Survey of Techniques for Large-Scale Distributed Graph Processing

Jonathan Brophy
Information and Computer science Department
University of Oregon
jbrophy@cs.uoregon.edu

Abstract— **With the recent increase of relational data such as social networks, web graphs, and protein interactions, techniques such as *MapReduce*[1] are not sufficient to efficiently handle data of this nature. This survey presents the motivation behind developing optimized solutions to process large-scale graphs, starting with a non-distributed framework called *GraphChi*. This paper moves on to explore the latest distributed graph processing abstractions such as *GraphLab, GPS, Trinity, and GraphX*, in which each system has derived influence from earlier works, namely *Pregel*. With many options to choose from, this paper highlights the key differences from system to system. For example, many real world graphs follow a power-law degree, in which the *PowerGraph* abstraction attempts to address. Finally, one characteristic common to all graph computational problems involves the partitioning of the graph data into different machines, and a new technique shows the benefits of stream partitioning that can be used in conjunction with the state-of-the-art offline partitioner, *Metis*[2], to further reduce network I/O and increase performance of distributed graph processing algorithms.**

Keywords— **Graph Processing, Large-Scale, Distributed, Stream Partitioning**

I. Introduction

A. Motivation

Much of the newly emerging data sets today manifest themselves as large interconnected graphs. This poses a real problem for researchers that want to learn from this data. Many current data-parallel processing frameworks, like *MapReduce*, are insufficient in computing the types of Machine Learning and Data Mining (MLDM) algorithms that these graphs require.

[1]http://hadoop.apache.org/
[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

A.1 Main Issues

Two questions arise while attempting to address the main motivation mentioned above. First, will a system built specifically to handle large graphs actually be any better than using state of the art data-parallel frameworks, such as *Spark*[3]? Second, does this system need to be distributed? If so, is the complexity and overhead of insuring fault tolerance, synchronous and/or asynchronous communication paradigms, and reducing potentially high network I/O involved in a distributed graph processing system worth the performance increase, if any?

B. Single Machine Graph Processing

This section attempts to answer the first question of whether or not a non-distributed graph processing system can beat some of the state-of-the-art data-parallel frameworks while computing the same MLDM algorithms.

Open-source *GraphChi* [1] is a disk-based system that is able to process large-scale graphs on a single modern machine without the hassle of managing a cluster and implementing fault tolerance between nodes. This system uses a parallel sliding windows method to efficiently handle large volumes of data in chunks, while minimizing the number of non-sequential read and writes.

B.1 Parallel Sliding Windows (PSW)

In order to compensate for the limited memory available to a single machine, *GraphChi* stores a graph G = (V,E) on disk. The vertices V are split into P disjoint intervals, where each interval contains a shard. Each shard stores the in-edges E that

[3]http://spark.apache.org/

have the destination vertex in that interval, ordered by source vertex.

PSW executes one interval p at a time, loading the respective shard into memory. The entire subgraph for interval p is created by reading the out-edges from consecutive shards, requiring p-1 reads. Then, a user-defined update-function executes for each vertex in parallel. Since update-functions can modify edge values, if two adjacent vertices are in the same interval, they are updated in sequential order, otherwise edges are updated safely in parallel. Finally, PSW writes the modified blocks back to the disk, replacing the old data.

B.2 Applications

The open source implementation of *GraphChi* supports a wide range of graph processing algorithms. In the context of graph mining, this system focuses on a well known algorithm, Pagerank [2], as well as Triangle Counting [3] and Connected Components [4].

This system has also implemented collaborative filtering in the form of the Alternating Least Squares (ALS) algorithm in an attempt to solve the Netflix movie rating prediction problem [4].

Finally, from the world of probabilistic graphical models, a special Belief Propagation (BP) algorithm [5] was devised to execute BP on a graph of webpages to determine whether a page is "good" or "bad".

B.3 Scalability and Performance

We now compare the performance of *GraphChi* to a *Hadoop* implementation, as well as several distributed graph processing systems. All performance measures for *GraphChi* were evaluated using a Mac Mini 2.5 GHz Intel i5 processor, 8GB RAM, 256GB SSD, and 750GB 7200 rpm hard drive.

GraphChi manages a 7x speedup (Table II) against a Hadoop cluster containing 1636 nodes. *GraphChi* shows relatively comparable performance to GraphLab performing Pagerank on the Yahoo-Web graph, but is beaten by an order of magnitude on the twitter data set against PowerGraph for both Pagerank and Triangle Counting. This is most likely due to the power-law degree of each vertex that many real world graphs follow.

GraphChi outpaces any strict data-parallel framework and provides the opportunity for anybody with a modern computer to run a range of MLDM algorithms on large-scale graphs in a reasonable amount of time.

[4]https://en.wikipedia.org/wiki/Connected_component_(graph_theory)

TABLE I
DATASETS FOR *GraphChi* EVALUATION

ID	Graph Name	Vertices	Edges
1	twitter-2010 [6]	42M	1.5B
2	domain [5]	26M	0.37B
3	Synthetic-1	105M	3.7B

TABLE II
GraphChi PERFORMANCE RESULTS

Algorithm	Graph	Competitor	GraphChi
Pagerank	2	GraphLab: 87s	132s
Pagerank	3	GPS: 144m	581m
Tri Count	1	Hadoop: 423m	60m
Pagerank	1	PowerGraph:3.6s	158s
Tri Count	1	PowerGraph:1.5m	60m

C. *Multi-Machine Graph Processing*

Although the performance of *GraphChi* is impressive, performance may still be an issue for various groups and companies. We have just seen examples of distributed graph processing systems beating an optimized single-machine graph processing method, which partially answers the second question posed in section A.1 of whether or not these graph processing systems need to be distributed.

The following sections in this paper go through the different distributed graph processing abstractions and highlight their key commonalities and differences to get a sense of which abstraction is most appropriate for a specific application.

II. Distributed Graph Processing

Now that we have seen some potential performance benefits in terms of execution time for a small set of distributed graph processing systems, it is important to look at each one in more detail. This survey looks at the 5 most recent (2012 and later) distributed graph processing techniques, *GraphLab, Trinity, GPS, PowerGraph, GraphX*, and one slightly older technique, *Pregel* (2010). The reason *Pregel* is included in this survey is because many of the more recent techniques borrow important concepts from it, so it is useful to highlight some of the foundational aspects of *Pregel*.

The issues mentioned below will act as a road map for this paper as the various aspects of distributed graph processing systems are evaluated.

First, this paper looks into the architecture of each system, as design decisions made in this area can have a big impact into what types of applications it supports and how efficiently it can run certain algorithms.

Second, the computational model for each abstraction is evaluated. This section also depicts which kind of communication paradigms they offer. Some abstractions may consider synchronous over asynchronous communication, while some use both.

Third, depending upon the computational model that the abstraction has chosen, the number of applications and algorithms they can support may vary from system to system.

Fourth, failures are always something that a distributed system must deal with, so the way in which these abstractions implement various forms of fault tolerance is very important.

Fifth, performance is evaluated for each system. How well do these systems really scale, and has all the effort into building these systems been worth it?

Finally, each abstraction is compared according to the API and usability of their system. Is the headache of trying to work with one of these techniques enough to switch to another one?

By comparing the techniques in each of these important aspects, we can get a sense of what the newest technology out there has to offer, and make an appropriate decision of what abstraction to use in regards to the problem at hand.

A. *Architecture*

The architecture of each system deals with how they represent graph data in a distributed environment, how the data is stored (whether it be in memory or on disk, or some combination of the two), and the communication and execution patterns between compute nodes.

A.1 Distributed Graph Representation

The *Pregel* [7] library divides a graph into partitions, each consisting of a set of vertices and all of those vertices' outgoing edges. The partitioning function is just a hash of the vertex ID MOD N, where N is the number of partitions, but users can replace this function. Persistent data is stored as files on a distributed storage system, whereas temporary data is stored on the local disk. *Graph Processing System (GPS)* [8] is an open-source implementation of the *Pregel* abstraction, and thus follows the same representation and execution models outlined in *Pregel*.

GraphLab [9] also uses a distributed file system to store the graph vertices and edges as atoms (binary compressed files containing commands to rebuild parts of the graph), as well as a third parameter D, which is user-defined data that can refer to model parameters, algorithm state, or statistical data.

Trinity [10] designed its own memory cloud, which is essentially a key-value store, where the value in a key-value pair is a blob of arbitrary size. The memory cloud consists of 2^p memory trunks, where each machine hosts multiple memory trunks. *Trinity* also implemented a circular memory management mechanism which tries to avoid memory gaps between a large number of key-value pairs. The graph is represented by cells (graph nodes) which contain a set of cellIds that represent its neighboring nodes. For directed graphs, a cell contains two sets of cellIds: one for incoming links, and another for outgoing links. It may also contain other information such as the name of the node, its description, etc.

GraphX [11] is different from the other graph processing systems in that it attempts to bridge the gap between distributed dataflow frameworks like *Hadoop* and *Spark* and graph processing abstractions like *Pregel*. *GraphX* is a thin layer that sits on top of *Spark* that aims to make the analytics pipeline more efficient by combining graphs with unstructured and tabular data. A property graph represents graph structured data, and this property graph can be represented as a pair of vertex and edge property collections. *GraphX* has a vertex collection containing the vertex properties uniquely keyed by the vertex identifier, and an edge collection containing the edge properties keyed by the source and destination vertex identifiers. These collections are built on the Spark RDD abstraction. New property graphs can be constructed by composing different vertex and edge property collections.

A.2 Communication and Execution Paradigms

Work in *Pregel* is based off a master and slave approach, where the master determines and assigns one or more partitions to each worker machine. The master then instructs each worker to perform supersteps until all active vertices are inactive.

GraphLab contains two distributed execution engines. The Chromatic Engine, which executes the set of scheduled vertices T partially asynchronously, and the Locking Engine, which is fully asynchronous and supports vertex priorities.

By assuming each vertex communicates mainly with a fixed set of vertices such as its neighbors, *Trinity* can prepare most of the messages that a vertex will need before it is scheduled to run.

PowerGraph [12] provides three different models of execution: Bulk Synchronous (Sync), which is a fully synchronous implementation of *PowerGraph*, Asynchronous (Async), an asynchronous implementation of *PowerGraph* that allows arbitrary interleaving of vertex-programs, and Asynchronous Serializable (Async+S), another asynchronous implementation of *PowerGraph* that guarantees serializability of all vertex-programs.

GraphX introduces a multicast join in which each vertex property is sent only to the edge partitions that contain adjacent edges.

B. Computation

The computational model chosen by each abstraction has large implications about how fast the system will perform on different algorithms with varying graph sizes.

B.1 Vertex-Centric Model

Pregel was one of the first graph processing systems to really embody the vertex-centric model of computation based on Valiant's Bulk Synchronous Parallel Model [13]. In this model, computations consist of a sequence of iterations, called supersteps. In each superstep, the framework invokes a user-defined function for each vertex in parallel. A vertex V in superstep S can read messages sent to it in superstep S-1, send messages to other vertices that will be received in superstep S+1, and modify the state of V and its outgoing edges.

Since communication is from superstep S to superstep S+1, the synchronicity of this model makes it easier to reason about the program semantics when implementing a specific algorithm. *Pregel* also employs a pure message passing model that can deliver messages asynchronously in batches. *GPS* uses the vertex-centric model, but it is important to note that *GPS* is able to repartition the vertices of the graph across compute nodes based on their message-sending patterns.

GraphLab uses a similar model with a slight difference. In contrast to *Pregel*'s update functions, which are initiated by messages and can only access the data in a message, *GraphLab* allows the user defined update functions complete freedom to read and modify any of the data on adjacent vertices and edges.

Trinity also uses the vertex-centric model, but with a variation different to *Pregel* and *GraphLab*. In each superstep in *Pregel*, a vertex may send messages to any other vertex, whereas *Trinity* restricts sending messages only to a fixed set of vertices (usually its neighbors). This restriction allows for multiple optimization opportunities, and is more conducive to algorithms like Pagerank and Shortest Path[6], where vertices only talk with their neighbors.

B.2 Edge-Centric Model

Graphs derived from real world data, such as social networks, typically have power-law degree distributions, implying that a small subset of the vertices connect to a large fraction of the graph. The *PowerGraph* abstraction exploits the structure of vertex-programs and explicitly factors computation over edges instead of vertices.

B.3 Gather Apply Scatter (GAS) Model

GraphX implements the GAS decomposition by having the system ask the vertex program for the value of messages between adjacent vertices rather than having the user send messages directly from the vertex program. This enables vertex-cut partitioning, improved work balance, and reduced data movement.

C. Supported Applications

There are many different kinds of algorithms one can run on a graph. Some of these algorithms may run faster or more efficiently on certain techniques than others due to the way these systems have designed their architectures, communication paradigms, and computational models. The types of algorithms that have been tested on the different abstractions are laid out in more detail below.

C.1 Pagerank

Almost every distributed graph processing system used Pagerank as one its algorithms to implement for two reasons. First, Pagerank is a very well known algorithm that fits nicely into a vertex-centric computational model, which almost all of these abstractions adopt, in one form or another. Second, since Pagerank is such a popular algorithm, it is easy to find benchmarks from other graph processing frameworks to compare with.

The performance of each abstraction is compared in a later section, where the design and implementation details of differing execution and communication paradigms really start to show themselves.

[6]https://en.wikipedia.org/wiki/Shortest_path_problem

C.2 Probabilistic Graphical Models

Exact inference algorithms on graphical models is usually intractable, but other algorithms can be computed at scale with the appropriate system. Only *GraphLab* and *PowerGraph* contained any experiments applying algorithms to graphical models.

GraphLab performed collaborative filtering on the Netflix movie rating prediction problem, and *PowerGraph* performed *Gibbs Sampling*[7] on a *Wikipedia*[8] web graph.

C.3 Clustering and Searching

In addition to the algorithms already mentioned, *Pregel* and one of its open-source implementations, *GPS*, focuses on finding the Single Source Shortest Paths (SSSP) in a large-scale graph. *GPS* and *GraphX* both implemented a connected components algorithm, while *Trinity* focused on *Breadth-First Search*[9] and People Search on a social graph.

D. Fault Tolerance

Before looking at the performance of these models on differing algorithms and graph sizes, we must look into each one's technique of handling failures. This is a major detail when designing any distributed system, and that challenge is only made more difficult when trying to perform graph processing in an elegant and efficient manor at scale.

D.1 Checkpoints

Pregel achieves fault tolerance through checkpointing, where the master instructs the workers to save the state of their partitions to persistent storage at the beginning of a superstep. If the master does not hear back from a worker, then the master marks that worker process as failed. When one or more workers fail, the current state of the partitions assigned to these workers is lost and the master reassigns graph partitions to the currently available

[7]https://en.wikipedia.org/wiki/Gibbs_sampling
[8]https://www.wikipedia.org/
[9]https://en.wikipedia.org/wiki/Breadth-first_search

set of workers. *GPS* does not spell out their fault tolerance design, but since it is an implementation of the *Pregel* abstraction, it is safe to say they they follow the checkpoint method explained above.

GraphLab also follows a checkpoint design with two methods: a synchronous method that suspends all computation while the snapshot is constructed, and an asynchronous method that incrementally constructs a snapshot without suspending execution. Synchronous snapshots are constructed by suspending execution of update functions, flushing all communication channels, and then saving all modified data since the last snapshot. The asynchronous solution is based on the Chandy-Lamport [14] snapshot algorithm but tailored specifically to the *GraphLab* data-graph and execution model.

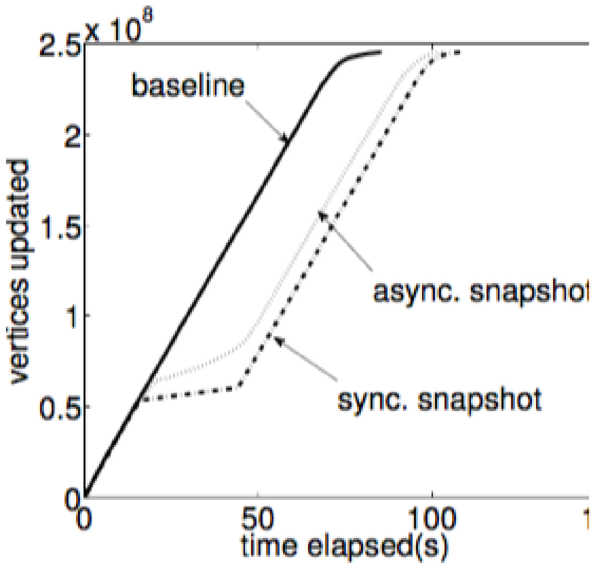

Fig. 1. *GraphLab* comparison of synchronous (completed in 109s) and asynchronous (completed in 104s) snapshot updates over 10 iterations.

After evaluating these two snapshot methods in the *GraphLab* abstraction, the difference is clear: synchronous snapshots halt execution while asynchronous snapshots only slow down execution (Fig. 1).

Trinity implements checkpoints every few supersteps for synchronous computation. For their asynchronous computation, they issue interruption signals periodically. Upon receiving this signal, all vertices pause after finishing the job at hand, and once the system has ceased, a snapshot is written to the persistent disk storage.

PowerGraph also takes snapshots of the data-graph, but does so between supersteps in the synchronous engine, and suspends execution in the asynchrnous engine to construct the snapshot.

D.2 Lineage-Based Fault Tolerance

Most of the graph processing systems only offer checkpointing as their means of fault tolerance. This is mainly due to the performance overhead in conjunction with the small incentive to implement recovery methods due to typically small mean times to failure of modern machines. As mentioned before, *GraphX* is built on *Spark*, which provides lineage-based fault tolerance [15] with negligible overhead as well as optional dataset replication.

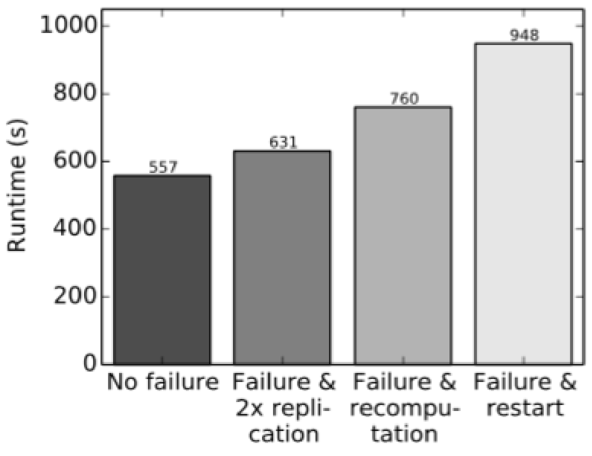

Fig. 2. *GraphX* running times for different fault tolerance mechanisms upon failure.

In contrast to other large-scale graph processing systems, data replication and recomputation in *GraphX* is faster than completely restarting after a failure (Fig. 2). There is also the added benefit that this fault tolerance is performed transparently by the dataflow engine.

D.3 Shared Table Maintenance

Trinity uses a shared addressing table to locate key-value pairs because a centralized implementation would introduce a bottleneck and a single source of failure. Using a shared table does introduce data inconsistency, but *Trinity* maintains a primary replica of the shared addressing table on a leader machine, and uses the fault-tolerant distributed file system (Trinity File System (TFS)) to keep a persistent copy of the primary addressing table. *Trinity* then uses heartbeat messages to detect machine failures. Upon detecting a failure, the leader machine reloads the data of the failed machine to other alive machines, updates the primary address table, and broadcasts it. If the leader machine fails, a new round of leader election will be triggered.

E. Performance

This section showcases the results of experimentation for each technique and attempts to answer the question of whether or not the pain and hassle of designing, implementing, and testing new large-scale graph processing systems is actually beneficial. This section will focus on the evaluation of running algorithms on realistic data sets (graphs that could actually be found in the real world), if provided by that system's report.

E.1 Single Source Shortest Path (SSSP)

The *Pregel* abstraction was tested with an SSSP implementation and run on a cluster of 300 multicore machines with a total of 800 worker tasks. It is important to note that since the execution time of these algorithms is relatively short, checkpointing was turned off for this experiment. The graphs built for this experiment use a log-normal distribution of outdegrees:

$$p(d) = \frac{1}{\sqrt{2\pi}\sigma d} e^{-(ln\ d - \mu)^2/2\sigma^2} \qquad (1)$$

By setting μ = 4 and σ = 1.3, the mean outdegree becomes 127.1, which resembles many real-world large scale graphs, such as web graphs or social networks, where most vertices have a relatively small degree but some nodes have a disproportionate number of the total edges. Five graphs were created and tested ranging from {10M, 100M, 200M, 500M, and 1B} vertices, with running times for these graphs being {~7s, ~90s, ~180s, ~420s, ~760s}, respectively. These times indicate that the running time basically increases linearly with the number of vertices in the graph, with the largest graph of 1B vertices and over 127B edges running in a little over 10 minutes.

E.2 Pagerank and Triangle Counting

GPS is the first of our graph processing systems to implement the Pagerank algorithm. This implementation ran Pagerank on a web graph of the .uk domain from 2007 (uk-2007-d:106M vertices, 3.7B edges) using 30 workers on 30 compute nodes.

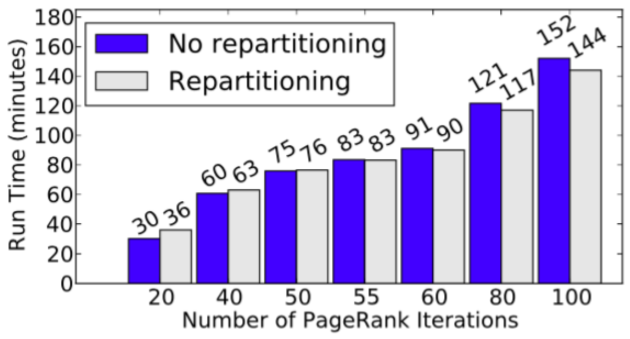

Fig. 3. *GPS* running times on Pagerank with and without dynamic repartitioning

At its biggest gains, dynamic repartitioning only improved the *GPS* running time of Pagerank by 1.13x (Fig. 3), which is modest at best. The area where dynamic repartitioning seemed to shine is in network I/O. In *GPS*, the master task turns dynamic repartitioning off once the number of vertices being exchanged is below a threshold, typically 0.1%. This occurs around 20 iterations of Pagerank, which is also where the network I/O performance really starts to increase. After 30 iterations of Pagerank, performance of network I/O increases by 2x. This shows the benefit of dynamic repartitioning on network I/O for long running algorithms.

Trinity also performed Pagerank using a synchronous vertex-centric computational model on R-MAT graphs [16]. The number of vertices vary from 64 million to 1024 million with each node having an average degree of 13. The fastest implementation involved 14 machines and had running times of {~3s, ~5s, ~11s, ~22s, ~40s} for {64M, 128M, 256M, 512M, 1024M} nodes, respectively. This means that for a graph of over 1B vertices and an average of 13.3B edges, *Trinity* can run one iteration of Pagerank in about 40s.

PowerGraph was designed with natural graphs in mind, focusing on graphs that follow a power law degree. Their experimentation involves a 64 node cluster of *Amazon EC2*[10] *Linux*[11] instances.

TABLE III
PowerGraph PAGERANK PERFORMANCE RESULTS AGAINST DISTRIBUTED DATA FLOW FRAMEWORKS.

System	Runtime	Machines
Hadoop	198s	50x8
Spark	97.4s	50x2
Twister [17]	36s	64x4
PowerGraph (sync)	3.6s	64x8

TABLE IV
PowerGraph TRIANGLE COUNTING PERFORMANCE RESULT AGAINST *Hadoop*.

System	Runtime	Machines
Hadoop	423m	1636x?
PowerGraph (sync)	1.5m	64x16

Both Pagerank and Triangle Counting were performed on a *Twitter*[12] follower network with 41 million vertices and 1.4 billion edges. The *PowerGraph* implementation of both algorithms is one to two orders of magnitude faster than published results of popular distributed data flow frameworks (Table III, Table IV).

GraphX is the last to perform Pagerank on a Twitter graph with 41.7 million vertices and 1.5 billion edges. This experiment uses 16 *Amazon* EC2 nodes, each with 8 virtual cores.

GraphX has shown (Fig. 4) that it is possible to achieve comparable performance to specialized graph processing systems using a general dataflow engine while gaining common dataflow features such as fault tolerance. *GraphLab* outperforms *GraphX* largely due to shared-memory parallelism; *GraphLab* without shared-memory parallelism is much closer in performance to *GraphX*.

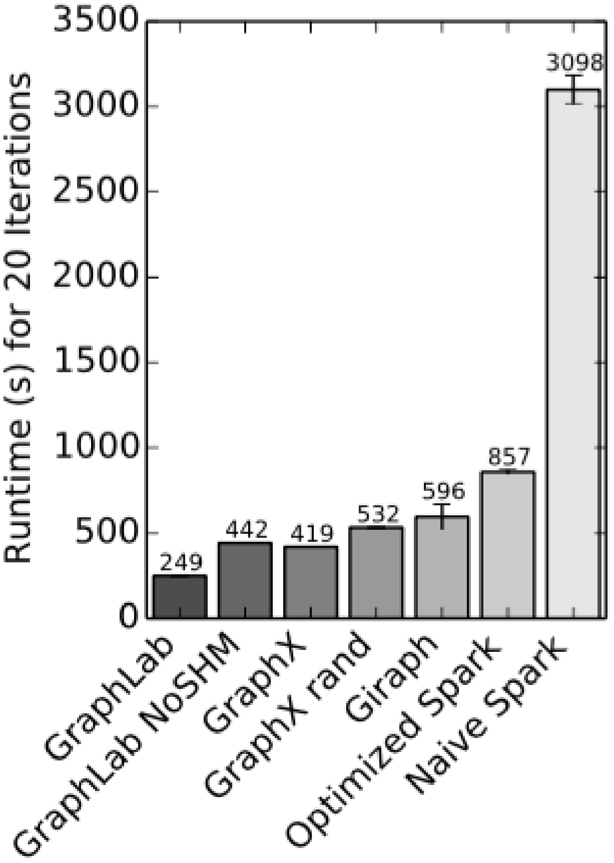

Fig. 4. *GraphX* running times on Pagerank against varying graph processing systems.

E.3 Netflix Prediction

One of the main experiments performed by the distributed *GraphLab* implementation is on the Netflix movie challenge, where an Alternating Least Squares (ALS) algorithm was chosen for the task. *GraphLab* measured its performance against *Hadoop* and a heavily optimized MPI implementation of ALS.

The *GraphLab* system used between 4 to 64 machines when comparing against *Hadoop* and the MPI implementation and found hat *GraphLab* performs between 40-60 times faster than *Hadoop*

[10]https://aws.amazon.com/ec2/
[11]https://www.linux.com/
[12]https://twitter.com/

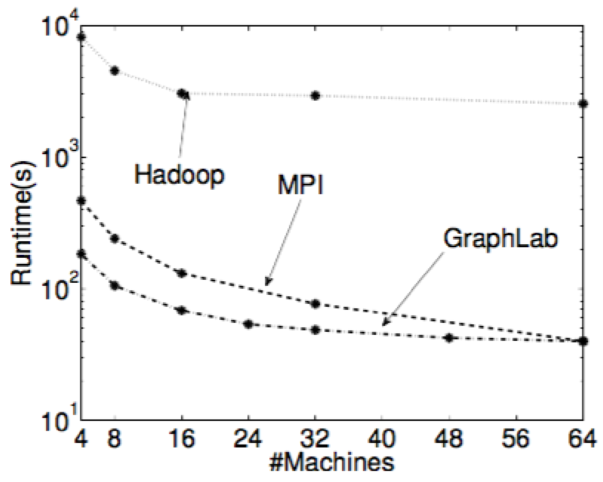

Fig. 5. Distributed *GraphLab* running times on the Netflix challenge against *Hadoop* and an MPI implementation.

(Fig. 5) and is comparable to the MPI implementation and actually slightly outperforms it.

F. Usability

This section analyzes how accessible each abstraction is and the hoops one must jump through to get their algorithm working with a particular system. If two graph processing systems perform similarly for a particular application, then a user may decide which system to use based on the programming effort required to get the results they want. This paper looks at the provided APIs for each system if there is one, and whether or not it is open source.

F.1 API

Pregel is an abstraction for a large scale graph processing system and thus provides a C++ API for other systems to implement. In order to write a *Pregel* program, one must subclass the predefined Vertex class, whose template arguments define three value types: vertices, edges, and messages. The user then overrides the Compute() method, which will be executed at each active vertex in every superstep. In this method, information can be queried about the current vertex and its edges, and messages can be sent to other vertices. This 'think like a vertex' approach to graph processing allows more intuitive reasoning about the algorithm that the user wants to implement, and *Pregel* has shown that satisfactory performance can be obtained with relatively little coding effort.

As mentioned previously, *GPS* is an implementation of the *Pregel* abstraction and follows their API. As an example of how expressive and easy to implement the vertex-centric computational model is with their API, *GPS* has implemented an HCC [18] algorithm to find the weakly connected components of an undirected graph in 17 lines of code using the Compute() method. Every vertex sets its own ID, then vertices set their values to the minimum value among their neighbors and their own value in iterations. When the vertex values converge, every vertex in that component has the smallest ID among the vertices in that component, and that ID identifies the weakly connected components.

The distributed *GraphLab* implementation is written in C++ and extends the original open-sourced shared memory *GraphLab* implementation.

Unfortunately, *Trinity* is not open source and thus makes comparisons with their system more difficult for specific applications, but they did design a high level language called TSL (Trinity Specification Language) for data and network communication modeling in *Trinity*. TSL provides object-oriented data manipulation for the underlying blob data in the memory cloud and facilitates data integration and system extension. For example, they implemented a sophisticated graph query language, TQL (Trinity Query Language), within this framework.

PowerGraph does provide an open source implementation of their abstraction which contains synchronous, asynchronous, and hybrid Asynchronous Serializable engines to suit the needs of many natural graph processing algorithms. Their paper does not provide specific details of the APIs and actual coding effort involved for processing an application.

GraphX provides an integrated graph and collections API which is sufficient to express existing graph abstractions and enable a much wider range of computation, allowing them to streamline the graph analytics pipeline which usually involves more than just the execution of the graph algorithm.

III. Partitioning

The last but important aspect to cover in distributed graph processing deals with how the graph is partitioned to different machines. The ideal situation is to have a reasonably balanced amount of work between all nodes in the computation. This is not always straightforward as you can have graphs where a small percentage of the vertices have most of the edges, and the rest of the vertices have small degrees, or certain algorithms deal with short computations and spend most of their time passing messages to and from other machines. There are generally three types of partitioning that take place: offline partitioning, dynamic repartitioning, and stream partitioning.

A. Offline Partitioning

This form of static partitioning occurs after a graph has been loaded and the effects of its partitioning relies on three factors: the graph algorithm being executed, the graph itself, and the configuration of the worker tasks across compute nodes.

METIS [19] is the state-of-the-art publicly-available software that divides a graph into a specified number of partitions, trying to minimize the number of edges crossing the partitions. By default, *METIS* balances the number of vertices in each partition. Not all graphs and graph algorithms are suitable for use of *METIS* and some of these graph processing systems have adopted their own methods for partitioning graphs.

GPS has experimented with different forms of static partitioning and has found that partitioning schemes that maintain workload balance among workers perform better than schemes that do not. One of their optimization techniques called LALP (large adjacency list partitioning) partitions high-degree vertices across workers. This optimization sees many benefits only if the algorithm satisfies two properties: vertices use their outgoing neighbors only to send messages and not for computation, and if a vertex sends a message, it sends the same message to all of its outgoing neighbors.

Trinity uses a multi-level partitioning algorithm [20] that is comparable to that of *METIS*.

In the case of natural graphs, most systems are forced to resort to hash-based (random) partitioning which has extremely poor locality. *PowerGraph* addresses this problem by evenly assigning edges to machines and allowing vertices to span multiple machines. Since each edge is stored exactly once, changes to edge data do not need to be communicated. However, changes to vertices must be copied to all the machines it spans, thus making storage and network overhead dependent on the number of machines spanned by each vertex. *GraphX* adopts this same mentality of vertex-cut partitioning that evenly assigns edges to machines and minimizes the number of times each vertex is cut.

A.1 Dynamic Repartitioning

As mentioned previously (Section II - E.2), *GPS* has also experimented with dynamic repartitioning based on how the vertices communicate with one another. Their results on the Pagerank algorithm indicate that the performance increase for running times is modest, but bigger gains come from the decreased network I/O that kicks in during later iterations of the algorithm.

B. Stream Partitioning

Depending on the graph and algorithm, existing graph partitioning heuristics can incur high computation and communication costs, sometimes as large as the actual computation itself. A new technique explores a lightweight streaming algorithm [21] that attempts to efficiently and effectively partition the graph as it is loaded into the cluster. The goal is to find a close optimal balanced partitioning that relies only on the graph structure and works regardless of the meta data with as little computation as possible.

After experimenting with over ten different heuristic methods, the one that works the best on the majority of the graphs is a method called Linear Deterministic Greedy. The idea behind this technique is that a vertex v is assigned to the partition where it has the most edges, and weight this by a penalty function based on the capacity of the partition, penalizing larger partitions:

$$ind = \arg\max_{i \in [k]} \{|P^t(i) \cap \Gamma(v)| w(t,i)\} \quad (2)$$

where w(t,i) is a weighted penalty function:

$$w(t,i) = 1 \text{ unweighted greedy}$$
$$w(t,i) = 1 - \frac{|P^t(i)|}{C} \text{ linear weighted}$$
$$w(t,i) = 1 - exp|P^t(i)| - C \text{ exp. weighted}$$

The experiment to test this heuristic involved two data sets: LiveJournal [22] with 4.6 million vertices and 77.4 million edges, and a Twitter graph with 41.7 million vertices and 1.468 billion edges. The LiveJournal experiment used 100 partitions on 10 Amazon EC2 nodes, while the Twitter data set was split into 400 pieces on 50 machines with 100 cores. Spark has two implementations of Pagerank: a Naïve version that sends a message on the network for each edge, and a more sophisticated combiner version that aggregates all messages between each partition.

TABLE V
STREAM GRAPH PARTITIONING VS HASH BASED.

Algorithm	Graph	Hash	Stream
Naïve PR	LJ	196.2s	181.5s
Combiner PR	LJ	155.1s	110.4s
Naïve PR	Twitter	1199.4s	969.3s
Combiner PR	Twitter	599.4s	486.8s

Linear Deterministic Greedy (LDG) was able to reduce the number of edges cut to 47,361,254 on the LiveJournal data set compared to 76,234,872 for hashing. This resulted in 7.5% and 28.8% speedups in the Naïve and Combiner Pagerank algorithms, respectively (Table V). LDG cut 1.341 billion edges for the Twitter graph while hashing cut 1.464 billion. This resulted in 19.2% and 18.8% speedups for the two algorithms, respectively.

This method of graph stream partitioning is not intended to act as a substitute for a full information graph partitioning, but rather to complement one. The results here show that a simple preprocessing step that considers graph edges can yield a large improvement on the running time with little overhead.

IV. Conclusions

We now see that there is a growing need for more complex and highly specialized systems to handle the increasing size and variability of graph data. There are many ways for a user to start processing their own graphs, and open-source implementations like *GraphChi* show that it is possible for anyone with a modern machine to perform large scale graph analytics in a larger, but reasonable amount of time.

Since graph structures can take on a variety of characteristics, and the graph algorithms to run on these graphs work well on some structures over others, it is hard to come up with one graph processing abstraction that will perform well for every size and structure graph with all algorithms. *Pregel* offers a great starting point with an easy to use API that makes vertex-centric computing straightforward and intuitive, while *GPS* extends this capability by implementing dynamic repartitioning of vertices during execution that can dramatically cut down on network I/O for long running algorithms.

Distributed *GraphLab* and *Trinity* both offer unique solutions for storing graphs with *GraphLab*'s shared-memory parallelism and *Trinity*'s circular memory with memory trunks. This enables both systems the capability to host a wide range of graph algorithms that can be run efficiently on either abstraction. Even though Trinity is not open-source, there is a lot of potential in its specification language to extend an already powerful system with even more capabilities.

PowerGraph recognizes that many real world graphs of interest have vertices that follow a power-law degree, and they take a slightly different approach from the other graph processing systems by partitioning out their edges, making the total storage and network overhead dependent upon the number machines that each vertex is spread across.

GraphX is one of the newest graph processing systems and really tries to take the best parts from distributed data flow frameworks and other graph processing abstractions and put them into one system that streamlines the entire analytics pipeline of data processing. Built on top of Spark, *GraphX* transparently takes advantage of the distributed data flow fault tolerance mechanisms while adopting *PowerGraph*'s idea of vertex-cut partitioning and maintaining a focus on vertex programming using the GAS decomposition as its computational model.

Finally, we have seen that *METIS* is the state-of-the-art offline partitioner, but a new technique called graph stream partitioning is a lightweight partitioner that partitions the data based solely on the graph structure while it is being loaded into the cluster. This can have significant performance improvements for many graph algorithms while incurring little computational cost.

All of the graph processing systems presented in this survey are exceptional tools that demonstrate beneficial performance on a variety of applications, but every problem is unique, and finding the right abstraction to use is essential in completing the desired task in an elegant and efficient manner.

REFERENCES

[1] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, 2012, pp. 31–46, USENIX.

[2] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999, Previous number = SIDL-WP-1999-0120.

[3] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," in *Algorithms and Models for the Web-Graph*, pp. 15–24. Springer, 2010.

[4] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management*, pp. 337–348. Springer, 2008.

[5] U Kang, Duen Horng, et al., "Inference of beliefs on billion-scale graphs," 2010.

[6] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010, SIGMOD '10, pp. 135–146, ACM.

[8] Semih Salihoglu and Jennifer Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, New York, NY, USA, 2013, SSDBM, pp. 22:1–22:12, ACM.

[9] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[10] Bin Shao, Haixun Wang, and Yatao Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2013, SIGMOD '13, pp. 505–516, ACM.

[11] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, Oct. 2014, pp. 599–613, USENIX Association.

[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, 2012, pp. 17–30, USENIX.

[13] Leslie G Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[14] K Mani Chandy and Leslie Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2012, NSDI'12, pp. 2–2, USENIX Association.

[16] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos, "R-mat: A recursive model for graph mining.," in *SDM*. SIAM, 2004, vol. 4, pp. 442–446.

[17] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 810–818.

[18] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[19] George Karypis and Vipin Kumar, "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[20] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang, "How

to partition a billion-node graph," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 568–579.

[21] Isabelle Stanton and Gabriel Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2012, KDD '12, pp. 1222–1230, ACM.

[22] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 29–42.