One Trillion Edges: Graph Processing at Facebook-Scale

Avery Ching
Facebook
1 Hacker Lane
Menlo Park, California
aching@fb.com

Sergey Edunov
Facebook
1 Hacker Lane
Menlo Park, California
edunov@fb.com

Maja Kabiljo
Facebook
1 Hacker Lane
Menlo Park, California
majakabiljo@fb.com

Dionysios Logothetis
Facebook
1 Hacker Lane
Menlo Park, California
dionysios@fb.com

Sambavi Muthukrishnan
Facebook
1 Hacker Lane
Menlo Park, California
sambavim@fb.com

ABSTRACT

Analyzing large graphs provides valuable insights for social networking and web companies in content ranking and recommendations. While numerous graph processing systems have been developed and evaluated on available benchmark graphs of up to 6.6B edges, they often face significant difficulties in scaling to much larger graphs. Industry graphs can be two orders of magnitude larger - hundreds of billions or up to one trillion edges. In addition to scalability challenges, real world applications often require much more complex graph processing workflows than previously evaluated. In this paper, we describe the usability, performance, and scalability improvements we made to Apache Giraph, an open-source graph processing system, in order to use it on Facebook-scale graphs of up to one trillion edges. We also describe several key extensions to the original Pregel model that make it possible to develop a broader range of production graph applications and workflows as well as improve code reuse. Finally, we report on real-world operations as well as performance characteristics of several large-scale production applications.

1. INTRODUCTION

Graph structures are ubiquitous: they provide a basic model of entities with connections between them that can represent almost anything. Facebook manages a social graph [41] that is composed of people, their friendships, subscriptions, likes, posts, and many other connections. Open graph [7] allows application developers to connect objects in their applications with real-world actions (such as user X is listening to song Y). Analyzing these real world graphs at the scale of hundreds of billions or even a trillion (10^{12}) edges with available software was very difficult when we began a project to run Facebook-scale graph applications in the summer of 2012 and is still the case today.

Table 1: Popular benchmark graphs.

Graph	Vertices	Edges
LiveJournal [9]	4.8M	69M
Twitter 2010 [31]	42M	1.5B
UK web graph 2007 [10]	109M	3.7B
Yahoo web [8]	1.4B	6.6B

Many specialized graph processing frameworks (e.g. [20, 21, 32, 44]) have been developed to run on web and social graphs such as those shown in Table 1. Unfortunately, real world social network are orders of magnitude larger. Twitter has 288M monthly active users as of 3/2015 and an estimated average of 208 followers per user [4] for an estimated total of 60B followers (edges). Facebook has 1.39B active users as of 12/2014 with more than 400B edges. Many of the performance and scalability bottlenecks are very different when considering real-world industry workloads. Several studies [14, 24] have documented that many graph frameworks fail at much smaller scale mostly due to inefficient memory usage. Asynchronous graph processing engines tend to have additional challenges in scaling to larger graphs due to unbounded message queues causing memory overload, vertex-centric locking complexity and overhead, and difficulty in leveraging high network bandwidth due to fine-grained computation.

Correspondingly, there is lack of information on how applications perform and scale to practical problems on trillion-edge graphs. In practice, many web companies have chosen to build graph applications on their existing MapReduce infrastructure rather than a graph processing framework for a variety of reasons. First, many such companies already run MapReduce applications [17] on existing Hadoop [2] infrastructure and do not want to maintain a different service that can only process graphs. Second, many of these frameworks are either closed source (i.e. Pregel) or written in a language other than Java (e.g. GraphLab is written in C++). Since much of today's datasets are stored in Hadoop, having easy access to HDFS and/or higher-level abstractions such as Hive tables is essential to interoperating with existing Hadoop infrastructure. With so many variants and versions

Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

of Hive/HDFS in use, providing native C++ or other language support is both unappealing and time consuming.

Apache Giraph [1] fills this gap as it is written in Java and has vertex and edge input formats that can access MapReduce input formats as well as Hive tables [40]. Users can insert Giraph applications into existing Hadoop pipelines and leverage operational expertise from Hadoop. While Giraph initially did not scale to our needs at Facebook with over 1.39B users and hundreds of billions of social connections, we improved the platform in a variety of ways to support our workloads and implement our production applications. We describe our experiences scaling and extending existing graph processing models to enable a broad set of both graph mining and iterative applications. Our contributions are the following:

- Present usability, performance, and scalability improvements for Apache Giraph that enable trillion edge graph computations.
- Describe extensions to the Pregel model and why we found them useful for our graph applications.
- Real world applications and their performance on the Facebook graph.
- Share operational experiences running large-scale production applications on existing MapReduce infrastructure.
- Contribution of production code (including all extensions described in this paper) into the open-source Apache Giraph project.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 provides a summary of Giraph, details our reasons for selecting it as our initial graph processing platform, and explains our usability and scalability improvements. Section 4 describes our generalization to the original Pregel graph processing model for creating more powerful application building blocks and reusable code. Section 5 details Giraph applications and their performance for a variety of workloads. In Section 6, we share our graph processing operational experiences. In Section 7, we conclude our work and describe potential future work.

2. RELATED WORK

Large-scale graph computing based on the *Bulk Synchronous Processing* (BSP) model [42] was first introduced by Malewicz et al. in the Pregel system [33]. Unfortunately, the Pregel source code was not made public. Apache Giraph was designed to bring large-scale graph processing to the open source community, based loosely on the Pregel model, while providing the ability to run on existing Hadoop infrastructure. Many other graph processing frameworks (e.g. [37, 15]) are also based on the BSP model.

MapReduce has been used to execute large-scale graph parallel algorithms and is also based on the BSP model. Unfortunately, graph algorithms tend to be iterative in nature and typically do not perform well in the MapReduce compute model. Even with these limitations, several graph and iterative computing libraries have been built on MapReduce due to its ability to run reliably in production environments [3, 30]. Iterative frameworks on MapReduce style computing models is an area that has been explored in Twister [18] and Haloop [13].

Asynchronous models of graph computing have been proposed in such systems as Signal-Collect [39], GraphLab [20] and GRACE [44]. While asynchronous graph computing has been demonstrated to converge faster for some applications, it adds considerable complexity to the system and the developer. Most notably, without program repeatability it is difficult to ascertain whether bugs lie in the system infrastructure or the application code. Furthermore, asynchronous messaging queues for certain vertices may unpredictably cause machines to run out of memory. DAG-based execution systems that generalize the MapReduce model to broader computation models, such as Hyracks [11], Spark [45], and Dryad [28], can also do graph and iterative computation. Spark additionally has a higher-level graph computing library built on top of it, called GraphX [21], that allows the user to process graphs in an interactive, distributed manner.

Single machine graph computing implementations such as Cassovary [23] are used at Twitter. Another single machine implementation, GraphChi [32], can efficiently process large graphs out-of-core. Latency-tolerant graph processing techniques for commodity processors, as opposed to hardware multithreading systems (e.g. Cray XMT), were explored in [34]. In Trinity [38], graph processing and databases are combined into a single system. Parallel BGL [22] parallelizes graph computations on top of MPI. Piccolo [36] executes distributed graph computations on top of partitioned tables. Presto [43], a distributed R framework, implements matrix operations efficiently and can be used for graph analysis. Several DSL graph languages have been built, including Green-Marl [25], which can compile to Giraph code, and SPARQL [26], a graph traversal language.

3. APACHE GIRAPH

Apache Giraph is an iterative graph processing system designed to scale to hundreds or thousands of machines and process trillions of edges. For example, it is currently used at Facebook to analyze the social graph formed by users and their connections. Giraph was inspired by Pregel, the graph processing architecture developed at Google. Pregel provides a simple graph processing API for scalable batch execution of iterative graph algorithms. Giraph has greatly extended the basic Pregel model with new functionality such as master computation, sharded aggregators, edge-oriented input, out-of-core computation, composable computation, and more. Giraph has a steady development cycle and a growing community of users worldwide.

3.1 Early Facebook experimentation

In the summer of 2012, we began exploring a diverse set of graph algorithms across many different Facebook products as well as related academic literature. We selected a few representative use cases that cut across the problem space with different system bottlenecks and programming complexity. Our diverse use cases and the desired features of the programming framework drove the requirements for our system infrastructure. We required an iterative computing model, graph-based API, and easy access to Facebook data. We knew that this infrastructure would need to work at the scale of hundreds of billions of edges. Finally, as we would

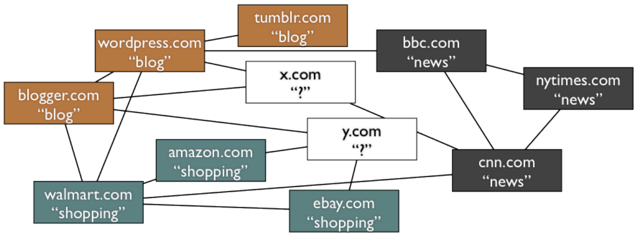

Figure 1: An example of label propagation: Inferring unknown website classifications from known website classifications in a graph where links are generated from overlapping website keywords.

be rapidly be developing this software, we had to be able to easily identify application and infrastructure bugs and have repeatable, reliable performance. Based on these requirements, we selected a few promising graph-processing platforms including Hive, GraphLab, and Giraph for evaluation. We used label propagation, among other graph algorithms, to compare the selected platforms. Label propagation is an iterative graph algorithm that infers unlabeled data from labeled data. The basic idea is that during each iteration of the algorithm, every vertex propagates its probabilistic labels (for example, website classifications - see Figure 1) to its neighboring vertices, collects the labels from its neighbors, and calculates new probabilities for its labels. We implemented this algorithm on all platforms and compared the performance of Hive 0.9, GraphLab 2, and Giraph on a small scale of 25 million edges.

We ended up choosing Giraph for several compelling reasons. Giraph directly interfaces with our internal version of HDFS (since Giraph is written in Java) as well as Hive. Since Giraph is scheduled as a MapReduce job, we can leverage our existing MapReduce (Corona) infrastructure stack with little operational overhead. With respect to performance, at the time of testing in late summer 2012, Giraph was faster than the other frameworks. Perhaps most importantly, the BSP model of Giraph was easy to debug as it provides repeatable results and is the most straightforward to scale since we did not have to handle all the problems of asynchronous graph processing frameworks. BSP also made it easy to implement composable computation (see Section 4.3) and simple to do checkpointing.

3.2 Platform improvements

Even though we had chosen a platform, there was a lot of work ahead of us. Here are some of the limitations that we needed to address:

- Giraph's graph input model was only vertex centric, requiring us to either extend the model or do vertex centric graph preparation external to Giraph.

- Parallelizing Giraph infrastructure relied completely on MapReduce's task level parallelism and did not have multithreading support for fine grain parallelism.

- Giraph's flexible types and computing model were initially implemented using native Java objects and consumed excessive memory and garbage collection time.

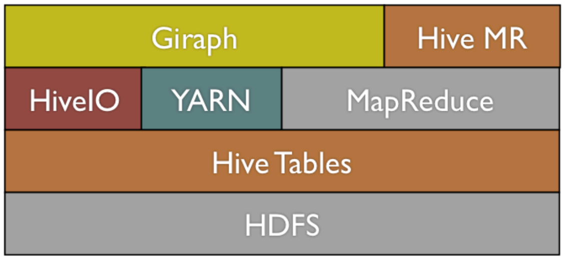

Figure 2: Giraph leverages HiveIO to directly access Hive tables and can run on MapReduce and YARN.

- The aggregator framework was inefficiently implemented in ZooKeeper and we needed to support very large aggregators (e.g. gigabytes).

We selected three production applications, label propagation, variants of PageRank [12], and k-means clustering, to drive the direction of our development. Running these applications on graphs as large as the full Facebook friendship graph, with over 1 billion users and hundreds of billions of friendships, required us to address these shortcomings in Giraph. In the following sections, first, we describe our efforts with flexible vertex and edge based input to load and store graphs into Facebook's data warehouse. Second, we detail our performance and scalability enhancements with parallelization approaches, memory optimization, and sharded aggregators. In addition, whenever relevant, we add a citation in the form of GIRAPH-XXX as the Giraph JIRA number associated with the work described that can be referenced at [5].

3.2.1 Flexible vertex/edge based input

Since Giraph is a computing platform, it needs to interface with external storage to read the input and write back the output of a batch computation. Similarly to MapReduce, we can define custom input and output formats for various data sources (e.g., HDFS files, HBase tables, Hive tables). The Facebook software stack is shown in Figure 2.

Datasets fed to a Giraph job consist of vertices and edges, typically with some attached metadata. For instance, a label propagation algorithm might read vertices with initial labels and edges with attached weights. The output will consist of vertices with their final labels, possibly with confidence values.

The original input model in Giraph required a rather rigid layout: all data relative to a vertex, including outgoing edges, had to be read from the same record and were assumed to exist in the same data source, for instance, the same Hive table. We found these restrictions suboptimal for most of our use cases. First, graphs in our data warehouse are usually stored in a relational format (one edge per row), and grouping them by source vertex requires an extra MapReduce job for pre-processing. Second, vertex data, such as initial labels in the example above, may be stored separately from edge data. This required an extra pre-processing step to join vertices with its edges, adding unnecessary overhead. Finally, in some cases, we needed to combine vertex data from different tables into a single one before running a Giraph job.

To address these shortcomings, we modified Giraph to allow loading vertex data and edges from separate sources (GIRAPH-155). Each worker can read an arbitrary subset of the edges, which are then appropriately distributed so that each vertex has all its outgoing edges. This new model also encourages reusing datasets: different algorithms may require loading different vertex metadata while operating on the same graph (e.g., the graph of Facebook friendships).

One further generalization we implemented is the ability to add an arbitrary number of data sources (GIRAPH-639), so that, for example, multiple Hive tables with different schemas can be combined into the input for a single Giraph job. One could also consider loading from different sources from the same application (e.g. loading vertex data from MySQL machines and edge data from Hadoop sequence files). All of these modifications give us the flexibility to run graph algorithms on our existing and diverse datasets as well as reduce the required pre-processing to a minimum or eliminate it in many cases. These same techniques can be used in other graph processing frameworks as well.

3.2.2 Parallelization support

Since a Giraph application is scheduled as a single MapReduce job, it initially inherited the MapReduce way of parallelizing a job, that is, by increasing the number of workers (mappers) for the job. Unfortunately, it is hard to share resources with other Hadoop tasks running on the same machine due to differing requirements and resource expectations. When Giraph runs one monopolizing worker per machine in a homogenous cluster, it can mitigate issues of different resource availabilities for different workers (i.e. the slowest worker problem).

To address these issues, we extended Giraph to provide two methods of parallelizing computations:

- Adding more workers per machine.
- Use worker local multithreading to take advantage of additional CPU cores.

Specifically, we added multithreading to loading the graph, computation (GIRAPH-374), and storing the computed results (GIRAPH-615). In CPU bound applications, such as k-means clustering, we have seen a near linear speedup due to multithreading the application code. In production, we parallelize our applications by taking over a set of entire machines with one worker per machine and use multithreading to maximize resource utilization.

Multithreading introduces some additional complexity. Taking advantage of multithreading in Giraph requires the user to partition the graph into n partitions across m machines where the maximum compute parallelism is n/m. Additionally, we had to add a new concept called *WorkerContext* that allows developers to access shared member variables. Many other graph processing frameworks do not allow multithreading support within the infrastructure as Giraph does. We found that technique this has significant advantages in reducing overhead (e.g. TCP connections, larger message batching) as opposed to more coarse grain parallelism by adding workers.

3.2.3 Memory optimization

In scaling to billions of edges per machine, memory optimization is a key concept. Few other graph processing systems support Giraph's flexible model of allowing arbitrary vertex id, vertex value, vertex edge, and message classes as well as graph mutation capabilities. Unfortunately, this flexibility can have a large memory overhead without careful implementation. In the 0.1 incubator release, Giraph was memory inefficient due to all data types being stored as separate Java objects. The JVM worked too hard, out of memory errors were a serious issue, garbage collection took a large portion of our compute time, and we could not load or process large graphs.

We addressed this issue via two improvements. First, by default we serialize the edges of every vertex into a byte array rather than instantiating them as native Java objects (GIRAPH-417) using native direct (and non-portable) serialization methods. Messages on the server are serialized as well (GIRAPH-435). Second, we created an *OutEdges* interface that would allow developers to leverage Java primitives based on FastUtil for specialized edge stores (GIRAPH-528).

Given these optimizations and knowing there are typically many more edges than vertices in our graphs (2 orders of magnitude or more in most cases), we can now roughly estimate the required memory usage for loading the graph based entirely on the edges. We simply count the number of bytes per edge, multiply by the total number of edges in the graph, and then multiply by 1.5x to take into account memory fragmentation and inexact byte array sizes. Prior to these changes, the object memory overhead could have been as high as 10x. Reducing memory use was a big factor in enabling the system to load and send messages to 1 trillion edges. Finally, we also improved the message combiner (GIRAPH-414) to further reduce memory usage and improve performance by around 30% in PageRank testing. Our improvements in memory allocation show that with the correct interfaces, we can support the full flexibility of Java classes for vertex ids, values, edges, and messages without inefficient per object allocations.

3.2.4 Sharded aggregators

Aggregators, as described in [33], provide efficient shared state across workers. While computing, vertices can aggregate (the operation must be commutative and associative) values into named aggregators to do global computation (i.e. min/max vertex value, error rate, etc.). The Giraph infrastructure aggregates these values across workers and makes the aggregated values available to vertices in the next superstep.

One way to implement k-means clustering is to use aggregators to calculate and distribute the coordinates of centroids. Some of our customers wanted hundreds of thousands or millions of centroids (and correspondingly aggregators), which would fail in early versions of Giraph. Originally, aggregators were implemented using Zookeeper [27]. Workers would write partial aggregated values to znodes (Zookeeper data storage). The master would aggregate all of them, and write the final result back to its znode for workers to access it. This technique was sufficient for applications that only used a few simple aggregators, but wasn't scalable due to znode size constraints (maximum 1 megabyte) and Zookeeper write limitations. We needed a solution that could efficiently handle tens of gigabytes of aggregator data coming from every worker.

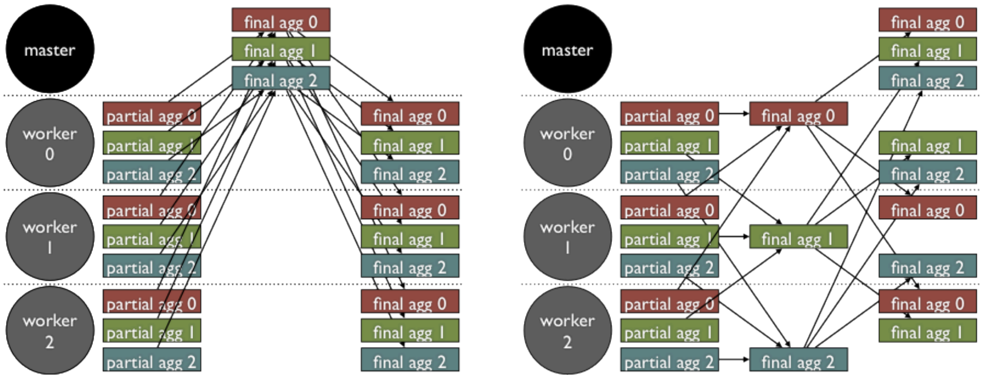

Figure 3: After sharding aggregators, aggregated communication is distributed across workers.

```
void Iterable<M> messages) {
  if (phase == K_MEANS)
    // Do k-means
  else if (phase == START_EDGE_CUT)
    // Do first phase of edge cut
  else if (phase == END_EDGE_CUT)
    // Do second phase of edge cut
}
```

Figure 5: Example vertex computation code to change phases.

To solve this issue, first, we bypassed Zookeeper and used Netty [6] to directly communicate aggregator values between the master and its workers. While this change allowed much larger aggregator data transfers, we still had a bottleneck. The amount of data the master was receiving, processing, and sending was growing linearly with the number of workers. In order to remove this bottleneck, we implemented sharded aggregators.

In the sharded aggregator architecture (Figure 3), each aggregator is now randomly assigned to one of the workers. The assigned worker is in charge of gathering the values of its aggregators from all workers, performing the aggregation, and distributing the final values to the master and other workers. Now, aggregation responsibilities are balanced across all workers rather than bottlenecked by the master and aggregators are limited only by the total memory available on each worker.

4. COMPUTE MODEL EXTENSIONS

Usability and scalability improvements have allowed us to execute simple graph and iterative algorithms at Facebook-scale, but we soon realized that the Pregel model needed to be generalized to support more complex applications and make the framework more reusable. An easy way to depict the need for this generalization is through a very simple example: k-means clustering. K-means clustering, as shown in Figure 4, is a simple clustering heuristic for assigning input vectors to one of k centroids in an n-dimensional space. While k-means is not a graph application, it is iterative in nature and easily maps into the Giraph model where input vectors are vertices and every centroid is an aggregator. The vertex (input vector) compute method calculates the distance to all the centroids and adds itself to the nearest one. As the centroids gain input vectors, they incrementally determine their new location. At the next superstep, the new location of every centroid is available to every vertex.

4.1 Worker phases

The methods *preSuperstep()*, *postSuperstep()*, *preApplication()*, and *postApplication()* were added to the Computation class and have access to the worker state. One use case for the pre-superstep computation is to calculate the new position for each of the centroids. Without pre-superstep computation, a developer has to either incrementally calculate the position with every added input vector or calculate it for every distance calculation. In the preSuperstep() method that is executed on every worker prior to every superstep, every worker can compute the final centroid locations just before the input vectors are processed.

Determining the initial positions of the centroids can be done in the preApplication() method since it is executed on every worker prior to any computation being executed. While these simple methods add a lot of functionality, they bypass the Pregel model and require special consideration for application specific techniques such superstep checkpointing.

4.2 Master computation

Fundamentally, while the Pregel model defines a functional computation by "thinking like a vertex", some computations need to be executed in a centralized fashion for many of the reasons above. While executing the same code on each worker provides a lot of the same functionality, it is not well understood by developers and is error prone. GIRAPH-127 added master computation to do centralized computation prior to every superstep that can communicate with the workers via aggregators. We describe two example use cases below.

When using k-means to cluster the Facebook dataset of over 1 billion users, it is useful to aggregate the error to see whether the application is converging. It is straightforward to aggregate the distance of every input vector to its chosen centroid as another aggregator, but at Facebook we also have the social graph information. Periodically, we can also compute another metric of distance: the edge cut. We can use the friendships, subscriptions, and other social connections of our users to measure the edge cut, or the weighted

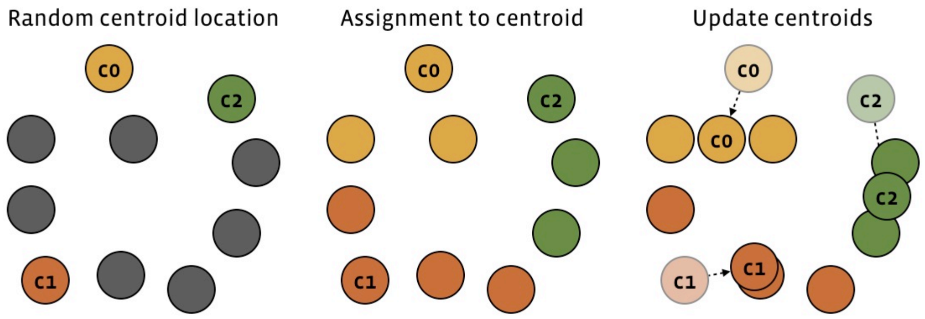

Figure 4: In k-means clustering, k centroids have some initial location (often random). Then input vectors are assigned to their nearest centroid. Centroid locations are updated and the process repeats.

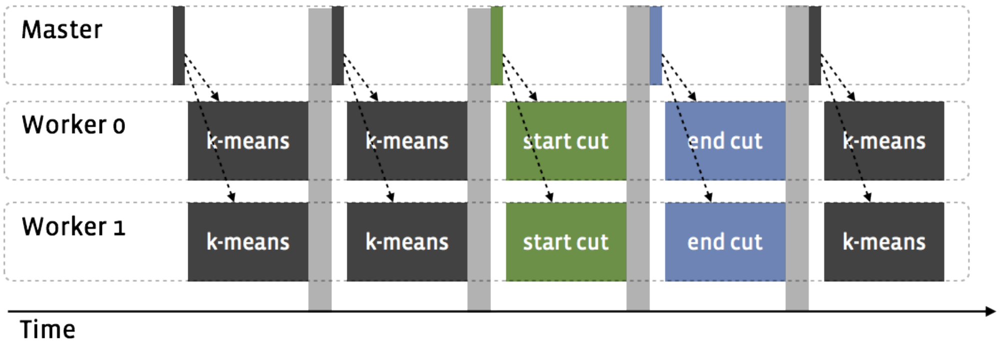

Figure 6: Master computation allows k-means clustering to periodically calculate edge cut computations.

edge cut if the edges have weights. Deciding when to calculate the edge cut can be the job of the master computation, for instance, when at least 15 minutes of k-means computation have been executed. With master computation, this functionality can be implemented by checking to see how long it has been since the last edge cut calculation. If the time limit has been exceeded, set a special aggregator (i.e. execute edge cut) to true. The execution workflow is shown in Figure 6. The vertex compute code only needs to check the aggregator value to decide whether to begin calculating an edge cut or continue iterating on k-means.

Executing a coordinated operation such as this one on the workers without master computation is more complicated due to clock skew, although possible, for instance with multiple supersteps to decide on whether to do the edge cut calculation.

Another example of the usefulness of master computation can be found in the example PageRank code in the Pregel paper [33]. In the example, every vertex must check whether the desired number of iterations has been completed to decide to vote to halt.

This is a simple computation that needs to be executed exactly once rather than on every vertex. In the master computation there is a haltComputation() method, where it is simple to check once prior to starting a superstep whether the application should continue rather than executing the check on a per vertex basis.

4.3 Composable computation

In our production environment, we observed that graph processing applications can be complex, often consisting of "stages", each implementing distinct logic. Master computation allows developers to compose their applications with different stages, but is still limiting since the original Pregel model only allows one message type and one message combiner. Also, the vertex compute code gets messy as shown in Figure 5. In order to support applications that do distinctive computations (such as k-means) in a cleaner and more reusable way, we added composable computation. Composable computing simply decouples the vertex from the computation as shown in Figure 7. The Computation class abstracts the computation from the vertex so that different types of computations can be executed. Additionally, there are now two message types specified. M1 is the incoming message type and M2 is the outgoing message type. The master compute method can choose the computation class to execute for the current superstep with the method setComputation(Class<? extends Computation> computationClass). The master also has a corresponding method to change the message combiner as well. The infrastructure checks to insure that all types match for computations that are chained together. Now that vertex computations are decoupled to different Computation implementations, they can be used as building blocks for multiple applications. For example, the edge cut computations can be used in a variety of clustering algorithms, not only k-means.

In our example k-means application with a periodic edge cut, we might want to use a more sophisticated centroid initialization method such as initializing the centroid with a random user and then a random set of their friends. The computations to initialize the centroids will have different message types than the edge cut computations. In Table 2,

Table 2: Example usage of composable computations in a k-means application with different computations (initialization, edge cut, and k-means)

Computation	Add random centroid / random friends	Add to centroids	K-means	Start edge cut	End edge cut
In message	Null	Centroid message	Null	Null	Cluster
Out message	Centroid message	Null	Null	Cluster	Null
Combiner	N/A	N/A	N/A	Cluster combiner	N/A

```
public abstract class Vertex<
    I, V, E, M> {
  public abstract void compute(
    Iterable<M> messages);
}

public interface Computation<
    I, V, E, M1, M2> {
  void compute(Vertex<I, V, E> vertex,
    Iterable<M1> messages);
}

public abstract class MasterCompute {
  public abstract void compute();

  public void setComputation(
    Class<? extends Computation> computation);

  public void setMessageCombiner(
    Class<? extends MessageCombiner> combiner);

  public void setIncomingMessage(
    Class<? extends Writable> incomingMessage);

  public void setOutgoingMessage(
    Class<? extends Writable> outgoingMessage);
}
```

Figure 7: Vertex was generalized into Computation to support different computations, messages, and message combiners as set by MasterCompute.

we show how composable computation allows us to use different message types, combiners, and computations to build a powerful k-means application. We use the first computation for adding random input vectors to centroids and notifying random friends to add themselves to the centroids. The second computation adds the random friends to centroids by figuring out its desired centroid from the originally added input vector. It doesn't send any messages to the next computation (k-means), so the out message type is null. Note that starting the edge cut is the only computation that actually uses a message combiner, but one could use any message combiner in different computations.

Composable computation makes the master computation logic simple. Other example applications that can benefit from composable computation besides k-means include balanced label propagation [41] and affinity propagation [19]. Balanced label propagation uses two computations: compute candidate partitions for each vertex and moving vertices to partitions. Affinity propagation has three computations: calculate responsibility, calculate availability, and update exemplars.

4.4 Superstep splitting

Some applications have messaging patterns that can exceed the available memory on the destination vertex owner. For messages that are *aggregatable*, that is, commutative and associative, message combining solves this problem. However, many applications send messages that cannot be aggregated. Calculating mutual friends, for instance, requires each vertex to send all its neighbors the vertex ids of its neighborhood. This message cannot be aggregated by the receiver across all its messages. Another example is the multiple phases of affinity propagation - each message must be responded to individually and is unable to be aggregated. Graph processing frameworks that are asynchronous are especially prone to such issues since they may receive messages at a faster rate than synchronous frameworks.

In social networks, one example of this issue can occur when sending messages to connections of connections (i.e. friends of friends in the Facebook network). While Facebook users are limited to 5000 friends, theoretically one user's vertex could receive up to 25 million messages. One of our production applications, friends-of-friends score, calculates the strength of a relationship between two users based on their mutual friends with some weights. The messages sent from a user to his/her friends contain a set of their friends and some weights. Our production application actually sends 850 GB from each worker during the calculation when we use 200 workers. We do not have machines with 850 GB and while Giraph supports out-of-core computation by spilling the graph and/or messages to disk, it is much slower.

Therefore, we have created a technique for doing the same computation all in-memory for such applications: superstep splitting. The general idea is that in such a message heavy superstep, a developer can send a fragment of the messages to their destinations and do a partial computation that updates the state of the vertex value. The limitations for the superstep splitting technique are as follows:

- The message-based update must be commutative and associative.
- No single message can overflow the memory buffer of a single vertex.

The master computation will run the same superstep for a fixed number of iterations. During each iteration, every vertex uses a hash function with the destination vertex id of each of its potentials messages to determine whether to send a message or not. A vertex only does computation if its vertex id passes the hash function for the current superstep. As an example for 50 iterations (splits of our superstep) our friends-of-friends application only uses 17 GB of memory per iteration. This technique can eliminate the need to go out-of-core and is memory scalable (simply add more iterations to proportionally reduce the memory usage).

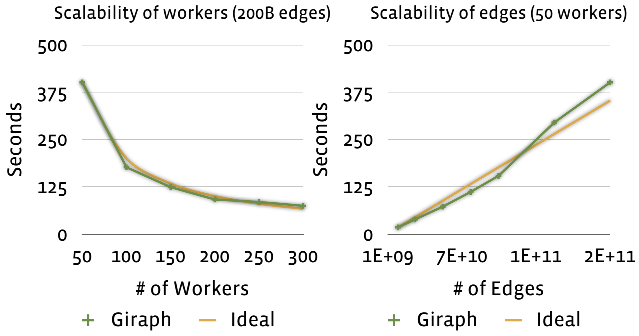

Figure 8: Scaling PageRank with respect to the number of workers (a) and the number of edges (b)

5. EXPERIMENTAL RESULTS

In this section, we describe our results of running several example applications in Giraph when compared to their respective Hive implementations. Our comparison will include iterative graph applications as well as simple Hive queries converted to Giraph jobs. We use two metrics for the performance comparison: CPU time and elapsed time. CPU time is the time measured by the CPU to complete an application while elapsed time is the time observed by the user when their application completes. All of the described applications are in production. Experiments were gathered on production machines with 16 physical cores and 10 Gbps Ethernet. The Giraph experiments were all conducted in physical memory without checkpointing, while Hive/Hadoop has many phases of disk access (map side spilling, reducer fetching, HDFS access per iterations, etc.) causing it to exhibit poorer performance. We used the default hash-based graph partitioning for all experiments.

5.1 Giraph scalability

With a simple unweighted PageRank benchmark, we evaluated the scalability of Giraph on our production clusters. Unweighted PageRank is a lightweight computational application that is typically network bound. In Figure 8a, we first fixed the edge count to 200B and scale the number of worker from 50 to 300. There is a slight bit of variance, but overall Giraph performance closely tracks the ideal scaling curve based on 50 workers as the starting data point. In Figure 8b, we fix the number of workers at 50 and scale the problem size from 1B to 200B edges. As expected, Giraph performance scales up linearly with the number of edges since PageRank is network bound.

5.2 Applications

We describe three production iterative graph applications as well as the algorithmic pseudocode that have been run in both Hive and Giraph. The Giraph implementation used 200 machines in our tests and the Hive implementations used at least 200 machines (albeit possibly with failures as MapReduce is built to handle them).

5.2.1 Label propagation

```
Computation 1: Send neighbors edge weights and
label data
1. Send my neighbors my edge weights and labels.
```

Table 3: Label propagation comparison between Hive and Giraph implementations with 2 iterations.

Graph size	Hive	Giraph	Speedup
701M+ vertices 48B+ edges	Total CPU 9.631M secs	Total CPU 1.014M secs	9x
	Elapsed Time 1,666 mins	Elapsed Time 19 mins	87x

```
Computation 2: Update labels
1. If the normalization base is not set,
   create it from the edge weights.
2. Update my labels based on the messages
3. Trim labels to top n.
4. Send my neighbors my edge weights and labels.
```

Label propagation was described briefly in Section 3.1. This algorithm (similar to [46]) was previously implemented in Hive as a series of Hive queries that had a bad impedance mismatch since the algorithm is much simpler to express as a graph algorithm.

There are two computation phases as shown above. We need to normalize edges and Giraph keeps only the out edges (not the in edges) to save memory. So computation 1 will start the first phase of figuring out how to normalize the incoming edge weights by the destination vertex. The master computation will run computation 2 until the desired iterations have been met. Note that trimming to the top n labels is a common technique to avoid unlikely labels being propagated to all vertices. Trimming both saves a large amount of memory and reduces the amount of network data sent.

In Table 3, we describe the comparison between Hive and Giraph implementations of the same label propagation algorithm. From a CPU standpoint, we save 9x CPU seconds. From an elapsed time standpoint, the Giraph implementation is 87x faster with 200 machines versus roughly the same number of machines for the Hive implementation. Our comparison was executed with only 2 iterations primarily because it took so long in Hive/Hadoop. In production, we run minor variants of label propagation with many different types of labels that can be used as ranking features in a variety of applications.

5.2.2 PageRank

```
Computation: PageRank
1. Normalize the outgoing edge weights if this is
   the first superstep.  Otherwise, update the page
   rank value by summing up the incoming messages
   and calculating a new PageRank value.
2. Send every edge my PageRank value weighted by
   its normalized edge weight.
```

PageRank [12] is a useful application for finding influential entities in the graph. While originally developed for ranking websites, we can also apply PageRank in the social context.

This algorithm above describes our implementation of weighted PageRank, where the weights represent the strength of the connection between users (i.e. based on the friends of friends score calculation). Weighted PageRank is slightly more expensive than unweighted PageRank due to sending each out

Table 4: Weighted PageRank comparison between Hive and Giraph implementations for a single iteration.)

Graph size	Hive	Giraph	Speedup
2B+ vertices 400B+ edges	Total CPU 16.5M secs	Total CPU 0.6M secs	26x
	Elapsed Time 600 mins	Elapsed Time 19 mins	120x

Table 5: Performance comparison for friends of friends score between Hive and Giraph implementations.)

Graph size	Hive	Giraph	Speedup
1B+ vertices 76B+ edges	Total CPU 255M secs	Total CPU 18M secs	14x
	Elapsed Time 7200 mins	Elapsed Time 110 mins	65x

edge a specialized message, but also uses a combiner in production. The combiner simply adds the PageRank messages together and sums them into a single message.

We ran an iteration of PageRank against a snapshot of some portion of the user graph at Facebook with over 2B vertices and more than 400B social connections (includes more connections than just friendships). In Table 4, we can see the results. At 600 minutes per iteration for Hive/Hadoop, computing 10 iterations would take over 4 days with more than 200 machines. In Giraph, 10 iterations could be done in only 50 minutes on 200 machines.

5.2.3 Friends of friends score

```
Computation: Friends of friends score
1. Send friend list with weights to a hashed
   portion of my friend list.
2. If my vertex id matches the hash function on
   the superstep number, aggregate messages to
   generate my friends of friends score with the
   source vertices of the incoming messages.
```

As mentioned in Section 4.4, the friends of friends score is a valuable feature for various Facebook products. It is one of the few features we have that can calculate a score between users that are not directly connected. It uses the aforementioned superstep splitting technique to avoid going out of core when messaging sizes far exceed available memory.

From Table 5 we can see that the Hive implementation is much less efficient than the Giraph implementation. The superstep splitting technique allows us to maintain a significant 65x elapsed time improvement on 200 machines since the computation and message passing are happening in memory. While not shown, out-of-core messaging had a 2-3x overhead in performance in our experiments while the additional synchronization superstep overhead of the superstep splitting technique was small.

5.3 Hive queries on Giraph

While Giraph has enabled us to run iterative graph algorithms much more efficiently than Hive, we have also found it a valuable tool for running certain expensive Hive queries in a faster way. Hive is much simpler to write, of course, but we have found many queries fall in the same pattern of *join* and also *double join*. These applications involve either only input and output, or just two supersteps (one iteration of message sending and processing). Partitioning of the graph during input loading can simulate a join query and we do any vertex processing prior to dumping the data to Hive.

Table 6: Double join performance in Hive and Giraph implementations for one expensive Hive query.)

Graph size	Hive	Giraph	Speedup
450B connections 2.5B+ unique ids	Total CPU 211 days	Total CPU 43 days	5x
	Elapsed Time 425 mins	Elapsed Time 50 mins	8.5x

Table 7: Custom Hive query comparison with Giraph for a variety of data sizes for calculating the number of users interacting with a specific action.)

Data size	Hive	Giraph	Speedup
360B actions 40M objects	Total CPU 22 days	Total CPU 4 days	5.5x
	Elapsed Time 64 mins	Elapsed Time 7 mins	9.1x
162B actions 74M objects	Total CPU 92 days	Total CPU 18 days	5.1x
	Elapsed Time 129 mins	Elapsed Time 19 mins	6.8 x
620B actions 110M objects	Total CPU 485 days	Total CPU 78 days	6.2x
	Elapsed Time 510 mins	Elapsed Time 45 mins	11.3x

In one example, consider a join of two big tables, where one of them can be treated as edge input (representing connections between users, pages, etc.) and the other one as vertex input (some data about these users or pages). Implementing these kind of queries in Giraph has proven to be 3-4x more CPU efficient than performing the same query in Hive.

Doing a join on both sides of the edge table (double join) as shown in the above example query is up to 5x better as shown in Table 6.

As another example, we have a large table with some action logs, with the id of the user performing the action, the id of the object it interacted with, and potentially some additional data about the action itself. One of our users was interested in calculating for each object and each action description how many different users have interacted with it. In Hive this would be expressed in the form: "select count(distinct userId) ... group by objectId, actionDescription". This type of query is very expensive, and expressing it in Giraph achieving up to a 6x CPU time improvement and 11.3x elapsed time improvement, both are big wins for the user as shown in Table 7.

Customers are happy to have a way to get significantly better performance for their expensive Hive queries. Although Giraph is not built for generalized relational queries, it outperforms Hive due to avoiding disk and using customized primitive data structures that are particular to the underlying data. In the future, we may consider building a more generalized query framework to make this transition easier for a certain classes of queries.

5.4 Trillion edge PageRank

Our social graph of over 1.39B users is continuing to grow rapidly. The number of connections between users (friendships, subscriptions, public interaction, etc.) is also seeing explosive growth. To ensure that Giraph could continue to operate at a larger scale, we ran an iteration of unweighted PageRank on our 1.39B user dataset with over 1 trillion social connections. With some recent improvements in messaging (GIRAPH-701) and request processing improvements (GIRAPH-704), we were able to execute PageRank on over a trillion social connections in less than 3 minutes per iteration with only 200 machines.

6. OPERATIONAL EXPERIENCE

In this section, we describe the operational experience we have gained after running Giraph in production at Facebook for over two years. Giraph executes applications for ads, search, entities, user modeling, insights, and many other teams at Facebook.

6.1 Scheduling

Giraph leverages our existing MapReduce scheduling framework (Corona) but also works on top of Hadoop. Hadoop scheduling assumes an incremental scheduling model, where map/reduce slots can be incrementally assigned to jobs while Giraph required that all slots are available prior to the application running. Preemption, available in some versions of Hadoop, causes failure of Giraph jobs. While we could turn on checkpointing to handle some of these failures, in practice we choose to disable checkpointing for three reasons:

1. Our HDFS implementation will occasionally fail on write operations (e.g. temporary name node unavailability, write nodes unavailable, etc.), causing our checkpointing code to fail and ironically increasing the chance of failure.

2. Most of our production applications run in less than an hour and use less than 200 machines. The chance of failure is relatively low and handled well by restarts.

3. Checkpointing has an overhead.

Giraph jobs are run in non-preemptible FIFO pools in Hadoop where the number of map slots of a job never exceeds the maximum number of map slots in the pool. This policy allows Giraph jobs to queue up in a pool, wait until they get all their resources and then execute. In order to make this process less error prone, we added a few changes to Corona to prevent user error. First, we added an optional feature for a pool in Corona to automatically fail jobs that ask for more map slots than the maximum map slots of the pool. Second, we configure Giraph clusters differently than typical MapReduce clusters. Giraph clusters are homogenous and only have one map slot. This configuration allows Giraph jobs to take over an entire machine and all of its resources (CPU, memory, network, etc.). Since Giraph still runs on the same infrastructure as Hadoop, production engineering can use the same configuration management tools and prior MapReduce experience to maintain the Giraph infrastructure.

Jobs that fail are restarted automatically by the same scheduling system that also restarts failed Hive queries. In production, both Hive and Giraph retries are set to 4 and once an Giraph application is deployed to production we rarely see it fail consistently. The one case we saw during the past year occurred when a user changed production parameters without first testing their jobs at scale.

6.2 Graph preparation

Production grade graph preparation is a subject that is not well addressed in research literature. Projects such as GraphBuilder [29] have built frameworks that help with areas such as graph formation, compression, transformation, partitioning, output formatting, and serialization. At Facebook, we take a simpler approach. Graph preparation can be handled in two different ways depending on the needs of the user. All Facebook warehouse data is stored in Hive tables but can be easily interpreted as vertices and/or edges with HiveIO. In any of the Giraph input formats, a user can add custom filtering or transformation of the Hive table to create vertices and/or edges. As mentioned in Section 3.2.1, users can essentially scan tables and pick and choose the graph data they are interested in for the application. Users can also turn to Hive queries and/or MapReduce applications to prepare any input data for graph computation. Other companies that use Apache Pig [35], a high-level language for expressing data analysis programs, can execute similar graph preparation steps as a series of simple queries.

6.3 Production application workflow

Customers often ask us the workflow for deploying a Giraph application into production. We typically go through the following cycle:

1. Write your application and unit test it. Giraph can run in a local mode with the tools to create simple graphs for testing.

2. Run your application on a test dataset. We have small datasets that mimic the full Facebook graph as a single country. Typically these tests only need to run on a few machines.

3. Run your application at scale (typically a maximum of 200 workers). We have limited resources for non-production jobs to run at scale, so we ask users to tune their configuration on step 2.

4. Deploy to production. The user application and its configuration are entered into our higher-level scheduler to ensure that jobs are scheduled periodically and retries happen on failure. The Giraph oncall is responsible for ensuring that the production jobs complete. Overall, customers are satisfied with this workflow. It is very similar to the Hive/Hadoop production workflow and is an easy transition for them.

7. CONCLUSIONS & FUTURE WORK

In this paper, we have detailed how a BSP-based, composable graph processing framework supports Facebook-scale production workloads. In particular, we have described the improvements to the Apache Giraph project that enabled us to scale to trillion edge graphs (much larger than those referenced in previous work). We described new graph processing techniques such as composable computation and superstep splitting that have allowed us to broaden the pool of potential applications. We have shared our experiences with several production applications and their performance improvements over our existing Hive/Hadoop infrastructure. We have contributed all systems code back into the Apache Giraph project so anyone can try out production quality graph processing code that can support trillion edge graphs. Finally, we have shared our operational experiences with Giraph jobs and how we schedule and prepare our graph data for computation pipelines.

While Giraph suits our current needs and provides much needed efficiency wins over our existing Hive/Hadoop infrastructure, we have identified several areas of future work that we have started to investigate. First, our internal experiments show that graph partitioning can have a significant effect on network bound applications such as PageRank. For long running applications, determining a good quality graph partitioning prior to our computation will likely be net win in performance. Second, we have started to look at making our computations more asynchronous as a possible way to improve convergence speed. While our users enjoy a predictable application and the simplicity of the BSP model, they may consider asynchronous computing if the gain is significant.

Finally, we are leveraging Giraph as a parallel machine-learning platform. We already have several ML algorithms implemented internally. Our matrix factorization based collaborative filtering implementation scales to over a hundred billion examples. The BSP computing model also appears to be a good fit for AdaBoost logistic regression from Collins et al. [16]. We are able to train logistic regression models on 1.1 billion samples with 800 million sparse features and an average of 1000 active features per sample in minutes per iteration.

8. REFERENCES

[1] Apache giraph - http://giraph.apache.org.

[2] Apache hadoop. http://hadoop.apache.org/.

[3] Apache mahout - http://mahout.apache.org.

[4] Beevolve twitter study. http://www.beevolve.com/twitter-statistics.

[5] Giraph jira. https://issues.apache.org/jira/browse/GIRAPH.

[6] Netty - http://netty.io.

[7] Open graph. https://developers.facebook.com/docs/opengraph.

[8] Yahoo! altavista web page hyperlink connectivity graph, circa 2002, 2012. http://webscope.sandbox.yahoo.com/.

[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*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[10] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[11] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[12] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[14] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1371–1382, New York, NY, USA, 2014. ACM.

[15] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 3:1–3:13, New York, NY, USA, 2012. ACM.

[16] M. Collins, R. E. Schapire, and Y. Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. hee Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *In The First International Workshop on MapReduce and its Applications*, 2010.

[19] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[21] 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, Broomfield, CO, Oct. 2014. USENIX Association.

[22] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 07/2005 2005. Accepted.

[23] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: the who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pages 505–514, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[24] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[25] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[26] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[29] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[30] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[31] 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*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[32] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[33] 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*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[34] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, pages 10–10. USENIX Association, 2011.

[35] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[36] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[37] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013.

[38] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[39] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[41] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

[42] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[43] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 197–210, New York, NY, USA, 2013. ACM.

[44] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[45] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[46] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Technical Report CMU-CALD-02-107, 2002.