GraphLab

a review of the distributed data analysis platform

Sebastian Straub
sebastian.straub@mailbox.tu-dresden.de

ABSTRACT

GraphLab was the first distributed computing platform that was able to efficiently handle graph algorithms with strong data dependencies as well as machine learning algorithms on a large scale. The developers achieved this by radically changing the way data was handled by the worker nodes: While customary frameworks like Apache Hadoop focus on the parallel batch processing of large lists of independent data sets stored in a distributed file system, GraphLab models all data in a distributed graph that is stored in memory and therefore allows fast access to interdependent data sets. This approach that to this day is unique among distributed computing frameworks allowed GraphLab to become an important player in the field of machine learning and data mining.

Since the release of GraphLab as a distributed computing platform in 2012, new competitors like Apache Spark and Giraph have emerged. Built on top of Hadoop, these frameworks try to tackle the issue of strong data dependencies with a rather customary approach – yet not without success. In this work, we will give a review of the development of GraphLab and introduce the core concepts behind the framework. After a short performance review, we will discuss the benefits of the different technology stacks used by GraphLab and it's competitors as well as the future prospects of the different players.

Unless noted otherwise, all statements in chapters 1 through 4 are based on the 2010 [3] and 2012 [2] papers that were published by Guestrin's team along with each major release of GraphLab.

1. INTRODUCTION

Machine learning (ML) tasks have become an integral part of data analytics for many businesses, and with the growing size of data sets that are being processed, there is a strong demand for ML algorithms that can handle big data problems. Yet even today, the proportion of multithreaded implementations among ML algorithms is rather limited, and the numbers get even worse when it comes to distributed solutions.

In 2010, when GraphLab was published, Hadoop was still the de facto standard among distributed computation frameworks. But while Hadoop scales very well for tasks with little to no computational dependencies among the tasks that are being processed, it fails when strong computational dependencies are given. Scheduling flexibility and globally shared state are also common requirements for many ML algorithms which Hadoop does solve efficiently on its own.

Hand-crafted solutions on the other hand can be highly efficient and scalable, but they tend to get complex (and therefore error-prone), as they force the developer to solve the same design issues over and over again.

GraphLab is a graph-based distributed computing framework which was designed with the special requirements of many ML algorithms in mind: It provides fast and consistent access to interdependent data as well as globally shared state and it provides the necessary scheduling flexibility. GraphLab was launched in 2009 at Carnegie Mellon University by Prof. Carlos Guestrin. The core components are written in C++ and released under a free software license.

In GraphLab, data is stored as vertices in a graph, while dependencies between them are encoded as directed edges. User-defined update functions can modify a vertex and it's adjacent (dependent) edges and vertices on the graph. The framework executes these update functions in parallel, following the selected scheduling strategy.

While the first version of GraphLab in 2010 was limited to the shared memory setting (i.e. it could only parallelize execution on a single machine), the second major release in 2012 extended the framework to the distributed setting. This required a proper solution for more design issues like distributed graph partitioning, shared memory access, scheduling and fault tolerance. The distributed version was a success, outperforming implementations of ML algorithms in Hadoop by two orders of magnitude and competing with hand-crafted solutions, while only a small fraction of the code is required for the actual implementation in GraphLab.

In 2013, Guestrin successfully launched Dato Inc., a company that develops proprietary extensions such as efficient ML algorithms and big data analytics solutions based on GraphLab.

2. GRAPHLAB: SINGLE MACHINE

When Guestrin's team started working on GraphLab in 2009, they focused their attention on the idea of having a highly parallel graph-based computation framework that scaled well enough for common machine learning algorithms.

Not having to consider issues like distributed memory or network latency allowed the team to release the first version of GraphLab already in 2010, and in order to provide for sufficient computing power, they ran their tests on a multi-socket machine with 4 CPUs and 64GB of shared memory. This caused a hard limit for the size of the data graph, which has to fit into memory at all times, but allowed for excellent performance and a rich variety of scheduling algorithms.

2.1 Related Work (2010)

In 2010, the competitors of GraphLab presented as follows: There were machine learning libraries, which could not handle big data, and there were distributed computing frameworks, which could not handle common ML tasks, or at least not efficiently.

Here's a short roundup of the alternatives at the time of the initial release of GraphLab:

- *scikit-learn*[1] is a popular free software machine learning library written in Python which provides implementations of a large variety of ML algorithms. The vast majority of them however is single-threaded and scikit-learn is not a framework that can be easily extended to support custom parallel or even distributed algorithms.
- *Apache Hadoop*[2] is the first free software distributed computing framework that can handle data on a petabyte scale. Hadoop's MapReduce engine can apply batch processing tasks on files that are stored in the distributed file system HDFS.
- *Dryad* was a research project from Microsoft that was discontinued in 2011. It modeled data flow in a directed acyclic graph (DAG) which allowed for a variety of operations, but resulted in strict execution plans with low scheduling flexibility. Only later projects like Apache Spark and Hive brought the DAG model to it's full potential.

2.2 Components

The GraphLab abstraction consists of a graph that holds all data and models dependencies between individual data sets. Additional information can be stored in a table that is initially empty, but can be filled during runtime by functions that inspect the state of the graph. The actual work is done through update functions that are executed in parallel, each on a single vertex and it's direct neighbourhood. A customizable scheduler has influence over the order of execution and the allowable degree of parallelism.

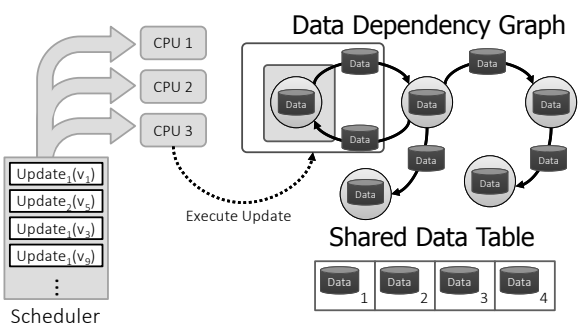

Figure 1: framework components

2.2.1 Data Model

The core data structure of the GraphLab framework is the *data graph* $G = (V, E)$, a directed graph where both vertices v and edges $(u \to v)$ can store arbitrary data in the form of C++ primitives or objects. Data associated with a vertex v is denoted as D_v and for edge $(u \to v)$ as $D_{u \to v}$. There are no limitations concerning the topology of the graph, but there is no way to change the graph's structure once it's been initialized.

GraphLab's secondary data structure is the *shared data table* (SDT), an associative map $T\,[\text{Key}] \to \text{Value}$ which also stores arbitrary values.

While the data graph models computational dependencies and stores the current program state, the purpose of the SDT is to support globally shared state.

2.2.2 Update Functions

Computations in GraphLab are performed through update functions that work on the neighbourhood of a vertex. The neighbourhood S_v of a vertex v (*scope of* v) is defined by v, it's adjacent edges (both incoming and outgoing) and all neighbouring vertices. The data associated with vertices and edges in S_v is denoted as D_{S_v}.

An update function is a stateless local computation on the neighbourhood of a vertex v, with read access to the SDT:

$$(D_{S_v}, T) \Rightarrow (D_{S_v}), \quad \text{short notation: } f(v)$$

A GraphLab program may consist of multiple update functions which are executed by the GraphLab engine in parallel.

2.2.3 Sync Mechanism

The sync mechanism is GraphLab's method of aggregating data over the entire graph. In it's most basic form, it is the equivalent of the higher order function *fold* over all vertices, where the result is written under a specific key into the SDT.

GraphLab provides different operation modes for the sync mechanism, which give the developer the freedom to weigh performance against strictness of execution, depending on the requirements of the algorithm.

The sync mechanism can interrupt the execution of update functions until the result has been written to the SDT, or it can run asynchronously with the update functions. While the blocking mode guarantees consistency of the aggregated results, higher performance can be achieved with the asynchronous execution model. As slightly decreased precision can be acceptable for many ML algorithms, the increase in performance often outweighs the benefits of absolute consistency.

The sync function itself can be executed sequentially or in parallel. While the sequential variant is easy to implement, it does not scale very well for large graphs. For a parallel execution, an additional function is required that combines the results of multiple folds.

For the sync mechanism to work, a key k, an initial (neutral) aggregation value r_0 and up to three functions have to be defined:

- *fold*: $(D_{S_v}, r_i) \Rightarrow r_{i+1}$
- *merge*: $(r_a, r_b) \Rightarrow r_c$
- *apply*: $(r_a) \Rightarrow (r_{a'})$

Fold takes the scope of a vertex and the so far aggregated result (which defaults to r_0) and updates the aggregated value based on the new input. Contrary to it's functional equivalent, GraphLab's fold function can have side effects,

[1] http://scikit-learn.org
[2] http://hadoop.apache.org

i.e. it can modify data in S_v according to the rules of a regular update function.

Merge is an optional function that combines the aggregated results from two fold functions that have been executed in parallel. When merge is not provided, fold can only run sequentially.

Apply is the last function that is called before the result is written to the SDT under the specified key k. It takes the result that was aggregated over all vertices and allows the developer to make some final adjustments.

2.2.4 Data Consistency

Because update and fold functions can be applied to vertices with overlapping scopes, the parallel execution may lead to race conditions and therefore data inconsistency. To mitigate this issue, GraphLab defines three consistency models which give update and fold functions exclusive access to their respective neighbourhood or a subset thereof.

Consistency Models.

The GraphLab framework ensures that update/fold functions that share components from different exclusion sets are never executed in parallel. The available consistency models are:

1. *full consistency*: No other function can access S_v while $f(v)$ is executed. Only vertices that do not share a common neighbour can be processed in parallel.

2. *edge consistency*: No other function can access v and it's adjacent edges while $f(v)$ is executed. Only non-adjacent vertices can be processed in parallel.

3. *vertex consistency*: No other function can access v while $f(v)$ is executed. All vertices can be processed in parallel.

The definitions of the consistency models do intentionally not prohibit unsafe writes to adjacent vertices and edges, but rather define a scope for a function that can be safely accessed and let the developer decide, whether certain unsafe operations should be allowed.

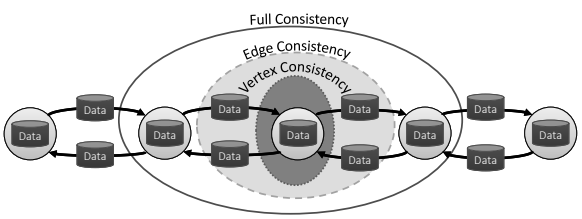

Figure 2: consistency models

While the full consistency model guarantees exclusive access to the entire neighbourhood, it also greatly decreases the amount of operations that can be executed in parallel. The vertex consistency model on the other hand allows maximum parallelism at the cost of unsafe access to S_v with just the exception of v itself.

Sequential Consistency.

To prove the correctness of a parallel execution, the developers of GraphLab introduce the concept of *sequential consistency*:

> "A GraphLab program is sequentially consistent if for every parallel execution, there exists a sequential execution of update functions that produces an equivalent result." [3]

This means that if a sequential algorithm is correct, independent of the execution order, then it's parallel equivalent is also correct, if it is sequentially consistent. In this case, sequential consistency is guaranteed if only data within the safe scope of the selected consistency model is accessed. In case of the full consistency model, this is always the case.

If however the order of execution is relevant, then a proper scheduling strategy is additionally required for a correct parallel execution.

2.2.5 Scheduling Strategies

The scheduling strategy determines in which order and to which degree of parallelism a list of tasks is executed. In this context, a task consists of a vertex and a function that is to be applied to this vertex.

While most of the concepts we have seen so far can be easily extended to the distributed setting, this is not the case for scheduling, where problems like communication between different machines, synchronization and locking are considerable obstacles. In the shared memory setting however, scheduling is a feasible problem and therefore, a variety of schedulers is already provided by GraphLab, along with a construction framework for custom implementations.

Base Schedulers.

GraphLab provides two schedulers that do not provide any control over the order of execution:

- *synchronous scheduler*: updates all vertices simultaneously, using data from the previous superstate. This is a blocking algorithm that switches between superstates in which the results of each update function are collected and written back to the graph only after the last update function has been executed.

- *round-robin scheduler*: updates all vertices sequentially, but in unspecified order, using the latest available data.

While the synchronous scheduler can execute update functions in parallel (not even bound by the rules of the consistency model), it also requires to synchronize once during each superstep and to write back the results, which causes some computational overhead. The round-robin scheduler on the other hand cannot run in parallel and is non-deterministic (results vary based on the execution order), but there is no computational overhead.

Task Schedulers.

More control over the order of execution and the tasks that are included into the scheduling strategy is provided by the two task schedulers, which start with an initial set of tasks and terminate as soon as the task queue is empty:

- *FIFO*: Allows update functions to add tasks to a queue, which are executed in the order in which they've been added

- *Prioritized*: Allows update functions to add tasks, but also to reorder them by defining different priorities for each newly added task

Each of these schedulers is available in a strict (sequential) and relaxed (parallel) version, the latter of which increases performance, but does not guarantee the exact order of execution.

Set Scheduler.

The set scheduler is GraphLab's scheduler construction framework. It is based on the synchronous scheduler, but gives the developer as much control over the order of execution as desired.

The set scheduler takes a set of vertices and an update function, applies this function to all vertices in the set, collects the results and writes them back into the graph; same as the synchronous scheduler would do. By providing a list of vertex set and update function pairs $[(Set\ \langle v \rangle\ ,\ f)]$, we gain more control over the order of execution, while retaining the amount of parallelism the set scheduler provides.

By leveraging the data dependencies that are represented in the graph, GraphLab can even launch the next task, before the current one has been completed, if there is no overlap between the scope of the vertices that are involved.

3. GRAPHLAB: DISTRIBUTED

Since the initial release of GraphLab in 2010 which focused on parallel machine learning algorithms in a shared-memory setting, the developers pushed toward an extension of the framework to the distributed setting. While the single machine approach can be very efficient, it is also the cause for a variety of constraints, which can be mitigated through expensive workstation hardware, but not resolved. The scalable and more cost-efficient way to go was to prepare the framework for the distributed setting. Also, in order to be able to compete with existing distributed computing frameworks like Hadoop, GraphLab actually had to be able to work in the same setting.

In 2012, the new version of GraphLab was released, described by the developers as a "high-level distributed abstraction that specifically targets the asynchronous, dynamic, graph-parallel computation found in many MLDM [(Machine Learning and Data Mining)] applications while hiding the complexities of parallel/distributed system design". [2]

3.1 Related Work (2012)

While we have seen that in 2010 the amount of GraphLab's direct competitors was fairly limited (see section 2.1), the situation has changed in the following two years:

- *Apache Giraph*[3] is "an iterative graph processing system built for high scalability" that was originally started by Google under the name *Pregel* [4]. While it is not specifically built for ML algorithms, GraphLab proves that efficient graph processing can set the foundation for distributed machine learning algorithms.

- *Apache Spark*[4] is a general-purpose cluster computing framework based on the DAG abstraction and extending the Hadoop platform with dozens of functional primitives beyond the already available map/reduce. Spark allows the developer to code in a functional style that is not limited to fixed map/reduce steps, which gives the necessary scheduling flexibility that is required by many ML algorithms, but (at least in 2012) did not provide any implementations of those.

- *Apache Mahout*[5] is a library that provides free implementations of distributed machine learning algorithms, many of which are based on the Hadoop platform. Because Mahout does not ship with it's own distributed computing platform, performance always depends on the underlying framework that is used.

While all of these frameworks have their advantages over the MapReduce pattern, which in comparison is a fairly simple and also limited approach, none of them was in a stable release state in 2012. Hadoop was still one of the biggest players in the field of distributed computing, making it the rival of choice for the GraphLab developers.

Besides using one of the existing frameworks, there is of course always the option to write a custom distributed implementation of a ML algorithm. Such implementations can be highly efficient, as they do not have to conform with any restrictions imposed by a framework, but they also require the developer to solve all obstacles imposed by distributed computing from scratch.

3.2 Problem Analysis

In order to extend GraphLab to the distributed setting, the developers had to solve a variety of new problems that do not have to be considered in the shared memory environment.

Graph partitioning is required if multiple nodes are to work on the same graph. A good partitioning scheme distributes data according to the computation power of each node, minimizes the number of edges that cross between machines and allows efficient load balancing on clusters of varying size. GraphLab supports several distributed graph partitioning heuristics and uses initial over-partitioning to allow for load balancing without repartitioning.

Inter-worker communication was realized with TCP/IP connections between workers using a custom communication protocol. The amount and type of data that has to be transferred depends on the execution engine which we will review in section 3.4.

Shared memory access is a problem that can be circumvented to a certain degree through efficient graph partitioning and messaging between individual workers. Still, GraphLab wants to maintain the concept of SDTs (see 2.2.1), which was realized by making a read-only table available in a distributed file system.

Fault tolerance is a valid concern when long-running jobs are executed on dozens of machines that consist of commodity hardware. GraphLab relies on a checkpoint mechanism that runs asynchronously (via sync functions, see 2.2.3) in regular intervals in order to minimize the performance impact. With a modified version of the Chandy-Lamport snapshot algorithm, GraphLab guarantees consistent snapshots under the condition that at least edge consistency is used on all update functions.

3.3 Process Overview

Before we take a closer look at the GraphLab engine, we will provide a summary of the distributed GraphLab process. Figure 3 shows the two main phases:

[3] *https://giraph.apache.org/*
[4] *https://spark.apache.org*
[5] *https://mahout.apache.org/*

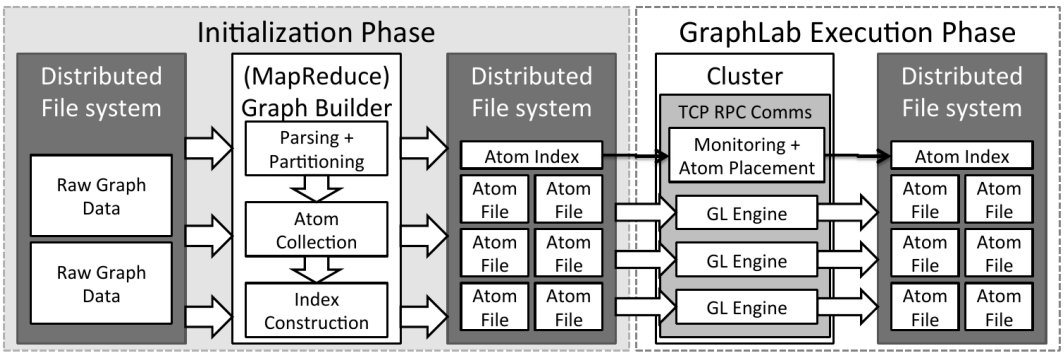

Figure 3: process overview

During **initialization** the graph is partitioned into the so called *atom* collection that contains way more partitions than there are workers currently available. Each atom contains information about edges to vertices stored in other atoms (called *ghosts*). All atoms are stored in the *atom index* which represents dependencies between atoms that result from the edges that cross atom borders. The entire graph partitioning scheme was implemented as MapReduce job, which results in higher performance and solves the distribution problem, as all atoms are stored in HDFS (or a compatible file system) anyway.

In the **execution** phase, each worker gets a number of atoms assigned, joins them to a single graph and resolves all internal ghosts (edges to atoms that are stored on the same machine). Then the GraphLab engine takes over and solves the desired computation by executing the user defined update functions. The resulting atoms are written back to the distributed file system.

3.4 Distributed Engines

In the shared memory setting, scheduling is a problem that can be solved rather easily, as we have seen in section 2.2.5. In the distributed setting however, where each machine contains only a partition of the data graph, a number of new problems arises:

- Enforcing the consistency model is hard, as each vertex may have edges to vertices that reside on different physical machines.
- Enforcing a specific order of execution is hard due to network latency between machines.
- Data transfer between different partitions (and therefore machines) is slow, in terms of both throughput and latency.

GraphLab provides two execution engines that solve these problems. The *chromatic engine* follows a synchronous scheduling strategy based on graph coloring, while the locking engine uses a fully asynchronous approach where workers acquire locks for every update function they execute.

3.4.1 Chromatic Engine

The approach of the chromatic engine is to identify subsets of vertices that can be updated in parallel in accordance with the selected consistency model. Therefore, the chromatic engine can be seen as an extension of the set scheduler that was introduced in section 2.2.5.

To find these subsets and communicate the to all machines, a vertex coloring is applied ahead of time, so that no adjacent vertices share the same color. Update functions are then executed synchronously on vertices of the same color on all machines. After each "color-step", as GraphLab refers to it, execution is halted until all workers have finished and the next color is selected.

Depending on the consistency model, different methods of graph coloring are required. Vertex coloring is sufficient for the edge consistency model, while the full consistency model requires a second-order vertex coloring (no vertex shares the same color as any vertex that is one or two hops away). The vertex consistency model on the other hand can be trivially implemented by assigning the same color to all vertices.

To further increase performance without sacrificing correctness, changes to ghosts are communicated asynchronously while they are made. This results in a more efficient network use and decreases synchronization time, as most of the updates have already been transferred.

The chromatic engine works best for problems whose structure always allows for a representation with few colors (e.g. two-colorable graphs) or where dependencies between edges are low. Drawbacks are imposed by the strict requirement of a full synchronization between all workers after each color-step and the low scheduling flexibility.

3.4.2 Locking Engine

Instead of analyzing dependencies ahead of time, the locking engine just runs update functions that were scheduled in a (possibly prioritized) queue on each machine and acquires locks for all resources, both locally and remote, that require exclusive access, as specified by the consistency model.

Each machine is only allowed to run update functions on local vertices. The ghosting system guarantees that all data in the scope of a vertex is cached locally, which reduces network load. Before an update function can be executed, a lock request is generated and sent to all machines that store vertices within the scope of the function through a distributed continuation passing scheme. This works by passing a message containing all lock requests through a ring of all worker nodes. As each lock request must contain all vertices that are required for an update function to work and the machines are asked for locks sequentially, deadlocks cannot occur.

Only after all locks have been granted, the update function is executed by the requesting worker. Local results

are stored immediately and changes to ghosts are passed on to the continuation passing scheme, along with the release messages for the locks.

To reduce blocking times, all lock requests are pipelined, which means that update functions are not necessarily executed in the order imposed by the (priority) queue, but as soon as the lock for the update function has been granted. Still, higher priority is given to update functions that are earlier in the queue.

The locking engine provides decent scheduling flexibility through priority queues and scales almost linearly for sufficiently large graphs. Performance can be increased by increasing the pipeline length (at the cost of reduced control over the order of execution) and degrades for graphs with a very high clustering coefficient, as large portions of the graph may have to be locked for a single update function to be executed.

4. PERFORMANCE ANALYSIS

As of 2015, there was no thorough independent performance analysis of GraphLab in comparison to other distributed computing frameworks. Therefore we will present the benchmark results that were published by the developers of GraphLab in 2012 [2].

The GraphLab developers performed their tests on Amazon EC2 with up to 64 nodes, each with a (then) state-of-the-art quad core CPU, 22GB RAM and a 10 Gigabit Ethernet connection. For the purpose of the benchmark, three applications have been implemented and tested:

- Netflix movie recommendation: A recommender system based on *collaborative filtering* that predicts the rating of a user for arbitrary movies by comparing the ratings that were provided by that user to those of users with similar taste. It was implemented using the alternating least squares (ALS) algorithm, which works on a sparse users by movies matrix. In GraphLab, the matrix was represented as a bipartite graph where each user has weighted edges to all movies they rated.
- Video Co-segmentation (CoSeg): This application "identifies and clusters spatio-temporal segments of video (...) that share similar texture and color characteristics" [2] which can be useful in computer vision tasks (e.g. robotics). The implementation uses Gaussian Mixture Models to estimate the best label for each fixed-sized partition of each frame from color and texture statistics. The algorithm then connects adjacent partitions in space (2D) and time (1D) and applies Loopy Belief Propagation to smooth the results.
- The third application is about Named Entity Recognition (NER), which can be defined as "the task of determining the type (e.g., Person, Place, or Thing) of a noun-phrase (e.g., Obama, Chicago, or Car) from its context (e.g., 'President ___', 'lives near ___', or 'bought a _')" [2]. As with the Netflix movie recommendations, the data graph for this task is bipartite: noun-phrases are connected by an edge to their respective contexts. The application uses the CoEM algorithm which predicts a label for all noun-phrases from a small set of pre-labeled noun-phrases.

Figure 4 visualizes the benchmark results: The left plot shows the relative performance gain of each application when more machines are added and the right plot shows the absolute runtime (log scale) of the Netflix movie recommendation problem for different numbers of nodes and implementations. The three implementations that were compared are GraphLab, Hadoop/MapReduce and a custom implementation based on the Message Passing Interface (MPI), a communications protocol used for programming parallel computers.

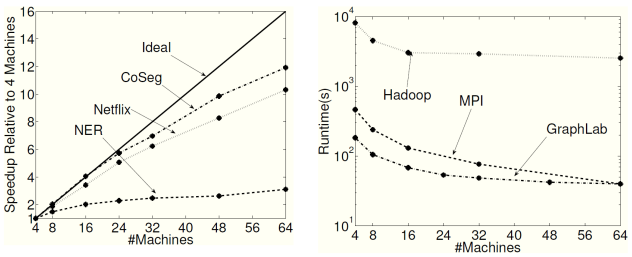

Figure 4: Performance analysis of GraphLab. [2]
Left: Scaling of 3 different ML problems on the GraphLab platform.
Right: Comparison of runtimes for the Netflix problem.

For the Netflix application, the Chromatic Engine was used, as there are no dynamic scheduling requirements and the graph is trivially two-colorable. The results show that the collaborative filtering algorithm scales well, but not ideally, with an increasing number of nodes. The results also show, that the GraphLab implementation is two orders of magnitude faster than a comparable Hadoop implementation and can compete well with a custom MPI implementation of the specific algorithm, although it cannot be verified how much effort the developers put into the implementation of their competitors.

The CoSeg application was implemented using the Locking Engine. This is owed to the fact that there is no trivial graph coloring for the representation of this problem and that the filtering algorithm can benefit greatly from dynamic prioritized scheduling. The test results show that the fully asynchronous locking engine can scale almost linearly up to 32 machines and continue in with a 10x performance gain for 16x more machines.

Even though the NER application is two-colorable (and therefore the chromatic engine was used), it stands out among the other two algorithms by scaling far worse, even for a small number of machines. The authors credit this problem to the high density of the graph, which does not allow for an efficient partitioning and therefore cause a lot of traffic between cluster nodes. And indeed, a network traffic analysis showed that transfer rates for the NER application were usually above 100 MB/s, while for the other application's traffic never exceeded an average of 10 MB/s.

Graph Processing Performance.

Apart from the analysis of GraphLab's performance as a distributed machine learning platform, which so far was only done by the GraphLab developers themselves, there are publications that measured GraphLab's performance in comparison to other graph processing tools, using only graph analytics problems.

In a benchmark by Guo et. al. [1], GraphLab's perfor-

mance was measured against Apache Hadoop, Apache Giraph, Neo4j and Stratosphere. In most scenarios, GraphLab clocked in second after Neo4j, except for large problems where Neo4j failed entirely, as it is not a distributed computing framework. They also showed, that GraphLab was massively slowed down by single-threaded file loading, which in some cases would take up more than 90% of the runtime. Yet even so, due to their much lower computation time, GraphLab and Apache Giraph (which had similar issues) were the best performing distributed graph processing tools in the benchmark.

In a more recent test parcours created by the developers of GraphX [6], the graph processing module of the Apache Spark project, the situation was similar: GraphLab was able to get ahead of Apache Giraph and it performed better than GraphX. Still, the graph loading time, though greatly improved, was still an obstacle for GraphLab, which made GraphX take the lead when the runtime of the whole processing pipeline was measured.

5. CURRENT DEVELOPMENT

The development of GraphLab is still in active progress, but the focus has shifted away from the open source core towards the proprietary extensions that Dato Inc. provides. Besides a variety of ML algorithms and data analytics solutions, they also include bindings to other big data ecosystems like Amazon S3, Apache Hadoop and Spark. Still, these features come at a cost: According to dato.com the software licensing fees are currently at $4,000 per machine and year. The proprietary nature of these vital extensions (as well as the price tag) may be reasons for the popularity of some of GraphLab's new competitors.

As of 2015, probably the most important competitor of GraphLab is MLlib [5], the machine learning library by and for Apache Spark. MLlib is a collection of ML algorithms and utilities which are implemented using Spark's DataFrames. Unlike GraphLab, Spark does not rely on a distributed data graph; it uses the more traditional way of distributing data using an underlying distributed file system. What makes it different from frameworks like Hadoop is the large variety of distributed higher-order functions and a scheduling flexibility that was not provided by the comparably simple MapReduce approach. Also, it benefits from the close integration into the Spark ecosystem, which solves a lot of problems (like loading times, pre- and post processing) that GraphLab had to hassle with for some time. While Spark's MLlib is still in an experimental state and usually less performant than GraphLab, it is free software that is in active development by a dedicated community which GraphLab, by it's now predominantly proprietary nature, can not benefit from.

6. CONCLUSION

GraphLab is a graph-based distributed computing platform tailored for data analytics tasks and specifically for machine learning algorithms. It's approach of local update functions that can only access the neighbourhood of a vertex makes it unique among other frameworks in that it allows to execute algorithms on both large and highly dependent data sets without having the entire graph in the memory of a single machine. The negative impact of network latency is greatly reduced by the ghosting system and the asynchronous exchange of data between workers. A dynamic scheduling system allows for shorter iteration times and therefore faster convergence for many analytics problems.

With these properties, GraphLab stands out among other distributed computing frameworks like Hadoop, which are predominantly batch oriented. Through it's different approach, GraphLab was able to outperform comparable implementations of machine learning algorithms in Hadoop by two orders of magnitude and made it possible to solve problems with heavy computational dependencies without the need to swap data to disk.

Five years after the release of GraphLab, new competitors have appeared. Apache Spark has seen a steep rise in popularity and with it's subprojects GraphX and MLlib it is targeting the same domain as GraphLab. While both try to solve problems of the same kind, they do so using an entirely different technology stack that is also reflected in the style of programming: Spark with it's functional-style data streaming approach and GraphLab using the vertex-centric update functions.

When it comes to performance, GraphLab often clocks in first, but the gap is closing and it stands to discuss, whether GraphLab's largely closed-source approach will win this battle against the dedicated developer communities of competing free software projects.

7. REFERENCES

[1] Guo, Y., Biczak, M., Varbanescu, A. L., Iosup, A., Martella, C., and Willke, T. L. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 395–404.

[2] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[3] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new framework for parallel machine learning. *CoRR abs/1006.4990* (2010).

[4] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. 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, ACM, pp. 135–146.

[5] Meng, X., Bradley, J. K., Yavuz, B., Sparks, E. R., Venkataraman, S., Liu, D., Freeman, J., Tsai, D. B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M. J., Zadeh, R., Zaharia, M., and Talwalkar, A. Mllib: Machine learning in apache spark. *CoRR abs/1505.06807* (2015).

[6] Xin, R. S., Crankshaw, D., Dave, A., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: Unifying data-parallel and graph-parallel analytics. *CoRR abs/1402.2394* (2014).