# Survey on Distributed Large Graph Processing System

Shravan Kale
Computer and Information Science Department
University of Oregon
*shravank@cs.uoregon.edu*

*Abstract*—**Graph Processing Systems must cater to the large size of the graphs that are generated today. Web and Social graphs are up to the size of a billion vertices and edges that contain knowledge which needs to be analyzed. Existing parallel systems solutions aren't always efficient and distributed systems tend to overcome some of the challenges of a parallel system like providing fault tolerance and cheaper costs of individual machines. Hence the motivation to survey the currently popular systems and the ones proposed across the literature. We also asses Facebook's modification of an existing system to scale to their large graph of a trillion edges. Throughout this survey paper, we compare such systems that are built for different requirements.**

*Keywords*— **Distributed Graph Processing System, Pregel, GPS, Mizan, MOCGraph, Arabesque, Tux2, Graph Mining, Machine Learning**

## I. Introduction

Processing large graphs is challenging because of the sheer size of the graphs. Lot of the BigData generated today can be modeled and efficiently processed by expressing it as graphs. This expression lets us exploit many of the graph algorithms and provides a platform for new graph-based algorithms.

There exist many parallel graph processing systems not reviewed in this survey which are efficient and even better than distributed systems though limited by the constraint of memory. There exist parallel systems which can scale to store large graphs but this scaling adds the issue of fault tolerance, increased network I/O and the expensive cost of maintaining said systems which do not make this system pragmatic. Hence the need for distributed graph processing systems which can address the challenges faced by the parallel systems.

One distributed graph processing system does not fit the needs of all the graph processing applications. Due to the modularity of the system, there is scope for iterative and innovative improvement of its many modules. This is the reason why this topic is so popular in the research community spawning many systems catering to different applications and performance requirements.

Some of these modules that have been worked upon are:

- Graph Partitioning
- Load balancing
- Load balancing
- Different Graph Structures
- Scalability

Through the survey paper, we look at such systems that improve the performance of some of these modules and the special features that they offer.

This survey paper covers some systems that are not general purpose systems and cater to a particular domain. Graphs are generally assumed to have homogeneous vertices but some systems like a recommendation system require heterogeneous vertices. There exists a system to cover this special requirement. Some application like graph mining and machine learning have their own challenges and requirement. We look at two such systems that focus on dealing with these challenges.

Pregel is a scalable general purpose system for implementing graph algorithms over graph representations in a large scale distributed environment. For any survey on graph processing systems, Google's Pregel would be found at the center as many systems build upon it's "think like a vertex" processing model. There are some papers which use features from this system but also challenge the model and demonstrating promising results. Hence

Pregel was chosen as the root paper. It also suggests an alternative to the popular MapReduce. Implementing graph algorithms in MapReduce adds more of an overhead since the state of the graph has to be passed from one stage to the next. The iterative step nature of Pregel reduces the complexity of programs written for MapReduce.

An important aspect of these systems is their ease of use. These systems are implemented and provide application programming interfaces for anyone to use an existing system and execute their graphs. The very notion of computing over a vertex introduces the ease of implementation. This paper hence also covers the APIs of some systems and their implementation.

## II. Different Graph Proccesing Systems

### A. Pregel

#### A.1 The system

Pregel[1] is Google's proprietary system for Distributed Graph Processing System. As a system, it was designed for vertex-centric algorithms with message passing capabilities. It runs iteratively in what it defines its greatest feature supersteps such that vertices can send messages to itself and other vertices from current superstep to the next one and also receive messages from the previous superstep. Vertices can also modify their own state by modifying user-defined values, flags etc, modify the state of its edges and add-remove its edges as required by any algorithm defined by the user.

Since the message passing takes place between supersteps and using checkpointing, the model takes care of race conditions and deadlocks. The paper reasons that message passing model is better for network latency than remote-read.

#### A.2 Computation model

Pregel is a vertex-centric system wherein during each superstep the vertices are executing in parallel across different workers across different clusters. The vertices can communicate with each between the supersteps using a message passing model. The vertices execute until they decide to halt and can be re-invoked by a message passed to the inactive vertex. Such a design ensure that not all the vertices are active if they are unnecessary to the computation. The algorithm terminates when all the vertices decide to halt and there exist no messages in transit like shown in figure 1. The output of the computation can be a graph which is isomorphic to the input graph unless an algorithm like clustering add or removes the edges of the vertices.

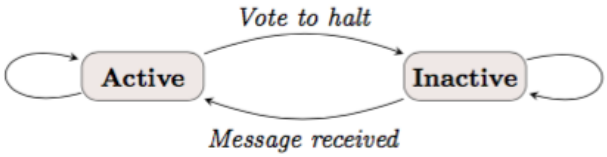



Fig. 1. Vertex State Machine

#### A.3 Architecture

Graph Processing systems like Pregel are implemented with workers on a cluster having a single master that coordinates all activities and is responsible for certain things like fault-tolerance, storing global values, worker reassignment etc. The workers compute on the vertices in parallel between supersteps.

#### A.4 The Application Programming Interface

Pregel lets the programmer subclass the vertex class to create a node and define vertex/edge value as required. Pregel provides access to functions using a C++ API. A user can simply override a function like compute() that is executed at active vertices in every superstep. The function can also get the current value of the state of a vertex, modify the value, add-remove edges of the vertices and edges. Since the modifications are confined to a superstep, it ensures that other vertices don't access values of a previous step.

#### A.5 Special Features

Pregel relies on message passing between vertices for information sharing. A vertex can send messages to its neighbors by iterating over it's known edges and or to any other vertex whose ID might be explicitly provided. The paper does not

mention the routing of messages to non-neighbours and it could be handled by the worker on the cluster it's working on. The iterative nature of pregel allow for a delivery guarantee, no duplicates but does not guarantee ordering of messages.

Another feature that can reduce network I/O and buffer load is an optimization called combiners that can be defined explicitly by a user by function overriding. Messages sent to a single vertex can be reduced if the operations are commutative and associative hence reducing the number of messages sent and reducing network I/O. The paper evaluates a reduction in message traffic using reducers.

Global co-ordination and monitoring can be achieved by user overridable aggregators. Apart from local aggregators, sticky aggregators can be used who maintain a global state throughout the supersteps. The paper defines a few use-cases that a user can define for their own implementations.

Topology mutations are handled by partial ordering that makes sure that none of the mutation requests change the graph structure in a undesired way due to the possibility of conflicting and out of order arrival of add-remove messages. It also provides the user to define a handler to resolve said conflicts by defining rules that govern the resolution of those conflicts. Pregel also provides local mutations for vertices that avoid conflict.

### A.6 Partitioning

Pregel does not focus on performance and hence it does not consider different partitioning techniques. Modularity of Pregel allows the user to define its own techniques, as a single technique may not suit every application. As a default, it uses hash partitioning using the mod of number of vertices as it's hash function. Though the system does allow the user to define multiple partitions per worker to execute in parallel.

### A.7 Fault Tolerance

The master and workers exchange periodic ping messages to check their active status. This determines the termination and or reassignment of the partition assigned to a worker.

Fault tolerance in Pregel is handled by a system

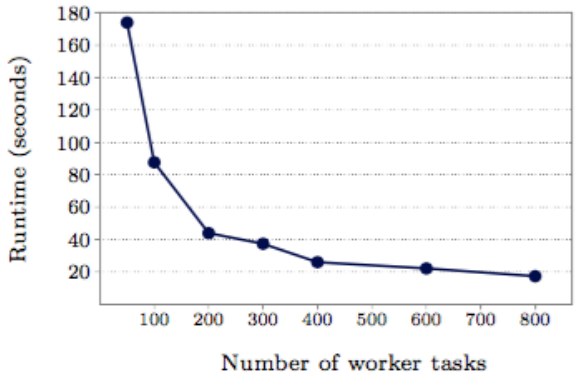



Fig. 2. SSSP-1 billion vertex binary tree: vary-ing number of worker tasks scheduled on 300 multicore machines

called checkpointing. Where in the state of a partition is stored by the vertex at the start of a superstep. The state is saved by writing it to storage. This does add an overhead on the runtime but can be disabled during run time if required. The frequency of checkpoints can be balanced against that of recovery cost. If any of the workers fail, their partition is reassigned by the master to a new worker. Though this new worker might have to repeat the supersteps of all the partitions, if the saved state is that of an older superstep. Confined Recovery uses the logs of the messages of the partitions for recovery of missing partitions. This method is resource efficient and has enough disk bandwidth to not bottleneck the I/O.

### A.8 Scalability

The paper experiments to test the scalability of system using the shortest path algorithm on binary trees using default partitioning and disabled checkpointing due to low probability of failure in the test case.

From figure 2 and 3 we can see that as the no.of workers across clusters are scaled the run time reduces providing a speedup of 10x by using 16x times workers. Whereas for graphs of increasing number of vertices the run time increases linearly and not exponentially. Hence proving the scalability of the the system. Further experiments are conducted on log-normal random graphs trying to em-

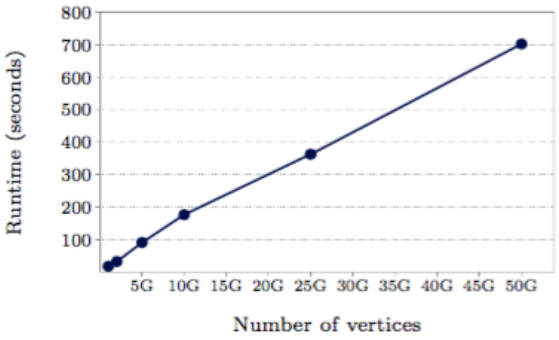



Fig. 3. SSSP-binary trees: varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

ulate real world graphs. The performance results remained the same.

### *B. GPS: A graph processing system*

#### B.1 The system

GPS: A graph processing system[2] is an open-source extension of the Pregel system. It focuses on extending some features of Pregel to improve performance. The paper details the features and some experiments.

#### B.2 Computation model

Just like Pregel GPS also takes a directed graph as input. Each vertex of the graph stores a user-defined value along with a flag signaling the activity or inactivity of the said vertex. The worker, master and superstep paradigm of Pregel remains the same through GPS. Though unlike Pregel the master and the workers communicate using Apache Mina and use the Hadoop Distributed File System to store data like the graph and saving states during checkpoints(a recovery measure).

#### B.3 Implementation

GPS is implemented with workers on a cluster having a single master that coordinates all activities and is responsible for certain things like fault-tolerance, storing global values, worker reassignment etc.

In GPS the workers run three threads. One thread for iterating over the vertices and running the compute() function, another Apache Mina thread for passing messages over the network and a message parser thread for parsing the incoming message buffer. Since these buffers are large there exist only two points of synchronization between the above threads. The rest of the implementation remains same as Pregel.

#### B.4 The Application Programming Interface

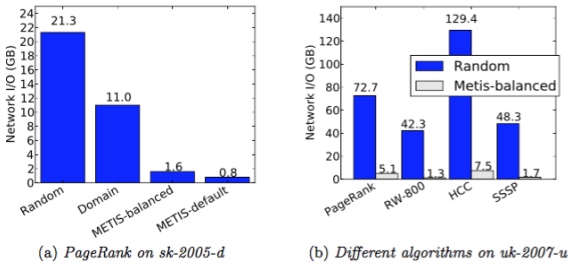
GPS provides to similar functionality like Pregel using functions. In addition, it also defines to the voteToHalt() function which a vertex can call to terminate itself. As previously discussed GPS provides access to a master.compute() function which is subclasses the master class, initialized at the beginning of a superstep. This class has access to global objects and can also store private objects that are not visible to the vertices. These are used for algorithms that require vertex-centric computations as well as global computations like k-means clustering algorithm.

#### B.5 Special Features

GPS also uses the bulk synchronous processing based message passing system used by pregel. GPS uses a custom implementation of aggregators in Pregel called Global Objects. At the beginning of every superstep these are available to all vertices and they can update the objects in their local map which is then aggregated to the master at the end. Some other optimizations of GPS include using combiners for messages at the message receiving workers and by using canonical objects to reduce the memory cost of creating objects in Java.

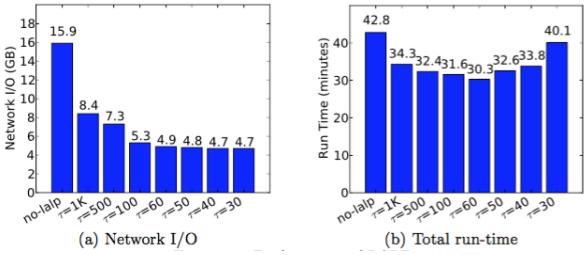
#### B.6 Partitioning

In GPS the input graph is stored as a hash map with vertex ID as the key and it's neighbors as it's values. Additionally, the value of vertices and edges can be given too. By default, GPS uses the same partitioning scheme as Pregel. Since GPS focuses on performance, the paper proves that well-partitioned graphs improve performance. It suggests using Random, METIS-default, METIS-balanced and domain based partitioning for different graphs, on different algorithms and worker con-



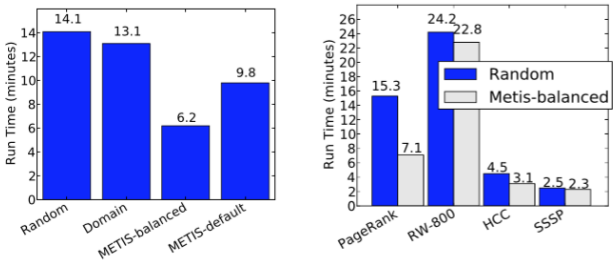

(a) *PageRank on sk-2005-d* (b) *Different algorithms on uk-2007-u*

Fig. 4. Network I/O, different partitioning schemes



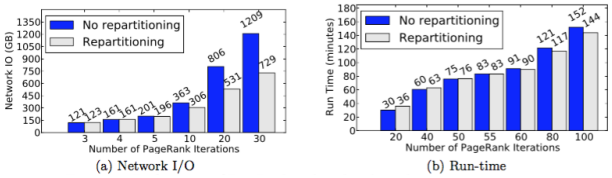

(a) Network I/O (b) Total run-time

Fig. 6. Performance of LALP




(a) *PageRank (50 iter.) on sk-2005-d* (b) *Different algorithms on uk-2007-u*

Fig. 5. Run-time, different partitioning schemes




(a) Network I/O (b) Run-time

Fig. 7. Performance of PageRank with and without Partioning

figuration. Random partitioning provides balance but also increases the network I/O dues to the increased message passing. We can introduce an intelligent skew to reduce said message passing to achieve a balance between the two. The paper introduces counter-measure for the skew created by the partitioning schemes.

We take network I/O and run-time as the two performance measurement metrics. From Figure 6.a compares the write per GB(measure of network I/O) for various Partitioning algorithm and 6.b compares compares the performance of various algorithms for given two partitioning techniques. METIS-default performs the best in terms of graph partitioning techniques. From figure 7.a we observe that METIS-balanced provides the most reduction in run-time. Where as in figure 7.b we see that PageRank benefits the most using METIS-balanced since PageRank involves high amount of network I/O which we reduce to gain the observed run time. Work-load balance is another factor that affects performance as the run-time is impacted by the worker with most no.of vertices in case of skewed partitions generated by the techniques. The paper shows experimental results that by increasing the number of partitions per worker it achieve an averaging and balancing effect which improves the runtime of techniques like METIS-default that skew the distribution of vertices amongst partitions.

GPS suggests another optimization technique called Large Adjacency-List partitioning(LALP) that benefits only certain algorithms. The adjacency lists of high degree vertices are partitioned across workers. This technique is beneficial for real-world graphs which have few very high degree nodes. From the figure above we can observe that for a reducing threshold of vertex degree the network I/O decreases as per vertex messages decrease. Though we can see that there exists an optimal threshold for reducing the run-time as the per worker map size increases with decrease in the threshold value.

The other suggested main feature of GPS is dynamic repartitioning of the vertices during supersteps. In this method the graphs are repartitioned during the supersteps to reduce network I/O and run-time. This feature has it's own limitation that the algorithm needs to run for a certain number of iterations for the feature to show its benefit and not just incur a cost greater than what it's trying to reduce. We can see this from the figure above that

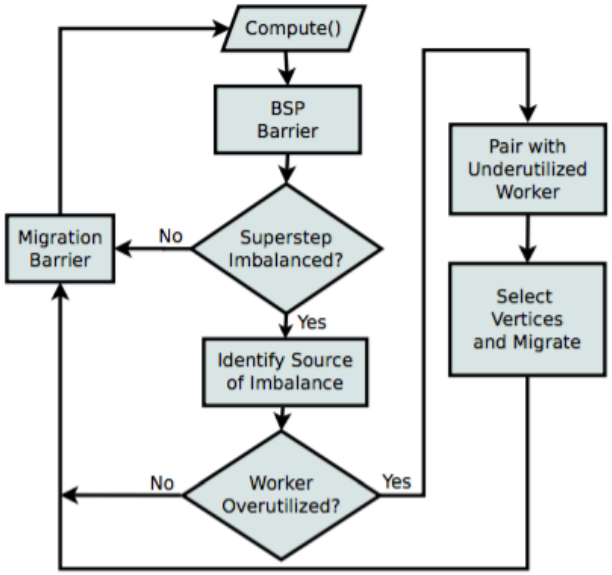



Fig. 8. Summary of Mizan's Migration Planner

dynamic partitioning shows reduction in network I/O and run-time required after a certain no.of iterations using random partitioning. Domain-based partitioning provides even better performance.

### B.7 Fault Tolerance

Inherits the fault tolerance implementation of Pregel

## C. *Mizan*

### C.1 The system

Mizan[3] was created from pregel as a system for adaptive and dynamic load balancing for algorithms whose runtimes are not known. The system was designed to be agnostic to the graph structure. Load balancing is done by monitoring the vertices during execution time and creating a vertex transfer plan at the end of the superstep to minimize the variation across the workers. It uses a hash table based service to keep a track of the vertices.

### C.2 Computation model

The model of Mizan is similar to Pregel except for it's dynamic load balancing feature. This involves monitoring of runtime characteristics and

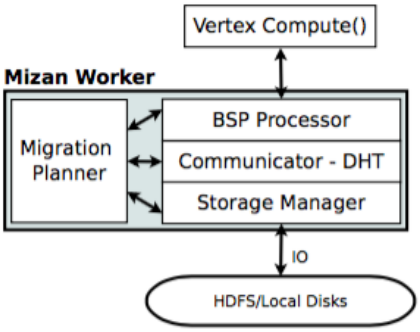



Fig. 9. Architecture of Mizan

planning the migration. The monitoring of vertices involves the outgoing messages to remote workers that incur network cost, the incoming messages which affect performance because of limited buffer size and their execution time. Migration planning involves planning migration with respect to the metric that causes the most imbalance of the workload. Figure 8 shows the summary of the migration steps.

### C.3 Architecture

The architecture of Mizan is illustrated in Figure 8. THe BSP Processor implements the Pregel API with compute class and does the barrier synchronization. The storage manager ensures the correctness of the graph data and the other data structures. The Hadoop Distributed File System(HDFS) is used for graph storage and checkpointing if any. The Communicator uses Message Passing Interface(MPI) for message passing between the workers and maintains information of vertex ownership. The migration planner plans and migrates the vertices for dynamic load balancing.

### C.4 Special Features

Mizan uses a Distributed Hash Table lookup service to distribute the overhead of searching vertices over all workers. Vertices use a hash function to find the default worker of target vertex which stores the updated location. The updates of these tables takes place before the superstep has ended to ensure consistency. Upon migration the location of

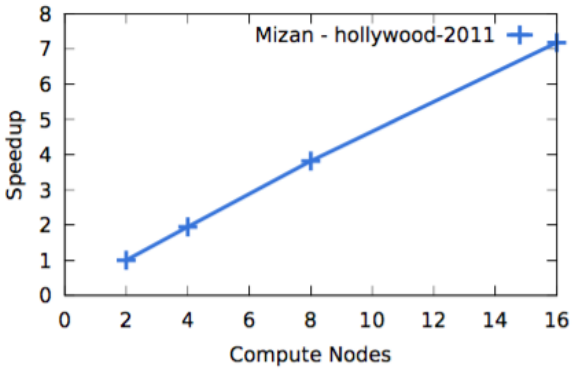



Fig. 10. Speedup on Linux Cluster of 16 machines using PageRank on the hollywood-2011 dataset

the reallocated vertices is broadcasted to all workers that have asked for the vertex or have previously cached it. When vertices are migrated their state including their queued messages have to be transferred to the new worker, Mizan uses a technique called delayed migration where the migration is done over two steps to minimize the cost of vertices with large state size. This is done without any location mismatch.

### C.5 Partitioning

Mizan uses predefined or user-defined initial partitioning scheme but uses dynamic load balancing or repartitioning during execution. First it identifies the source of imbalance, then selects the strongest objective for migration. It then causes an averaging effect by pairing works with heavy and low workload. After selection of vertices, it migrates them hence re-partitioning the graph dynamically.

On a given dataset Mizan does incur an overhead because of the vertex migration but it perform better than static Mizan(without dynamic load balancing).

### C.6 Scalability

Mizan was tested against a linux cluster and an IBM supercomputer. From Figure 10 we can see that the speedup increases linearly for the linux cluster but flattens for the supercomputer as more workers involves an overhead of message passing.

## *D. PowerGraph*

### D.1 The system

PowerGraph[4] address the challenges faced by existing systems when computing over natural graphs that have highly skewed power-law degree distributions. Natural graphs tend to have a smaller subset of vertices that connect to a large fraction of the graph. The system exploits the structure of the said graphs for computation to provide better performance and scalability.

### D.2 Computation model

PowerGraph uses a Gather, Apply and Scatter model for vertex computations. In this model, the vertex values of neighbours of a vertex are gathered during the Gather phase using the sum operation. Hence the values must be commutative and associative. The apply phase is just like the compute function of pregel wherein the computation is done on the vertex. The new updated value is sent to to the neighbors of a vertex in the scatter phase. In pregel, the gather phase is implemented using the compute function with combiners. The three phases are incorporated with a vertex-program function that is executed per vertex in parallel.

### D.3 Implementation

It uses a Hadoop Distributed File System(HDFS) to import the graph and other data. The implementation of PowerGraph is done in three ways:

Bulk Synchronous Implementation: The three phases Gather, Apply and Scatter run synchronously on vertices are called minor-step. A superstep comprises of all the minor-steps. Updated vertex and edge values are available at the start of every minor-step as they are committed at the end of the previous minor-step. This ensures a deterministic execution. Though some algorithms may converge slowly or may not converge at all.

Asynchronous Implementation: Updated vertex and edge values from the apply and scatter function are immediately visible to the neighbors for their own computation. Since active processors are executed immediately once network and processor

resources are available this ensures their effective utilization. This leads to faster convergence of algorithms like that of greedy graph coloring algorithm. Though determinism is limited by available resources.

Asynchronous Serializable Implementation: The performance of the asynchronous implementation is limited by the availability of resources. A serializable implementation prevents parallel execution of adjacent vertex-programs by using a parallel locking protocol. Unlike other systems which use sequential locking which is unfair to higher-degree vertices parallel locking addresses those limitations.

### D.4 Special Features

Delta Caching: Vertex-program is triggered on vertices when there are changes made to its neighbors. The vertex-program invokes the gather function which may end up gathering same old values of the neighboring vertex does introduce redundancy. Powergraph caches the results of the previous gather phase and updates it when a change(delta) is received from the scatter function. If nothing is returned from the scatter function then the cache is dropped to add the new value from the next gather function. If this cached value is available it saves the computation required for another redundant gather function calls.

From Figure 11 we can see that Delta caching reduces the runtime as the iterations of PageRank increase on a twitter dataset.

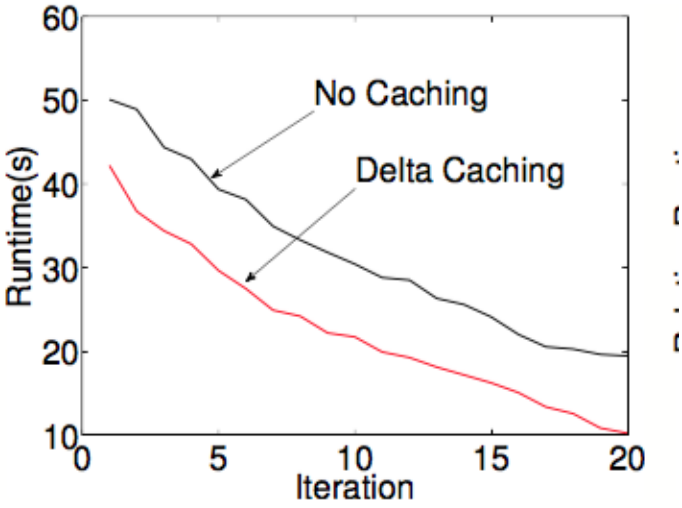



Fig. 11. The PageRank per iteration runtime on the Twitter graph with and without delta caching

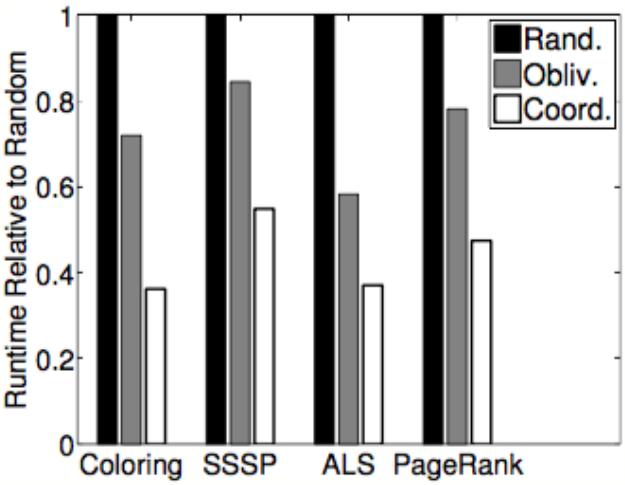



Fig. 12. The effect of partitioning on runtime

### D.5 Partitioning

Partitioning large power-law graphs is challenging and Pregel-like systems generally resort to random hashing of vertices which has poor locality. The system addresses this by partitioning the edges randomly and evenly across the workers where the edges need to be replicated. Though the PowerGraph abstraction allows the vertex-program to span across machines. This means that for a vertex over two machines the communication overhead only includes the the accumulation and updation of data by message passing as the gather, apply functions can be parallelized.

The system also uses balanced p-way vertex-cut to minimize the no.of workers spanned by a single vertex. The paper also suggest a greedy method of vertex-cut where the edges are not partitioned randomly but by using a greedy heuristic to minimize expected replication factor. This greedy algorithm has two implementations, Coordinated where there exists a global table for each machine to update the value of previous edges to and Oblivious where the machine maintains a local table but runs the heuris-

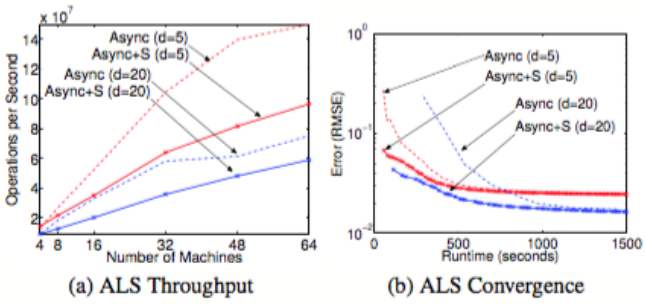



Fig. 13. (a) The throughput of ALS measured in millions of User Operations per second. (b) Training error (lower is better) as a function of running time for ALS application

tic independently. We can see the variation in performance across various algorithms from figure 12

### D.6 Fault Tolerance

PowerGraph uses the same checkpointing system as Pregel. For synchronous implementation, the state is saved only between two supersteps whereas for it's asynchronous implementation execution is suspended to save the state.

### D.7 Scalability

The scalability of the asynchronous(Async) and asynchronous serializable(Async+S) is evaluated on the Alternating Least Squares(ALS) algorithm. The d values is a parameter of the ALS algorithm that controls it's complexity. We can see from figure 13 that Async+S performs better than Async in terms of throughput and convergence.

## *E. MOCGraph*

### E.1 The system

MOCgraph[5] is proposed as a distributed graph processing system that reduces the memory footprint by processing the messages in a streaming manner to save buffer space and disk I/o if and when needed. This technique is called as Message Online Computing(MOC). Reducing the footprint allows the system to scale to larger graphs and do more complex analytics that might require larger space. The system also extends MOC for external storage to support out-of-core execution. It compares itself with the combiner feature to reduce

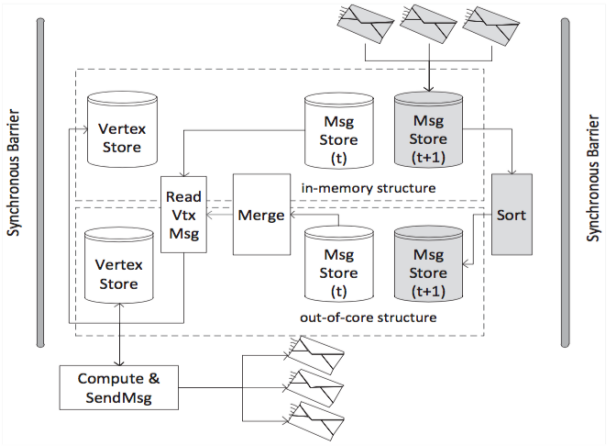



Fig. 14. Data Flow in Original Giraph at Superstep t

memory footprint used on Pregel which it finds restrictive and suggests MOC as an alternative.

### E.2 Computation model

The paper compares its MOC model with that of Pregel-model and Pregel-model with combiners. It' finds that both of them store the entire incoming message and the new message to be sent to the next superstep thus holding them in memory. Although the combiner model reduces the space by combining messages per vertex, the space requirement is still large and depends on the size of the message. This model executes over 2 supersteps. On the other hand, MOC updates the value of the vertex for every batch of the incoming message thus reducing the need to store the entire message in memory. This also facilitates the ability to access the vertex value, produce and process the value of the messages in the same superstep unlike that of the Pregel-systems.

Asynchronous Computation: Messages sent by the vertex are the latest value of the vertex at sending time. This speeds up the convergence of graph algorithms.

Synchronous Computation: For every superstep I the message sent at first is restricted to the first updated value of the vertex instead of the one available when the message is sent. Since the final value is known before the end of the superstep, this value can be determined hence preserving deter-

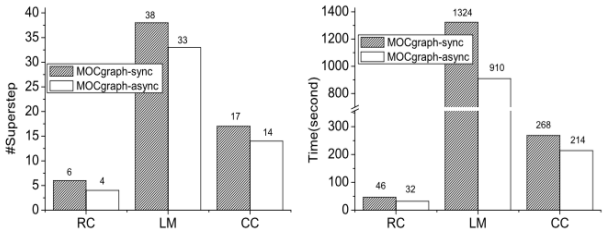



Fig. 15. Superstep and Time Cost for both sync and async mode: we choose Twitter as the dataset, and set the number of workers 23, 23, 46 for Reachability, Connected Components and Landmark, respectively. The number of landmarks is set to 25. All cases are running with sufficient memory

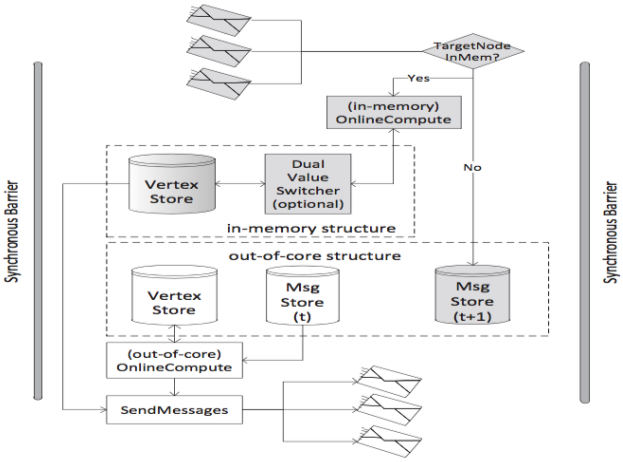



Fig. 16. Data Flow in Original Giraph at Superstep t

minism. The synchronous model is implemented by not storing the intermediate value but only the final value received from an unexpected vertex to save space.

From figure 15 we can see that due to the sending of messages based on their most recent value graph algorithms converge faster for the asynchronous executions. They reduce the no.of supersteps required and the execution time.

### E.3 Implementation

MOCgraph is built on top of Apache Giraph using the MOC model.

Data Storage: The vertex is data is stored in memory in a vertex store whereas the messages are not stored for in-core computation as they are streamed. For out-of-core computation, a separate vertex store and a message store is used to store vertices and messages respectively.

### E.4 The Application Programming Interface

The compute function used in pregel is split into onlineCompute() and sendMessages(). The former takes care of the message computation and vertex update while the later takes care of sending messages in the current superstep. CreateNewVersion() specifies the generation of new values based on old one for synchronous executions.

### E.5 Special Features

The paper suggests certain optimizations for reducing the disk I/O for out-of-core executions. One of the methods it suggests is edge separation in which the edges are removed for some graph analytic applications which don't use edges to store information but only to pass messages. This safely sacrifices some edges to store more vertices in memory.

Some vertex partitions need to be swapped between memory and disk depending upon the messages that come in. This frequent swapping can create disk I/O that needs to be reduced. The paper introduces a concept call hot-aware partitioning in which partitions that are receiving the most messages hence the term hot are placed into the memory to minimize the disk I/O

From Figure 17 we can see that the combination of edge separation and hot-aware partitioning works very well and reduces the execution time consistently for PageRank, Connected Components and Landmark Index Construction.

### E.6 Partitioning

Uses existing partitioning methods that can be defined by the user.

### E.7 Fault Tolerance

It uses the same checkpointing system like that of Pregel but has the less added overhead of the incoming messages to be saved in a superstep. For in-memory executions, no messages are stored

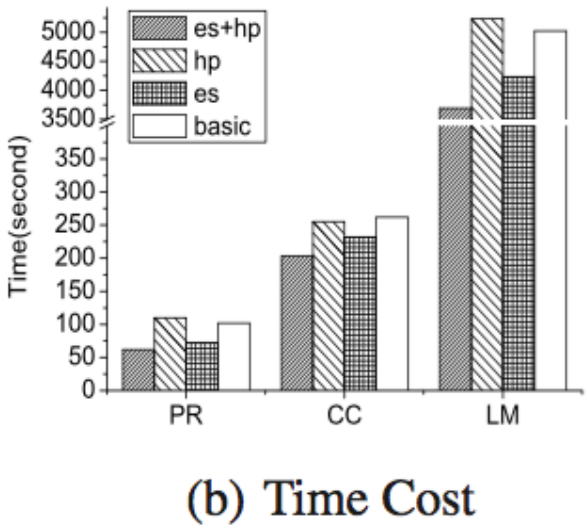



Fig. 17. Effects for edge separation and hot-aware repartition

whereas for out-of-core executions the messages are checkpointed by storing in the message store.

### E.8 Scalability

The scalability is tested by reducing the memory per worker and evaluating the execution time of various algorithms mentioned in the figure 18. We can see the MOCGraph and MOCGraph-asyn consistently outperform Giraph and even Giraph with combiner narrowly. This is because of Giraphs inefficient out-of-core executions which adds more overhead because of the disk I/O.

## *F. TUX2*

### F.1 The system

Tux2[6] is developed as a distributed graph processing system specialized for machine learning algorithms using graph computations as the authors believe that previous systems like GraphLab[7] lacked the flexibility and performance benefits offered by Tux2. The suggested system also outperforms existing systems like PowerGraph.

### F.2 Computation model

Tux2 uses a model called MEGA model which is a staged based model described as a model that provides more flexibility than the Gather-Apply-Scatter(GAS) model of computation used by PowerGraph A stage is defined as a computation using a user-defined function on a set of vertices and it's edges. The types of stages are Mini-batch, Exchange, GlobalSync and Apply(MEGA).

- Exchange iterates over all the edges of a set of vertices with computing a user defined function on all the edges. This function has set parameters to update the vertex, edge data.
- Apply iterates over a set of vertices to synchronize the master and the mirror vertices generated during vertex-cut partitioning. The apply function invoked by master vertex updates the global state of the masters followed by the mirror vertices.
- GlobalSync: Aggregates data across a set of vertices and synchronizes context across workers.
- Mini-batch comprises of small stages that are executed iteratively per mini-batch which is a set of vertices or edges.

### F.3 Implementation

For every partition created, it's assigned to a process which acts as a worker by iterating over all the vertices and passing messages along the edges of the vertices. It also acts as a master by synchronizing the data between the master vertex and mirror vertices created by the vertex-cut partitioning technique. Tux2 uses multi-thread parallelization with the server and worker role for the same thread for local computation over the mirror vertices and synchronizing with the master vertex of that same partition.

### F.4 Special Features

Heterogeneous Data Layout: Graph engines generally assume the homogeneity of vertices in a graph but some problems like machine learning problems might require the use of heterogeneous vertices like a bipartite graph. This heterogeneity is extended to the master and mirror vertex concept explained earlier. The vertices can also be stored separately to improve data locality. Stale Synchronous Parallel(SSP): Each iteration of TUX2 is executed in mini-batches where the process works on a batch of vertices and edges. SSP introduces

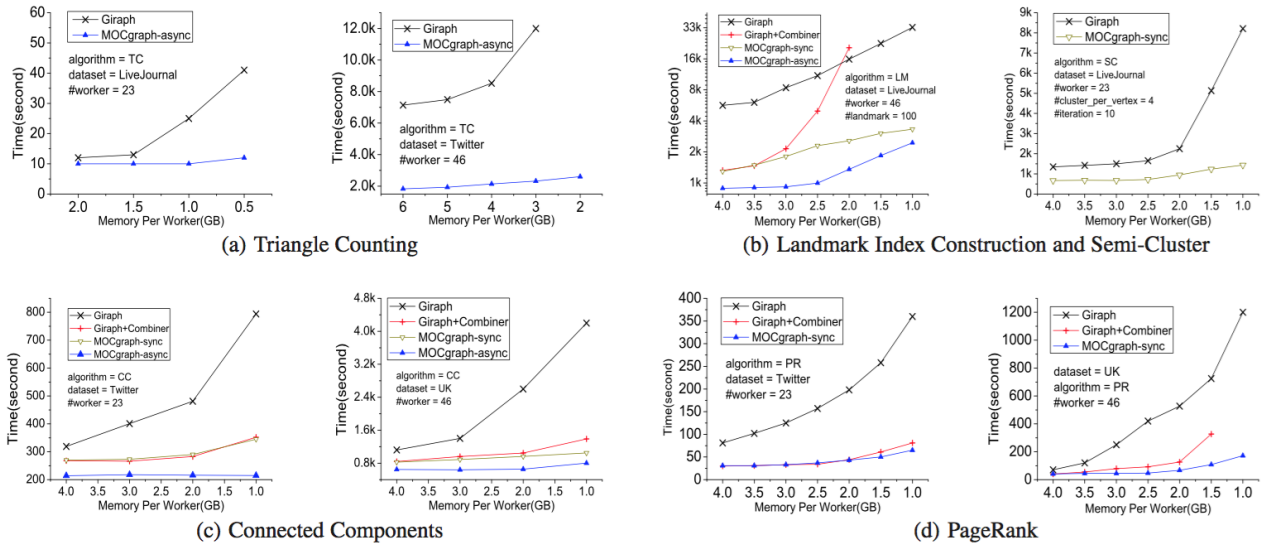



Fig. 18. Memory Usage and Speedups: Lines marked as Giraph represent tasks running with Giraph. MOCgraph and Giraph can run both in-memory and out-of-core. Giraph+Combiner stands for tasks running with Giraph and Combiner, which can run all in memory, or with message partitions all in memory but vertex partitions out of core.

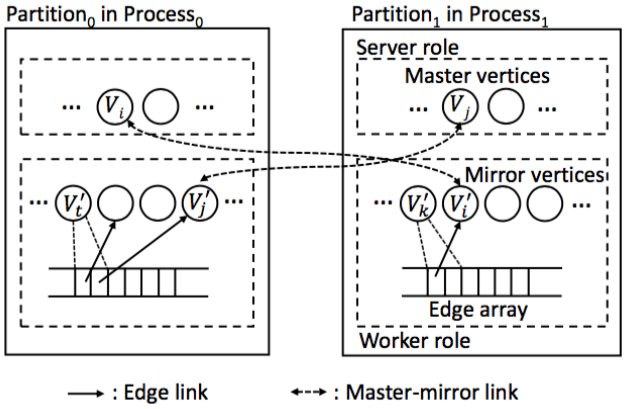



Fig. 19. Graph placement and execution roles in TUX2

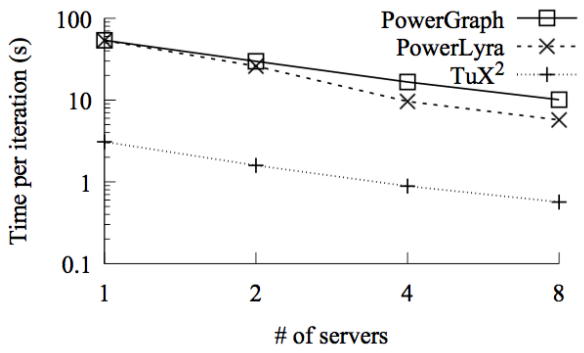



Fig. 20. Tux2 vs PowerGraph and PowerLyra

a bounded slack parameter which dictates the drift of the execution of a mini-batch. The worker can execute batch at t only if all the batches up to clock t-s-1have been updated.

## F.5 Partitioning

For partitioning, Tux2 uses the vertex-cut partitioning technique. In this technique, the vertex with higher degrees is partitioned with replicated vertices. One of the vertices acts like a master storing the global state and the mirrors store the local cached copy. The vertices and edges are stored in arrays for optimized traversal.

## F.6 Scalability

The system is compared to PowerGraph and it is an improvement PowerLyra though we focus on PowerGraph for this survey paper. Using the twitter dataset it is evident from figure 20 that, Tux2 outperforms PowerGraph. This is due to the Gather-Apply-Scatter model of PowerGraph which

requires two phases per iteration that introduces an overhead. Whereas the two stages exchange and apply required from the MEGA model for Tux2 take much less time.

### *G. Arabesque*

#### G.1 The system

Arabesque[8] is presented as a specialized distributed graph processing problems that can cater to graph mining algorithms that require an exploratory processing of sub-graphs for large graphs. It uses a filter-process process model to filter explored sub-graphs and process as required by the algorithm. The solution suggested is a scalable solution built for various graph mining problems.

#### G.2 Computation model

Arabesque introduces a new paradigm called "think like an embedding" (TLE) instead of Pregel's "think like a vertex"(TLV). An embedding represents a subgraph of a pattern to be mined. Graph mining algorithms cannot be formulated using the TLV paradigm as they involve the mining of the structure and the labels of a graph. It compares it's TLE model with think like a pattern(TLP) model used for a centralized approach but finds that TLP creates load unbalances between workers in a distributed environment.

The computation is done through a sequence of steps called as exploration steps. In each step, given an input of embeddings candidate embeddings are generated using either edge-based or vertex-based expansion which can be defined by the user. The candidates are filtered and processed using user-defined filter and process functions. It also provides the ability to aggregate multiple embeddings as defined by the user. The system implements guarantee of not exploring the extensions of candidates that are already filtered out.

#### G.3 Implementation

Arabesque is implemented over Giraph which uses a BSP(Bulk Synchronous Processing) model. The vertices are considered as the workers and the edges are the communication channel. Though the vertices and edges are not that of the input graph.

#### G.4 The Application Programming Interface

Arabesque defines functions like filter() and process() for filtering and processing the embeddings. Aggregation is done using a MapReduce-like model which provides map() and reduce() functions with the output provided with the readAggregate(). Special functions for aggregation of output are also provided with mapOutput() and reduceOutput(). The terminationFilter() terminates an embedding after it's processed and before it's added to the output of the current step.

#### G.5 Special Features

• Graph Exploration: Arabesque uses a coordination free exploration strategy to avoid redundant work. It does so by using an embedding canonicality check that checks the uniqueness of an embedding before the filter function to avoid duplicates.
• Storing the embeddings: Since algorithms generate a huge number of embeddings they need to be stored in an efficient way. They are stored as vertex ID or edge ID integers in a structure called Over-approximating Directed Acyclic Graphs which are similar to prefix trees but over-approximate the set of sequences to be stored. This method trades space complexity for computation complexity as the embeddings have to be recomputed.

#### G.6 Partitioning

All the embeddings in arabesque for each pattern are partitioned by doing round-robin on blocks of embedding. These blocks have a cost attached to them which are generated from the structure of ODAG. These elements of equal cost are used as blocks for partitioning. This is a recursive process wherein if a block's cost needs to be split, it's split from the second array or level of the ODAG.

## III. Graph Processing System for One Trillion Edges

A very important aspect of a distributed graph processing system is the data itself. Some graphs like the Facebook social graph is quite a unique one since it hosts about billions of people with undirected relationships of about trillion edges. Researchers at Facebook took on the challenge to do

analytics on this data using their existing limitations like data stored on HDFS for interoperation with their Hadoop Structure. Here are some findings from the paper One Trillion Edges[9] by Facebook

They chose Apache Giraph which is a Pregel like a system They tested this system for scalability while suggesting improvements to suit their needs. Some of the improvements included:

- Modifying Giraph to take the input of vertex and edges from different sources as giraph assumed the placement of edge data in the same source as the vertex data.
- Adding workers per machine and multithread support for fine grain parallelism as originally Giraph is executed as a single MapReduce job which parallelizes by increasing the mappers per job.
- Using Netty framework [netty.io] for large data aggregation as the ZooKeeper implementation of giraph was inefficient.

They also chose to make some extensions to the compute model of Pregel by implementing:

- Multiple Worker Phases like preSuperstep(), postSuperstep() etc to support efficient execution of certain algorithms frequently used on the facebook dataset like k-means algorithm
- Master computation for centralized computations while Pregel only supports parallel computations through compute().
- Superstep splitting for messages that must not be aggregated in which the fragment of the message is sent to the destination for partial computation

Some of the scalability tests conducted on the system

From Figure 21a we can see that Giraph performs similarly to the ideal scaling curve for 50 workers. From figure 22b it is evident that Giraph scales linearly close to ideal scaling for one 1 to 200B edges. As seen in figure 22 the system was also compared with hive for wighted pagerank and it results in 26x speed up for total CPU time and 120x speedup for total elapsed time for one iteration in both cases

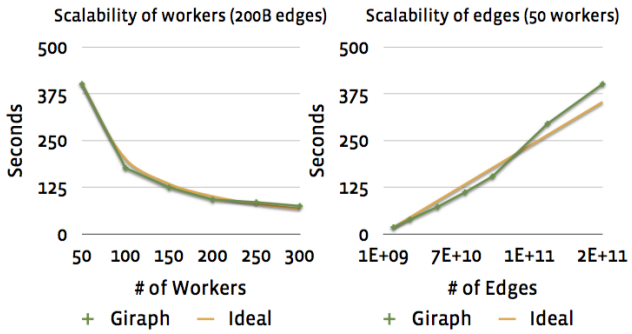



Fig. 21. Scaling PageRank with respect to the number of workers (a) and the number of edges (b)

| 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 |

Fig. 22. Weighted PageRank comparison between Hive and Giraph implementations for a single iteration.

## IV. Conclusions and Future Work

Large graph processing systems is a hot topic in the research community and there is active development on existing and new systems. This makes it harder to survey all the systems that are present. Hence my conclusion would be based only on these papers.

A common trend I found throughout the papers was the absence of consideration of the fault tolerance of the system for testing scalability. I agree with the argument made that for measuring the performance deltas enabling the fault tolerance system for systems that are less than likely to fail introduces a run-time overhead. Though there is a scope for experimental research on the scalability of these systems with fault tolerance.

A scope for improvement for these systems would be the fault tolerance system. More or less of these papers use the same checkpointing version. Other techniques could be looked at for improving the overhead of disk I/O required by the checkpointing style of fault tolerance systems.

Another common trend I found is that most of these graph processing systems are built for static

graph analytics. There is a scope for research on time-series graphs as they represent the dynamic structure of the real-world graphs.

The comparisons made in the survey are theoretical and need to be backed by test results on common grounds. This topic could be revisited in a few years when the development of these systems plateau.

## REFERENCES


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

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

[3] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, New York, NY, USA, 2013, EuroSys '13, pp. 169–182, ACM.

[4] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2012, OSDI'12, pp. 17–30, USENIX Association.

[5] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 377–388, Dec. 2014.

[6] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou, "Tux: Distributed graph computation for machine learning," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017, pp. 669–682, USENIX Association.

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

[8] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, New York, NY, USA, 2015, SOSP '15, pp. 425–440, ACM.

[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015.