s.mark \leftarrow$ 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:     $unified\_active\_list \leftarrow unified\_active\_list \cup v$
15:     $unfinished\_qlist \leftarrow$ 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
```

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 2.6 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

Algorithm 2.8 Modification in algorithm of building the active list for IQT

```
1:  ▷ done for any master/mirror by Scatter-Batch
2:  for any vertex $v$ which got changed do
3:      while $s \in v.succ()$ do
4:          $s.mark \leftarrow$ 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 $\leftarrow$ 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:     $unified\_active\_list \leftarrow unified\_active\_list \cup v$
16:     $active\_qlist[v.id]$ $\leftarrow$ activeq_bitset;
17:     if !$v.mirrors.empty()$ then
18:         Send_G-Active_to_mirrors($v$);
19:     end if
20: end for
```

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 2.6, 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 2.8, line 11). Finally, now all the machines have the tracking information, the active_qlist is constructed (line 16 of Algorithm 2.8.

2.1.3 Optimization: Simple Distributed Reuse

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.

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 $\underline{m_h}$ has the current value, it is responsible for sending the value to other machines to ensure all the machines have required data

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]*)

Figure 2.2: Reuse equations to update vertices in Reuse-Batch.

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 2.2 shows the reuse equations for SSSP, SSWP and Viterbi [9].

2.2 Evaluations

In this Section we evaluate the proposed different version of MultiLyra. We starts with describing the experimental setup, then we evaluate the Basic MultiLyra before evaluating FQT, IQT, and Reuse optimization. Finally we compare MultiLyra's speedup with the one achieve by the related work Quegel [38].

2.2.1 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),

Table 2.3: 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) [9]	Float

Table 2.4: Real world input graphs.

Input Graph	**#Edges**	**#Vertices**	**#Queries**
Twitter (TT) [5, 8]	2.0B	52.6M	1K
LiveJournal (LJ) [6, 10]	69M	4.8M	1K

and Viterbi (VT) [9] (see Table 2.3). We use two input graphs listed in Table 2.4 - 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.

Table 2.5: Running 1024 queries on the No-Batching Baseline framework (PowerLyra).

G	Algorithm	Exch-msg	Recv-msg	Gather	Apply	Scatter	Sync	Exe. 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%	36,246
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%	14,287

Table 2.6: 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×

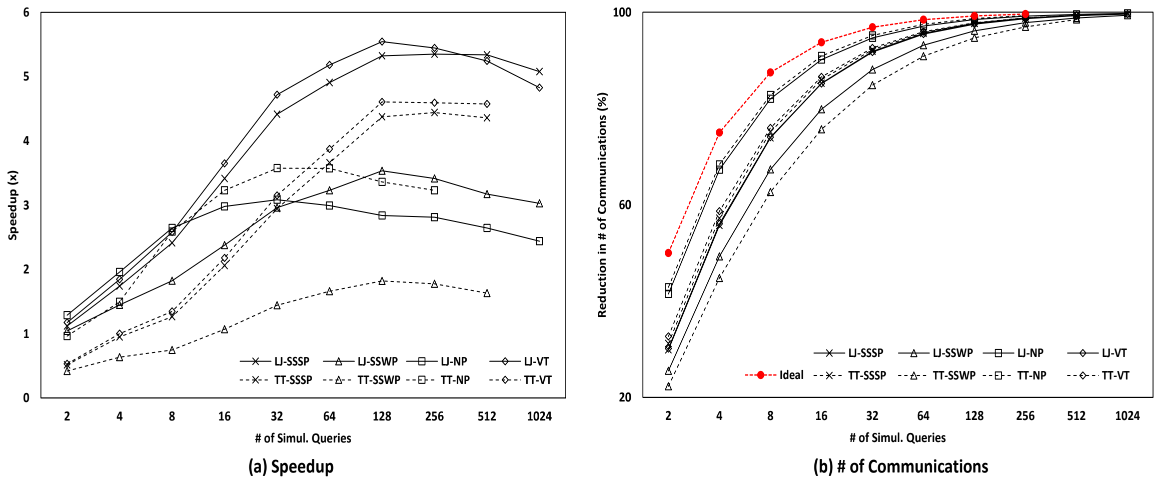


Figure 2.3: Speedup (a) and the percentage of reduction in number of communications (b) when running 1K queries with different batch sizes for four algorithms(SSSP, SSWP, NP and VT) on two input graphs, Twitter (TT) and LiveJournal (LJ)

2.2.2 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 2.5 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.

The speedups obtained by Basic MultiLyra are shown in Figure 2.3(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 2.3(b) shows the percentage of reduction in the number of communications when the

Table 2.7: 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×

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 2.3(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 2.6 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 2.7 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 2.8 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 2.8: Percentage of the number of communication in Data Category needed in Gather and Apply phases.

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

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.

2.2.3 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 2.9 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%

Table 2.9: The percentage of time on average for which a completed query performs wasteful processing.

G	Algorithm	#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%

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 2.10. As we expected from Table 2.9, 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

Table 2.10: IQT and FQT in details for the selected batch sizes and their speedup over the baseline (Table 2.5).

G	Algorithm	#Batch	V	Exch-msg	Recv-msg	Gather	Apply	Scatter	Sync	Seedup	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×	

each vertex in an iteration. We repeated the same experiment for IQT. Table 2.10 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.

2.2.4 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 2.4 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 2.11.

Table 2.11: 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×

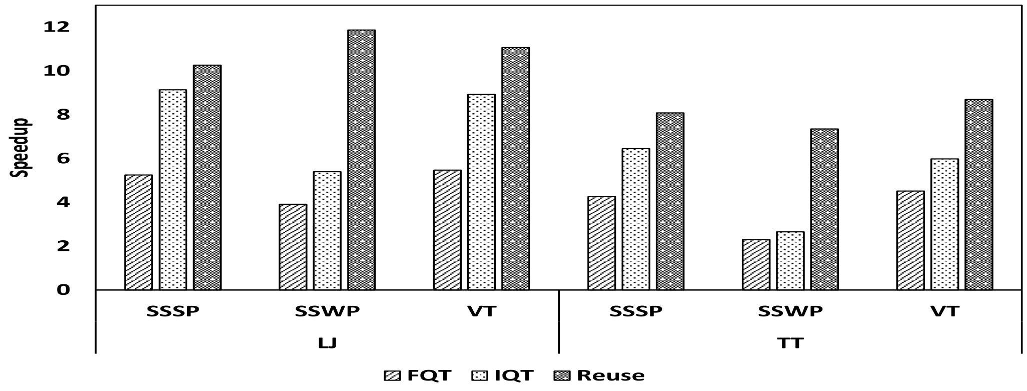


Figure 2.4: Reuse vs. FQT and IQT.
The NP algorithm is omitted because application of Reuse to NP is unsafe.

2.2.5 IQT Batching Vs. Quegel Batching

Quegel [38] 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 [43] (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 IQT.

The speedups obtained by IQT batching over Quegel batching are given in Table 2.12 for different batch sizes. As we can see, the speedups of IQT over Quegel increase as batch 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.

Table 2.12: Speedups of IQT batching over Quegel batching on a four machine cluster.

G	Algorithm	#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×

2.3 Summary

In this chapter, 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 the related-work Quegel.

Chapter 3

ExpressWay: Faster Convergence Technique

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. 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 chapter, 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.

3.1 Background Review

Now that real-world graphs are huge (e.g., *Friendster* [15] 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.

3.1.1 Single Query: Gemini

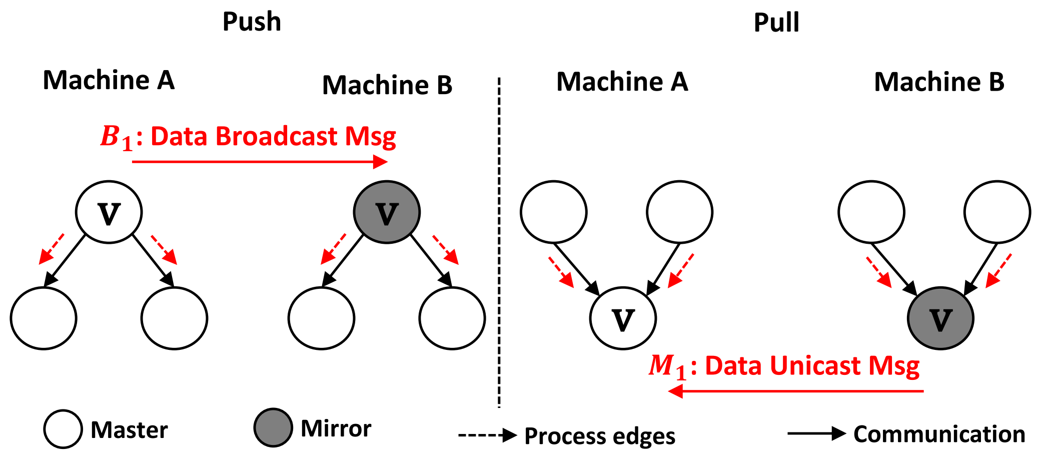


Figure 3.1: Communication Pattern in Gemini for Push and Pull Modes. In Push, the Updated Value of Vertex v is Sent to All Machines No Matter Whether They Need It Or Not. Gemini Overlaps Communications with Computations Hiding Message-Passing Overhead.

Table 3.1: Running 10 queries on the single-query Baseline framework (Gemini) using different modes (i.e., push-pull, push-only, and pull-only).

		Gemini: Time (seconds)		
G	**Algorithm**	**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

Many frameworks have been developed to run a single graph query efficiently in different platforms. 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$\times$ 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). Next, we discuss how *Gemini* operates in terms of computation and communication for each of these modes (see Figure 3.1).

– *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 3.1 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 to their respective outgoing remote neighbors, as depicted in Figure 3.1, 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 3.3.

– *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 3.1, 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 3.1 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 3.4 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*.

3.1.2 Multiple Queries: MultiLyra

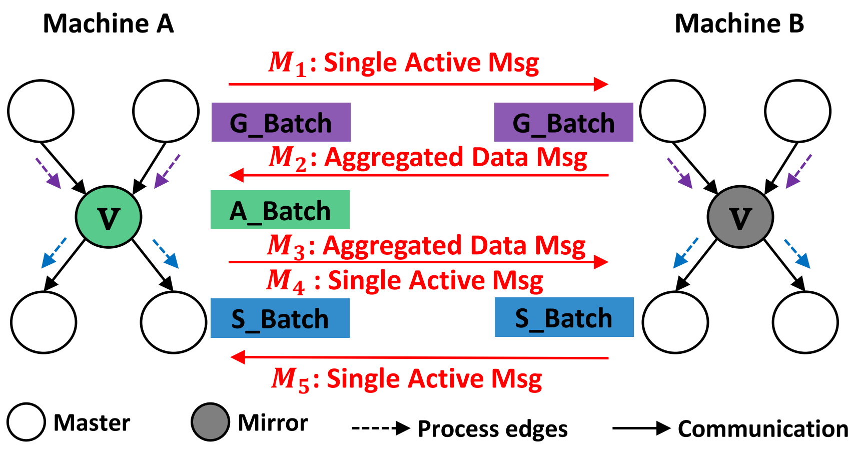


Figure 3.2: Communication Pattern in MultiLyra GAS Model: Five Messages are Needed for Each Vertex Processing; Two of Them Carry Vertex Values and Three are Active Messages. MultiLyra Does Not Overlap Communications with Computations

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 3.2, 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 3.2), 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

Table 3.2: Running 10 queries concurrently on the batching Baseline framework (MultiLyra) using different modes (i.e., Basic, FQT, and IQT).

		MultiLyra: Time (seconds)		
G	**Algorithm**	**Basic**	**FQT**	**IQT**
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

data message, based on one of the modes Basic_batch, FQT, and IQT (explained in the next paragraph), including the data for all the active queries (*M2* in Figure 3.2). 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 3.2). 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 3.2).

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* 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* fits better for the midsize batches. Finally, *IQT* 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 3.2 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.

3.2 Distributed 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 distributed scenarios that are proposed aiming at utilizing the most benefit that Highways can offer. Finally we explain the *ExpressWay* setup in various frameworks with an example and algorithm. The method to identify the highways is similar to Core Graph proposed in [55] and [56] with one key difference. Highways in *ExpressWay* are marked by reordering the edges of the same graph unlike the Core Graph idea in which two versions of the input graph are loaded to the system.

3.2.1 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. The author in [55] shows that we only need a few number of high-degree vertices and having more than 20 high-degree vertices doesn't significantly improve the effectiveness of highways in making more vertices to converged to their final value. Instead, it unnecessarily increases the number of highways in the graph. Therefore, We picked 20 high-degree vertices for building the highways. To determine these 20 high-degree vertices, we sorted all vertices based on their degrees and select the top 20.

Forward Query Evaluation - After identifying the 20 high-degree vertices, we apply our algorithm to these vertices on 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 in regards to only highways. We examine all the

Algorithm 3.1 Identifying Highways on a Given Graph: Finding High-Degree vertices

```
1:  $D[V]$: array for collecting degree of each vertex
2:  $H$: high-degree vertex set
3:  for each $v \in V$ do
4:      $D[v]$ = OutDegree($v$) + InDegree($v$)
5:  end for
6:  $H$ = Index of 20 high values on array $D[V]$
```

Algorithm 3.2 Identifying Highways on a Given Graph: Forward/Backward Evaluations

```
1:  ▷ Forward Query Evaluation
2:  for each $h \in H$ do
3:      $E_{forward}(h)$ = SOLVE ( $G(V, E)$, DIRECTION $f$ )
4:      $E_{highways}$ = $E_{highways} \cup E_{forward}(h)$
5:  end for
6:
7:  ▷ Backward Query Evaluation
8:  for each $h \in H$ do
9:      $E_{backward}(h)$ = SOLVE ( $G(V, E)$, DIRECTION $b$ )
10:     $E_{highways}$ = $E_{highways} \cup E_{backward}(h)$
11: end for
```

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 the edges of the input graph reordered. Considering the input graph with only highways creates a hypothetical graph that retains the same

Algorithm 3.3 Identifying Highways on a Given Graph: Solve Function to create the hypothetical smaller graph.

```
1: Input: Graph $G(V, E)$
2: Output: $G(V, E_{highways})$; $E_{highways}$ contains $highways$
3:
4: function SOLVE ( $G(V, E)$, DIRECTION $d$ )
5:     Evaluate Query $Q(s)$ on $G(V, E)$
6:     for all $e(u, v) \in E$ do
7:         if $Q(s)$ updates $Q(s).Val(u)$ then
8:             if $(Q(s).Val(u) \bigoplus w(u, v) = Q(s).Val(v))$ then
9:                 if $(d == f)$ then
10:                    $E_{highways}(h) = E_{highways}(h) \cup \{ e(u, v) \}$
11:                else ▷ ( $d == b$ )
12:                    $E_{highways}(h) = E_{highways}(h) \cup \{ e(v, u) \}$
13:                end if
14:            end if
15:        end if
16:    end for
17: end function
```

Algorithm 3.4 Identifying Highways on a Given Graph: Connectivity of the Highways.

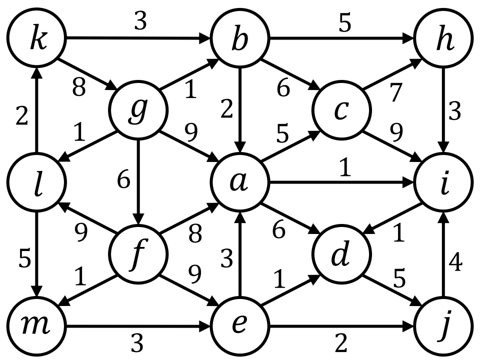
```
1: for all $v \in V$ do
2:     if (OutDegree(v)$\neq$ 0) $\wedge$ (OutEdges(v) $\cap$ $E_{highways}) = \phi$ then
3:         Add an out edge of $v$ to $E_{highways}$
4:     end if
5: end for
```

vertices as the original but has a significantly reduced number of edges, now termed highways. In Algorithms 3.1 to 3.4, 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 (see Algorithm 3.1). According to algorithm 3.2, after pinpointing the twenty high-degree vertices, a forward query evaluation is performed (see Algorithm 3.2 lines 1-5). For each vertex in our high-degree vertex set H, the $Solve$ function from Algorithm 3.3 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 (see Algorithm 3.2 lines 7-11). Again, the $Solve$ function from Algorithm 3.3 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 through the highways. We iterate over the graph's vertices. If there is a vertex to which no highways in the $E_{highways}$ set is connected, an outgoing edge of that vertex will be added to $E_{highways}$ to ensure connectivity of highways (see Algorithm 3.4). The resulting graph contains all the highways reordered in the beginning of the edge list for each vertex, enabling accelerated distributed graph processing. The $Solve$ function in Algorithm 3.3 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 will be added to $E_{highways}$, considering both forward and backward directions.

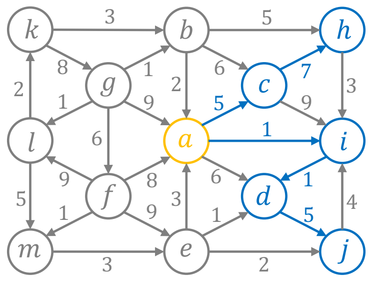
3.2.2 An Example To Demonstrate ExpressWay Construction

Let us demonstrate Algorithms 3.1 to 3.4 using an example. As you can see in Figure 3.3(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 3.3(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 3.3(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 3.3(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 the Figure 3.4, 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



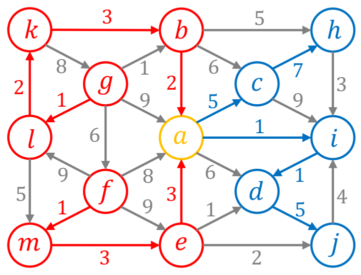
(a) Full Graph.



SSSP Results – Forward Direction

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

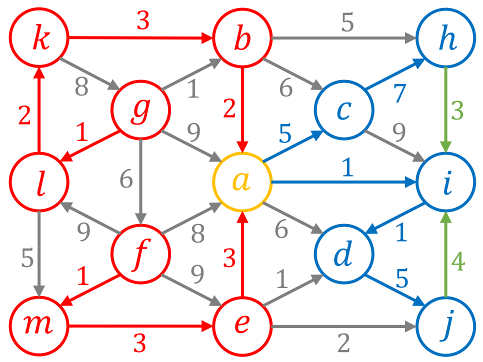
(b) Forward Query Evaluation for Vertex a.



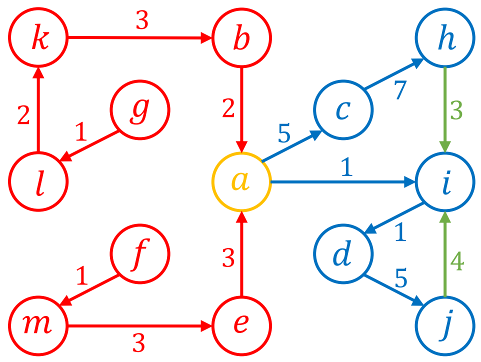
SSSP Results – Backward Direction

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.

Figure 3.3: Example to Show the Steps for Identifying the Highways on a Graph for the Single Source Shortest Path Algorithm and High-degree Vertex a.

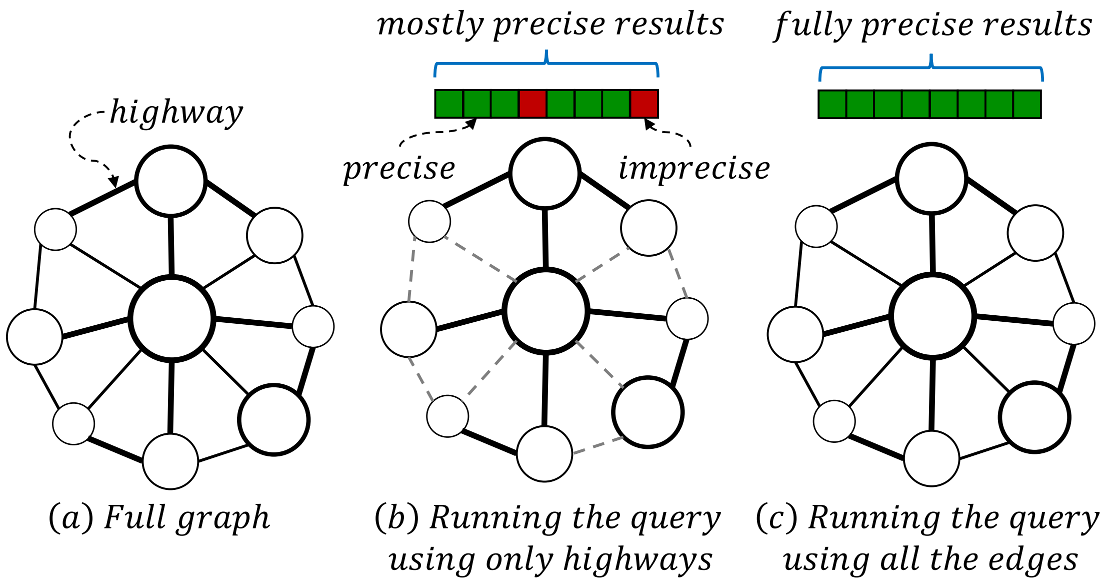

mostly precise results
fully precise results
highway
precise
imprecise
(a) Full graph
(b) Running the query using only highways
(c) Running the query using all the edges

Figure 3.4: Query Evaluation Using the Highways in a Graph.

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.

3.2.3 ExpressWay Plocies & Scenarios

We proposed four types of policies for using the expressway in the graph.

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, 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, ExFSin combines ExWaySin and ExWayFull. Please note that running graphs fully on highways in a single ma-

Algorithm 3.5 *Expressway* employed in the Gather phase of the MultiLyra GAS model. Similarly, it applies to the Pull mode 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 $q \in active\_queries\_for(v)$ do
12:         for $e \in edge\_list$ 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
```

chine is fast enough not imposing any communication cost. Therefore, combining ExWaySin with ExWayHalf is not feasible.

Algorithm 3.6 *Expressway* employed in the Scatter phase of the MultiLyra GAS model. Similarly, it applies to Push mode for the single query platform assuming batch_size is 1.

```
1: ▷ Expressway for scatter/push
2: Input: active vertex v, Expressway_Enable
3: Output: makes the next frontier
4:
5: function S_BATCH ( $v$, Expressway_Enable)
6:     edge_list = out_edges_of($v_i$)
7:     if Expressway_Enable then
8:         edge_list = highways_of($v_i$, out_edge=true)
9:     end if
10:    for $q \in active\_queries\_for(v_i)$ do
11:        for $e \in edge\_list$ do
12:            if ($e$.dst().value[$q$] $\bigoplus$ agg($v$.value[$q$], $e$.data())) then
13:                active_list ← $e$.dst().id
14:            end if
15:        end for
16:    end for
17: end function
```

3.2.4 ExpressWay Setup

Algorithms 3.5 and 3.6 show 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 these algorithms in the MultiLyra batching GAS model while one can similarly apply *ExpressWay* to the Pull/Push modes of 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).

Algorithm 3.7 *Expressway* employed in Main loop of the MultiLyra GAS model. Similarly, it applies to Main loop for the single query platform assuming batch_size is 1.

```
1:  ▷ Main loop
2:  Input: G, Expressway_Enable=$true$, ExHalf=$false$, i_threshold
3:  Output: The final result for the running algorithm
4:
5:  function Run( G, Expressway_Enable, ExHalf)
6:      i ← 0
7:      while !active_list.empty() do
8:          if Highway.isDone() or (ExHalf and i=i_threshold) then
9:              Expressway_Enable = $false$
10:             active_list ← all $v$.ids visited
11:         end if
12:         ▷ run in parallel for each active $v$
13:         collected_data = G_batch($v$, Expressway_Enable)
14:         A_batch($v$, collected_data[$v$])
15:         S_batch($v$, Expressway_Enable)
16:         i++;
17:     end while
18: end function
```

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 Algorithms 3.5 and 3.6, lines 7-9 for Gather and Scatter, respectively). 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 in Algorithms 2.2. 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 3.3. 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 14 of Algorithm 3.7. Scatter will also loop over the highways only when the $Expressway\_Enable$ flag is set to $true$, as shown in Algorithm 3.6, lines 7-9. 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 13 of Algorithm 3.6.

Finally in Algorithm 3.7 line 8 - 11, it 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 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.

3.3 Evaluations

To Evaluate the proposed faster convergence technique, we first analyze *ExpressWay* using, Gemini, the most efficient single query framework thoroughly. Then, we pick the least and the

Table 3.3: Equations used to aggregate the data to update vertex v for any query which propagated through incoming edge e coming from the incoming neighbor u (i.e., its source). Algorithms: SSSP-single source shortest path; SSWP-single source widest path; SSNP - single source narrowest path; and VT - Viterbi.

Algorithm	**Aggregating Equation**
SSSP	*v.vlaue* = Min (*v.value*, *u.value* + *e.data*)
SSWP	*v.vlaue* = Max (*v.value*, Min (*u.value* , *e.data*))
SSNP	*v.vlaue* = Min (*v.value*, Max (*u.value* , *e.data*))
VT	*v.vlaue* = Max (*v.value*, *u.value / e.data*)

Table 3.4: Real-world input graphs along with their number of vertices and the number of edges.

Input Graph	**#Edges**	**#Vertices**
Twitter WWW (TTW) [8]	1.5 B	41.6 M
Twitter MPI (TT) [5]	2.0 B	52.6 M
Friendster (FS) [15]	2.6 B	68.3 M

most speedup delivery scenarios to apply it on MultiLyra to evaluate it in the context of batching. We picked the basic batching version of MulitLyra since we consider small batches of 10 queries.

3.3.1 Experimental Setup

We implemented *ExpressWay* using Gemini [3] which advances the distributed graph processing via its NUMA-Aware design for a single query, and MultiLyra [17] 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) [9] (see Table 3.3). We use three large input graphs listed in Table 3.4, with a billion edges named TTW, TT, and FS. For each input graph and for each al-

Table 3.5: Speedup of running 10 random SSSP queries on highway only vs. full edges.

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

gorithm, 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., a total of 64 threads), 256 GB memory, and runs Rocky Linux release 8.5.

3.3.2 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 3.5 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 vertex when using only *highways*. The column "ExWayFull" on Table 3.6 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 3.6, column "Highway/Edge".

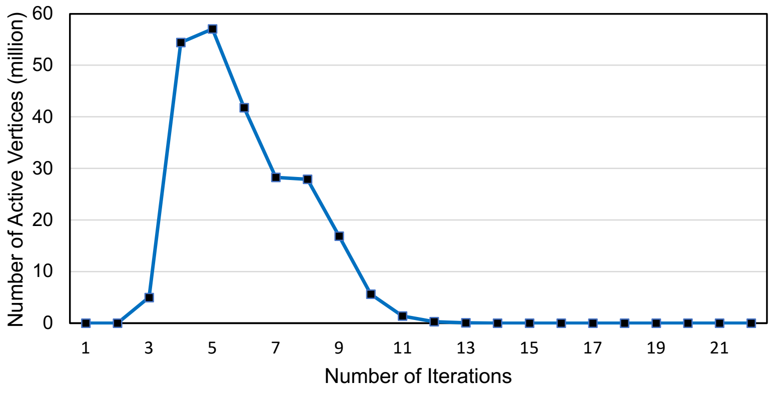


Figure 3.5: The Frontier Size Over the Iterations of a SSSP Query on TTW.

Therefore, the speedups displayed in Table 3.5 represent the ideal performance gains that *ExpressWay* policies aim to converge toward. Table 3.7 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 3.7 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 3.5 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 3.6 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 3.5 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 3.6 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. When the evaluation on highways is finished in each machine before transferring to the the full edges, master and mirror vertices need to exchange their value in favor of consistency to make sure all the veritces have their most updated value.

Table 3.6: The ratio of vertices that get converged to the final result by only running on Highways. The average results of running 10 random queries.

G	Algorithm	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

Table 3.7: Reduction in Number of updates, communications, and the edges processed in Gemini when running 10 random SSSP queries on the input graphs using all the edges vs. highways only.

		$\times 10^9$		
G	**# of**	**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×

3.3.3 Single Query: Gemini

Table 3.8: ExpressWay speedup over Gemini when running 10 random queries one by one.

		Time (s)	**ExpressWay: Speedup**		
G	**Algorithm**	**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×

We applied the three scenarios discussed above, i.e., *ExFDis*, *ExHDis*, and *ExFSin*, and implemented *ExpressWay* as discussed in Section 3.2.4 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 3.8 shows the execution time in seconds for the baseline (*Gemini*) as well as the speedups achieved by the *Express-Way* 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 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 3.8, delivers better speedups for VT on FS.

3.3.4 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 3.3, to the TTW and TT graphs. Table 3.9 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.

Table 3.9: ExpressWay speedup over MultiLyra when running a batch of 10 random queries concurrently.

		Time (s)	**ExpressWay: Speedup**	
G	**Algorithm**	**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×

3.4 Summary

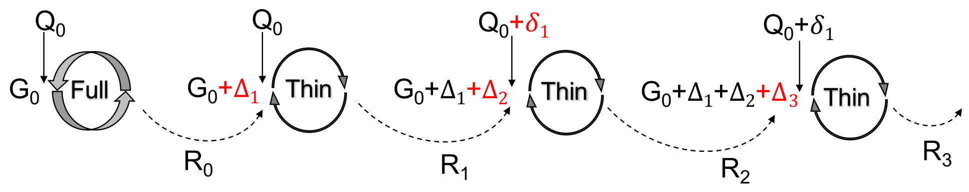
In this chapter, we presented *ExpressWay*, a technique to determine the important edges called *Highways* in a large graph, aiming at accelerating the distributed evaluation of iterative graph queries. Highways are selected from those edges that deliver the aggregated final value between their endpoints. 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.

Chapter 4

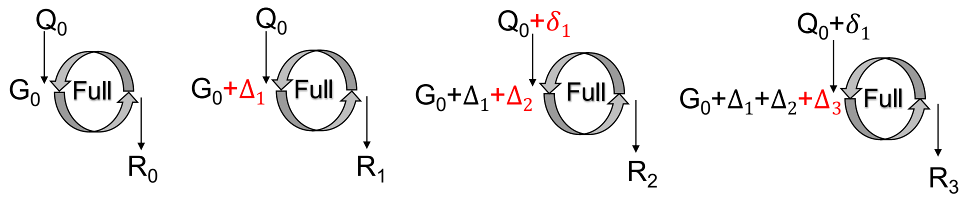
BEAD: Incremental Evaluation Technique

Simultaneous evaluating a batch of iterative graph queries on a distributed system enables amortization of high communication and computation costs across multiple queries. As demonstrated in Chapter 2, *MultiLyra*, batched graph query processing can deliver significant speedups and scale up to batch sizes of hundreds of queries.

In this chapter, we greatly expand the applicable scenarios for batching by developing *BEAD*, a system that supports Batching in the presence of Evolving Analytics Demands. First, *BEAD* allows the graph data set to evolve (grow) over time, more vertices (e.g., users) and edges (e.g., interactions) are added. In addition, as the graph data set evolves, *BEAD* also allows the user to add more queries of interests to the query batch to accommodate new user demands. The key to the superior efficiency offered by *BEAD* lies in a series of incremental evaluation techniques that leverage the results of prior request to "fast-foward" the evaluation of the current request.



(a) BEAD: Efficient Incremental Evaluations.



(b) MultiLyra: Independent Full Evaluations.

Figure 4.1: *BEAD* vs. *MultiLyra*: evaluating the sequence of requests (G_0, Q_0), $(G_0 + \Delta_1, Q_0)$, $(G_0 + \Delta_1 + \Delta_2, Q_0 + \delta_1)$, $(G_0 + \Delta_1 + \Delta_2 + \Delta_3, Q_0 + \delta_1)$, producing results R_0, R_1, R_2, and R_3.

Let $Eval(G, Q) \to R$ denote the evaluation of a batch of queries Q on graph G with results R, a basic functionality of *MultiLyra*. Next, we explain how *BEAD* generalizes the capabilities of *MultiLyra* by efficiently handling the following evolving analytics demands:

(i) Growing Graph. Assume the initial graph is G_0, the initial query batch is Q_0, and their evaluation $Eval(G_0, Q_0)$ yields results R_0. Then, the graph grows with a set of additions Δ, which may include new edges and/or vertices. Instead of fully reevaluating Q_0 on the new graph $G_0 + \Delta$ as in *MultiLyra*, *BEAD* exploits prior results R_0 to incrementally evaluate $G_0 + \Delta$, denoted as $Inc(G_0 + \Delta, Q_0, R_0)$.

(ii) Growing Graph and Query Batch. In addition to the growing graph $G_0 + \Delta$, new queries of interests δ are also added to the query batch occasionally, that is, $Q_0 + \delta$. Even in this scenario, *BEAD* can still manage to leverage prior results R_0 to streamline the full evaluation

$Eval(G_0{+}\Delta, Q_0{+}\delta)$ to an incremental evaluation $Inc(G_0{+}\Delta, Q_0{+}\delta, R_0)$, without compromising the correctness of the results.

Figure 4.1 illustrates how a sequence of requests are handled by *BEAD* and how they can be evaluated, though inefficiently, using *MultiLyra*. In this example, after the initial evaluation of queries on the original graph, first reevaluation is required due to changes in graph (Δ_1), then due to changes in both graph and query batch (Δ_2 and δ_1), and finally due to changes in the graph (Δ_3). In Figure 4.1(a), *BEAD* efficiently evaluates the queries by taking advantage of the query results from the prior evaluation, while in Figure 4.1(b) *MultiLyra* evaluates queries on the corresponding graphs independently from scratch.

Furthermore, in the presence of new user request while the old is still being processed (i.e., user interruption), instead of waiting for the previous request to finish, *BEAD* permits anytime evaluation of the new request $Eval(G_0{+}\Delta, Q_0{+}\delta)$ using the unconverged results from $Eval(G_0, Q_0)$, denoted as $\approx R_0$. That is, before the convergence of Q_0 on G_0, user may interrupt the evaluation, make additions to the graph and the query batch, and carry out new evaluation incrementally.

We have developed a prototype of *BEAD* that builds upon the *MultiLyra* prototype and compared their performance on multiple input graphs and multiple kinds of graph queries. Experiments demonstrate that *BEAD*'s batched evaluation of 256 queries, following graph changes that add up to 100K edges to a billion edge Twitter graph and also query changes of up to 32 new queries, outperforms *MultiLyra*'s batched evaluation by factors of up to 26.16$\times$ and 5.66$\times$, respectively.

4.1 MultiLyra Review

Distributed platforms offer a promising way for processing large real-world graphs by partitioning the input graph across a cluster of machines. *MultiLyra* [17] adopts the hybrid-cut graph partitioning strategy first introduced by *PowerLyra* [2]. For low-degree vertices, it distributes the vertices along with their edges evenly among machines (i.e., edge-cut), such that these vertices can be processed locally. In comparison, for high-degree vertices, it distributes their edges evenly among machines (i.e., vertex-cut) to better balance the workload. When a high-degree vertex is partitioned and replicated across multiple machines, one of the replicas is selected as the master and the rest become the mirrors.

4.1.1 Basic GAS Model

MultiLyra follows the GAS (**G**ather-**A**pply-**S**catter) model first introduced in *PowerGraph* [1] to perform BSP-style [24] iterative graph computations. For simultaneously processing a batch of queries, like the SSSP queries as shown below:

$$\{\text{SSSP}(v_1),\ \text{SSSP}(v_2),\ \cdots,\ \text{SSSP}(v_n)\}$$

MultiLyra maintains a unified active list such that a vertex is active if it is active for at least one of the queries in the batch. For a single active vertex, it integrates the processing of all queries in the batch to amortize the overhead across queries.

Algorithm 4.1 summarizes the iterative algorithm. First, it initializes the `unified_activeList` based on the given batch of queries (Line 3). Line 4-10 initialize the status of each query. After initializing the vertex values at line 12-14, the main loop of iterative graph processing is shown at

Algorithm 4.1 Batching in MultiLyra – The GAS model.

```
1:  function EVAL($G, Q, mode$)
2:      ▷ Initialize the unified list & query status of active vertices
3:      $unified\_activeList \leftarrow \langle q_1, q_2, ..., q_n \rangle \in Q$
4:      if $mode$ == $IQT$ then
5:          ▷ $S_i$ is a bitset indicating active queries for vertex $i$
6:          $q\_status \leftarrow \langle S_1, S_2, ..., S_N \rangle$ where $S_{q_i}.set\_bit(i)$
7:      else                                                          ▷ mode = FQT
8:          ▷ $s_i$ is a bit indicating if query $i$ is still unfinished
9:          $q\_status \leftarrow \langle s_1, s_2, ..., s_n \rangle$ where $s_i = 1$
10:     end if
11:     ▷ Initialize the vertex values
12:     for each vertex $v \in G$ do
13:         $R[v][] \leftarrow INIT\_VAL$
14:     end for
15:     while !$unified\_activeList.empty()$ do
16:         Exch_Batch($q\_status$)                                   ▷ $S_1$
17:         $unified\_activeList, q\_status \leftarrow$ Recv_Batch()    ▷ $S_2$
18:         Gather_Batch($q\_status, mode$)                            ▷ $S_3$
19:         Apply_Batch($q\_status, mode$)                             ▷ $S_4$
20:         Scatter_Batch($q\_status$)                                 ▷ $S_5$
21:     end while
22:     return $R$
23: end function
```

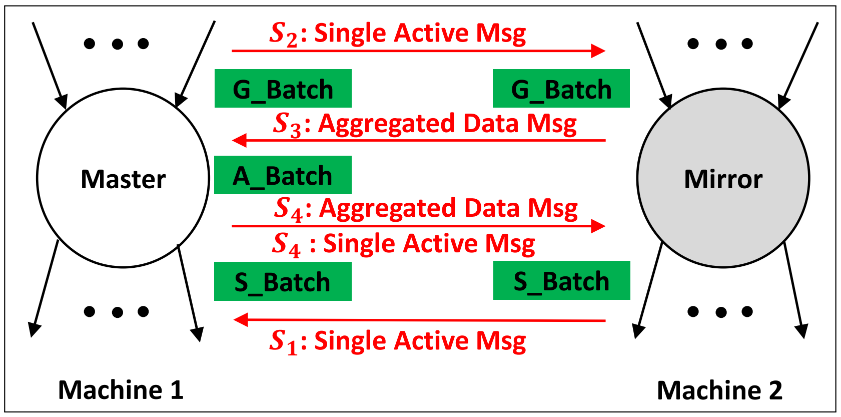


Figure 4.2: Communication needed in steps S_1 through S_5 for an active vertex in one iteration of MultiLyra. Please note since Scatter_Batch (i.e., S_5) marks the successors locally, it does not perform any communication.

line 15-21. At high level, there are five steps during each iteration of the processing, referred to as S_1 to S_5. Figure 4.2 shows the communications needed by the vertex in each step. Next we summarize the work performed by each step:

S_1: Exch-Batch - In this step, all the local mirrors that were scheduled to be active for the current iteration send an activation message to their masters that reside on the remote machines, so that the master would be informed to become active. Note that one single activation message is sufficient per local mirror to cover all the queries.

S_2: Recv-Batch - All the masters that are either informed by their local neighbors or through an activation message are added to the `unified_activeList`. After that, each of them sends one single activation message to their mirrors to inform them participating the gather phase.

So far, the first two steps have made an consensus on which vertices are active in the current iteration. Based on this, the next three steps perform the GAS operations.

S_3: Gather-Batch - All the active vertices, including both mirrors and masters, on each machine collect data along their incoming edges for all the queries in the batch. In addition, the mirrors send their portion of the locally gathered data to their masters, via a data message, so that the masters are aware of all the data they need globally for the Apply-Batch.

S_4: Apply-Batch - Based on the collected data from the last step, values of all vertices (except mirrors) are first updated (depending on the graph applications) for all the queries. Then, to maintain the consistency across machines, when a vertex value is updated by at least one of the queries, the vertex values of all queries are sent to their mirrors in one aggregated data message to reflect the updates. Along with this message, one single activation message is also sent to the mirrors for the following-up scattering step.

S_5: Scatter-Batch - All active vertices (including mirrors and masters) whose values have changed at least for one of the queries during S_4 inform (schedule) their out-neighbors for processing in the next iteration, which may include both local masters and local mirrors of remote vertices.

The above five steps form the basic version of *MultiLyra*. Given an active vertex, it performs integrated processing of all queries in each GAS phase, thus amortizing the overhead.

4.1.2 FQT & IQT

Note that, in the basic version, a unified active list is maintained, which does not distinguish the active vertices among queries and is unaware the completion of queries. This leads to wasteful computations and communications. To improve the efficiency, *MultiLyra* provides two optimized ways of tracking the status of queries and vertices:

- FQT tracks if any of the queries in the batch has been finished.

- IQT tracks which queries need to be evaluated in one iteration for each vertex.

V.data[] = $[\mathbf{d_{q_1}}, \mathbf{d_{q_2}}, \mathbf{d_{q_3}}, \mathbf{d_{q_4}}, \mathbf{d_{q_5}}, \mathbf{d_{q_6}}, \mathbf{d_{q_7}}, \mathbf{d_{q_8}}]$

$$\textbf{(a) FQT} \begin{cases} \mathbf{Q\_Status}: \langle U, F, U, U, U, F, U, U \rangle \\ \mathbf{Data\_Msg}: [\mathbf{d_{q_1}}, \mathbf{d_{q_3}}, \mathbf{d_{q_4}}, \mathbf{d_{q_5}}, \mathbf{d_{q_7}}, \mathbf{d_{q_8}}] \end{cases}$$

$$\textbf{(b) IQT} \begin{cases} \mathbf{Q\_Status[v]}: \langle I_v, F, I_v, U, I_v, F, I_v, U \rangle \\ \mathbf{Data\_Msg}: [\mathbf{d_{q_4}}, \mathbf{d_{q_8}}] + \mathbf{00010001} \end{cases}$$

Figure 4.3: Data Message Compression. F indicates the query has finished, U indicates the query has not finished and is still running, and I_v indicates that query is inactive for active vertex v in the current iteration.

These two fine-grained tracking methods enable *MultiLyra* to support compressed data messages, as shown in Figure 4.3. These strategies use one bitset or array of bitsets, for *FQT* or *IQT* respectively, to hold the statuses of queries in the running batch (Algorithm 4.1 lines 4-10) and keeping it updated along with the active list in *Recv_Batch*. Moreover, *FQT* and *IQT* can skip the vertex evaluation for inactive or finished queries.

In general, *IQT* works better for large batches of queries in the scenario where high number of queries have not finished but are inactive for a vertex. So the performance benefits of *IQT* can easily eclipse its tracking overhead. *FQT*, on the other hand, works more efficiently for the relatively smaller batch sizes where the opportunity to take advantage of inactive queries is relatively low; thus, it only tracks finished queries.

4.2 *BEAD*: Distributed Support Of Batching For Evolving Analytics Demands

Despite the promise of *MultiLyra* [17] in amortizing the communication and synchronization overheads across a batch of simultaneously-running queries, it is designed for only static scenarios, where the graph and queries are fixed. In this section, we introduce *BEAD* which generalizes *MultiLyra* and adapts its batching strategies to scenarios of evolving analytics demands – growing graphs and batch of queries.

4.2.1 Batching With Graph Updates

Given the full evaluation of query batch Q_0 on graph G_0: $Eval(G_0, Q_0) \rightarrow R_0$, this subsection describes how *BEAD* computes the results of Q_0 on the updated graph $G_0 + \Delta$, which are denoted as R_1. Instead of making another round of full evaluation, *BEAD* updates the existing results R_0 to obtain R_1, by performing lightweight incremental computation $Inc(G_0 + \Delta, Q_0, R_0)$ $\rightarrow R_1$, as described in Algorithms 4.2 and 4.3.

Basically, *BEAD* takes over the evaluation process from the most recent completed evaluation on graph G_0. First, it inserts the new edges and vertices in Δ to the graph (Algorithm 4.2 at Line 2) which yields a new graph G_1. Based on the insertions, it carefully initializes the vertex values, and updates the active vertex list and query status to reflect the addition of new edges and vertices (Algorithm 4.3). In specific, it first grows the vertex value array by the number of new vertices (Algorithm 4.3 at Line 3). Then, the two key initializing steps are performed.

First, the Vertex Data Values for all the existing vertices are initialized to their data value in R_0, which is passed on by the previous step in *BEAD* system as was shown in Figure 4.1(a).

Algorithm 4.2 BEAD: Reevaluating for Graph Update ($G_0 + \Delta$) - Inc().

```
1:  function Inc($G_0, Q_0, R_0, \Delta, mode$)
2:      $G_1 \leftarrow G_0$.update($\Delta$)                    ▷ Add new edges and vertices
3:      $R_1 \leftarrow$ Initialize($G_1, Q_0, R_0, \Delta, mode$)
4:      while !unified_activeList.empty() do
5:          ▷ ComputeIteration() performs steps $S_1$ through $S_4$ from Algorithm 4.1
6:          ComputeIteration(q_status, mode)
7:          Scatter_Batch(q_status)
8:      end while
9:      return $R_1$
10: end function
```

This allows them to achieve faster convergence to the final result. For all the new vertices from Δ, the vertex data values initialize to the INIT_VAL, defined by the graph algorithm (Line 4-10, Algorithm 4.3). Note that having R_0 as the initial data values for the existing vertices not only helps these vertices converge efficiently on the new graph but can also help the new vertices converge faster.

Second, the List of Active Vertices that are affected by Δ need to be added to the initial list of active vertices. Vertex v, either existing or new, is an affected vertex with respect to Δ if there is at least one new edge in Δ that points to v (Line 12-14, Algorithm 4.3). This guarantees that the evaluation continues towards re-convergence by scattering through the new edges. Note that all queries for these vertices need to be active as well which is accomplished by setting their corresponding bit to 1 in the query tracking bitset. In Algorithm 4.3 at line 16, the query tracking bitset array is updated for each affected vertex in case of *IQT*, and for *FQT*, at line 20, we only need

Algorithm 4.3 BEAD: Reevaluating for Graph Update ($G_0 + \Delta$) - Initialize().

```
1: function INITIALIZE($G_1, Q_0, R_0, \Delta, mode$)
2:     ▷ Initialize $R_1$
3:     $R_1.set\_size(G_1.num\_vertices())$
4:     for each vertex $v \in G_1$ do
5:         if $v \in \Delta$ then                                  ▷ v is a new vertex
6:             $R_1[v][] \leftarrow$ INIT_VAL
7:         else                                   ▷ Otherwise initialize $R_1$ using $R_0$
8:             $R_1[v][] \leftarrow R_0[v][]$
9:         end if
10:    end for
11:    ▷ Initialize unified_activeList with all the affected vertices
12:    for each edge $e \in \Delta$ do                      ▷ scatter through new edges
13:        $v \leftarrow e.dest()$
14:        unified_activeList ← unified_activeList ∪ $v$
15:        if mode == IQT then
16:            q_status ← q_status ∪ $S_v.set\_all()$
17:        end if
18:    end for
19:    if mode == FQT then                                ▷ Make all queries active
20:        q_status ← $\langle s_1, s_2, ..., s_n \rangle$ where $s_i = 1$
21:    end if
22:    return $R_1$
23: end function
```

to set all the query statuses to unfinished (i.e., 1) in the single bitset with the size of the number of queries (n).

After the above preparation, *BEAD* enters the evaluation loop, just like the full evaluation (Line 4-8 of Algorithm 4.2). To save space, the first four steps are combined into one function in Algorithm 4.2 (Line 6). We leave Step 5 out as some other code needs to be inserted later between Step 4 and Step 5 (see Section 4.2.2).

Mode selection - As mentioned earlier in Section 4.1.2, *IQT* works better for large batches of queries in the scenario where high number of queries have not finished but are inactive for a vertex, while *FQT* works more efficiently for the small batch sizes leveraging its negligible overhead, where the opportunity to take advantage of inactive queries is low. However, it is not quite the same case for the incremental evaluation when the graph evolves. In this scenario, another factor, the amount of changes to the graph Δ, in addition of the batch size is also important for proper mode selection. When Δ is small, even for large batches, fewer queries are made active for more iterations. This causes the fraction of finished queries in the batch of running queries to be large for most of the iterations. In this scenario, *FQT* works better since the scenario behaves in a manner similar to having the smaller batch of queries. In the scenario of the large Δ, more queries are unfinished during most of the iterations, thus *IQT* performs better.

4.2.2 Batching With Simultaneous Graph And Query Updates

As the graph grows, it becomes natural that the user wants to expand the query batch with new interesting queries, denoted as δ. In this section, we describe how *BEAD* enables the expansion of query batch as the graph grows, that is,

Table 4.1: Equations used in UpdateUsingR_0() (Algorithm 4.7 - line 7) to update vertices for any new query $q \in \delta$ which reaches the source vertex v_{old} of a query q_{old} in Q_0.

Algo.	**Update Equation**
SSSP	$\underline{R_1[v][q]}$=Min($\underline{R_1[v][q]}$, $\underline{R_1[v_{old}][q]}$+$\underline{R_0[v][q_{old}]}$)
SSWP	$\underline{R_1[v][q]}$=Max($\underline{R_1[v][q]}$, Min($\underline{R_1[v_{old}][q], R_0[v][q_{old}]}$))
Viterbi	$\underline{R_1[v][q]}$=Max($\underline{R_1[v][q]}$, $\underline{R_1[v_{old}][q] \times Ro[v][q_{old}]}$)
BFS	$\underline{R_1[v][q]}$=Min($\underline{R_1[v][q]}$, $\underline{R_1[v_{old}][q]}$+$\underline{R_0[v][q_{old}]}$)

$$Inc(G_0 + \Delta, Q_0 + \delta, R_0) \rightarrow R_1.$$

BEAD carries out the incremental evaluation for the new query batch $Q_0 + \delta$ on the evolved graph $G_0 + \Delta$, by using the existing result R_0. The key lies in the suitable initialization of the vertex values, query tracking bitsets, and the list of active vertices. The incremental evaluation can be either simultaneous or ordered. Each policy has its own advantages and opportunities. Next, we discuss the policies supported by *BEAD* in detail, then present an integrated algorithm that can be run under different modes and employ different policies.

Policy I: Simultaneous Evaluation – In this scenario, both changes in graph and queries (i.e., Δ and δ) are considered simultaneously by *BEAD*. That is, *BEAD* evaluates the new queries δ alongside the old queries Q_0 in one evaluation as a larger batch $Q_0 + \delta$ on the larger graph $G_0 + \Delta$, in a way that the existing R_0 can be leveraged. The rationale for simultaneous evaluation is that considering all changes together may cause total number of iterations required to be less than the ordered evaluations described next.

Policy II: Ordered Evaluation – In comparison, this option considers Δ and δ in an order: either Δ-first or δ-first. In the case of Δ-first, *BEAD* first computes the results of Q_0 on $G_0 + \Delta$ till convergence, then evaluates $Q_0 + \delta$ on $G_0 + \Delta$ till convergence. By contrast, δ-first

Algorithm 4.4 BEAD: Reevaluating for Both Graph and Query Updates ($G_0 + \Delta$, $Q_0 + \delta$) - Inc().

```
1: function INC($G_0, Q_0, R_0, \Delta, \delta, mode$)
2:     $G_1 \leftarrow G_0$.update($\Delta$)                                  ▷ Add new edges and vertices
3:     $Q_1 \leftarrow Q_0$.update($\delta$)                                  ▷ Add new queries
4:     $R_1 \leftarrow$ InitializeVArray($G_1, Q_1, R_0, \Delta, \delta, mode$)
5:     InitializeLists($\Delta, \delta, mode$)
6:     $V_{old} \leftarrow$ Source vertices of the top three high out-degree $\in Q_0$
7:     while !unified_activeList.empty() do
8:         ▷ ComputeIteration() performs steps $S_1$ through $S_4$ from Algorithm 4.1
9:         ComputeIteration(q_status, mode)
10:        if $\delta \neq \phi$ then
11:            Update($V_{old}$, $R_1, \delta, R_0$)
12:        end if
13:        Scatter_Batch(q_status)
14:    end while
15:    return $R_1$
16: end function
```

ordering first computes the results of $Q_0+\delta$ on G_0 till convergence, then evaluates $Q_0+\delta$ on $G_0+\Delta$ till convergence. The rationale for using ordered approaches is that, in the case where the size of Δ is much larger than δ, computing them separately allows IQT and FQT to be used for handling Δ and δ, respectively.

Algorithms 4.4 to 4.7 describes the incremental evaluation of *BEAD* that directly supports Policy I, but it can be used to emulate and thus support Policy II as well (which will be shown later).

Algorithm 4.5 BEAD: Reevaluating for Both Graph and Query Updates ($G_0+\Delta$, $Q_0+\delta$) - Initialize Vertex Array.

```
1: function InitializeVArray($R_1, G_1, Q_1, R_0, \Delta, \delta$)
2:     $R_1.set\_size(G_1.num\_vertices())$
3:     for each vertex $v \in G_1$ and each $q \in Q_1$ do
4:         $R_1[v].set\_size(Q_1.size())$
5:         if $v \in \Delta$ or $q \in \delta$ then                ▷ $v$ or $q$ are new
6:             $R_1[v][q] \leftarrow INIT\_VAL$
7:         else                                  ▷ Otherwise initialize $R_1$ using $R_0$
8:             $R_1[v][q] \leftarrow R_0[v][q]$
9:         end if
10:    end for
11:    return $R_1$
12: end function
```

The high-level structure of Algorithms 4.4 to 4.7 is similar to that of Algorithms 4.2 to 4.3, except several key differences.

Expanding Data Structures – After updating both the graph and the batch of queries, Algorithm 4.4 in lines 4-5 expands the vertex array and query bitset by calling Algorithms 4.5 and 4.6, respectively. Particularly, the number of values stored at each vertex is grown by the number of new queries (Line 4 in Algorithm 4.5), the same happens to the q_status array (Line 3 in Algorithm 4.6).

Algorithm 4.6 BEAD: Reevaluating for Both Graph and Query Updates ($G_0 + \Delta$, $Q_0 + \delta$) - Initialize Queries and Unified Active List.

```
1:  function InitializeLists($\Delta, \delta, mode$)
2:      ▷ Initialize unified_activeList with all the affected vertices
3:      $q\_status$.add_bitset_size_by($\delta.size()$)
4:      $unified\_activeList \leftarrow$ source vertices $v_q$ of all $q \in \delta$
5:      if $mode == IQT$ then
6:          for each $q \in \delta$ do $q\_status[v_q]$.set_bit($q$)
7:      end if
8:      for each $e \in \Delta$ do
9:          $v \leftarrow e.dest()$
10:         $unified\_activeList \leftarrow unified\_activeList \cup v$
11:         if $mode == IQT$ then
12:             $q\_status[v]$.set_all()
13:         end if
14:     end for
15:     if $mode == FQT$ then
16:         $q\_status$.set_all();
17:     end if
18:     return $R_1$
19: end function
```

Initialization for the New Queries – All the vertex values corresponding to the new queries are set to the initial value INIT_VAL (Line 5-6 in Algorithm 4.5). After that, the source

Algorithm 4.7 BEAD: Reevaluating for Both Graph and Query Updates ($G_0 + \Delta$, $Q_0 + \delta$) - Update().

```
1: function UPDATE($V_{old}$, $R_1, \delta, R_0$)
2:     if any $v_{old} \in V_{old}$ has been activated by any $q \in \delta$ then
3:         ▷ Send the current data of $v_h$ to other machines
4:         ClusterSynced($R_1[v_{old}][]$)
5:         ▷ Update current value of all vertices for queries in $\delta$
6:         ▷ Using equations in Figure 4.1
7:         $R_1 \leftarrow$ UpdateUsing$R_0(R_1, \delta, R_0, v_{old})$
8:     end if
9: end function
```

vertices of all the new queries are also added to the unified_activList to start the evaluation of new queries (Line 4 in Algorithm 4.6). In addition, if IQT is selected, it sets the corresponding bit for each new query – initializing the statuses of new queries to be active.

Enabling Indirect Incremental Computations – In addition, *BEAD* manages to take advantage of old results R_0 to achieve faster convergence for the new queries in δ. As shown in Algorithm 4.4 (Line 10-12), a new step, called *Update*, is inserted to the main loop right before *Scatter-Batch*, which updates all vertex data of new queries using the equations from Figure 4.1 for faster convergence. In specific, one old query $q_{old} \in Q_0$ is selected and its results are used by the update equations to improve all the vertex values of a new query q. However, to apply the update equations, the source vertex of q_{old} should be reachable from the source vertex of q – the source vertex of q_{old} should be activated by the new query in the current iteration (Line 2 in Algorithm 4.7). One intuitive heuristic for selecting the old query q_{old} is selecting the one with the highest out-degree source vertex who is more likely to be reached by a new query. In our case, *BEAD* selects three queries whose source vertices have the top three out-degrees (Line 6 in Algorithm 4.4). Finally, before apply the update equations (Line 7 in Algorithm 4.7), the results of the selected old query need to be synchronized across machines (Line 4 in Algorithm 4.7).

Now we present how Algorithms 4.4 - 4.7 can be used to emulate different policies: simultaneous evaluation (i.e., $\Delta||\delta$); Δ-first evaluation (i.e., $\Delta \rightarrow \delta$); and δ-first evaluation (i.e., $\delta \rightarrow \Delta$).

$\Delta||\delta$

USING $Eval(G_0, Q_0) \rightarrow R_0$

EVALUATE $Inc(G_0 + \Delta, Q_0 + \delta, R_0) \rightarrow R$

By Calling Algorithm 4.4::Inc($G_0, Q_0, R_0, \Delta, \delta, IQT$)

$\Delta \rightarrow \delta$

USING $Eval(G_0, Q_0) \rightarrow R_0$

EVALUATE $Inc(G_0 + \Delta, Q_0, R_0) \rightarrow R_1$

By Calling Algorithm 4.4::Inc($G_0, Q_0, R_0, \Delta, \phi, IQT$)

EVALUATE $Inc(G_0 + \Delta, Q_0 + \delta, R_1) \rightarrow R$

By Calling Algorithm 4.4::Inc($G_1, Q_0, R_1, \phi, \delta, FQT$)

$\delta \rightarrow \Delta$

USING $Eval(G_0, Q_0) \rightarrow R_0$

EVALUATE $Inc(G_0, Q_0 + \delta, R_0) \rightarrow R_1$

By Calling Algorithm 4.4::Inc($G_0, Q_0, R_0, \phi, \delta, FQT$)

EVALUATE $Inc(G_0 + \Delta, Q_0 + \delta, R_1) \rightarrow R$

By Calling Algorithm 4.4::Inc($G_0, Q_1, R_1, \Delta, \phi, IQT$)

As shown above, by feeding an empty set ϕ alternatively to Algorithm 4.4, both ordered evaluations can be realized. Note that we do not list FQT for $\Delta||\delta$, as it does not perform as well as IQT, for the reasons we mentioned earlier.

4.2.3 Interruption Handling

Finally, we consider the situation in which the user presents a new request while the prior is still being processed. One way to handle this interruption is waiting for the old request (say Δ_1 and δ_1) to converge then starting the processing of the new request (say Δ_2 and δ_2), which we referred to as following convergence. Instead of waiting for the old request to complete, *BEAD* chooses to merge the processing of both the old and new requests, such that the total processing time could be reduced. We refer to this more proactive option as anytime interruption. Algorithms 4.8 and 4.9 describe how anytime interruption works in the presence of a new user request while the old request is being processed. Right after the new request interruption is received (Line 12 in Algorithm 4.8), *BEAD* combines the new request with the old meanwhile leverages the current intermediate results of the old request (i.e., $\approx R_1$), to enable the incremental evaluation (Algorithm 4.8 - Line 14 by calling InterruptHandling from Algorithm4.9).

Algorithm 4.8 BEAD: Reevaluating for Anytime Simultaneous Update ($G_0 + \Delta$,$Q_0 + \delta$) - Inc().

```
1: function Inc($G_0, Q_0, R_0, \Delta, \delta, mode$)
2:     $G_1 \leftarrow G_0$.update($\Delta$)                          ▷ Add new edges and vertices
3:     $Q_1 \leftarrow Q_0$.update($\delta$)                          ▷ Add new queries
4:     $V_{old} \leftarrow$ Source vertices of the top three high out-degree $\in Q_0$
5:     $R_1 \leftarrow$ InitializeVArray($G_1, Q_1, R_0, \Delta, \delta, mode$)      ▷ see Algorithm 4.5
6:     while !unified_activeList.empty() do
7:         ComputeIteration(q_status, mode)          ▷ Steps $S_1$ to $S_4$ from Algorithm 4.1
8:         if $\delta \neq \phi$ then
9:             Update($V_{old}, R_1, \delta, R_0$)                 ▷ see Algorithm 4.7
10:        end if
11:        Scatter_Batch(q_status)
12:        if UserInterruptReceived() then
13:            ▷ Accommodate the new request
14:            $R_1 \leftarrow$ InterruptHandling($G_1, Q_1, R_1, mode$)
15:        end if
16:    end while
17:    return $R_1$
18: end function
```

4.3 Evaluations

We compare *BEAD* against the *MultiLyra* baseline under the following scenarios. A part of the graph (50%, 70% and 90%) is randomly selected from the full graph and chosen as G_0, the

Algorithm 4.9 BEAD: Reevaluating for Anytime Simultaneous Update ($G_0 + \Delta$,$Q_0 + \delta$) - InterruptHandling().

```
1: function INTERRUPTHANDLING($G_1, Q_1, \approx R_1, mode$)
2:     $\Delta', \delta' \leftarrow$ UserInterrupt.get_new_request()
3:     $G_1 \leftarrow G_1.update(\Delta')$                ▷ Add new edges and vertices
4:     $Q_1 \leftarrow Q_1.update(\delta')$                ▷ Add new queries
5:     ▷ Reinitialize $R_1$ based on prior unfinished results $\approx R_1$
6:     $R_1 \leftarrow$ Initialize($G_1, Q_1, \approx R_1, \Delta', \delta', mode$)
7:     return $R_1$
8: end function
```

first version of the graph. Then, additional portions of the graph are added to G_0 in batches of Δ to emulate a growing graph. All the Δ batches were randomly chosen of different sizes – 1k, 10k, 100k for LJ and 10k, 100k, and 1000k edges for TT; Since TT has roughly ten times the number of vertices as LJ, Δ sizes chosen for TT are ten times that of LJ. Additional δ queries are added to Q_0 to reflect the growing batch of queries. These additional randomly chosen δ queries are added to Q_0 in increments of 8, 16 and 32 queries.

As *BEAD* is built on top of *MultiLyra*, to be self-contained and to demonstrate the promises of batching evaluation, we briefly report the baseline performance (more details in Chapter 2).

Table 4.2: Real world input graphs.

Input Graph	**#Edges**	**#Vertices**
Twitter (TT) [5, 8]	2.0B	52.6M
LiveJournal (LJ) [6, 10]	69M	4.8M

4.3.1 Experimental Setup

We developed *BEAD* by integrating the implementations of incremental evaluation algorithms into the *MultiLyra* from Chapter 2. Our evaluation covers four common graph applications - Single Source Shortest Path (SSSP), Single Source Widest Path (SSWP), Breadth First Search (BFS), and Viterbi (VT) [9]. Two input graphs are listed in Table 4.2, including the Twitter graph (TT) with 2 billion edges and the LiveJournal graph (LJ) with 69 millions of edges. For each input graph and for each algorithm, we generated the queries by randomly selecting the source vertices.

All experiments were run on a cluster of four homogeneous machines. Each machine has 32 Intel Broadwell cores and 256GB memory, and runs CentOS Linux release 7.4.1708.

4.3.2 MultiLyra – Scalability With Batch Sizes

We first show the benefits of batching achieved by *MultiLyra* for G_0 (50%, 70%, 90%) during the evaluation of a total of 256 SSWP queries. For each G_0 (of LJ), we ran the SSWP queries first one by one (i.e., non-batching which is equivalent to PowerLyra [2]) and then in batches of 64, 128, and 256 queries (in IQT mode). Table 4.3 shows execution time in seconds and the speedups of batching over non-batching. The results show that batching in *MultiLyra* brings more speedups as the batch size increases, meanwhile the gains decreases as the batch size approaches 256 queries (more details in Chapter 2). Also note that the total number of iterations is reduced dramatically, as the number of iterations for a batch of queries is determined by the "slowest" query, rather than the sum of those for all the queries as in the non-batching case.

Table 4.3: Total Execution Time of running 256 SSWP queries using MultiLyra on LJ to compute $Eval(G_0, Q_0) \rightarrow R_0$ with varying batch sizes.

$\mathbf{G_0}$	**Batch Size**	**#Iter.**	**Time (s)**	
50%	**1(non-batching)**	9286	2759.42	**Speedup**
	64	400	538.48	5.13×
	128	200	432.81	6.38×
	256	100	431.58	6.39×
70%	**1(non-batching)**	9420	2462.62	**Speedup**
	64	400	592.93	4.15×
	128	200	495.03	4.98×
	256	100	457.38	5.38×
90%	**1(non-batching)**	10060	2713.6	**Speedup**
	64	400	642.39	4.22×
	128	200	527.05	5.15×
	256	100	483.57	5.61×

4.3.3 Graph Updates: BEAD Vs. MultiLyra

In this section, we compare the handling of graph updates Δ by *BEAD* that incrementally reevaluates batch of queries Q_0, with *MultiLyra* that must evaluate queries Q_0 on graph $G_0 + \Delta$ from scratch. Table 4.4 presents the speedups obtained by *BEAD* over *MultiLyra* for evaluating 256 queries in a single batch on the updated graph $G_0 + \Delta$, where G_0 is 50% and Δ is set to 100K edges and 10K edges for TT and LJ, respectively. Both *BEAD* and *MultiLyra* ran in *IQT* mode. The last column of Table 4.4 reports the execution time for *MultiLyra*.

Speedups delivered by BEAD range from 6.21× for SSWP to 26.16× for SSSP on TT and from 3.99× for Viterbi to 5.34× for SSSP on LJ. Note that generally higher speedups are achieved for the larger TT graph than for the smaller LJ graph. This indicates that the savings in

Table 4.4: Speedups of BEAD over MultiLyra when computing $Inc(G_0 + \Delta, Q_0, R_0)$ given $Eval(G_0, Q_0) \rightarrow R_0$, where $G_0 = 50\%$.

$\mathbf{G_0}$	**Graph Algo.**	**Batch Size**	$\mathbf{\Delta}$	**BEAD**		**MultiLyra**
				Speedup	**#Iter**	**Exe. Time**
	SSSP	**256**	**100K**	26.16×	11	1141.5s
TT	**SSWP**	**256**	**100K**	6.21×	100	2753.6s
(50%)	**BFS**	**256**	**100K**	15.00×	7	510.2s
	VT	**256**	**100K**	18.61×	21	1506.9s
	SSSP	**256**	**10K**	5.34×	26	337.7s
LJ	**SSWP**	**256**	**10K**	4.15×	45	431.6s
(50%)	**BFS**	**256**	**10K**	4.00×	11	111.0s
	VT	**256**	**10K**	3.99×	19	195.0s

work achieved by *BEAD*'s incremental algorithm are greater for the larger TT graph. The overall speedup for *SSWP* on TT is lower than those of the other three graph algorithms. This is because a few queries in Q_0 take much longer to converge than the rest of the queries for *SSWP* – note the very high number of iterations for *SSWP* shown in **#Iter** column in Table 4.4. Consequently, for most iterations, only a few queries are actually active (25 active queries after iteration 15), limiting the benefits of batching.

Next, we perform more detailed experiments, for *SSSP* on TT and *SSWP* on LJ, to study the sensitivity of performance benefits (*BEAD* over *MultiLyra*) with respect to a number of factors, including: (a) Varying Δ – for TT, this was varied across 10k, 100k, and 1000k while for LJ it was varied across 1k, 10k, and 100k; (b) Varying the size of G_0 – for both TT and LJ this was varied across 50%, 70%, and 90%; (c) Varying batch size – the 256 queries were run in one batch of 256, 2 batches of 128, and 4 batches of 64; and (d) using IQT vs FQT. Table 4.5 and Table 4.6 present the results for *SSSP* on TT, and Table 4.7 and Table 4.8 present the results for *SSWP* on LJ. The speedups are calculated by comparing the execution time of each configuration with the execution time of the corresponding *MultiLyra* configuration.

Following are our observations from the above experiments.

(a) Sensitivity to Varying Δ – When the size of the changes to graph increases, the speedup of the incremental evaluation of *BEAD* decreases since more computation is needed to attain convergence to the final result. Table 4.5 shows that *BEAD* on SSSP, for $G_0 = 50\%$, achieves a maximum speedup of 33.37× when $\Delta = 10K$ and a minimum speedup of 7.86× when $\Delta = 1000K$. Similar trend is also observed in the case of SSWP on LJ (see Table 4.7).

(b) Sensitivity to Varying G_0 – Table 4.5 and Table 4.7 show that *BEAD*'s speedups decrease when larger portions of the graph are loaded as G_0. Since larger parts of the graph are more connected for larger G_0, it starts longer evaluation waves through the graph as the graph grows. These tables show the maximum speedups of: 33.37× for $G_0 = 50\%$ vs. 25.91× for $G_0 = 70\%$ for SSSP on TT; and 4.53× for $G_0 = 50\%$ vs. 4.34× for $G_0 = 70\%$ for SSWP on LJ.

(c) Sensitivity to Varying Q_0 – We ran 256 queries divided into varying batch sizes to study impact of varying Q_0 size. Our results from Table 4.5 and Table 4.7 show that although *BEAD*'s speedups for different sizes of Q_0 vary, the variation across different Δ sizes is mostly small and no specific size of Q_0 gives the best speedups across different Δ sizes.

(d) Sensitivity to IQT / FQT – As mentioned earlier, *IQT* and *FQT* are two modes of evaluation that enable the opportunities to shrink not only the amount of computations but also the amount of data communicated between master and mirror vertices hosted on different machines. To examine if these two modes are still relevant during *BEAD*'s incremental evaluation as the graph

Table 4.5: Sensitivity study of running SSSP on TT using BEAD when graph changes: $Inc(G_0 + \Delta, Q_0, R_0)$ given $Eval(G_0, Q_0) \rightarrow R_0$.

				Mode		
				IQT		**FQT**
G$_0$	**Δ**	**# $\times$Q$_0$**	**#Iter.**	**Time (s)**	**Speedup**	**Speedup**
50%	**10K**	**4 $\times$ 64**	37	69.46	27.79$\times$	27.77$\times$
		2 $\times$ 128	19	47.59	33.37$\times$	31.59$\times$
		1 $\times$ 256	10	41.36	27.60$\times$	27.09$\times$
	100K	**4 $\times$ 64**	44	65.11	29.65$\times$	22.58$\times$
		2 $\times$ 128	22	50.44	31.49$\times$	26.12$\times$
		1 $\times$ 256	11	43.63	26.16$\times$	25.45$\times$
	1000K	**4 $\times$ 64**	72	126.32	15.28$\times$	13.75$\times$
		2 $\times$ 128	40	121.26	13.10$\times$	10.04$\times$
		1 $\times$ 256	23	145.30	7.86$\times$	5.26$\times$
70%	**10K**	**4 $\times$ 64**	40	81.24	24.15$\times$	20.63$\times$
		2 $\times$ 128	22	60.09	25.91$\times$	21.78$\times$
		1 $\times$ 256	12	52.08	25.07$\times$	24.47$\times$
	100K	**4 $\times$ 64**	52	88.78	22.10$\times$	14.32$\times$
		2 $\times$ 128	29	68.92	22.59$\times$	17.43$\times$
		1 $\times$ 256	16	63.52	20.56$\times$	20.27$\times$
	1000K	**4 $\times$ 64**	66	122.00	16.08$\times$	14.39$\times$
		2 $\times$ 128	34	90.33	17.24$\times$	13.24$\times$
		1 $\times$ 256	17	83.59	15.62$\times$	12.93$\times$
90%	**10K**	**4 $\times$ 64**	32	84.88	25.64$\times$	20.26$\times$
		2 $\times$ 128	18	59.22	27.25$\times$	26.68$\times$
		1 $\times$ 256	9	50.01	27.85$\times$	27.72$\times$
	100K	**4 $\times$ 64**	41	89.27	24.38$\times$	15.80$\times$
		2 $\times$ 128	22	67.45	23.92$\times$	21.21$\times$
		1 $\times$ 256	12	54.53	25.54$\times$	24.29$\times$
	1000K	**4 $\times$ 64**	50	102.26	21.28$\times$	17.56$\times$
		2 $\times$ 128	26	68.95	23.40$\times$	23.07$\times$
		1 $\times$ 256	14	61.96	22.48$\times$	20.48$\times$

Table 4.6: Extra number of communications needed for running SSSP on TT using BEAD when graph changes to compute: $Inc(G_0 + \Delta, Q_0, R_0)$ given $Eval(G_0, Q_0) \rightarrow R_0$.

			Message Type		
$\mathbf{G_0}$	$\mathbf{\Delta}$	**#** $\times$ $\mathbf{Q_0}$	**Active**	**Data**	**Total** ($\times 10^6$)
50%	**10K**	**4** $\times$ **64**	20.54%	79.46%	0.04
		2 $\times$ **128**	20.82%	79.18%	0.03
		1 $\times$ **256**	21.08%	78.92%	0.03
	100K	**4** $\times$ **64**	20.09%	79.91%	0.57
		2 $\times$ **128**	20.79%	79.21%	0.44
		1 $\times$ **256**	21.22%	78.78%	0.36
	1000K	**4** $\times$ **64**	21.54%	78.46%	69.02
		2 $\times$ **128**	21.67%	78.33%	67.06
		1 $\times$ **256**	21.74%	78.26%	65.74
70%	**10K**	**4** $\times$ **64**	18.72%	81.28%	0.05
		2 $\times$ **128**	19.37%	80.63%	0.03
		1 $\times$ **256**	19.91%	80.09%	0.02
	100K	**4** $\times$ **64**	21.16%	78.84%	1.16
		2 $\times$ **128**	21.49%	78.51%	1.08
		1 $\times$ **256**	21.73%	78.27%	1.01
	1000K	**4** $\times$ **64**	20.04%	79.96%	32.47
		2 $\times$ **128**	20.01%	79.99%	24.02
		1 $\times$ **256**	19.94%	80.06%	14.80
90%	**10K**	**4** $\times$ **64**	17.28%	82.72%	0.03
		2 $\times$ **128**	17.47%	82.53%	0.02
		1 $\times$ **256**	17.94%	82.06%	0.01
	100K	**4** $\times$ **64**	18.66%	81.34%	0.27
		2 $\times$ **128**	19.03%	80.97%	0.19
		1 $\times$ **256**	19.35%	80.65%	0.14
	1000K	**4** $\times$ **64**	18.12%	81.88%	3.44
		2 $\times$ **128**	18.64%	81.36%	2.67
		1 $\times$ **256**	19.02%	80.98%	2.09

Table 4.7: Sensitivity study of running SSWP on LJ using BEAD when graph changes: $Inc(G_0 + \Delta, Q_0, R_0)$ given $Eval(G_0, Q_0) \to R_0$.

				Mode		
				IQT		**FQT**
G_0	**Δ**	**# ×Q_0**	**#Iter.**	**Time (s)**	**Speedup**	**Speedup**
50%	**1k**	**4** × **64**	155	118.96	4.53×	4.57×
		2 × **128**	81	103.98	4.16×	4.15×
		1 × **256**	45	107.77	4.00×	4.03×
	10k	**4** × **64**	155	119.92	4.49×	4.56×
		2 × **128**	81	102.14	4.24×	4.36×
		1 × **256**	45	103.75	4.15×	4.21×
	100k	**4** × **64**	155	126.93	4.24×	4.09×
		2 × **128**	81	109.28	3.96×	3.31×
		1 × **256**	45	126.77	3.40×	2.81×
70%	**1k**	**4** × **64**	178	136.75	4.34×	4.24×
		2 × **128**	130	168.99	2.86×	2.89×
		1 × **256**	73	171.02	2.67×	2.55×
	10k	**4** × **64**	178	142.20	4.17×	4.14×
		2 × **128**	130	167.32	2.89×	2.90×
		1 × **256**	73	178.26	2.57×	2.53×
	100k	**4** × **64**	178	137.46	4.31×	4.26×
		2 × **128**	130	168.05	2.88×	2.87×
		1 × **256**	73	175.88	2.60×	2.33×
90%	**1k**	**4** × **64**	265	233.74	2.75×	2.93×
		2 × **128**	140	218.56	2.41×	2.16×
		1 × **256**	87	201.21	2.40×	2.10×
	10K	**4** × **64**	265	233.71	2.75×	2.87×
		2 × **128**	140	226.76	2.32×	2.04×
		1 × **256**	87	203.87	2.37×	2.03×
	100K	**4** × **64**	265	255.06	2.52×	2.47×
		2 × **128**	140	229.02	2.30×	2.06×
		1 × **256**	87	227.71	2.12×	1.76×

Table 4.8: Extra number of communications needed for running SSWP on LJ using BEAD when graph changes to compute: $Inc(G_0 + \Delta, Q_0, R_0)$ given $Eval(G_0, Q_0) \rightarrow R_0$.

			Message Type		
$\mathbf{G_0}$	$\mathbf{\Delta}$	**#** $\times$ $\mathbf{Q_0}$	**Active**	**Data**	**Total** ($\times 10^6$)
50%	**1K**	**4** $\times$ **64**	39.78%	60.22%	0.36
		2 $\times$ **128**	39.76%	60.24%	0.34
		1 $\times$ **256**	39.68%	60.32%	0.31
	10K	**4** $\times$ **64**	37.98%	62.02%	0.38
		2 $\times$ **128**	38.77%	61.23%	0.35
		1 $\times$ **256**	39.14%	60.86%	0.32
	100K	**4** $\times$ **64**	15.85%	84.15%	24.31
		2 $\times$ **128**	15.82%	84.18%	24.18
		1 $\times$ **256**	15.78%	84.22%	24.09
70%	**1K**	**4** $\times$ **64**	35.91%	64.09%	0.26
		2 $\times$ **128**	35.96%	64.04%	0.26
		1 $\times$ **256**	35.77%	64.23%	0.25
	10K	**4** $\times$ **64**	34.66%	65.34%	0.27
		2 $\times$ **128**	35.32%	64.68%	0.27
		1 $\times$ **256**	35.43%	64.57%	0.25
	100K	**4** $\times$ **64**	26.93%	73.07%	0.43
		2 $\times$ **128**	30.34%	69.66%	0.35
		1 $\times$ **256**	32.49%	67.51%	0.29
90%	**1K**	**4** $\times$ **64**	34.93%	65.07%	0.45
		2 $\times$ **128**	35.02%	64.98%	0.45
		1 $\times$ **256**	35.01%	64.99%	0.45
	10K	**4** $\times$ **64**	34.30%	65.70%	0.47
		2 $\times$ **128**	34.70%	65.30%	0.46
		1 $\times$ **256**	34.84%	65.16%	0.45
	100K	**4** $\times$ **64**	15.34%	84.66%	16.08
		2 $\times$ **128**	15.33%	84.67%	15.99
		1 $\times$ **256**	15.32%	84.68%	15.94

grows, we collected the number of messages communicated, as shown in Table 4.6 and Table 4.8. After dividing these messages according to their type, Active vs. Data, it can be seen that 60-84% of the communications are in form of Data messages which is similar to our observations for non-incremental *MultiLyra* communication in Chapter 2. Thus, as in case of *MultiLyra*, during incremental evaluation *IQT* typically outperforms *FQT*. The only exception is Table 4.7 where for small query batch size and/or small Δs, *FQT* performs slightly better leveraging its low overhead of tracking only the finished queries while *IQT* performs better in larger batch sizes and Δs.

4.3.4 Graph And Query Updates: BEAD Vs. MultiLyra

In this section, we evaluate *BEAD* when both the graph and the batch of queries simultaneously grow. We evaluate all three policies discussed earlier in Section 4.2.2: (i) applying graph change then the query change ($\Delta \rightarrow \delta$), (ii) applying the query change then graph change ($\delta \rightarrow \Delta$), and (iii) simultaneously applying both changes ($\Delta \| \delta$). The initial setup is running an original batch of 256 queries (Q_0) for different algorithms on 50% of TT and LJ (G_0). The new batch of queries δ can be of size varying among 8, 16, and 32, while the graph updates Δ are set to 100K for TT and 10K for LJ. Table 4.9 shows the speedups of *BEAD* under the three policies. The baseline execution times were collected by running the same batch of $Q_0 + \delta$ queries on the updated graph $G_0 + \Delta$ using *MultiLyra*.

As shown in Table 4.9 where the best speedups are marked in red, the ordered evaluation ($\Delta \rightarrow \delta$ and $\delta \rightarrow \Delta$) obtains better performance comparing to simultaneous evaluation ($\Delta \| \delta$). The reason can be understood as follows. During the simultaneous evaluation, the new queries (δ) and old queries (Q_0) are merged into a larger batch ($Q_0 + \delta$). As the old queries were started earlier

Table 4.9: Speedups of *BEAD* over *MultiLyra* on Simultaneous Graph and Query Updates: computing $Inc(G_0 + \Delta, Q_0 + \delta, R_0)$ given $Eval(G_0, Q_0) \rightarrow R_0$, where $G_0 = 50\%$ and $Q_0 = 256$.

				BEAD			
				$\Delta{\rightarrow}\delta$	$\delta{\rightarrow}\Delta$	$\Delta \| \delta$	
G	**Algorithm**	Δ	δ	**Speedup**	**Speedup**	**Speedup**	MultiLyra
TT	**SSSP**	**100K**	**8**	5.39×	5.28×	2.67×	1241.6s
			16	4.94×	5.02×	2.50×	1303.0s
			32	3.88×	4.02×	2.25×	1429.1s
	SSWP	**100K**	**8**	5.57×	5.66×	5.01×	2891.6s
			16	5.42×	5.39×	4.84×	2887.0s
			32	5.33×	4.98×	4.30×	3131.8s
	BFS	**100K**	**8**	5.10×	5.39×	2.89×	543.7s
			16	4.02×	4.21×	2.28×	562.7s
			32	3.24×	3.46×	2.08×	571.4s
	VT	**100K**	**8**	4.13×	4.25×	2.33×	1571.4s
			16	3.39×	3.55×	1.99×	1576.7s
			32	2.67×	2.79×	1.80×	1725.8s
LJ	**SSSP**	**10K**	**8**	3.19×	3.28×	2.18×	343.3s
			16	2.92×	2.74×	1.86×	363.5s
			32	2.65×	2.60×	1.85×	399.9s
	SSWP	**10K**	**8**	3.23×	3.25×	2.66×	456.6s
			16	3.20×	3.09×	2.43×	471.5s
			16	2.96×	2.76×	2.28×	469.4s
	BFS	**10K**	**8**	2.73×	2.65×	1.80×	112.6s
			16	2.69×	2.53×	1.65×	121.3s
			32	2.40×	2.38×	1.51×	125.4s
	VT	**10K**	**8**	2.88×	2.73×	2.15×	211.4s
			16	2.52×	2.50×	1.82×	214.0s
			32	2.16×	2.03×	1.74×	221.0s

than the new queries, their vertex values tend to converge earlier. Once their values are converged, they become the overhead of the following iterative evaluation, slowing down the progress of the new queries. Next, we examine how the ordering between the graph updates and query updates affect the performance, that is, $\Delta \rightarrow \delta$ vs. $\delta \rightarrow \Delta$.

Δ-First vs. δ-First Evaluation – As indicated in Table 4.9 under columns $\Delta \rightarrow \delta$ and $\delta \rightarrow \Delta$, in general, whether applying the graph change Δ at the first place for Q_0 or at second place for $Q_0+\delta$ with the availability of stable results from the previous step only makes limited differences in performance. However, since the evaluation of sub-batch δ starts from scratch (despite the use of indirect incremental computations), it could take more iterations to traverse the changed graph $G_0 + \Delta$ than the original graph G_0. This effect is more significant when the original graph G_0 is relatively small or the graph change Δ is relatively large. In our setup, graph TT is about 29X larger than LJ in terms of the number of edges, but its update batch size is only 10X larger than that of LJ. Consequently, as shown in Table 4.9, when comparing the speedups on TT, with the those on LJ, δ-First evaluation works better on TT, whereas Δ-First evaluation shows superiority on LJ, the smaller graph. For example, Δ-first evaluation obtains a maximum speedup of 5.39$\times$ for *SSSP* on TT whereas δ-first evaluation achieves a maximum speedup of 3.28$\times$ for *SSSP* on LJ. Note that the speedups after including new queries δ, in addition to graph updates Δ, are lower than those with only graph updates (comparing to Table 4.4) because although queries in Q_0 terminate rapidly, the queries in δ being new take much longer time.

4.3.5 Interruption Handling

Finally, we evaluate *BEAD* in the scenario of interruption – a new request (say Δ_2 and δ_2) arrives in the middle of the incremental evaluation for the prior request (say evaluating $Q_0 + \delta_1$ on graph $G_0 + \Delta_1$). For this evaluation, we first ran 256 queries (Q_0) for each algorithm on the 50% input graphs (G_0) using *BEAD*. Then, we let *BEAD* incrementally evaluate the first request – the

Table 4.10: Performance of BEAD under User Interruptions computing $Inc(G_0 + \Delta_1 + \Delta_2, Q_0 + \delta_1 + \delta_2, R_0)$ in two requests given $Eval(G_0, Q_0) \rightarrow R_0$, where $G_0 = 50\%$.

					Latency (Seconds)		
					Interruption Points		
G	**Algo.**	**Q$_0$**	**Δ_1:δ_1**	**Δ_2:δ_2**	**50%**	**75%**	**100%**
TT	**SSSP**	**256**	**100K:16**	**10K:8**	469.94	465.49	489.41
	SSWP	**256**	**100K:16**	**10K:8**	667.24	719.30	795.71
	BFS	**256**	**100K:16**	**10K:8**	215.06	208.57	231.69
	VT	**256**	**100K:16**	**10K:8**	747.49	770.42	787.78
LJ	**SSSP**	**256**	**10K:16**	**1K:8**	200.12	230.57	236.12
	SSWP	**256**	**10K:16**	**1K:8**	144.53	161.26	169.83
	BFS	**256**	**10K:16**	**1K:8**	68.63	82.92	85.14
	VT	**256**	**10K:16**	**1K:8**	115.24	136.44	138.18

updated query batch $Q_0 + \delta_1$ on the updated graph $G_0 + \Delta_1$, where δ_1 = 16 and Δ_1 is 100K for TT and 10K for LJ. After that, in the middle of this evaluation, at the points when 50%, 75%, and 100% of the evaluation has been done (in terms of elapsed time), the second request (Δ_2 and δ_2) from the user interrupts BEAD and asks for updated evaluation, that is, $Q_0 + \delta_1 + \delta_2$ on $G_0 + \Delta_1 + \Delta_2$. The 50% and 75% scenarios correspond to the anytime interruption strategy used by BEAD whereas the 100% scenario mimics the following convergence strategy that can be used alternatively (see Section 4.2.3).

Table 4.10 reports the results of the above experiments. The 100% scenario (i.e., following convergence) ensures that the precise results R_1 are available to the incremental computation of the second request, while in 50% and 75% scenarios only the approximate results $\approx R_1$ are available. We observe that the immediately starting of the second request using $\approx R_1$ (i.e., anytime interruption) leads to lower response latency for the second request. Although using precise results R_1 can reduce the work performed in evaluating the second request, waiting to compute the second request outweighs this benefit for the interruption points of 50% and 75%.

4.4 Summary

In this Chapter, we generalized *MultiLyra* from Chapter 2 to consider scenarios in which analytics demands of the user evolve. While *MultiLyra* delivers high performance by solving batches of queries simultaneously, *BEAD* achieves the same in the presence of changes to the graph and/or query set. Thanks to its incremental query evaluation, *BEAD* delivers significant speedups over *MultiLyra* in handling the evolving demands. Experiments demonstrate that *BEAD*'s batched evaluation of 256 queries, after graph changes that add up to 100K edges to a billion edge Twitter graph and also query changes that add up to 32 new queries, outperforms batched evaluation by *MultiLyra* by factors of up to 26.16$\times$ and 5.66$\times$ respectively.

Chapter 5

Related Work

This chapter discusses the various research works in literature that are related to this thesis. 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 [10], 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 [4, 14, 13, 30, 29] (e.g., Galois [13], Ligra [14], and GRACE [29]), GPUs, and custom accelerators [44] [45] [47] as well as platforms in the distributed environment (e.g., Pregel [12], GraphLab [11], GraphX [7], PowerGraph [1], PowerLyra [2], ASPIRE [26], and CoRAL [28]). 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 [38], Congra [22], The More the Merrier [23], SimGQ [20], and SimGQ+ [21]).

5.1 Single Query Graph Processing

Shared-memory systems on a single machine lack scalability while the distributed ones are able to load large graphs into the combined memory of multiple machines delivering scalability. Nguyen et al. in [13] present a lightweight shared-memory infrastructure called Galois for Domain-Specific Languages (DSLs) that automatically schedules fine-grain tasks with application-specific priorities. Kusum et al. in [30] used input reduction techniques and built a system on top of Galois to prcoess larger graphs. Ligra [14] traverses the input graph efficiently leveraging its shared-memory abstraction for vertex algorithms based on the Bulk Synchronous Parallel (BSP) [24] model. On the other hand, there are frameworks such as GRACE [29] that enable asynchronous execution using message passing model. To Process extremely large graphs on a single multicore machine many out-of-core processing systems have been proposed such as (GraphChi [31], X-Stream [33], GridGraph [37], DynamicShards [36], Turbograph [35], Flashgraph [34], and Bishard [32]).

Alternately distributed systems that combine memories of multiple machines to handle large graphs can be used. The most relevant ones include PowerGraph [1], PowerLyra [2] and Gemini [3]. 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). However, PowerLyra [2] 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 [19]. It uses edge-cut for low-degree vertices making the computation of low-degree vertices local to each machine while using vertex-cut to evenly distribute the incoming edges of the high-degree vertices among the machine to achieve the computation load balance. 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 [3] tries to achieve scalability while maintaining the intra-machine efficiency. 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. 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.

5.2 Batched Query Graph Processing

Although distributed Quegel [38] was designed to solve a batch of concurrent queries by efficiently sharing memory and computing resources among the queries, but its performance relies on an expensive hub indexing pre-computation. In addition, Quegel's applicability is limited to point-to-point queries [43] as opposed to the more general point-to-all queries evaluated by MultiLyra and BEAD. Finally, Quegel can overlap the evaluation of only a few queries as it employs pipelined parallelism. By contrast, MultiLyra expanded the batching capability to simultaneously evaluating hundreds of iterative queries.

For the shared-memory setting, Chengshuo et al. developed SimGQ [20], an online system that efficiently evaluate a batch of concurrent queries by reusing results of common subcomputations of different queries in the batch. Later they extend it by developing SimGQ+ [21] to evaluate point-to-point queries [43]. In the most recent work, Glign[50] improves evaluation of a batch of concurrent queries by aligning their associated graph traversals.

5.3 Evolving Graph Processing

Most of the above systems focus on evaluating graph queries on fixed input graphs that do not change over time. However, real-wolds graphs are changing all the time, e.g. social media networks update when a new user joins (i.e., new vertex) or new friendship occurs (i.e., new edge). Many frameworks have been proposed to solve graph analytics problems in the changing graph scenarios. Kickstarter [27] and Graphbolt [40] track the dependencies to enable fast query processing on streaming graphs while Naiad [25] employs incremental algorithms. A distributed streaming graph processing framework has been proposed in Kineograph [41] that enables push and pull modes focusing on incremental computation. In another work, a snapshot of the current version of the graph is taken by Tornado [39] to evaluate the graph queries. Note that batched evaluation of queries by BEAD is different from the above mentioned query evaluation over streaming graphs. First, BEAD performs continuous evaluation of a batch of queries, that is, following updates, the queries must be reevaluated. In contrast, in streaming graphs, individual queries are evaluated (not batches) and they are evaluated only upon request (not continuously). Finally, [23] evaluates a batch of queries but it is specialized for BFS and [22] executes different queries in different processes making it inefficient in comparison to BEAD.

5.4 Graph Processing Accelerators

Jetstream [44] is distinguished as the pioneering accelerator designed for streaming graphs with support for incremental algorithms. It adeptly manages both accumulative and monotonic graph algorithms using an event-driven computational framework. This framework limits access to a select subset of graph vertices, skillfully reutilizes previous query outcomes to minimize repetition, and refines the memory access sequence to maximize memory bandwidth use. MEGA [48] emerges as the first accelerator for evolving graphs (i.e., graphs that change over time). It is equipped to handle incremental event-driven streaming of edge additions and can simultaneously process multiple snapshots. MEGA employs CommonGraph [45, 47, 49], a new technique aimed at the incremental handling of evolving graphs, which enhances efficiency by sidestepping costly deletions.

Chapter 6

Contributions and Future Work

6.1 Contributions

In this thesis we proposed comprehensive distributed batching techniques that amortize the communication, synchronization, and computation costs of iterative graph queries over the real-world large-scale graphs in both fixed and evolving analytics demands.

We first introduced MultiLyra (Section 2), a distributed batching system whose scalability enables simultaneous evaluation of hundreds of iterative graph queries over a fix graph which is published in BigData'19 [17]. MultiLyra achieves outstanding throughput and higher scalability utilizing various optimization techniques such as unified active list across queries, fine-grained query status methods and a distributed reuse technique.

Next, we introduced ExpressWay (Chapter 3) to further improve the efficiency and throughput of MultiLyra by enabling distributed faster convergence for various iterative graph algorithms by taking different policies. ExpressWay currently is under submission. ExpressWay prioritize the important edges in the input graph and runs the query only using the selected edges reducing the amount of computation and communication. After this initialization step, majority of vertices has their final or pre-final values leading faster convergence when ExpressWay runs the graph query using all edges.

Finally in Chapter 4, we present BEAD whose incremental evaluation techniques greatly expand MultiLyra to support evolving both the graph and the batch of running queries which is published in BigData'20 [18]. BEAD proposes various incremental scenarios, i.e., only the graph evolves, only the batch of query evolves. or both graph and batch of running queries evolve simultaneously. In addition, Bead's *anytime interruption* technique enables the user to interrupt the running experiment early in case of receiving new updates for the graph or batch of running queries and still utilizing the benefits of the incremental computation.

6.2 Future Work

6.2.1 Non-Monotonic Queries

This thesis mainly focused on the monotonic graph algorithms in which vertex values starts from an initial value and monotonically decrease or increase throughout the evaluating iterations until it can not change anymore (i.e., converged). Although, non-monotonic algorithms can benefit from unified active list but most of the optimization technique we proposed do not apply to such queries. Therefore, one can explore batching system for non-monotonic queries to study on either new optimization techniques that works for these queries or how to customize the existing ones such as status query tracking to better fit the problem.

6.2.2 Heterogeneous Queries

Our observations from Sections 2 and Section 4 lead us to an important missing extension to our proposed distributed batching system that is a support for the heterogeneous queries in the running batch. Lets get back to the example of online shopping center which aims analyzing the behavior of the most important customers. In previous studies (i.e., *Quegel*, *MultiLyra* and *BEAD*), it is assumed that the same behavior of different customers are being analysed. Therefore, all queries in the running batch are the same while analyzing different customers (starting from different source vertices).

However, in practice users can be interested in different aspects of each important customers' behavior. This leads to have different kinds of queries with different dynamic behavior in the same running batch (e.g., SSSP along with PageRank queries). In other words, in the real-world situation we are dealing with a batch of heterogeneous graph queries. Having such queries with different behavior in the same running batch dramatically affect the efficiency of the unified active list since the majority of the active vertices in each iteration are active only for a multiple number of queries.

Bibliography

[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] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis (SC), pages 1-11, 2010.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. *http://snap.stanford.edu/data*, June 2011.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] M. Then, M. Kaufmann, F. Chirigati, T-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H.T. Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. In *Proceedings of the VLDB Endowment*, 2015.

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

[25] D. Murray, F. McSherry, R. Isaacs, M. Isard, P.Barham, and M. Abadi. Naiad: a timely dataflow system. In SOSP, pages 439–455, 2013.

[26] 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, 2014.

[27] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 237-251, 2017.

[28] 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, 2017.

[29] 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.

[30] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu. Efficient Processing of Large Graphs via Input Reduction. In ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 245-257, 2016.

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

[32] K. Najeebullah, K. U. Khan, W. Nawaz and Y-K. Lee. BiShard Parallel Processor: A Disk-Based Processing Engine for Billion-Scale Graphs. In International Journal of Multimedia and Ubiquitous Engineering, 9(2):199-212, 2014.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In 24th ACM Symposium on Operating Systems Principles (SOSP), pages 472-488, 2013.

[34] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flash-Graph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In 13th USENIX Conference on File and Storage Technologies (FAST), pages 45-58, 2015.

[35] W-S. Han, S. Lee, K. Park, J-H. Lee, and M-S. Kim, and J. Kim and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pages 77-85, 2013.

[36] K. Vora, G. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Distributive Disk-based Graph Algorithms. USENIX Annual Technical Conference (ATC), pages 507-522, June 2016.

[37] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In USENIX Annual Technical Conference (ATC), pages 375-386, 2015.

[38] 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.

[39] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In SIGMOD, pages 417–430, 2016.

[40] M. Mariappan and K. Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. EuroSys, pages 1–16, 2019.

[41] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In EuroSys, pages 85–98, 2012

[42] 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.

[43] 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.

[44] S. Rahman, M. Afarin, N. Abu-Ghazaleh and 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.

[45] M. Afarin, C. Gao, S. Rahman, N. Abu-Ghazaleh and 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.

[46] 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.

[47] M. Afarin, C. Gao, S. Rahman, N. Abu-Ghazaleh and 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.

[48] C. Gao, M. Afarin, S. Rahman, N. Abu-Ghazaleh and Rajiv Gupta, MEGA Evolving Graph Accelerator In *56th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), 2023.

[49] M Afarin, C Gao, S Rahman, N Abu-Ghazaleh, R Gupta. Graph Analytics on Evolving Data In *arXiv preprint arXiv:2308.14834*.

[50] 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.

[51] 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.

[52] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special interest tracks and posters of the 14th International Conference on World Wide Web*, pp. 1094-1095. 2005.

[53] A. Kan, J. Chan, J. Bailey, and C. Leckie. A query based approach for mining evolving graphs. In *Proceedings of the Eighth Australasian Data Mining Conference-Volume 101*, pp. 139-150. 2009.

[54] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. In *Proceedings of the VLDB Endowment 4*, no. 11, pp. 726-737, 2011.

[55] X. Jiang. Runtime Optimizations for Evaluating Batches of Graph Queries. University of California, Riverside, 2023.

[56] X. Jiang, M. Afarin, Z. Zhao, N. Abu-Ghazaleh, R. Gupta. Core Graph: Exploiting Edge Centrality to Speedup the Evaluation of Iterative Graph Queries. In *Proceedings of the Nineteen European Conference on Computer Systems* (EuroSy)s, 2024.