

Scaling Apache Giraph to a trillion edges

By Avery Ching on Wednesday, August 14, 2013 at 10:30pm

Graph structures are ubiquitous: they provide a basic model of entities with connections between them that can represent almost anything. Flight routes connect airports, computers communicate to one another via the Internet, webpages have hypertext links to navigate to other webpages, and so on. Facebook manages a social graph that is composed of people, their friendships, subscriptions, and other connections. [Open graph](#) 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 impossible last year. We needed a programming framework to express a wide range of graph algorithms in a simple way and scale them to massive datasets. After the improvements described in this article, [Apache Giraph](#) provided the solution to our requirements.

In the summer of 2012, we began exploring a diverse set of graph algorithms across many different Facebook products as well as 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 fast access to Facebook data. Based on these requirements, we selected a few promising graph-processing platforms including Apache Hive, GraphLab, and Apache Giraph for evaluation.

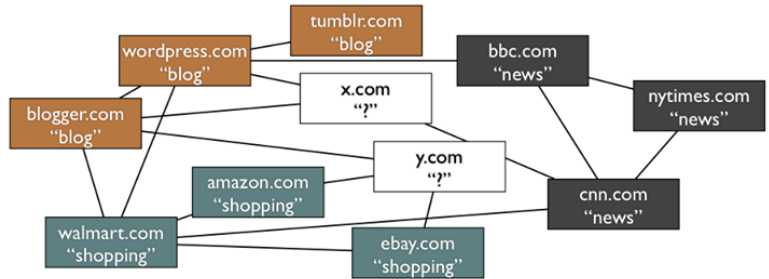
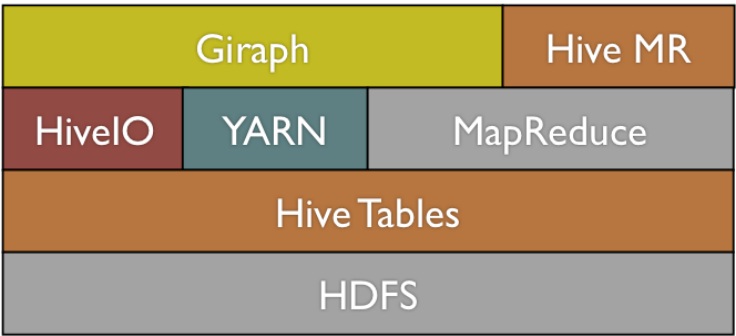


Figure 1: An example of label propagation: Inferring unknown website classifications in a graph where links are generated from overlapping website keywords.

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) and talks directly to Hive. Since Giraph runs 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 Giraph was faster than the other frameworks - much faster than Hive. Finally, Giraph's graph-based API, inspired by Google's [Pregel](#) and Leslie Valiant's bulk synchronous parallel computing model, supports a wide array of graph applications in a way that is easy to understand. Giraph also adds several useful features on top of the basic Pregel model that are beyond the scope of this article, including master computation and composable computation.



Facebook Engine

Notes by Facebook Engineering

All Notes

Get Notes via RSS
Embed Post
Report

SUGGESTED PAGES

- Franchise Cricket League**
479 people like this.
[Like](#)
- Vasavi vanitha Bangalore south**
88 people like this.
[Like](#)
- Give Way - Democracy on the road**
231 people like this.
[Like](#)
- Melbourne Florida Cricket**
3 people like this.
[Like](#)
- mHaaak - Feel Comfortable**
32 people like this.
[Like](#)
- Nasc-Chembirika**
218 people like this.
[Like](#)
- Senthikumar Photography**
130 people like him.
[Like](#)

- Sumit Singh Rawat** likes Ananth Kota's post.
- Saurabh Jain** likes Suman Mazumdar's post.
- Manikandan Venkat** likes Pooja Roy's post.
- Jamuna Krishnaraj** likes Forum for Hindu Awakening's post.
- Manjunath M Melavanki** likes Sakshar Sh's photo.
- Keshava Murthy** likes Ritu Bala's post in Flowers and Quotes
- Shriram Neelakandan**
- Sourabh Kathavate**
- Bhagwati Lal Suthar**
- Chandan Gowda**
- Waheed Aman**
- Nagendra Prasad**
- Raja Tripathy**
- Santosh Panda**
- MORE FRIENDS (3)**
- Manickavasagam S**

Unable to connect to chat.
Check your Internet connection.

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

After choosing a platform, we set to work to adapt it to our needs. We selected three production applications (label propagation, variants of page rank, and k-means clustering) to drive the direction of our development. Running these applications on graphs as large as the full Facebook friendship graph (over 1 billion users and hundreds of billions of friendships) required us to add features and major scalability improvements to Giraph. In the following sections, first, we describe our efforts with flexible vertex and edge based input and Hive IO in order to load and store graphs into Facebook's data warehouse. Second, we detail our performance and scalability enhancements with multithreading, memory optimization, and sharded aggregators.

Flexible vertex/edge based input

Since Giraph is a computing platform, it needs to interface with external storage in order to read the input and write back the output of a batch computation. Similar to MapReduce, custom input/output formats can be defined for various data sources (e.g., HDFS files, HBase tables, Hive tables).

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. We found this restriction suboptimal for most of our use cases: 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. Furthermore, vertex data (such as initial labels in the example above) has to be joined with its edges.

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. All of these modifications give us the flexibility to run graph algorithms on our existing datasets as well as reduce the required pre-processing to a minimum or eliminate it in many cases.

HiveIO

Since Facebook's data warehouse is stored in Hive, one of our early requirements was efficient reading from/writing to Hive tables. Hive does not support direct querying of its tables (only via HQL). One possible alternative, HCatalog, didn't support Facebook's internal Hadoop implementation, so we created HiveIO.

[HiveIO](#) provides a Hadoop I/O format style API that can be used to talk to Hive in a MapReduce job. Our Giraph applications use HiveIO to read graphs in both edge and vertex oriented inputs up to four times faster than Hive. It is also used to write the final vertex-oriented output back to Hive. HiveIO is being used by several other teams at Facebook for accessing Hive data and contains shell binaries and other useful tools for interacting with Hive.

Multithreading

Since a Giraph application runs as a single MapReduce job, it can be easily parallelized by increasing the number of workers (mappers) for the job. Often though, it is hard to share resources with other Hadoop tasks running on the same machine due to differing requirements and resource expectations. When Giraph takes all the task slots on a machine in a homogenous cluster, it can mitigate issues of different resource availabilities for different workers (slowest worker problem). For these reasons, 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 and using multithreading to maximize resource utilization.

Memory optimization

In the 0.1 incubator release, Giraph was a memory behemoth 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 couldn't load or process large graphs. This issue has been fixed via two improvements.

First, by default we serialize every vertex and its edges into a byte array rather than keeping them around as native Java objects (GIRAPH-417). Messages on the server are serialized as well (GIRAPH-435). We also improved on the serialization performance by using native Java serialization as the default. Second, we started using Java primitives based on FastUtil for specialized edge stores (GIRAPH-528).

Given that there are typically many more edges than vertices, we can 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 around 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 ability 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 page rank testing.

Sharded aggregators

Aggregators 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. Originally, aggregators were implemented using Zookeeper. 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 okay 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.

To solve this issue, first, we bypassed Zookeeper and used **Netty** 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.

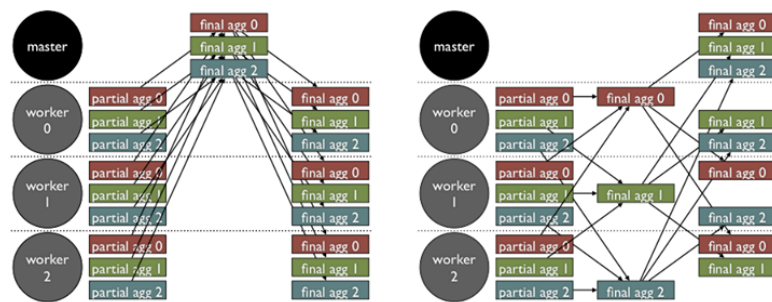


Figure 3: Before sharded aggregators, aggregator communication was centralized. After sharded aggregators, aggregated communication is distributed across workers.

In the sharded aggregator architecture, 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 other workers. Now, aggregation responsibilities are balanced across all workers rather than bottlenecked by the master.

Scalability/performance results

All of these improvements have made Giraph fast, memory efficient, scalable and easily integrated into our existing Hive data warehouse architecture. On 200 commodity machines we are able to run an iteration of page rank on an actual 1 trillion edge social graph formed by various user interactions in under four minutes with the appropriate garbage collection and performance tuning. We can cluster a Facebook monthly active user data set of 1 billion input vectors with 100 features into 10,000 centroids with k-means in less than 10 minutes per iteration. In recent relevant graph processing literature, the largest reported real-world benchmarked problem sizes to our knowledge are the Twitter graph with 1.5 billion edges (i.e.

<http://www.select.cs.cmu.edu/publications/paperdir/osdi2012-gonzalez-low-gu-bickson-guestrin.pdf>) and the Yahoo! Altavista graph with 6.6 billion edges (i.e.

<http://www.kdd.org/sites/default/files/issues/14-2-2012-12/V14-02-04-Kang.pdf>); our report of performance and scalability on a 1 trillion edge social graph is 2 orders of magnitude

beyond that scale.

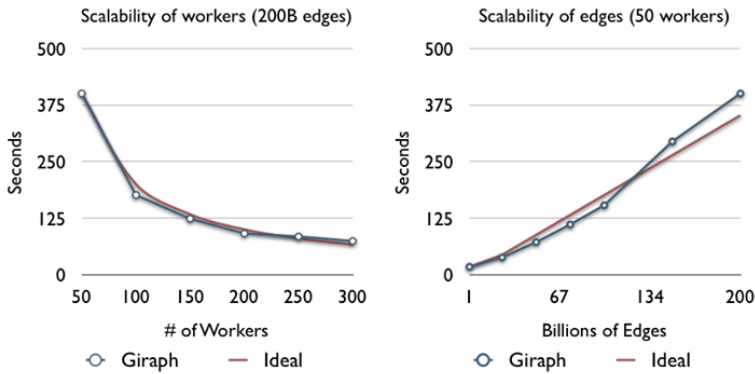


Figure 4: Giraph scales near linearly with the number of workers or the problem size.

Conclusion

With these improvements to Giraph, we have been able to launch production applications that help a variety of teams with challenges they couldn't solve with Hive/Hadoop or other custom systems in the past. We are working closely with many teams toward leveraging Giraph for challenging large-scale iterative computation in areas such as search, ads, ranking, clustering, and graph partitioning.

While we base our production code on the trunk branch of Giraph, the Giraph community has recently released a 1.0.0 version that provides all of the features described in this article and a stable API for developers to use. Documentation for the site has been greatly improved including a simple [page rank example](#) to get started. Feel free to try out either trunk or the 1.0.0 release [here](#).

Our work is by no means complete. There are many new applications to implement, more performance optimizations and computing models to investigate, and even larger scalability challenges ahead. This is the first in a series of articles about graph processing, so stay tuned and thanks for reading!

Avery Ching, Nitay Joffe, Maja Kabiljo, Greg Malewicz, Ravi Murthy, and Alessandro Presta work on Facebook's data infrastructure graph processing team.

Like

Comment

Share

646

Top Comments

176 shares

Write a comment...

Kamugisha Maurice To those that understand this has immense implications

Like · Reply · 10 · August 14, 2013 at 10:38pm

1 Reply

Babu Srinivasan very interesting

Like · Reply · 2 · August 20, 2013 at 11:00am

Chuck Woolley ...Your Server For My Area Has A Lot To Be "Desired" For The Past 2 Week's. I'm Talking "Desktop (3) Computers" At My Residence Run At A "Snail Pace." My "I-Phone" Work's Fine Like The Desktop's Use To Perform." Yes, This Is A Complaint Being Registered With You." I'm Sure I'm Not The Only User Having This Problem.....

Like · Reply · 2 · August 19, 2013 at 3:42pm · Edited

Miko Aro awesome read!

Like · Reply · 2 · August 19, 2013 at 9:18am

1 Reply

Mahesh CR A trillion edges..mind boggling..makes our product's current capability seem like a toy! Am grinning at line that ends with "...on a small scale of 25 million edges"..reckless I tell you

Like · Reply · 2 · August 15, 2013 at 3:15pm

John Wang Someone will make huge amount of money when they figure out how to run this to solve most business problems out there

Like · Reply · 1 · August 21, 2013 at 8:38pm

Anthony Yee "On 200 commodity machines we are able to run **an iteration of page rank** in under four minutes.". How many iterations and how much time does the entire ranking take?

Like · Reply · 1 · August 17, 2013 at 12:46am

Jordan Dea-Mattson Very interesting, informative, and cool!



Like · Reply · 1 · August 16, 2013 at 10:29pm

MD Fahin I am going to be engineer

Like · Reply · 1 · August 16, 2013 at 5:32pm

Jack Vaughan The Graph is the bomb!

Like · Reply · 1 · August 16, 2013 at 12:54am

-  **Vishnu M Satam** Great work..
Like · Reply · 1 · August 15, 2013 at 10:31pm
-  **Bill McColl** Great work. Nice to see BSP at tera-edge scale.
Like · Reply · 1 · August 15, 2013 at 3:03pm
-  **Manny Veloso** How does this compare to the NSA/Accumulo graph stuff?
<http://es.slideshare.net/.../big-graph-analysis-issues-nsa>
Like · Reply · 1 · August 15, 2013 at 4:58am
-  **Raxit Majithiya** First of all, it's very interesting how you gyz scaled graph traversal using Giraph.
Which label propagation algorithm do you use ? Can you provide reference about it ?
Like · Reply · 1 · August 14, 2013 at 11:10pm
-  **Sumit Dhamija** <https://www.facebook.com/mantraforengineering>
Like · Reply · 1 · August 17, 2013 at 10:19pm
- 2 Replies
-  **Deepak Dubbey** Any system problem,
Like · Reply · 1 · August 15, 2013 at 6:39am
-  **Chiranjeeb Ghosh** anything on blockchain technology?
Like · Reply · February 24, 2015 at 11:02pm
-  **Zeeshan Haider** Brilliant... very interesting and informative, thanks for sharing...
Like · Reply · February 15, 2015 at 5:48am
-  **Hiram Tobias** Like one
Like · Reply · September 3, 2014 at 4:15am
-  **Srinath Sudharsan** Reshma Giri... see this link courtesy : chandrasekar
Like · Reply · June 11, 2014 at 8:44pm
-  **Rvi Sharma** Does Apache Giraph have query language also like Cypher Query Language in Neo4J?
Like · Reply · December 13, 2013 at 4:21pm
-  **Karthik Venkatesan** Very Interesting
Like · Reply · October 31, 2013 at 12:59am
-  **Darren Doucet** Thanks for share! The work resembles compiling/connecting brain synapses..... Keeping posting info like this & I might need to start a career minor in another field! LOL
Like · Reply · September 4, 2013 at 6:17am
-  **प्रेम न्त** studing CSE @ GTU, INDIA
Like · Reply · August 28, 2013 at 10:30am
-  **Krishnaraj Subburayalu** Nice
Like · Reply · August 27, 2013 at 4:36pm
-  **Aman Saxena** <http://bughunters007.blogspot.in/>
Like · Reply · August 22, 2013 at 8:24pm
-  **James Martin** Once you send the data to the NSA (sorry I meant once you ignore the fact that the NSA is hooked into your upstream connections grabbing everything thanks to you giving them your SSL keys) with their computer power they should be able to reduce the runtime to nanoseconds
Like · Reply · August 17, 2013 at 3:51am

[View more comments](#)

27 of 52