ElGA: Elastic and Scalable Dynamic Graph Analysis

Kasimir Gabert*
Georgia Institute of Technology
Atlanta, Georgia, USA
kasimir@gatech.edu

Kaan Sancak
Georgia Institute of Technology
Atlanta, Georgia, USA
kaan@gatech.edu

M. Yusuf Özkaya
Georgia Institute of Technology
Atlanta, Georgia, USA
myozka@gatech.edu

Ali Pınar
Sandia National Laboratories
Livermore, California, USA
apinar@sandia.gov

Ümit V. Çatalyürek†
Georgia Institute of Technology
Atlanta, Georgia, USA
umit@gatech.edu

ABSTRACT

Modern graphs are not only large, but rapidly changing. The rate of change can vary significantly along with the computational cost. Existing distributed graph analysis systems have largely been designed to operate on static graphs. Infrastructure changes in these systems need to occur when the system is idle, which can result in significant wasted resources or the inability to cope with changes.

We present ElGA, an elastic and scalable dynamic graph analysis system. Using a shared-nothing architecture and consistent hashing, ElGA can scale elastically as the graph grows or more computation is required. By applying sketches, we perform an edge partitioning of the graph where high degree vertices can be split among multiple nodes. ElGA supports both synchronous and asynchronous vertex-centric applications that operate in batches on a continuously changing graph.

We experimentally demonstrate that ElGA outperforms state-of-the-art static systems while supporting client queries, elastic infrastructure changes, and dynamic algorithms.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Mathematics of computing** → **Graph algorithms**; • **Information systems** → **Data access methods**; **Data layout**.

KEYWORDS

Distributed graphs systems, dynamic graphs, elastic computing

ACM Reference Format:
Kasimir Gabert, Kaan Sancak, M. Yusuf Özkaya, Ali Pınar, and Ümit V. Çatalyürek. 2021. ElGA: Elastic and Scalable Dynamic Graph Analysis. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3480857

*Also with Sandia National Laboratories.
†Also with Amazon Web Services. This publication describes work performed at the Georgia Institute of Technology and is not associated with Amazon.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3480857

1 INTRODUCTION

Graph analysis is a crucial part of many data analysis pipelines. Numerous graph algorithms and systems have been developed to tackle a large range of problems. Most graph algorithms and systems have been developed as *static* graph systems—they start by loading an input graph, perform some computation, save results, and finally shutdown. Many large graphs are generated from continuous processes, such as website visits, computer network traffic, and more. As graphs have grown in scale, increasing attention has been made towards treating them as *dynamic* graphs, where they change over time and remain in-memory on a system [11]. High performance systems that handle massive dynamic graphs, those in the hundreds of billions of edges, are increasing in importance [16].

There have been dozens of large, distributed graph systems developed over the last decade [63]. In many cases, dynamic graphs are *partially supported*: the underlying system may support some form of updates, and many graph algorithms can be continued from a prior state. Unfortunately, such systems tend to not work well in practice on rapidly changing graphs due to architectural decisions resulting in large start-up and tear-down costs. Many dynamic graph algorithms, especially those which are locally persistent [5, 72] and well-suited for vertex-centric distributed systems [62], may have significant runtime variance between batches of edge changes [31]. Similarly, the rate of change of graphs can vary significantly [71]. A crucial yet overlooked factor in dynamic graph systems is *elasticity*: the ability of a system to rapidly support scaling computational resources up and down, to match demand, meet constraints, or optimize an objective [4, 8, 35, 42, 66].

Why is elasticity important, and especially so for dynamic graph systems? First, dynamic graphs change over time. As they grow, their scale changes, and over time the underlying infrastructure requirements, such as memory, change. Second, graphs can experience periods of relative calm and periods of significant bursts of changes. Scaling up during bursts and down during calm periods can provide both cost savings and performance gains. Third, even among identically sized batches, some batches necessarily take much longer to process than others [5, 31, 72]. In many cases the cost of the batch is not quickly predicable and so the ability to elastically scale based on the runtime behavior of the batch is important. Fourth, for both static and dynamic graphs, many iterative algorithms require less work for later iterations. Scaling down accordingly can bring significant cost savings [41, 70, 82].

Overall, elastic systems can make trade-offs between runtime performance, memory demand, and cost. Inelastic systems need to continuously consume resources demanded for the peak scale. Elasticity has recently been recognized as an open and important research direction in distributed graph systems for low-latency queries on dynamic graphs [41].

1.1 Design Goals

Motivated to provide an elastic dynamic graph processing system for modern, large graphs, we introduce five specific design goals for such a system. Let P be the number of processors in the system.

Goal 1. *The system can operate on graphs with hundreds of billions of edges and skewed degree distributions.*

Goal 2. *At any point in time, all system participants operate with $O((n+m)/P+P)$ memory, where n and m are the current number of vertices and edges, respectively.*

Goal 3. *The system is scalable, and so the only dependence on P for frequent operations is $O(\log P)$.*

Goal 4. *The graph is dynamic and can be continuously updated. Maintenance algorithms should optimize for low-latency and low-variance runtimes and should support concurrent queries.*

Goal 5. *The system can scale up or down, manually or automatically, during computation to meet demand.*

1.2 Contribution

We propose ElGA, a distributed graph system that is able to perform computation on rapidly changing graphs and can do so while the underlying infrastructure scales up and down. Inspired by key-value stores that scale from single nodes to global systems [87], ElGA uses a shared-nothing architecture [77] with a reliable and scalable message passing system. The main question is how to identify which graph edge belongs to which processing agent given a continuously changing graph. ElGA uses consistent hashing [52] to solve this. Prior dynamic graph partitioning approaches that address skewed degree distributions have required information about all vertices in memory [1] (taking $O(n)$ space), which does not meet Goal 2 and introduces challenges for processing concurrent graph changes from independent sources. We remove this dependence through the use of sketches [36] where a small, fixed amount of memory contains partitioning information for the whole graph.

Through our design, ElGA meets the above goals. At the same time, our architecture provides better runtime performance than existing static distributed graph systems. We show that distributed graph systems which are both *dynamic* and *elastic* do not have to sacrifice performance or scalability against their static graph system counterparts. We release ElGA as an open source system to encourage development and further improvements to distributed, dynamic, and elastic graph systems[1].

The remainder of this paper is as follows. In § 2 we provide necessary background and a review of existing distributed systems. In § 3 we describe ElGA, including its programming model and architectural details. In § 4 we perform an experimental evaluation and finally in § 5 we conclude.

[1] ElGA is available at https://github.com/GT-TDAlab/ElGA.

2 BACKGROUND AND RELATED WORK

2.1 Graphs and Dynamic Models

In this section, we introduce the graphs that we consider along with the enabled class of algorithms. We focus on *directed* graphs.

Definition 2.1. A *directed graph* $G = (V, E)$ is a set of vertices V and a set of edges E, where each edge $e = (u, v) \in E$ has a source $u \in V$ and a destination $v \in V$.

In the remainder, we refer to directed graphs as graphs.

Definition 2.2. A *graph algorithm* $\mathcal{A} : G \to O(G)$ takes a graph G as input and produces an output $O(G)$.

Definition 2.3. A *dynamic graph* is an infinite turnstile stream of edge changes, $D = (c_1, c_2, \dots)$, where $c = (d, u, v) \in D$ consists of an *action* d, which either indicates the insertion or removal of an edge, and the edge itself, (u, v) with $u \in V, v \in V$.

At a point i in the stream, a graph G^i can be formed by applying $c_1, c_2, \dots, c_i$ starting from the empty graph $G^0 = (\emptyset, \emptyset)$.

Definition 2.4. A *batch* $\Delta_{i,j}$ is a segment of the stream from i to j with $i \leq j$, and so $\Delta_{i,j} = (c_i, c_{i+1}, \dots, c_{j-1}, c_j)$.

Definition 2.5. A *dynamic graph algorithm* is one that can maintain a graph algorithm's computation on the infinite stream. The dynamic graph algorithm $\mathcal{B} : (G^i, O(G^i), \Delta_{i,j}) \to O(G^j)$ produces the output $O(G^j)$ at position j, starting from the graph G^i, the previous output $O(G^i)$ and the batch $\Delta_{i,j}$.

Definition 2.6. A *query* is an operation that retrieves part of the output $O(G^i)$, typically consisting of the result for a given vertex.

Suppose the latest update is c_j. The goal of dynamic graph algorithms is to run quickly so that queries return outputs from $O(G^{j'})$, where $j' \leq j$ and $j - j'$ is small with the hope that $O(G^{j'})$ is close to $O(G^j)$. Predicting or measuring this gap is challenging [72].

We note that dynamic graphs and algorithms are different from *temporal* graphs and algorithms [54], which operate over the entire history of the graph, and from *streaming* graphs and algorithms [64], which operate with sublinear memory limits.

Many dynamic algorithms are vertex-centric and communicate over edges. These are known as *locally persistent* [5, 72]. Within vertex-centric algorithms there are *synchronous* algorithms, where each vertex can only be processed once before receiving updates from all neighbors, *asynchronous* algorithms, where vertices are processed when all necessary neighbor messages are provided, and *partially synchronous* where asynchrony occurs with larger, global synchronous levels [34]. In many distributed graph systems a vertex may be duplicated for load balancing across multiple independent nodes, which we call replicas.

2.2 Distributed Graph Systems

2.2.1 Classification of Distributed Graph Systems. There are numerous distributed graph systems serving a large variety of purposes. There are two main dimensions that determine what graph system is suitable for a given application: the properties of the input graph and the desired algorithms. In this section we provide a classification to explain where *dynamic graph systems* fit in.

Starting from the graph properties, the first classification point is whether the input can fit into memory across the system or not. If the graph is too large to fit, streaming graph algorithms [64], sparisification [27], sampling [58], approximations [49], and other randomization strategies, e.g., [3], can be used. We address graphs that can fit into memory across the entire system. The next point is whether the graph is static, i.e., it will not change, or dynamic. If it is static, there are block-based methods, vertex-centric methods, among others [63]. We focus on dynamic graphs. Next, we consider three different algorithm needs. First, some algorithms only perform neighbor traversals or similar subgraph lookups, typically following vertex label constraints. Graph databases are used in this case [12]. Second, some algorithms require a temporal history of the graph, and so temporal graph systems are used [44, 54]. Third, algorithms may be used for real-time or interactive applications and need results on the most recent graph [11]. Our focus is on this need. In the next section, we drill down based on what the desired latencies of the algorithms are and what kind of scale the graph requires.

2.2.2 Dynamic Graph Systems. There are numerous distributed graph systems that have been proposed and developed and many have some notion of a changing, or dynamic, underlying graph. [11] provides a thorough review of prior systems. Table 1 presents a comparative overview of dynamic and elastic graph systems; the columns and systems are described in the following.

Table 1: Dynamic and elastic properties of graph systems.

System	Open Source	Scalable	Partially Dynamic	Fully Dynamic	Elastic
Pregel [62]		✓			
Blogel [89]	✓	✓			
Hadoop/Giraph [38]	✓	✓			
Spark/GraphX [37]	✓	✓			
Flink/Gelly [17]	✓	✓			
Spark/GraphX/Sprouter [2]	✓	✓	✓		
Spark/GraphX/CellIQ [47]		✓	✓		
Spark/GraphTau [48]		✓	✓		
GraM [88]		✓	✓		
Timely/Delta-BiGJoin [6]		✓	✓		
Kineograph [20]		✓	✓		
InfoSphere/UNICORN [78]			✓	✓	
Flink/Gelly Streaming [17]	✓		✓	✓	
ZipG [53]			✓	✓	
ChronoGraph [28]			✓	✓	
Naiad [68]			✓	✓	
Kickstarter [84]	✓		✓	✓	
Concerto [56]			✓	✓	
Vaquero et al. [83]			✓	✓	
Spark/GraphX/EdgeScaler [69]			✓		✓
GRAPE [30, 32, 33]	✓		✓		✓
FaRM/A1 [16]		✓	✓		✓
JoyGraph [82]		✓			✓
Graphless [81]		✓			✓
iGiraph [39, 40]					✓
Joker [51]					✓
ElGA	✓	✓	✓	✓	✓

Definition 2.7. A system is *scalable* if it meets Goals 1–3: it has been shown to run on large graphs, does not require storing all vertices or edges on one node, and does not depend linearly on the number of processors in time-sensitive operations.

There are many scalable systems and programming models developed for static graphs and with fixed infrastructure, including Pregel [62], Blogel [89], Giraph [38], GraphX [37], and Gelly [17]. These systems provide programming models [63] and, in many cases, robust and flexible code bases that dynamic graph systems develop on top of.

Definition 2.8. A system is *partially dynamic* if it can re-use prior output to continue an incremental computation.

Several systems have been developed that are partially dynamic, but do not specifically support low-latency updates. Sprouter [2] and CellIQ [47] address dynamic partitioning costs, but do not address the batch startup costs or elasticity. GraphTau [48] can operate in a partially dynamic manner but was built to provide temporal analysis of graphs. GraM [88] focuses largely on static graphs with fixed partitioning. Delta-BiGJoin [6] is an approach implemented in Timely Dataflow [68] that is optimized for efficient subgraph queries. Kineograph [20] operates on snapshots of a dynamic graph and leaves elasticity as a future goal.

Definition 2.9. A system is *fully dynamic* if it supports low-latency queries and low-latency batch processing, meeting Goal 4.

There are many fully dynamic distributed graph systems, which address the need to run low-latency dynamic graph algorithms. To a large extent these systems have not focused on scalability or elasticity. UNICORN [78] is built on InfoSphere Streams and provides low-latency incremental graph updates, but it has not been shown to scale to large graphs or to provide elasticity. Gelly Streaming [17] focuses on streaming graph algorithms and does not address elasticity or directly support locally persistent algorithms. ZipG [53] provides low-latency queries but focuses on subgraph extraction and does not address elasticity. ChronoGraph [28] is a fully dynamic system built in NodeJS, but not elastic. Naiad [68] is a general purpose dataflow system. KickStarter [84] maintains an approximate computation and re-computes exactly on-demand. Concerto [56] uses elastic key-value stores internally, but does not explicitly support elasticity. [83] re-partitions internally on graph changes but does not address elasticity.

Definition 2.10. A system is *elastic* if it can rapidly scale its infrastructure, manually or automatically, to meet demand [42, 66].

There are elastic graph systems, but to date these have not focused on dynamically changing graphs. GRAPE [30, 32, 33] consists of a series of work, some of which address partially dynamic graph algorithms and others which address various aspects of elasticity. The GRAPE model is not scalable as it requires a global view of the graph on a single machine. A1 [16] is a dynamic attributed graph database that is in-memory and communicates over

RDMA by building on FaRM [24]. A1 inherits FaRM's elasticity, but is limited in its scope to querying subgraphs and retrieving vertex and edge attributes; it does not support dynamic graph algorithms. JoyGraph [82] demonstrates the advantages of elasticity for static graphs, but does not address dynamic graphs. Graphless [81] operates in the serverless model but does not support dynamic graphs. iGiraph [39, 40] elastically scales during computation to provide cost and performance benefits, but does not support dynamic graphs. Joker [51] is a dataflow system that addresses elasticity, however it has not been developed to address graph problems.

2.3 Achieving Elasticity in Clouds

Consistent hashing was developed as a caching technique to decrease hot spots in large, distributed data sets [52]. The idea is to place servers and data uniformly at random in a ring, and then map data to the closest server. Hashes can be used on server IDs and data keys ensure uniform placement of both servers and data. Importantly, when a server joins or leaves, the data mapped to it moves to only a few neighboring servers and all other data has no movement. This effectively provides both smoothness for partition keys and reasonable load-balance across the servers.

Chord [76] was one of the first distributed systems to use consistent hashing. Since then, it has been foundational for many systems that have churn in participants, such as distributed hash tables [50], and in cloud systems that are highly elastic, such as Dynamo, Amazon's key-value store [23]. Consistent hashing has been applied for graphs by treating vertices as keys, providing in high-quality partitioning [1]. However, such approaches only partition vertices or require global information about each vertex.

2.4 Sketches

Sketches are tools that allow for approximate results using sublinear space [36]. The first sketch that counts distinct types of elements in a stream is Count Sketch [19]. It works by creating a small matrix and, for each element in the data stream, incrementing or decrementing one cell in each row based on row-specific hashes for the data. The total count is then approximated by the average value of the cells. The CountMinSketch [22] improves the Count Sketch by only going in *one direction*. Instead of adding or subtracting, CountMinSketch will only add values. A minimum instead of the average is used. For a fixed probability and an additive error of εm after m elements, its space complexity is $\Omega(1/\varepsilon)$. We are not aware of prior uses of sketches in dynamic graph partitioning.

3 ElGA

The key to achieving scalability and elasticity in a distributed graph system starts with how the vertices and edges are partitioned among the processing units in the system. Due to the irregular nature of graphs, balancing workloads is crucial. When new compute resources are added or removed, we want to both keep the system well-balanced and to limit data movement, allowing both the graph algorithm runtime and elastic scaling to be quick. Once the partition of each edge is determined, the rest of the system falls naturally into place: any communication through an edge can communicate with the edge owner and vertex-centric programs will work.

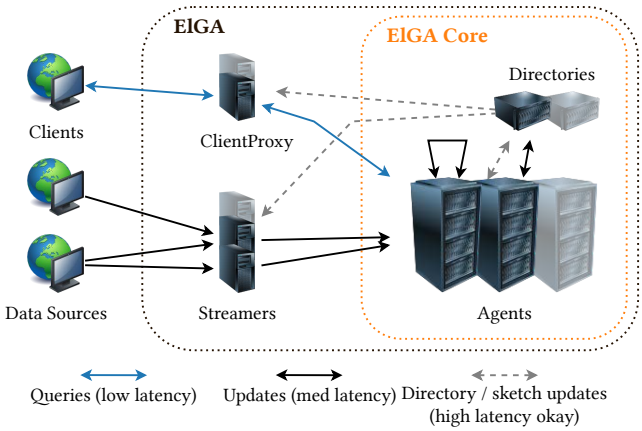


Figure 1: An overview of the components of ElGA.

Our key contribution consists of a new way of placing the edges given challenging constraints: the graph is dynamically changing, the nodes owning edges are being added and removed, and we cannot use more than a (small) constant amount of global storage.

We solve this by pulling together concepts from cloud computing and streaming algorithms. First, every component in ElGA that needs to be scalable builds on consistent hashing. This forms the backbone of ElGA. Second, ElGA is flexible with receiving messages out-of-order and/or destined for the wrong node. It buffers such messages appropriately and forwards them to the best known destination to achieve eventual consistency. Third, in any situation where global knowledge of the graph is required, we apply sketches. This allows us to globally make decisions about vertices using a small constant amount of memory which can be updated over time.

In the remainder of this section we provide an overview of our system, describe its programming model, and discuss important architectural details.

3.1 System Overview

A high-level overview of ElGA is shown in Figure 1. ElGA follows a shared-nothing design [77]. This means that each entity is single threaded and only communicates via message passing. ElGA is composed of several entities: Agents run graph algorithms and hold graphs in memory; Streamers send graph updates to Agents; and ClientProxies proxy end-user queries to Agents to receive algorithm results. We call Agents, Streamers, and ClientProxies "Participants", as they all need to determine which Agent has ownership of a given edge. The system boundary provides the entry point for other programs to interact with ElGA. We have support for simple TCP and UNIX socket protocols for crossing the system boundary.

Finally, there is a directory system in ElGA which both informs Participants which Agent is responsible for what edge and facilitates synchronization as needed in bulk-synchronous algorithm steps. The directory itself needs to be scalable, as the number of Participants can grow and each Participant establishes a connection to the directory system.

ElGA contains low-latency, medium-latency, and high-latency connections. Low-latency connections need to be returned as quickly

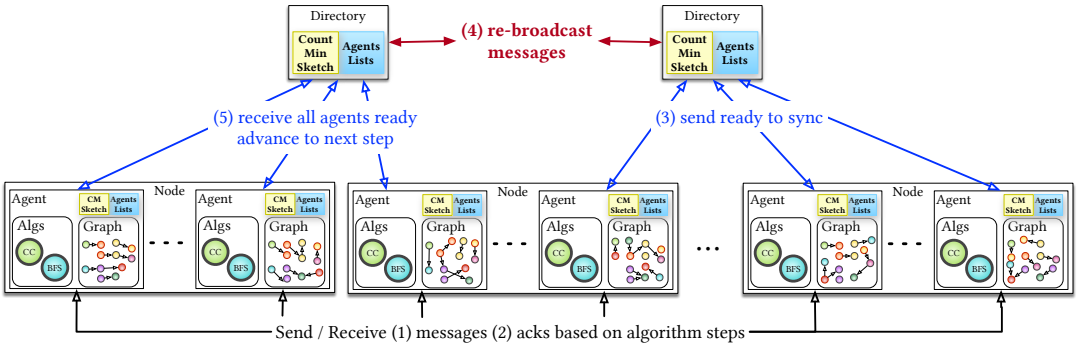


Figure 2: Agents sending messages to neighboring vertices and advancing to the next superstep through Directories.

as possible. These are used by the clients to query specific algorithm results or properties of vertices. Medium-latency connections consist of updates that are on some critical path for reducing the staleness of a client query. This consists of both updates to the graph and updates required by a batch algorithm to finish computation. Finally, there are connections that allow high-latency which support elastic scaling and load balancing, but are not on any critical path for future client queries.

3.2 ElGA Core – Programming Model

ElGA follows a locally persistent dynamic graph algorithm model [5, 72]. Each algorithm is executed when it has some changed state, either a message from a neighbor or replica. The algorithm runs from the perspective of a vertex. It can save local state and send messages along its edges. This is a common model for scalable graph systems, and while there are many alternative models, such as edge-centric [73], subgraph-centric [80], and block-centric [89], this model continues to perform with state-of-the-art results [7].

To support more complicated dynamic graph algorithms than only bulk-synchronous parallel algorithms, such as PageRank, ElGA has support for receiving specific instructions from an algorithm for which edges to expect changes on. If it needs to wait for a result from another vertex, then it places itself in the waiting set for that vertex, at the given iteration. Only when that specific message is received will the vertex be removed from the waiting set. When a vertex is no longer waiting on any messages, it enters an active state and can be processed again.

The process of performing an iteration in this model is shown in Figure 2. First, **(1)** Agents will send messages—as the algorithm dictates—to various neighbors and **(2)** wait to receive acknowledgements from each. When each internal vertex is inactive **(3)** Agents report they are ready for the next superstep. **(4)** Directories re-broadcast ready messages among themselves. Finally, **(5)** when all Agents are ready, the superstep is advanced and vertices which are marked to become active will begin processing.

3.3 Directory System

Inside of the directory system, there are Directories and a single DirectoryMaster (not shown in Figure 1). The DirectoryMaster serves as a bootstrap service for ElGA: it is queried once by any component to find a Directory that is open to receiving communication and only queried again if the current Directory leaves the system. When Agents join or leave, or the graph changes enough to impact load balancing, Agents inform their respective Directory server. To keep each Directory in sync, all Directories internally broadcast messages appropriately. This update then propagates throughout the system and eventually, all Participants will use the latest directory. During the gap when some Participants have stale information, messages will be forwarded.

The full broadcast size is $O(P + dw)$, as each Agent's IP address and connection information are sent, P is the number of Agents (one Agent per core), d is the CountMinSketch depth, and w the CountMinSketch width. Each Directory also keeps track of the current batch ID, a monotonically increasing clock used to bootstrap Agents and ensure consistency. As shown in Figure 2, Directories are also used to facilitate global synchronization by waiting for their connected Agents to be ready and broadcasting state changes.

3.3.1 Sketch Size. As sketches are broadcast throughout ElGA, we want to ensure that they are appropriately sized. For a given number of edges, we can compute the size desired for a given error at a probability. Consider a point in the stream with m edges seen. Let d_i be the degree of vertex i at this point and $\hat{d}_i$ be the estimated degree. There are two parameters: the width of the table and the depth. The width can be set to $\lceil e/\varepsilon \rceil$ and the depth $\lceil \log 1/\delta \rceil$, and the sketch guarantees an additive error $\hat{d}_i \leq d_i + \varepsilon m$ with probability $1 - \delta$. If a graph will have 100 billion edges, we can get a 99.965% probability of each degree estimation within just over 1 million if we use a width of 2^{18} and a depth of 8. If the vertex replication threshold is 2 million, this table size will suffice and fits in 8 MB. We explore various table sizes in § 4.5.

3.4 Agents

Agents are responsible for holding the graph in memory and carrying out the computation on the graph. They provide the vertex-centric model for algorithms to run in and manage all operations on components of the graph.

Agents themselves are relatively straightforward. They operate as a state machine and, during computation, either execute the algorithms on their vertices, send updates to other Agents, or receive updates from Agents. They continuously poll on their communication channel and act on whatever packet they receive. First, the packet is inspected to ensure that the endpoint remains valid. If the receiving Agent is no longer the correct destination, the packet is forwarded to the latest, correct Agent. Each packet with edge

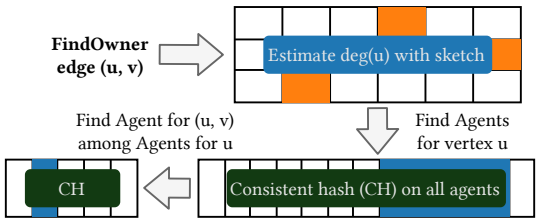


Figure 3: Participants look up Agents for edges by estimating the degree and then applying two consistent hashes.

data may contain iteration information. If it is for an iteration in the future, the packet is stored until the computation can catch up. In ElGA's asynchronous mode, vertices are individually processed when they no longer have any outstanding updates they are waiting on. In the synchronous mode, each vertex must have no outstanding updates. In between supersteps, some vertices may have updates that are sent to their replicas, in the case that the vertex is high-degree and split across multiple Agents. Garbage collection of received neighbor values and other book-keeping data occur at superstep boundaries.

While a batch is running, the graph does not change: any edge changes are buffered. Once the batch is over, these updates can be processed and the Agent becomes ready to perform a computation again. Depending on the algorithm, only updated (or *active*) vertices will be processed at the next iteration.

3.4.1 Finding an Edge. Each Participant receives enough information from its Directory to find the owner of any edge. This process is the heart of ElGA's load balancing scheme, and it enables dynamic graphs and elastic scaling. This process is outlined in Figure 3.

A participant will first perform a query into the CountMinSketch to identify how many Agents are responsible for handling the given vertex's edges. This is a biased approximation as it may exceed the degree but not underestimate it. A participant will then use a consistent hash to find the starting Agent and return Agents for the number of replications, based on the degree. It will then perform a second consistent hash on those Agents to find the final Agent that is responsible for the edge. Each consistent hash can be implemented with a binary search over the Agent list taking $O(\log P)$. Querying the degree estimate takes $O(d)$, where d is typically 8.

Concretely, each consistent hash consists of the following. Each Participant has a list of all the Agents. Each Agent ID is then hashed and added to a vector. To query for a vertex, the vertex ID is then hashed and the next highest Agent in the vector is selected as its owner. If the CountMinSketch indicates a replication factor of $k > 1$, then the Participant needs to select between the next k-highest Agents in the vector. In this case, a second hash is performed on the destination of the vertex to choose which of those k Agents.

For efficiency reasons, if only *some* Agent responsible for the vertex is required, e.g., for a vertex query, then the last consistent hash is bypassed and one replica is chosen at random.

3.4.2 Virtual Agents. Despite using two-levels of consistent hashing and sketching, creating only as many Agents as hardware cores still yields some load imbalance. To address this problem, we introduce *virtual agents*. Instead of putting a single ID into the vector, an Agent will put a variable number of IDs into the vector. We experimentally determined 100 as a good number (see § 4.5). This significantly improves the load balance but increases the lookup time by a constant factor (O(log 100)). Future work could explore dynamically adjusting the number of virtual agents over time based on memory or computation pressure or for heterogeneous systems.

3.4.3 Agent Elasticity and Sketch Updates. If an Agent joins or leaves, or receives an update to its CountMinSketch, it needs to ensure that it does not have any edges that need to migrate to another Agent. In ElGA, we take a straightforward approach of recomputing the correct destination for all current edges. Any edges that are in the wrong Agent are removed and forwarded appropriately. No state change can occur until receiving acknowledgements.

When an Agent leaves, it only signals it is leaving to the Directory. After receiving the next directory update, it evaluates its edges normally and determines that they all need to leave. Only when it has no edges and has waited a period of time will it disconnect.

To handle fully elastic autoscaling, ElGA comes with an API for metric collection and autoscalers. We implemented a simple reactive autoscaler that computes the exponential moving average of a metric and scales to the average divided by a scaling factor. We implemented Agent metrics for graph change rates, client query rates, and superstep times. Metrics are passed to Directories.

3.5 Communication

Communication in ElGA occurs using ZeroMQ [43]. ZeroMQ is designed as a networking layer that promotes various communication patterns. For example, it provides a publish/subscribe pattern, where multiple subscribers can receive select updates from a publisher. Internally, ZeroMQ handles message routing and resiliency. We configure ZeroMQ to use TCP between nodes and its interprocess protocol within a node. ZeroMQ operates on separate threads, allowing for overlapped computation and communication management, including packet routing, buffering, and filtering.

ElGA uses a simple serialization and deserialization protocol on top of ZeroMQ messages. The first byte of any message is a packet type which determines how a Participant will handle the message. ElGA's protocols typically involve direct memory copies into ZeroMQ's network buffers.

For low-latency messages, we use the `REQ/REP` model, which is designed for blocking requests and responses.

For messages with medium-latency, we use the `PUSH` communication pattern, which is a non-blocking send. This allows the client to continue executing while ZeroMQ finishes sending the message. This can be crucial when there is a large backlog of messages or potential networking issues. In cases where an explicit acknowledgment is required, a second `PUSH` is then sent in return.

Finally, we use the `PUB/SUB` model for high-latency messages. Here, Participants will subscribe to the messages they are interested in: Agents subscribe to synchronization barriers from Directories, all Participants subscribe to directory updates, and so on. As ElGA uses only a single byte for the message type, filtering subscriptions in ZeroMQ is efficient. ZeroMQ internally will route, filter, and duplicate the messages as intended.

There is an overhead with ZeroMQ: using Mellanox ConnectX/5 NICs, we benchmarked the latency of an MPI send at around 1 μs, a raw TCP send at 4 μs and a send through ZeroMQ at over 20 μs.

4 EXPERIMENTS

In this section we describe our experiments to understand and evaluate the design decisions and performance of ElGA. Following standard distributed graph system experimental methodologies [29], we run five independent trials for each experiment. We report the means and, assuming a t-distribution as the sample size is small, we show the 95% confidence intervals for the mean. All results were checked for correctness among the baselines and ElGA, and, when applicable, against ground truth.

We implemented ElGA in C++17 and use ZeroMQ 4.3.4. Our dynamic graph is stored as a flat hash map with vectors. We configured all systems to use 64-bit integers for vertex IDs. We compiled with GCC 8.3.1 using `O3`. We store both in and out edges.

4.1 Experimental Environment

Our experimental environment consists of 65 servers, each with dual-socket Intel Xeon E5-2683 v4 processors with 16 cores each, 512 GB RAM, and three local SSDs. One server is dedicated to filesystem and other metadata, and so 64 servers are used for computation. Each has Mellanox ConnectX 5 NICs and are connected to an Arista 7500E switch, which supports 100 Gbps. Jumbo frames were enabled. We run Ceph [86] on each node to provide a distributed filesystem for storing edge lists and auxiliary information and we run HDFS [15] for the state-of-the-art graph systems that we used as baselines, as they depend on it. None of our experiments compare graph loading times and and so the use of HDFS or Ceph is not being evaluated; we chose Ceph as it achieves higher performance on our system, however the baselines are implemented with HDFS. We use CentOS 8 and performed basic system tuning using `tuned`'s `hpc-compute` profile.

4.2 State-of-the-art Baselines

While there are no known dynamic and elastic baselines, we evaluate against Blogel [89] and GraphX [37]. We chose Blogel as it has been shown to be the state-of-the-art static system [7] and we chose GraphX both as it exhibits strong performance and is the platform for many snapshot-based partially dynamic systems (see § 2.2.2). We confirm previous results [7] that show that the Voronoi partitioning variant of Blogel (which we call Blogel-Vor) is not competitive against their simple vertex partitioning implementation (which we call Blogel) or even GraphX, even when partitioning time is ignored. Unfortunately, Voronoi is the only partitioning strategy implemented for the algorithms in our experiments.

We extensively explored the performance of both Blogel and GraphX in terms of MPI libraries and Java systems, Java garbage collection parameters, and more. We found that using dedicated SSDs for scratch storage for GraphX, along with the G1 garbage collection operating in parallel, a dynamic number of executors, and initial heap allocations of almost all available memory allowed GraphX to be competitive against Blogel and other systems. We configured GraphX to use its three main built-in partitioning strategies. We found that Java 8 provided the fastest runtimes for GraphX, although the loading times were faster with Java 11. As such, we use OpenJDK Java 8 throughout. For Blogel, we tried different compilers and MPI libraries and conclude that using OpenMPI 3.1.4 and GCC, as suggested in the original papers, still provides the fastest end-to-end runtimes. We use GCC 8.3.1 with `O3`. We found using 8 MPI ranks per node for Blogel was fastest, whereas GraphX was fastest with per-stage dynamic executors choosing the number of cores from 1 to 64 during runtime. We used the best found settings for all shown experiments.

Together, these two baselines represent the *hardest cases* for ElGA—if we compete well against them, we show that our design and implementation which supports dynamic graphs could even be used for end-to-end improvements in static cases. Our experiments show this is indeed the case. We stress that we do not include loading, partitioning, or saving time, as the baselines are not built to handle dynamic graphs and have unoptimized code in each area.

4.3 Static and Dynamic Algorithms

We evaluate with two iterative vertex-centric algorithms commonly used in distributed graph system benchmarks [7] that are implemented on Blogel, GraphX, and ElGA: PageRank and weakly connected components (WCC). In PageRank, at each iteration, a vertex receives messages from each in-neighbor, aggregates them with a sum, scales the value, and sends its values out to its out-neighbors. In WCC, a vertex aggregates and sends with a minimum instead of a sum and only sends updated minimums, but to both in- and out-neighbors. In the static case, WCC initializes each vertex to a unique identifier. In the dynamic case with insertions, known as the incremental case, each vertex retains its old or initial component information and only vertices directly modified in the batch are activated. When a vertex receives a message, it becomes active.

To effectively evaluate the graph systems, we ensured that all algorithms are the same across each system, including termination conditions. As such, the performance differences come from the systems themselves. We ensure our implementation's correctness by comparing against the baselines and ensured floating point values were correct up to 10^{-8}. For both baselines, we used the respective algorithms distributed with the baselines, from the original authors. We observed each system perform the same number of supersteps.

While PageRank and WCC are common graph algorithms, important future work includes studying the performance of other algorithms in ElGA which exhibit different bottlenecks and communication patterns [46].

4.4 Datasets

ElGA is designed to address large graphs which cannot fit in shared memory. As such, we focus our evaluation on large datasets. We use datasets from LAW [14] and SNAP [60]. We evaluate on representative datasets from social networks, web crawls, product purchases, location, citation, and email data. We further test with the largest published and available synthetic graphs from the LDBC Graphalytics benchmark [46]: the Graph500 30-scale graph [18, 67] and the largest currently published Datagen `fb` and `zf` graphs [29].

In most cases, the available datasets are not as large as modern commercial or real-world datasets and these publicly available graphs are all easily run on a single, shared-memory system. To evaluate ElGA's performance over the variety of datasets it is intended for, along with the intended scale, we use A-BTER [74]. This takes an existing graph—that can fit into shared memory—computes degree and clustering coefficient distributions, and then generates

Table 2: The graphs used in our experiments.

Graph	A-BTER Scale	n	m	EL Size
Twitter-2010 [55]	-	42 M	1.5 B	25 GB
Friendster [90]	-	65 M	1.8 B	31 GB
UK-2007-05 [13, 14]	-	105 M	3.7 B	63 GB
Datagen-9.3-zf [29]	-	555 M	1.3 B	34 GB
Datagen-9.4-fb [29]	-	29 M	2.6 B	65 GB
Email-EuAll [59]	×5000	1.3 B	5.6 B	105 GB
Skitter [59]	×200	339 M	6.3 B	119 GB
LiveJournal [9, 61]	×100	484 M	8.6 B	161 GB
Amazon0601 [57]	×2000	807 M	9.8 B	183 GB
Graph500-30 [18, 67]	-	448 M	17 B	319 GB
Gowalla [21]	×10000	2.0 B	28 B	568 GB
Patents [59]	×1000	3.7 B	33 B	673 GB
Pokec [79]	×1000	1.6 B	44 B	898 GB
Pokec [79]	×2500	4.0 B	112 B	2.3 TB

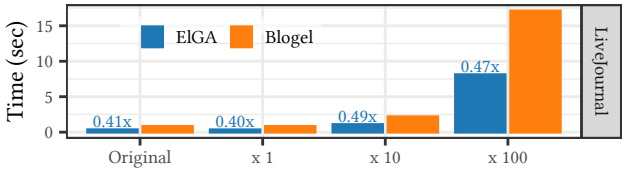


Figure 4: Per-iteration runtime of PageRank on LiveJournal with three A-BTER generated LiveJournal-like graphs. The relative runtimes (shown in blue), i.e., ratio between ElGA's and Blogel's runtimes remain consistent.

random *scaled up* graphs that share the same distributions and properties. It is important to test on larger datasets as unexpected inefficiencies and complications can arise at scale. By using A-BTER, we are able to experiment on a variety of representative graphs *at scale*. Table 2 shows the datasets used in our experiments.

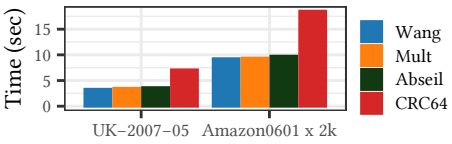
To understand how performance may change with A-BTER synthetic graphs, in Figure 4 we show the runtime on the original LiveJournal [9, 61] graph, a synthetic version with no scale difference (×1), and two scaled-up versions (×10 and ×100). While there are some relative differences in system performance as the graphs' scale increases, the runtime at the same scales matches well for each system. These results extend to the other graphs we tested, showing that A-BTER is a valuable tool for evaluating performance of graphs with varying structure at larger scales.

We extended A-BTER to stream edge updates, allowing ElGA to directly receive the graph as it is generated.

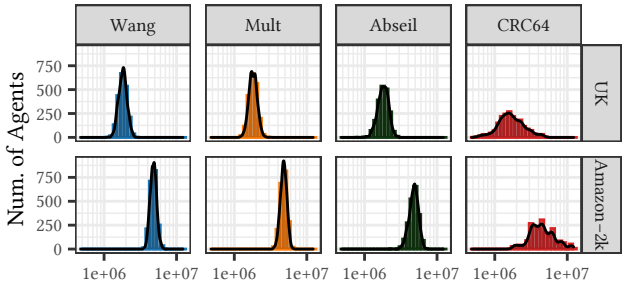
While all of these graphs are temporal and dynamic, the given datasets do not have edge deletion or insertion information. As such, we model their dynamic change by first deleting a random sample of edges and second adding the sample back in, as a batch.

4.5 Design Choices

We want to understand the impact of various design choices in ElGA. First, we look at the hash function. The hash function plays a crucial role: it is used, along with virtual Agents and the CountMinSketch, for every edge *access* to find the corresponding Agent. It needs



(a) PageRank Iteration Runtime



(b) Histograms of Edges per Agent

Figure 5: The hash function has a large impact on the runtime. We found that Wang's 64-bit integer hash performs the best. The runtime performance follows the quality of the edge distributions. Ideal is a single vertical line.

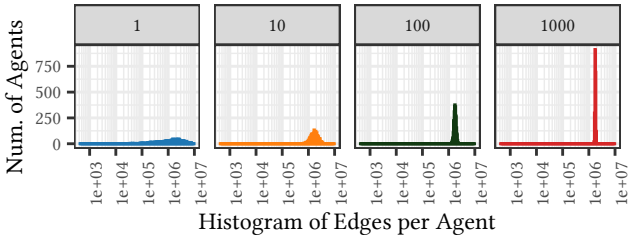
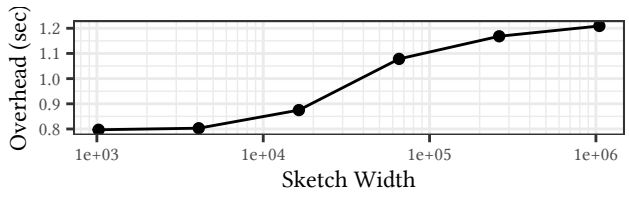


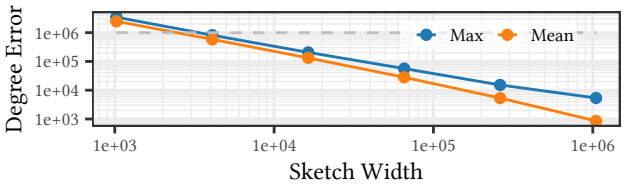
Figure 6: The load balance distributions for 2048 Agents as the number of virtual agents per Agent is varied from 1 to 1000 for Twitter-2010. Beyond 100 improvements do not outweigh the computational cost (not shown here).

to be both fast and of high quality, that is, result in a uniform distribution. We evaluate different hash functions in Figure 5. Mult is from [75], Abseil is a non-deterministic hash similar to Mult used in the Abseil C++ library, and CRC64 [25]. Due to the number of calls to the consistent hashing system, we use a 64-bit ring and avoid more expensive hash functions, such as the commonly used cryptographic ones [23]. Thomas Wang's [85] is the best performing hash function tested and we use it for the remaining experiments.

Second, we show the impact of the number of virtual agents. With more virtual agents, a larger number of nodes will need to receive or send new edges when an Agent leaves or joins and each consistent hash lookup is more expensive. However, the number of edges per transfer decreases and the load balance improves. In Figure 6, we show the load balance distribution for Agents as the number of virtual agents *per Agent* is varied for Twitter-2010. We found a similar behavior for all graphs. At 100 virtual agents per Agent the distribution both has a high quality and sufficiently low lookup rates, and so we use this value for all graphs.



(a) Overhead per PageRank Iteration on Twitter-2010



(b) Maximum and Average Degree Errors on Twitter-2010

Figure 7: The runtime cost of resolving edges to Agents along with the degree estimation error as the table width varies. With a replication threshold even as low as 2 million, any max degree error below the dashed line (at 1 million) means the sketch will result in no replication error.

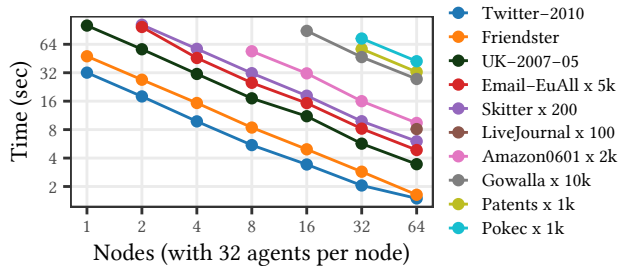


Figure 8: The scalability of ElGA reporting PageRank iterations as the number of nodes are varied. The larger graphs run out of memory on small numbers of nodes.

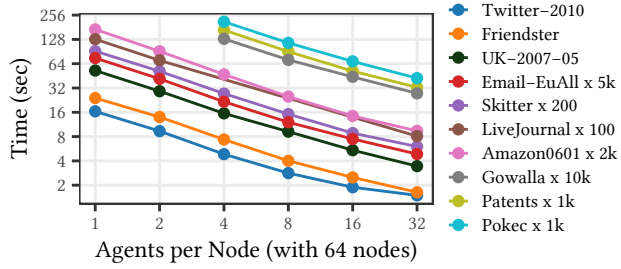


Figure 9: The scalability of ElGA reporting PageRank iterations as the number of Agents per node are varied.

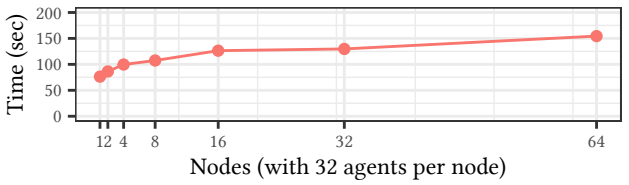


Figure 10: ElGA's weak scaling with the Pokec dataset. The scale ranges from ×39 to ×2500. A horizontal line is ideal.

Third, we evaluate our CountMinSketch parameters. Our sketch approach enables high-degree vertices to be split across multiple Agents within ElGA. Each split incurs an overhead, and so we only want to target vertices that cause significant load imbalance or memory pressure and reduce the number of unnecessary replications. In Figure 7, we explore the CountMinSketch width parameter. We vary the width and compute both the runtime overhead per PageRank iteration and the maximum and average error in degrees. We set the threshold high for replicating, at 10^7, and so we can use a small sketch size of $10^{4.2}$, below the inflection point for added overhead and without replication error across our datasets.

4.6 Scalability

We next study our scalability and show that the shared-nothing architecture is able to both strongly and weakly scale. We focus on PageRank and show that with ElGA it scales up, both with more cores per machine and more machines. In Figure 8, we vary the number of nodes and report the per-iteration PageRank time. For each graph, adding more nodes results in lower runtimes. As more memory is required per node with fewer nodes, the large graphs stop fitting into memory and so we cannot report their runtimes. In Figure 9, we keep the number of nodes fixed at our cluster size, 64, and instead vary the number of Agents that we run on each node. Similarly, adding more Agents results in faster runtimes.

Next, we look at weak scaling. We show the per-iteration runtime of PageRank as we scale the Pokec dataset from 1.7 billion edges to 112 billion, while keeping the degree and clustering coefficient distributions within 2% error. The results are in Figure 10. With only a few nodes, and a significantly reduced amount of communication, ElGA performs better per edge than itself at a larger scale. However, at these scales many real-world graphs will not fit into memory. Above 16 nodes our scaling is close to ideal, a horizontal line.

4.7 Comparison with Static State-of-the-art

We compare PageRank iterations against *static baselines* in Figure 11. Given Blogel-Vor's poor performance against Blogel we do not show it in the results. GraphX runs out of memory on the largest graphs. ElGA is designed to handle a constantly and rapidly changing graph, yet we outperform the baselines even when ignoring partitioning time and other static costs of those systems. This is a surprising result. Both Blogel, the second fastest, and ElGA use C++. Blogel uses a CSR internally to hold the graph which is faster than our flat hash maps (but do not easily support dynamic graphs). Further, Blogel uses MPI, and as we showed in § 3.5 MPI has 20× lower packet latencies on our cluster. The underlying filesystems's I/O performance is not included in the timing results here.

As our algorithms are the same, we attribute our performance to an interesting property not well explored in distributed and scalable graph systems: even with bulk-synchronous parallel algorithms, allowing messages to arrive out-of-order, with communication handled in separate threads, keeping global state limited, and using a shared-nothing server-based polling system inside each Agent provide significant benefits. We are able to accommodate the increase in computation with per-edge Agent lookups as Blogel is fastest with 8 cores per node, likely due to MPI allreduces saturating the network, and we take advantage of all cores.

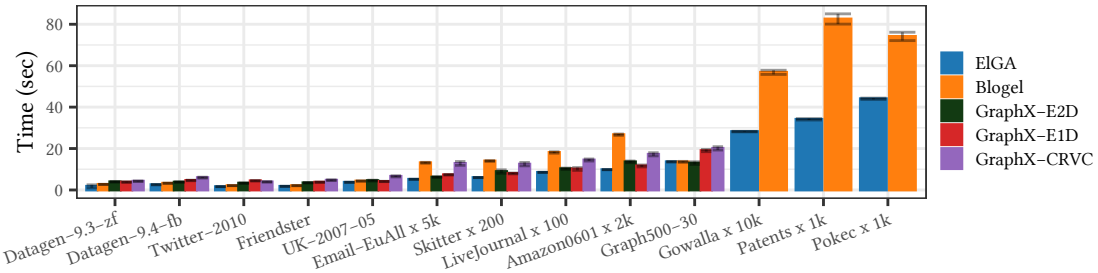


Figure 11: ElGA's per-iteration PageRank runtime compared against Blogel and GraphX, using 64 nodes. GraphX includes a significant partitioning overhead (not shown here) and ran out of memory on the larger graphs. A t-test shows ElGA is fastest with $p < 0.0005$ in all datasets except for Graph500-30, which is inconclusive with $p > 0.05$. 95% confidence intervals are shown.

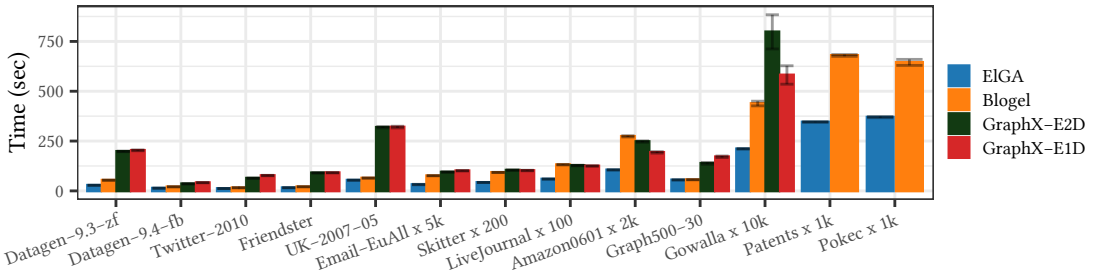


Figure 12: The weakly connected components runtime for ElGA, Blogel, and GraphX. In all cases, a t-test shows ElGA is fastest with $p < 0.0005$ (except Graph500-30, where ElGA is fastest with $p < 0.03$). 95% confidence intervals are shown.

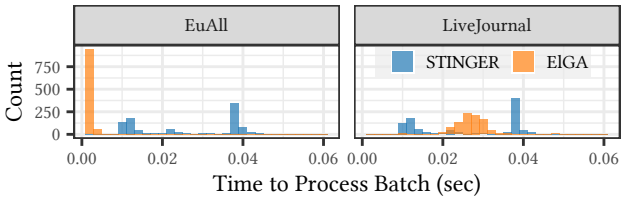


Figure 13: ElGA and STINGER maintaining components.

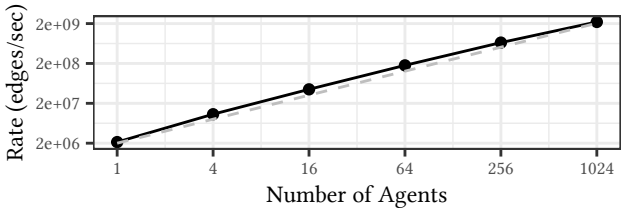


Figure 14: The insertion rate of edges from Skitter. Agents densely fill nodes, so 64 Agents run across two nodes. The dashed line shows ideal linear scaling.

To evaluate weakly connected components, we had to fix a bug in Blogel where it does not consider an undirected form of the graph. We did this by symmetrizing the input graph (which along with other I/O costs is not shown). The results are shown in Figure 12. We were not able to run GraphX with CRVC partitioning as it ran out of memory on almost all graphs. Similarly, Blogel-Vor did not provide competitive results and so is not shown. All of our results show better performance for our tuned baselines than experiments reported in [7] (with Haswell, not Ivy Bridge CPUs and more RAM).

4.8 Comparison with Single Node Systems

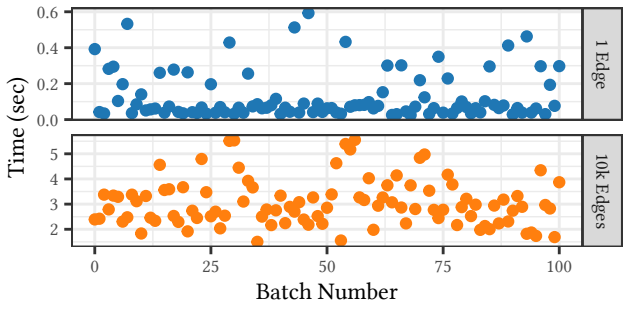
We next consider the COST [65] of running ElGA on a single node compared with a specialized inherently shared-memory algorithm and system. We are only aware of one publicly available implementation of a dynamic WCC, which is part of STINGER [26]. We were unable to run STINGER on billion-edge graphs; we instead run LiveJournal and EuAll at their original scale. In Figure 13, we show the insertion of the last 1000 edges. STINGER can likely optimize for some easy batches due to its global view. It has a bimodal distribution that is surprisingly similar across graphs. For LiveJournal, ElGA's median runtime is 0.027 seconds; STINGER's is 0.032.

Note that STINGER does not include any distributed support and uses OpenMP to parallelize. We also compared with GAPbs [10], a shared-memory parallel static graph system. GAPbs takes 0.94 seconds, including building its CSR from an in-memory edge list and running WCC.

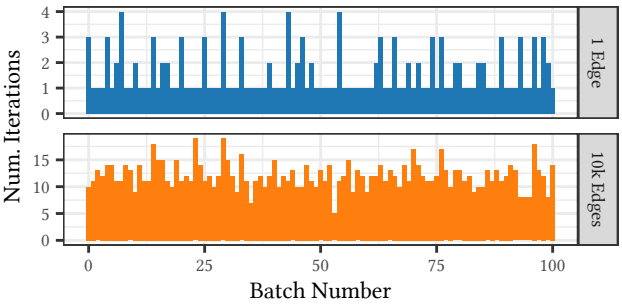
4.9 Dynamic Behavior and Elasticity

Even though for static cases it scales and performs well, ElGA is primarily designed to be both *dynamic* and *elastic*. An important consideration for a dynamic graph system is the rate of edge changes it can accept. In Figure 14 we show the edge insertion rate for Skitter as the number of nodes changes. We configure half of the cluster to be Streamers to send the graph. The performance is above 2 million edges per second per Agent and scales well.

We next show how important having a *dynamic* system can be. For snapshot-based systems or partially dynamic systems, such as GraphX, the standard approach is to initialize the iterative algorithm with prior outputs, re-initialize any new or changed vertices, and



(a) The runtime for each batch. Note differing scales.

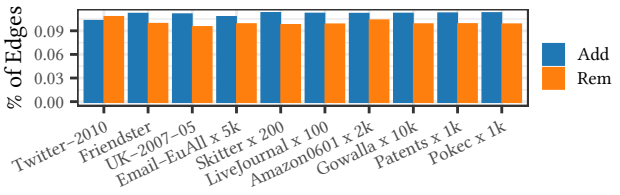


(b) The iterations until convergance for each batch. Note differing scales.

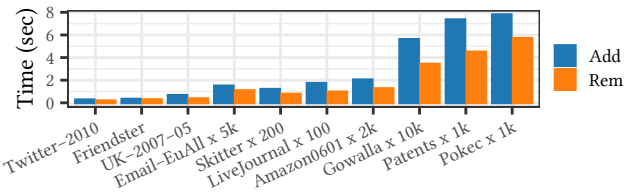
Figure 15: Maintaining connectivity on Twitter-2010. From scratch, ElGA takes 14 seconds. (Not shown here: GraphX takes over 49 seconds for one iteration, due to startup/teardown costs; its recomputation takes 66 seconds.)

run the iterative algorithm to convergence. If the new graph differs only slightly from the prior graph, the number of iterations may be small, resulting in significant savings over re-computing from scratch. This is the model used by many of the extensions to make GraphX dynamic, e.g., Sprouter [2] and EdgeScaler [69]. A fully dynamic system, on the other hand, can quickly change parts of the graph and only compute with vertices that are active, closer to an asynchronous model. As our dynamic baseline, we *completely ignore partitioning costs* in GraphX and instead use the strategy above, re-initializing only changed vertices and running WCC. By ignoring partitioning costs, we show the best achievable performance if a perfect elastic load balancer is put into GraphX. In Figure 15, we show ElGA inserting 100 batches into Twitter-2010. Even on single edge changes, our GraphX baseline never took less than 49.45 seconds. Given ElGA's minimum, average, and maximum runtimes of 0.025, 0.12, and 0.59 seconds on single edge changes, we achieve speedups between 83× to 1962×.

It is important to be able to scale both up and down as the graph size and rate of changes varies. Our architecture is designed to support this. In Figure 16 we show the ratio of edges that moved across ElGA when an Agent joins and a random one leaves for each graph. This shows that ElGA can elastically scale as needed without incurring significant overheads. In Figure 17, PageRank runs for five iterations on Gowalla starting with 16 nodes. After one iteration, an operator manually scales the cluster from 16 to 64 nodes. ElGA elastically scales and continues the computation,



(a) The percent of edges moved when adding and then removing an Agent



(b) The total time to add and then remove a single Agent

Figure 16: The cost of adding and removing one Agent, starting from 2048. Multiple Agent changes amortize the cost.

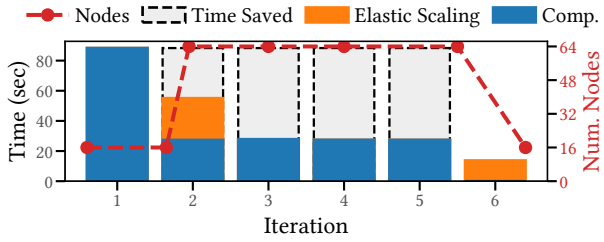


Figure 17: PageRank running on Gowalla, manually scaled to 64 nodes during computation and then back to 16.

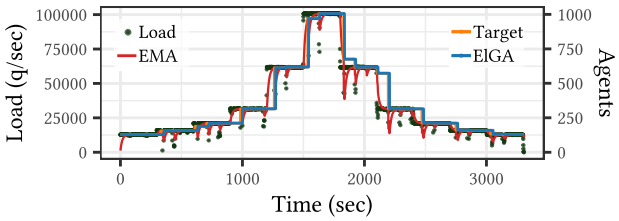


Figure 18: Fully elastic autoscaling in ElGA. ElGA converges quickly to match the autoscaling target (Target).

improving the overall runtime. Finally, after the computation is over, the cluster is reduced back, providing cost savings.

We next evaluate ElGA's elasticity. We implemented a reactive autoscaler that takes a fraction of the exponential moving average (EMA) of 30 seconds of client PageRank vertex query rates to determine the target Agent count. It then waits for 60 seconds before potentially scaling again to allow the EMA to stabilize. Any suitable autoscaler or scaling measures can be used [45]. We varied client request rates with a step function to emulate sudden workload changes on Skitter. The results are shown in Figure 18. ElGA quickly converges to the autoscaler's target, evidenced by the mostly overlapping blue and orange lines, and hence elastically matches the load.

5 CONCLUSION

As graphs continue to grow, many dynamic and large-scale graph algorithms, together with distributed graph processing systems, have been developed. The majority of these efforts, however, fail to consider the high variance from both the distributed graph's natural rate of change and the inherent variance in dynamic computations. We present ElGA, an elastic and scalable dynamic graph analysis system. ElGA performs computation as the underlying graph changes, and can scale as computational demands change. It accomplishes this by combining a shared-nothing architecture with consistent hashing and, to handle heavily skewed graphs, count-min sketches. We show that ElGA outperforms state-of-the-art graph processing systems in terms of runtime on both static and dynamic graph algorithms over variety of real world graphs and their scaled-up replicas.

ACKNOWLEDGMENTS

We thank M. Mucahid Benlioglu for his help on an earlier version of the software. This work was funded in part by the NSF under Grant CCF-1919021 and in part by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. Streaming graph partitioning: An experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.

[2] T. Abughofa and F. Zulkernine. Sprouter: Dynamic graph processing over data streams at scale. In *International Conference on Database and Expert Systems Applications*, pages 321–328. Springer, 2018.

[3] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14, 2012.

[4] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2017.

[5] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, page 32–42. SIAM, 1990.

[6] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.

[7] K. Ammar and M. T. Özsu. Experimental analysis of distributed graph systems. *Proceedings of the VLDB Endowment*, 11(10):1151–1164, 2018.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] 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.

[10] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[11] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *arXiv preprint arXiv:1912.12740*, 2019.

[12] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*, 2019.

[13] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[14] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[15] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53(1-13):2, 2008.

[16] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao, et al. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 329–344, 2020.

[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[18] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[19] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[20] 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 *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[21] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090, 2011.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[24] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[25] ECMA. Data interchange on 12,7 mm 48-track magnetic tape cartridges - dlt1 format.

[26] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[27] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

[28] B. Erb, D. Meissner, J. Pietron, and F. Kargl. Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 78–87, 2017.

[29] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630, 2015.

[30] W. Fan, C. Hu, M. Liu, P. Lu, Q. Yin, and J. Zhou. Dynamic scaling for parallel graph computations. *Proceedings of the VLDB Endowment*, 12(8):877–890, 2019.

[31] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 155–169. ACM, 2017.

[32] W. Fan, P. Lu, W. Yu, J. Xu, Q. Yin, X. Luo, J. Zhou, and R. Jin. Adaptive asynchronous parallelization of graph algorithms. *ACM Transactions on Database Systems (TODS)*, 45(2):1–45, 2020.

[33] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)*, 43(4):1–39, 2018.

[34] A. Fidel, N. M. Amato, L. Rauchwerger, et al. Kla: A new algorithmic paradigm for parallel graph computations. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 27–38. IEEE, 2014.

[35] S. K. Garg, S. Versteeg, and R. Buyya. A framework for ranking of cloud computing services. *Future Generation Computer Systems*, 29(4):1012–1023, 2013.

[36] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. *External memory algorithms*, 50:39–70, 1999.

[37] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[38] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[39] S. Heidari and R. Buyya. A cost-efficient auto-scaling algorithm for large-scale graph processing in cloud environments with heterogeneous resources. *IEEE*

Transactions on Software Engineering, 2019.

[40] S. Heidari, R. N. Calheiros, and R. Buyya. igiraph: A cost-efficient framework for processing large-scale graphs on public clouds. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 301–310. IEEE, 2016.

[41] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Computing Surveys (CSUR)*, 51(3):1–53, 2018.

[42] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, 2013.

[43] P. Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[44] P. Holme and J. Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[45] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 75–86, 2017.

[46] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, et al. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

[47] A. Iyer, L. E. Li, and I. Stoica. Celliq: Real-time cellular network analytics at scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 309–322, 2015.

[48] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.

[49] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. Asap: fast, approximate graph pattern mining at scale. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, pages 745–761, 2018.

[50] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *International Workshop on Peer-to-Peer Systems*, pages 98–107. Springer, 2003.

[51] B. Kahveci and B. Gedik. Joker: Elastic stream processing with organic adaptation. *Journal of Parallel and Distributed Computing*, 137:205–223, 2020.

[52] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[53] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1149–1164, 2017.

[54] V. Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[55] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[56] M. M. Lee, I. Roy, A. AuYoung, V. Talwar, K. Jayaram, and Y. Zhou. Views and transactional storage for large graphs. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 287–306. Springer, 2013.

[57] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5–es, 2007.

[58] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006.

[59] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[60] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[61] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[62] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[63] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[64] A. McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[65] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[66] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[67] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[68] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[69] D. Presser, F. Siqueira, L. Rodrigues, and P. Romano. Edgescaler: effective elastic scaling for graph stream processing systems. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 39–50, 2020.

[70] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell. Supporting on-demand elasticity in distributed graph processing. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 12–21. IEEE, 2016.

[71] @raffi. New tweets per second record, and how! https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how, 2013. Accessed June 28, 2021; Archive at https://web.archive.org/web/20210628160850/https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.

[72] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1-2):233–277, 1996.

[73] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[74] G. M. Slota, J. W. Berry, S. D. Hammond, S. L. Olivier, C. A. Phillips, and S. Rajamanickam. Scalable generation of graphs for benchmarking hpc community-detection algorithms. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[75] G. L. Steele Jr, D. Lea, and C. H. Flood. Fast splittable pseudorandom number generators. *ACM SIGPLAN Notices*, 49(10):453–472, 2014.

[76] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[77] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[78] T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 1321–1326, 2014.

[79] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*, 2012.

[80] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From" think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[81] L. Toader, A. Uta, A. Musaafir, and A. Iosup. Graphless: Toward serverless graph processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 66–73. IEEE, 2019.

[82] A. Uta, S. Au, A. Ilyushkin, and A. Iosup. Elasticity in graph analytics? a benchmarking framework for elastic graph processing. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 381–391. IEEE, 2018.

[83] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *2014 IEEE 34th international conference on distributed computing systems*, pages 144–153. IEEE, 2014.

[84] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, pages 237–251, 2017.

[85] T. Wang. Integer hash function, 1997. https://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm.

[86] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[87] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[88] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421, 2015.

[89] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[90] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

This paper includes a variety of experiments that run both PageRank and weakly connected components on existing public graphs and their scaled-up versions. We ran on the Carnac cluster at Sandia National Laboratories. We used 64 nodes plus one additional node for metadata connected via an Arista 7500E switch with 100Gbps networking.

The experiments were run by using pdsh to start ElGA executables on each node. The executables were pinned to cores using `numactl`. The underlying filesystem was a Ceph filesystem and everything was setup on CentOS. The Ceph MDS and Hadoop metadata servers were run on a separate, 65-th machine. The baselines were run with OpenMPI installed via Spack, and the latest versions of Hadoop and Spark downloaded and installed by untarring into a shared directory and running `sbin/start-all.sh`. Makefiles and other compilation scripts were created for Blogel. `mimalloc` 1.7.0 was used for compiled binaries.

Both Blogel and ElGA were compiled with G++ version 8.3.1 with `-O3`. We ran GraphX, Hadoop, and Spark with OpenJDK version 1.8.0-292. We used Spark 3.1.1 compiled with Hadoop 3.2 support. We used Hadoop 3.2.2. Spark was configured with the following defaults, which was found to produce the fastest GraphX runtimes:

```
spark.network.timeout 10000000
spark.executor.heartbeatInterval 9000000
spark.driver.memory 5g
spark.executor.memory 400g
spark.shuffle.service.enabled false
spark.executor.extraJavaOptions -XX:+UseG1GC -Xms400g
    -XX:ConcGCThreads=20 -XX:ParallelGCThreads=20
spark.dynamicAllocation.initialExecutors 1/2/4/8/16
spark.dynamicAllocation.minExecutors 1/2/4/8/16
spark.dynamicAllocation.maxExecutors 1/2/4/8/16
```

The following values were used in the Spark env:

```
SPARK_WORKER_CORES=64
SPARK_WORKER_MEMORY=400G
SPARK_WORKER_DIR=/scratch/data/
SPARK_DAEMON_MEMORY=20G
```

A standalone 1TB Samsung 860 SSD was mounted at `/scratch/data`. The systems had the following networking related values set in `/etc/sysctl.conf`:

```
net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_no_metrics_save = 1
net.ipv4.tcp_max_syn_backlog = 65536
net.core.netdev_max_backlog = 65536
net.core.somaxconn = 65536

net.core.wmem_max = 12582912
net.core.rmem_max = 12582912
net.ipv4.tcp_rmem = 10240 87380 12582912
net.ipv4.tcp_wmem = 10240 87380 12582912
```

The systems were tuned with `tuned-adm profile hpc-compute`.

The scaling experiments were run by using pdsh to start ElGA on different numbers of nodes. The directory master and directory were all started on one of the compute nodes, but with overloaded IP addresses. The comparisons between different components of ElGA were all performed by recompiling after modifying the `CONFIG` flags inside of ElGA. To elastically scale up, new agents were added, again using pdsh, and to elastically scale down, `SIGINT` was issued using `kill` for the ElGA agents that were being removed. Note that `SIGINT` is captured by ElGA and a graceful elastic agent shutdown then occurs.

To create the ABTER-scaled graphs, the default values did not produce graphs with under 5% error for degree distributions or clustering coefficient distributions. By performing a parameter search (after modifying ABTER to report errors), we used the following non-default values.

Table 1: Non-default values used for ABTER scaling.

graph	scale	cavg
Gowalla	10000	2 (and cmax 0.9)
Email-EuAll	5000	0.46
Amazon0601	2000	1.75
Pokec	1000	2.08
Patents	1000	2.2
Skitter	200	2

The `scripts/` directory contains scripts to start ElGA, interact with the binary, and run the experiments. It contains a README that explains the variables to set and how to configure it for the specific cluster.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/GT-TDAlab/ElGA
Artifact name: ElGA
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz with 500 GB RAM

Operating systems and versions: CentOS 8.3

Compilers and versions: GCC 8.3.1, OpenJDK 1.8

Applications and versions: ABTER, ElGA

Libraries and versions: OpenMPI 3.1.4

Input datasets and versions: SNAP datasets and LAW graph datasets