Experimental Analysis of Distributed Graph Systems

Khaled Ammar, M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
{khaled.ammar, tamer.ozsu}@uwaterloo.ca

ABSTRACT

This paper evaluates eight parallel graph processing systems: Hadoop, HaLoop, Vertica, Giraph, GraphLab (PowerGraph), Blogel, Flink Gelly, and GraphX (SPARK) over four very large datasets (Twitter, World Road Network, UK 200705, and ClueWeb) using four workloads (PageRank, WCC, SSSP and K-hop). The main objective is to perform an independent scale-out study by experimentally analyzing the performance, usability, and scalability (using up to 128 machines) of these systems. In addition to performance results, we discuss our experiences in using these systems and suggest some system tuning heuristics that lead to better performance.

PVLDB Reference Format:
Khaled Ammar, M. Tamer Özsu. Experimental Analysis of Distributed Graph Systems. *PVLDB*, 11(10): 1151-1164, 2018.
DOI: https://doi.org/10.14778/3231751.3231764

1. INTRODUCTION

In the last decade, a number of graph processing systems have been developed. These are typically divided into graph analytics systems (e.g. Giraph) and graph database systems (e.g. Neo4j) based on the workloads they process. In this paper we focus on graph analytics systems. Many of these use parallel processing to scale-out to a high number of computing nodes to accommodate very large graphs and high computation costs. Single machine solutions have also been proposed, but our focus in this paper is on scale-out systems. Although each of the proposals are accompanied by a performance study, objective, independent, and comprehensive evaluation of the proposed systems is not widely available. This paper reports the results of our extensive and systematic performance evaluation of eight graph analytics systems over four real datasets with different characteristics. The choice of these eight systems is based on a classification discussed in Section 2 and include:

Proceedings of the VLDB Endowment, Vol. 11, No. 10

DOI: https://doi.org/10.14778/3231751.3231764

- Vertex-centric:
  - Synchronous: Giraph [3], GraphLab [24], Blogel-Vertex (Blogel-V) [47]
  - Asynchronous: GraphLab [24]
- Block-centric: Blogel-Block (Blogel-B) [47]
- MapReduce: Hadoop [4]
- MapReduce extensions: HaLoop [17], Spark/GraphX [25]
- Relational: Vertica [30]
- Stream: Flink Gelly [2]

The experiments generated more than 20 GB of log files that were used for analysis. The novel aspects of this study are the following:

- We study a comprehensive set of systems that cover most computation models (§ 2). Previous studies (e.g., [14, 27, 37]) consider only vertex-centric systems.
- Compared to previous studies, we use a wider set of real datasets: web graphs (UK200705, ClueWeb), social networks (Twitter), and road networks (world road network). Although, web graphs and social networks share some common properties, such as power-law distribution [18] and shrinking diameter [34], road networks are different, for example with their very large diameters (48K in our dataset).
- We suggest several system tuning heuristics and a number of enhancements to existing systems for improved performance and usability.
- This is the first study that considers the COST metric for parallelization (§ 5.13).
- We develop a visualization tool[1] that processes different system log files, extracts interesting information, and displays several types of figures for comparisons. Independent of the performance results we report, this tool is itself useful for experimental evaluation.

The major findings of our study are the following:

- Blogel is the overall winner. The execution time of Blogel-B is shortest, but Blogel-V is faster when we consider the end-to-end processing including data loading and partitioning (§ 5.1).
- Existing graph processing systems are inefficient over graphs with large diameters, such as the road network (§ 5.3, 5.6, 5.8).
- GraphLab performance is sensitive to cluster size (Section 5.4).
- Giraph has a similar performance to GraphLab when both systems use random partitioning (§ 5.5).

[1] `https://tinyurl.com/ya5plcr3`

- GraphX is not suitable for graph workloads or datasets that require large number of iterations (§ 5.6).
- General data parallelization frameworks such as Hadoop and Spark have additional computation overhead (§ 5.7) that carry over to their graph systems (HaLoop, GraphX). However, they could be useful when processing large graphs over shared resources or when resources are limited (§ 5.10).
- Vertica is significantly slower than native graph processing systems. Although, its memory footprint is small, its I/O wait time and network communication is significantly high (§ 5.11).

It can be claimed that some of the performance differences could be due to the choice of the implementation language (Java or C++). It is common knowledge that C++ has better overall performance than Java for multiple reasons. Although we are not aware of a system that has both C++ and Java implementations to conduct a more controlled experiment, the fact that GraphLab and Giraph have similar performance when they use the same partitioning algorithm (random) suggests that implementation language may not be a main factor. Nevertheless, this point requires further study.

We introduce the systems under investigation in Section 2 and the workloads in Section 3. Section 4 explains the experimental setup while Section 5 presents our quantitative results. Section 6 compares our work with related works, and we conclude in Section 7.

2. SYSTEMS

We evaluate seven graph processing systems in this study. All systems, except Vertica, read datasets from and write results to a distributed file system such as HDFS (Hadoop Distributed File System). Vertica 7.2.1 is a relational database system and it uses its own distributed storage. It is included in this study because of a recent claim that it performs comparable to native graph systems [30]. Hadoop 1.0.4, HaLoop, and Giraph 1.1.0 are developed in Java and utilize the Hadoop MapReduce framework to execute all workloads. Flink Gelly 1.4.2 has Scala and Java APIs, both use existing libraries to read and write data from HDFS. Blogel and GraphLab 2.2 are developed in C++; they use libraries to read and write from HDFS. Finally, GraphX is developed using Scala and run on top of Apache Spark. We comment on the differences in programming languages in Section 7.

We categorize parallel graph systems based on their computational model, which explains our choice of the systems under study. A summary of the features of these systems is given in Table 1. We also describe the special configurations used in this study.

2.1 Vertex-Centric BSP

Vertex-centric systems are also known as "think-as-a-vertex". Each vertex computes its new state based on its current state and the messages it receives from its neighbors. Each vertex then sends its new state to its neighbors using message passing. Synchronous versions follow the Bulk Synchronous Parallel (BSP) model that performs parallel computations in iterative steps, and synchronizes among machines at the end of each step. This means that messages sent in one iteration are not accessible by recipients in the same iteration; a recipient vertex receives its messages in the subsequent iteration. The computation stops when all vertices converge to a fixpoint or after a predefined number of iterations. This has been the most popular approach, and we study three systems in this category: Giraph, GraphLab, and Blogel-V.

2.1.1 Giraph

Giraph [3] is the open source implementation of Pregel [38], the prototypical vertex-centric BSP system. Giraph is implemented as a map-only application on Hadoop. It requires all data to be loaded in memory before starting the execution. Graph data is partitioned randomly using edge-cut approach, and each vertex is assigned to a partition.

Giraph API has one function, called `compute`. At every iteration, the `compute` function may update the vertex state based on its own data or based on its neighbors' data. The `compute` function may also send messages to other vertices.

In our experiments, we use Giraph 1.1.0, configure four mappers in each machine, and allow Hadoop to utilize 30GB memory in each machine.

2.1.2 GraphLab / PowerGraph

GraphLab [24] is a distributed graph processing system that is written in C++ and uses MPI for communication. Similar to Giraph, it keeps the graph in memory. However, it does not depend on Hadoop and it introduces several modifications to the standard BSP model:

- Instead of using one `compute` function, it has three functions: `Gather`, `Apply`, and `Scatter` (GAS). The GAS model allows each vertex to *gather* data from its neighbors, *apply* the compute function on itself, and then *scatter* relevant information to some neighbors if necessary.
- It uses vertex-cut (i.e., edge-disjoint) partitioning instead of edge-cut. This replicates vertices and helps better distribute the work of vertices with very large degrees. These vertices exist in social network and web graphs, because they follow the power-law distribution [35]. Replication factor of a vertex refers to the number of machines on which that vertex is replicated.

GraphLab 2.2 automatically uses all available cores and memory in the machine. It has multiple partitioning approaches that we study in further detail at Section 4.4.1.

2.1.3 Blogel-V

Blogel [47] adopts both vertex-centric and block-centric models (discussed in the next section). Blogel is implemented in C++ and uses MPI for communication between nodes. Blogel-V follows the standard BSP model. Its API has a `compute` function similar to Giraph.

2.2 Vertex-Centric Asynchronous

GraphLab [24] has an asynchronous mode where vertices can have access to the most recent data at other vertices within the same iteration. This avoids the overhead of waiting for all vertices to finish an iteration before starting a new one. Synchronization is achieved by distributed locking. Both versions of GraphLab use the same configurations.

2.3 Block-Centric BSP

This category is also known as graph-centric. The main idea is to partition the graph into blocks of vertices, and

Table 1: Graph processing systems

System	Memory /Disk	Architecture	Computing paradigm	Declarative Language	Partitioning	Synchronization	Fault Tolerance
Hadoop [20, 4]	Disk	Parallel	BSP	×	Random	Synchronous	re-execution
HaLoop [17]	Disk	Parallel	BSP-extension	×	Random	Synchronous	re-execution
Pregel/Giraph/GPS [38, 3, 43]	Memory	Parallel	Vertex-Centric	×	Random	Synchronous	global checkpoint
GraphLab [36]	Memory	Parallel	Vertex-Centric	×	Random Vertex-cut	(A)synchronous	global checkpoint
Spark/GraphX [48, 25]	Memory/Disk	Parallel	BSP-extension	×	Random Vertex-cut	Synchronous	global checkpoint
Giraph++ [45]	Memory	Parallel	Block-Centric	×	METIS	(A)synchronous	global checkpoint
Blogel [47]	Memory	Parallel	Block-Centric	×	Voronoi 2D	Synchronous	global checkpoint
Vertica [30]	Disk	Parallel	Relational	✓ (SQL)	Random	Synchronous	N/A

run a serial algorithm within a block while synchronizing blocks on separate machines using BSP. The objective is to reduce the number of iterations, which leads to reducing the synchronization overhead. The number of blocks is expected to be significantly less than the number of vertices in a large graph, hence the performance gain from decreasing network communication. There are two prominent block-centric systems: Giraph++ [45] and Blogel [47]. Our study investigates Blogel, because Giraph++ is built on an earlier version of Giraph that does not implement the more recent optimizations proposed for Giraph [19, 43].

Blogel-B [47] has a `compute` function for blocks, which typically includes a serial graph algorithm that runs within the block. Blogel-B partitions the dataset into multiple connected components using a partitioning algorithm based on Graph Voronoi Diagram (GVD) [22] partitioning. Additional partitioning techniques based on vertex properties in real graphs, such as 2-D coordinates (for road-network) or URL prefix (for web graph) have also been discussed, but we do not use these dataset-specific techniques in this study. We use the default parameters for Blogel-B's GVD partitioning [47].

2.4 MapReduce

MapReduce [20] is a distributed BSP data processing framework whose goal is to simplify parallel processing by offering two simple interfaces: `map` and `reduce`. It achieves data-parallel computation by partitioning the data randomly to machines and executing the map and reduce functions on these partitions in parallel. Hadoop [4] is the most common open source implementation of MapReduce. It has been recognized that Hadoop is not suitable for graph algorithms that are iterative, due to excessive I/O with HDFS and data shuffling at every iteration [17, 31, 36, 38, 48]. We nevertheless include Hadoop in this study, because there are cases where memory requirements will not allow other systems to run, and Hadoop is the only feasible alternative. Hadoop 1.0.4 is configured to use four mappers and two reducers in each machine. It is also granted 30GB on each machine.

2.5 MapReduce Optimized

Modified MapReduce systems, such as HaLoop [17] and Spark [48], address the shortcomings of MapReduce systems (in iterative workloads as graph processing) by caching reusable data between `map` and `reduce` steps and between iterations to avoid unnecessary scans of invariant data, and unnecessary data shuffling between machines.

2.5.1 HaLoop

The main objective of HaLoop optimizations is to reduce data shuffling and reduce network usage after the first iteration. HaLoop proposes several modifications to enhance Hadoop's performance on iterative workloads:

- A new programming model suitable for iterative programs, e.g., enabling loop control on the master node.
- Task scheduler in the master node is changed to be loop-aware. It keeps information about the location of sharded data, and tries to co-schedule tasks with data. This helps to decrease network communication.
- Slave nodes include a module for caching and indexing loop-invariant data that could be used in all iterations. The task tracker is modified to manage these modules.
- New support is introduced for fixpoint evaluation to optimize checking for convergence. The result of the last iteration is always locally cached to be used in the comparison instead of retrieving it again from HDFS.

HaLoop configuration is very similar to Hadoop's: four mappers, two reducers, and 30GB memory. However, in our environment HaLoop suffered from multiple errors because it keeps many files open. Therefore, we had to change the operating system's `nofile` limits.

2.5.2 Spark/GraphX

Similar to HaLoop, Spark caches dataset partitions for future use, but in memory instead of on local disk. The main feature of Spark is its fault tolerant in-memory abstraction, called resilient distributed datasets (RDD). GraphX [25] is a graph library that extends Spark abstractions to implement graph operations. It uses vertex-cut partitioning (similar to GraphLab). Every iteration consists of multiple Spark jobs. A developer can decide what data portions should be cached for future use. However, cached data cannot change because they are used as RDDs, which are immutable.

We run GraphX using the Spark 1.5.1 standalone mode to eliminate any overhead or performance gain from Yarn, Mesos, or any other systems that facilitate resource sharing. GraphX has many configuration parameters. We configured its workers and reducers so that they can use all available memory in each machine. By default, Spark uses all available cores.

2.6 Relational

These systems [23, 30] use a relational database as a back-end storage and query engine for graph computations. A graph can be represented as an edge and a vertex table. Transferring information to neighbors is equivalent to joining these tables, and then updating the answer column in

the vertex table. Each graph workload can be translated to a SQL query and executed on the tables.

Repeated joins over large vertex and edge tables is inefficient, and several optimizations have been proposed for Vertica [30]:

- Instead of updating multiple values in the vertex table (which also means random access to data on disk), it may be more efficient to create a new table instead, and replace the old table with the new one (sequential disk access) if the number of updates is large. If the number is very small, updating the table might be more efficient, but, it is not straightforward to estimate the number of updates beforehand.
- In traversal workloads, such as Single Source Shortest Path (SSSP), it is common to only process a few vertices at every iteration. Instead of starting from the complete vertex table and filter these vertices, it is more efficient to keep active vertices in a temporary table and use it during the join operation.

Several changes were made to the cluster to satisfy all Vertica OS-level requirements. Before we start our experiments, instead of loading the data to HDFS, we load the data as a table of edges to Vertica 7.2.1.

2.7 Stream Systems

There are a few systems in the literature, such as Timely and Differential Dataflow [5, 40], Naiad [41], Flink [1] and TensorFlow [7] that model computations as a series of operations. In these systems, a developer needs to define operators, and then connect them to describe the data flow among operators. Data are streamed and processed through these operators. These are general data processing systems that sometimes support iteration in their data flow, therefore they can process graph algorithms.

In our study, we consider Flink Gelly 1.4.2 [2] as a representative for this category. Gelly is the graph processing API built on top of Flink. It has two approaches: stream and batch. The stream reads data from an input stream and it pushes the received edges (or vertices) to the data flow as they arrive. The batch approach reads data from containers then process the whole dataset in the data flow operations described by the application developer. To be consistent with other systems, we use the batch approach in our experiments, which allow us to isolate the time required to read and prepare the graph from execution time.

3. WORKLOADS

In this study we consider four graph workloads: `PageRank`, `WCC` (weakly connected component), `SSSP` (single source shortest path) and `K-hop`. These are chosen because: (1) they are prominent in graph system studies, (2) they have different characteristics – some (e.g, PageRank and WCC) are analytic workloads that involve iterative computation over all the vertices in the graph while others (e.g., SSSP and K-hop) are known as online workloads that operate on certain parts of the graph, and (3) there are implementations of each of them over the evaluated systems. Although every system offers its own implementation of these workloads, we made small changes to ensure uniformity of the algorithm and implementation across the systems.

3.1 PageRank

PageRank has been the most popular workload for evaluating graph systems for iterative algorithms. In a nutshell, PageRank assigns an importance weight (rank) to each vertex based on its influence in the graph. This is achieved by an initial assignment of rank followed by iterative computation until a fixpoint of weights is found.

The iterative algorithm models a random walk agent that moves through the graph, such that when it is at vertex u, it may choose an outgoing edge from u with probability $1-\delta$ or jump to a random vertex with probability δ. Therefore, a vertex PageRank value $pr(v)$, in a graph $G(V, E)$, follows the following equation:

$$pr(v) = \delta + (1-\delta) \times \Sigma \frac{pr(u)}{outDegree(u)} \mid (u, v) \in E$$

where $outDegree(u)$ is the number of directed edges from vertex u to other vertices. Many implementations assume that $\delta = 0.15$ and start with an initial rank of 1 for each vertex. Iteratively vertex ranks are computed using this formula until the rank of each vertex converges to a value.

The standard PageRank implementation follows synchronous computation, such that all vertices are involved in computation until convergence. In our experiments, convergence means the maximum change in any vertex rank is less than the initial value. This definition is more suitable than stopping after a fixed number of iterations, because it takes into consideration the properties of each dataset. Asynchronous or synchronous implementations that allow converged vertices to opt-out early from computation result in approximate answers. We will discuss this error and accuracy in the detailed experiments in Section 5.3.

3.1.1 Self-edges issue in GraphLab

GraphLab could not compute the correct PageRank values, because it does not support self-edges, which exists in the real graphs we use in this study. Changing the system to allow self-edges using flags or other potential implementations is possible, but is outside the scope of this study, as it requires significant changes to GraphLab code. This issue was communicated to GraphLab developers.

3.1.2 Block-centric implementation

The block-centric implementation of PageRank in Blogel-B [47] also did not generate accurate results. The proposed algorithm has two steps: (1) Compute the initial PageRank value using block-computation and local PageRank; (2) Compute PageRank using vertex-computation. The first step constructs a graph of blocks, such that the weight of an edge between two blocks represents the number of graph edges between these two blocks. In the first iteration, each block runs local PageRank over local edges in the block, then it runs a vertex centric PageRank on the graph of blocks. This step continues until convergence.

The second step starts by initializing the PageRank of every vertex in a block b as $(pr(v) \times pr(b))$ (where $pr(v)$ is the initial vertex pagerank and $pr(b)$ is the block pagerank after the first step converged)[2]. The second step then runs until convergence. We also considered a version of this algorithm where each step runs for the number of iterations computed

[2]There are other possible initialization functions that may include block and vertex pagerank values or degrees.

earlier to guarantee conversion. However, it is not clear how many iterations each step would need to guarantee the same results as the other computation models. Therefore, in our experiments we follow the version of block-computation PageRank proposed in the original Blogel paper [47].

3.2 WCC

The objective of a weakly connected component (WCC) algorithm is to find all subgraphs that are internally reachable regardless of the edge direction. HashMin [31] is the straightforward distributed implementation of WCC. It labels each vertex in the input graph by the minimum vertex id reachable from the vertex, regardless of the edge directions. It starts out by considering each vertex to be in one component (i.e., each vertex id is its component id). Each vertex propagates its component id to its neighbors. The process terminates when a fixpoint is reached, i.e., no vertex changes its component id. This algorithm requires $O(d)$ iterations, where d is the graph diameter.

HashMin algorithm has been implemented in all of the systems under consideration. However, we found that the result generated by some of these implementations are not correct, because of failure to process both directions of an edge.We corrected Blogel [47] and Giraph [27] implementations by adding an extra task to the first iteration: creating reverse edges, when necessary. Since GraphLab [36] allows vertices to access both ends of an edge regardless to the edge direction, it does not suffer from this overhead. However, as we will show later, memory requirements of GraphLab is typically larger than Giraph and Blogel for this very reason.

3.3 SSSP and K-hop

The Single Source Shortest Path (`SSSP`) query is a graph traversal workload. It finds the shortest path from a given source vertex to every other vertex in the graph. Assuming the source node is u, a typical algorithm starts by initializing distance $dist(u, v) = \infty$ for any vertex $v \neq u$. Iteratively, using a breadth first search, the algorithm explores new vertices: at iteration i new vertices that are i hops away from the source vertex are considered. The number of iterations is $O(d)$. The algorithm stops when all reachable vertices are visited and $dist(u, v)$ is computed for all v.

The `K-hop` query is very similar to `SSSP`, but it is bounded by K hops. This query is relevant in evaluating graph systems because it is a traversal query, but its complexity (#iteration) does not depend on the graph diameter. We fix K to a small number, 3, to reduce the impact of graph diameter on the performance, and to represent multiple use cases, such as the friends-of-friends query and its potential indexes.

In the results reported in this paper, to be consistent with other studies in the literature, we only use a random start vertex, which is chosen for each graph dataset, and used consistently in all experiments.

4. EXPERIMENT DESIGN

The experimental setting of this study is summarized in Table 2.

Table 2: A summary of experiments dimensions

Dimension	potential values
Systems	Giraph, Blogel, Hadoop, HaLoop, GraphX, GraphLab, Vertica, Flink Gelly
Workloads	WCC, PageRank, SSSP, K−hop
Datasets	Twitter, UK, ClueWeb, WRN
Cluster Size	16, 32, 64, 128
Instance type	r3.xlarge

4.1 Infrastructure

All experiments are run on Amazon EC2 AWS r3.xlarge machines, each of which has 4 cores and 30.5 GB memory, Intel Xeon E5-2670 v2 (Ivy Bridge) processors, and SSD disks. They are optimized for memory-intensive applications and recommended for in-memory analytics. We test scalability over 16, 32, 64, and 128 machines (one master).

4.2 Evaluated Metrics

We measure two things: resource utilization and system performance. Each system is evaluated in isolation, with no resource sharing across systems or experiments. For resource utilization, we record CPU utilization for each process type (user, system, I/O, idle) and memory usage every second. We also record the total network traffic by measuring network card usage before and after workload execution. We report the following system performance metrics: (a) data-loading time, (b) result-saving time, (c) execution time, and (d) total response time (latency).

Data-loading time includes reading data from HDFS and graph partitioning, when necessary. Ideally, total response time should equal load+execute+save. However, we report it separately, because it represents the end-to-end processing time, and occasionally includes some overhead that might not be explicitly reported by some systems, such as the time of repartitioning, networking, and synchronization.

4.3 Datasets

Table 3 summarizes the characteristics of the datasets used in this study: Twitter[3], World road network (WRN)[4], UK200705[5], and ClueWeb[6]. These are among the largest publicly available graph datasets, and they cover a wide range of graph characteristics.

Table 3: Real Graph Datasets

Dataset	$\|E\|$	Avg./Max. Degree	Diameter
Twitter	1.46 B	35 / 2.9M	5.29
WRN	717 M	1.05 / 9	48 K
UK200705	3.7 B	35.3 / 975K	22.78
ClueWeb	42.5 B	43.5 / 75M	15.7

We partition all input graph datasets into chunks of similar sizes, and then load them to HDFS for all systems because this makes data loading more efficient for Blogel and GraphLab and has no impact on other systems. Note that the HDFS C++ library used in Blogel and GraphLab create a thread for each partition in the dataset. If there is only one data file, then only one thread executing on the master will read the entire graph, which significantly delays the loading process.

In order to ensure a fair comparison, we prepared a dataset format that matches the typical requirement of each system. Specifically, we use three graph formats: adj, adj-long, and edge. The adjacency (adj) format is a typical adjacency list; each line includes a vertex id and then the ids of all vertices it is connected with. If a vertex does not have an out-edge,

[3]http://law.di.unimi.it/webdata/twitter-2010
[4]http://www.dis.uniroma1.it/challenge9/download.shtml
[5]http://law.di.unimi.it/webdata/uk-2007-05/
[6]http://law.di.unimi.it/webdata/clueweb12/

it does not need to have a line for itself. The adjacency-long (adj-long) format requires each vertex to have a line in the dataset input file. Moreover, the first value after the vertex id is the number of neighbor vertices. Edge format has a line for each edge in the graph.

Hadoop, HaLoop, Giraph, and Graphlab use the adj format, which is the most concise format and significantly reduces the input size. Blogel needs to use the adj-long format for it to be able to create vertices that only have in-edges [6]. This limitation could be fixed by adding an extra superstep in all computations to create missing vertices. However, this solution adds an overhead on the computation performance and was not preferred by Blogel developers when they were contacted. Finally, GraphX and Flink Gelly use edge-list.

4.4 Configuration

We report below the experiments we performed to better understand and fix configurations of some of the systems.

4.4.1 Partitioning in GraphLab

GraphLab has two main partitioning options: "Random" and "Auto". Random assigns edges to partitions using a hash function. Auto chooses between three partitioning algorithms (PDS, Grid and Oblivious), in order, based on the number of machines in the cluster.

Typically, these partitioning methods try to decrease the replication factor (see Section 2.1.2) by minimizing the number of machines at which each vertex is stored. This would decrease the network communication between machines. The details of these algorithms are as follows:

- Grid assumes the cluster is a rectangle of height and width equal to X and Y and requires the number of machines $M = X \times Y$, such that $|X - Y| \leq 2$ for any positive numbers X and Y. Using a hashing function to map each vertex to a machine m, the vertex could be replicated to any machine in the same column or same row that include m. An edge between two vertices can be assigned to any partition that can include a replica for both vertices.
- PDS creates a perfect difference set [46] S of size $p+1$ if the number of machines $M = p^2 + p + 1$, where p is a positive prime number. Then, for each value i in S, it creates another set S_i by adding this value to $[0, M]$ mod M. Finally, using a hash function to map a vertex to a machine m, the vertex could be replicated to any machine in S_i such that $m \in S_i$. Again, an edge could be placed in a machine that can include both of its vertices.
- Oblivious is a greedy heuristics for edge placement to reduce the partitioning factor. Given an edge (u, v), such that S_u is the set of machines that include replicas of u and S_v is the set of machines that include replicas of v, the edge will be placed in the least loaded machine in S_e, such that:
  - if $S_u \cap S_v \neq \phi$ then $S_e = S_u \cap S_v$;
  - if $S_u = \phi$ and $S_v \neq \phi$ then $S_e = S_u$;
  - if $S_u = S_v = \phi$ then S_e is the set of all machines;
  - if $S_u \cap S_v = \phi$ then $S_e = S_u \cup S_v$.

Occasionally these algorithms do not reduce the replication factor when compared with random. For example, the difference between replication factor (Table 4) in random and auto for the Twitter dataset is not significant (less than 2×). Twitter dataset has some differences when compared with the the UK0705 dataset. For example, the maximum out-degree in the Twitter dataset is 3× the maximum out-degree in the UK0705 dataset, despite the fact that Twitter dataset is 3× smaller than UK0705. Unlike UK9795, the Twitter dataset has only one large component. Auto partitioning could not help GraphLab to enhance the efficiency of Twitter graph processing. At the same time, the auto replication factor for the UK0705 dataset is 5× less than random replication factor in the 32 cluster, though.

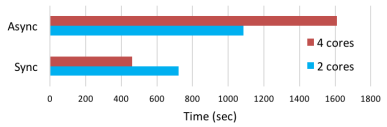


Figure 1: In a 16-machines cluster, GraphLab synchronized mode benefits from using all 4 cores in computation, while asynchronous computation does not because vertices do computation and communication on the same time.

4.4.2 CPU utilization in GraphLab

GraphLab, by default, uses all the cores in every machine. It reserves two cores for networking and overhead operations and uses the remaining cores for computations. Our experiments use GraphLab's default configuration. Nonetheless, we studied the value of this default configuration by changing the GraphLab code to use all available cores for computation (Figure 1). When we used all cores for computation, we obtained 40% improvement (with the synchronous computation) on a 16-machine cluster over 30 iterations of PageRank computation using the Twitter dataset. On the other hand, asynchronous computation requires multiple communications while some vertices are still in the computation phase. Due to the expensive repetitive context switching, asynchronous does not benefit, and sometimes even under-performs, when all cores are used for computation.

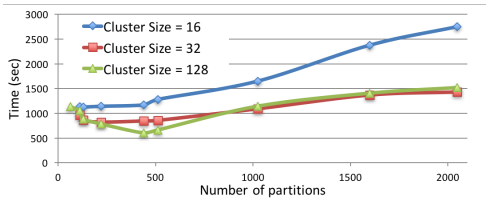
4.4.3 Number of Partitions in GraphX

By default, the number of partitions is equal to the number of blocks[7] in the input file. Based on our communication with Spark engineers, this default value may not be optimum. We found that the default number of partitions may lead to reasonable performance in the case of small datasets. However, since this number does not consider the amount of available cores, it may lead to under utilization of the cluster computing power. Figure 2 shows the influence of changing the number of partitions on two datasets (Twitter and UK0705) and three cluster sizes (32, 64, 128). The default number of partitions for the Twitter dataset is 440,

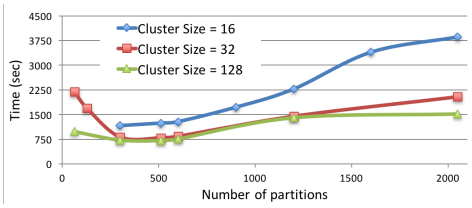
[7]The default block size in HDFS is 64 MB.

Table 4: The replication factor in GraphLab.

Dataset	Cluster Size	Random	Auto
Twitter	16	9.3	5.5
	32	13.3	9.8
	64	17.8	9.1
	128	22.5	15.2
WRN	16	NA	NA
	32	3.0	2.2
	64	3.0	3.0
	128	3.0	2.3
UK0705	16	5.7	NA
	32	15.8	3.6
	64	21.5	10.1
	128	27.1	4.5



(a) The default number of partitions in Twitter is 440.



(b) The default number of partitions in UK0705 is 1200.

Figure 2: Analysis of how number of partitions influence the performance of GraphX. The default number of partitions is not optimum.

which also achieves the best performance among all clusters. However, the default number of partitions for the UK0705 dataset is 1200, which is significantly larger than the number of cores[8]. Therefore, the performance of GraphX using the default number of partitions is significantly worse than other options. Number of partitions used in our experiments are summarized in Table 5.

Table 5: Number of partitions GraphX in different cluster sizes.

Dataset	#blocks	Cluster Size			
		16	32	64	128
Twitter	440	128	256	440	440
WRN	240	128	240	240	240
UK200705	1200	128	256	512	1024

5. RESULTS & ANALYSIS

We summarize experimental parameters in Table 2. The main experiment compares the performance of all systems with respect to all workloads, cluster sizes and datasets.

In this paper we only depict detailed performance results for PageRank on all datasets and cluster sizes (Figure 6) and results for Twitter dataset on all workloads and cluster sizes (Figure 5). We have similar results (and more) for other workloads, but space limitations do not allow us to include them. Instead, for other workloads, we summarize the results. The full set of experimental results will be reported in the longer version of the paper [9].

For all datasets except ClueWeb, we evaluate each system using all workloads across all cluster sizes; ClueWeb only fits in a cluster of 128 machines and those results are reported separately in Table 7. Empty entries in the result tables indicate that the execution did not successfully complete. There are multiple possible errors: timeout when an execution fails to complete in 24 hours (TO), out-of-memory at any machine in the cluster (OOM), MPI error which only happens with Blogel-B (MPI), and shuffle error which only happens with HaLoop (SHFL). The following abbreviations are used for system names: `BV` and `BB` (Blogel -V and -B), `G` (Giraph), `S` (Spark/GraphX), `V` (Vertica), `HD` (Hadoop), `HL` (HaLoop), `GL` (GraphLab), and `FG` (Flink Gelly). GraphLab experiments have six different versions identified by three symbols: (`A/S`) for asynchronous or synchronous computation, (`A/R`) for auto or random partitioning, and (`T/I`) for tolerance or iteration stopping criteria (discussed in PageRank workload in Section 3.1). For example, `GL-A-R-I` means GraphLab using asynchronous computation, random partitioning, and iteration stopping criteria.

5.1 Blogel: The Overall Winner

Vertex-centric Blogel (`BV`) has the best end-to-end performance. It is the only system that could finish the SSSP/WCC computation across all cluster sizes over WRN dataset, due to its large diameter. Moreover, it is the only system that could finish computations over ClueWeb in the 128-machine cluster. It achieves this performance because it does not have an expensive infrastructure (such as Hadoop or Spark), uses efficient C++ libraries, utilizes all CPU cores, and has a small memory footprint.

On the other hand, `BB` has the shortest execution time for queries that rely on reachability, such as WCC, SSSP, and K-hop for two reasons: (1) these queries benefit from Voronoi partitioning; and (2) block centric computation minimizes network overhead because it runs a serial algorithm with in each block before it communicates with other blocks. PageRank workload suffers from handling an awkward algorithm as discussed in Section 3.1. The purpose of running PageRank internally in each block is to start the global algorithm (considering all vertices in the graph) with a better initialization than a straightforward initialization of equal PageRank value for each vertex. However, it turns out that the algorithm used for this purpose does not generate good initial values, which hurt the overall performance. This causes the block-centric version to take more iterations and more execution time after running the local PageRank. This result matches the original Blogel results and was discussed with Blogel developers.

The existing version of `BB`, reads data from HDFS, runs Voronoi partitioning, stores partitions in HDFS, reads these partitions again, and then runs a workload. We found that the end-to-end performance of block-centric computation has a significant overhead, due to the partitioning phase and the I/O overhead of writing and reading from HDFS. Removing the I/O overhead between partitioning and workload execution results in 50% reduction of the overall end-to-end response time (Figure 3).

Finally, `BB` could not process the WRN and ClueWeb graph because the GVD partitioning phase failed. After each sampling round, the Voronoi partitioner uses the master to aggregate block assignment data from each worker to count the size of each block. During this process the MPI interface uses an integer buffer to store data offset at a different location for each worker. Since WRN has a large number of vertices, the number of bytes received is larger than the maximum integer leading to an overflow and system crash. This issue can happen when MPI library is used to aggregate a large number of data items. The issue is known to the MPI community[9] and is a problem with MPI rather than Blogel.

5.2 Exact vs. Approximate PageRank

The exact PageRank computation assumes that all vertices participate in the computation during all iterations. The approximate version allows vertices whose PageRank values have not changed (or changed by an amount smaller than a threshold) to become inactive and not participate in

[8]There are 4 cores per machine. A 128-machines cluster has 512 cores.

[9]`https://tinyurl.com/ybb9uu84`

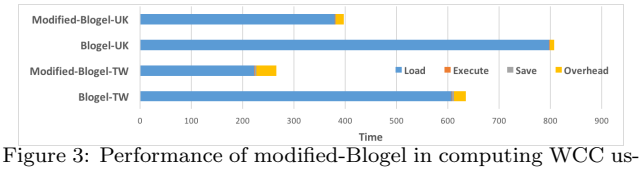


Figure 3: Performance of modified-Blogel in computing WCC using a cluster of 16 machines, without HDFS overhead between partitioning and workload execution. The load time, which represents data reading, partitioning, and shuffling before execution, has been significantly reduced.

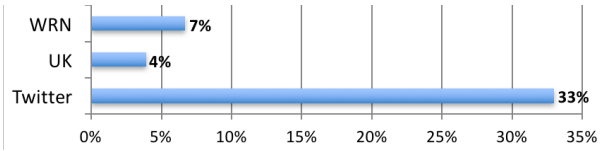


Figure 4: Percentage of updated verteces in case of approximate PageRank in comparison to an exact one.

further computations. GraphLab is the only system that facilitates the latter because active vertices can gather the ranks of their in-neighbors even if these neighbors are not active. However, this also increases its memory overhead. Therefore, GraphLab fails with OOM error when using random partitioning for the UK0705 dataset over 16 machines. GraphLab also fails to load the WRN dataset to memory in the 16 machine configuration, regardless of the partitioning algorithm. Approximate PageRank in GraphLab is the only implementation that outperforms Blogel exact implementation. Approximate answers are less expensive than exact ones because many vertices converge in the first few iterations. For example, Figure 4 shows the ratio between the number of vertex updates in the approximate and exact implementations for three different datasets.

5.3 Synchronous vs. Asynchronous

GraphLab is the only system that offers an asynchronous computation mode. However, it initiates thousands of threads per worker and allocates them to process vertices, leading to distributed lock contention as previously reported [27]. Therefore, PageRank asynchronous computation is typically slower than synchronous counterparts. Moreover, we found that asynchronous computation is only suitable for specific cluster sizes. It is not clear how to determine the right cluster size without trying multiple cluster configurations[10].

Unexpectedly, GraphLab asynchronous implementation failed with OOM while computing PageRank for the road network dataset using 128 machines. Further analysis indicates that this is due to distributed locking. Figure 7 shows memory usage for synchronous and asynchronous; each line represents a machine. Data load overhead (time and memory) is the same for both modes. While the synchronous mode finished within a reasonable time with reasonable memory usage, in the asynchronous mode, many vertices started to allocate more memory (without releasing them due to distributed locking), which slowed down the computation performance and caused the failure.

5.4 GraphLab partitioning optimization

GraphLab has two partitioning modes: "Random" and "Auto" (Section 4.4.1). Auto chooses between Grid, PDS an Oblivious. While Grid and PDF are faster than Oblivious, the latter does not have any requirements on the number of machines. Therefore, it gives priority to PDS or Grid, then uses Oblivious if neither are usable.

In our reports, load time includes reading and partitioning the dataset. Figure 6 shows that load time for GraphLab-auto is significantly smaller when using 16 or 64 machines (Grid) than the load time using 32 and 128 machines (Oblivious). This means that increasing the number of machines may lead to reduction of GraphLab performance.

5.5 Giraph vs. GraphLab

Giraph is very competitive with GraphLab when the latter uses random partitioning and runs a fixed number of iterations (similar to Giraph). Giraph was faster than GraphLab in the 16 and 32 clusters. However, as the cluster size grows, Giraph spends more time in requesting resources and releasing them because it uses MapReduce platform for resource allocation. Therefore, both systems perform similar in the 64 cluster, but GraphLab finally wins in the 128 cluster.

5.6 GraphX is not efficient when large number of iterations are required

GraphX/Spark is slower than all other systems in our study because it suffers from Spark overheads, such as data shuffling, long RDD lineages, and checkpointing. Previous publications [25] show that GraphX is efficient because it uses a special Spark prototype version that includes in-memory shuffling, which is not available in the latest release.

GraphX failed to compute WCC for the WRN dataset due to memory or timeout errors in all cluster sizes. It turns out that Spark fault-tolerance mechanism of maintaining RDD lineages is the culprit of memory errors. When the number of iterations grows, these lineages become long leading to high memory usage, and potential out of memory errors. Recent introduction of GraphFrames[11] should be a more efficient option for GraphX. We investigated GraphFrame implementations and found that many of its algorithms convert the input graph to GraphX format and then run GraphX algorithms. We also found that most algorithms have a default maximum limit on number of iterations to reduce the potential overhead of long lineage in RDDs. For example, SSSP has a limit of 10, otherwise it starts to checkpoint to avoid long lineages. Moreover, the default implementation of WCC requires checkpointing every two iterations. Checkpointing prevents lineage from being very long, but it leads to expensive I/O interactions with disk, which then leads to a timeout error. GraphFrames offers a version of the hash-min [31] algorithm originally used to compute WCC, called hash-to-min [32] that uses fewer iterations. We tested this implementation as well and found that it was competitive with hash-min in Blogel.

We noticed that Spark could not uniformly distribute partitions to workers. As depicted in Figure 8 some machines were assigned a large number of partitions. In a synchronous computation model, stragglers form in this case slowing down the workers who finish their tasks. In future Spark releases, this problem could be solved by implementing a more efficient load balancer.

In practice, the number of partitions should not be more than the number of blocks in the input file, because this forces Spark to read the same data block more than once.

[10]This was suggested by the GraphLab team in the official forum: `http://forum.turi.com/discussion/714/`

[11]`https://github.com/graphframes/graphframes`

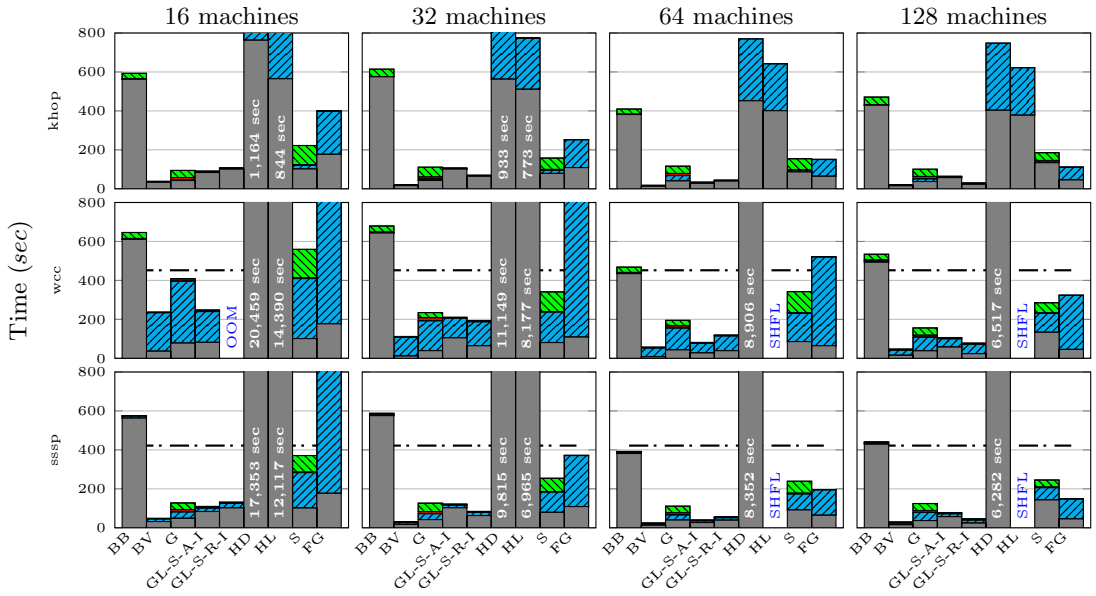
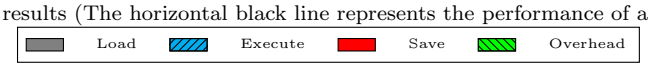


Figure 5: Twitter results (The horizontal black line represents the performance of a single thread implementation)



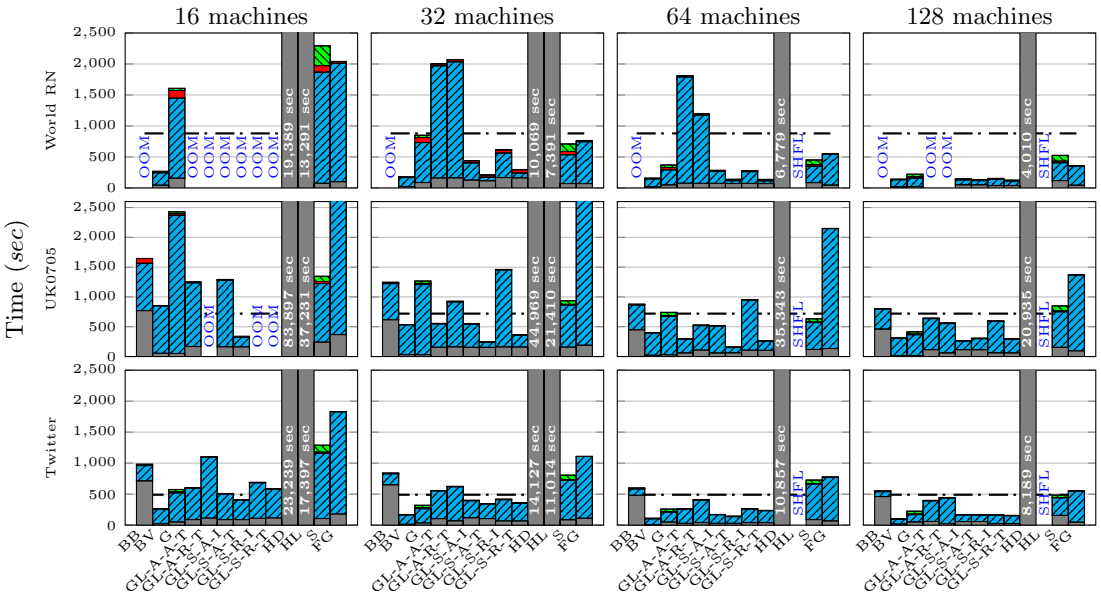
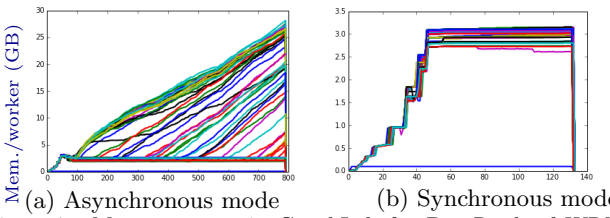


Figure 6: PageRank query results (The horizontal black line represents the performance of a single thread implementation)



(a) Asynchronous mode (b) Synchronous mode

Figure 7: Memory usage in GraphLab for PageRank of WRN using 128 machines. Each line represents the memory usage per worker per second; the X-axis is the time line for the computation in seconds. In the asynchronous mode, thousands of threads were created and allocated memory for vertices without releasing them quickly distributed locking which cause several machines to allocate large amount of memory, before the computation fails.

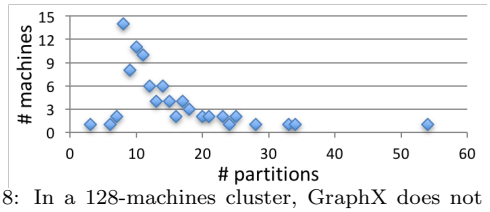


Figure 8: In a 128-machines cluster, GraphX does not balance number of partitions (1200) evenly to machines. A balanced distribution of workload would assign 1200/128 = 9.4 partitions to each machine. However, one machine has 54 partitions.

On the other hand, the number of partitions should not be less than the number of cores in the cluster, because this would lead to CPU under-utilization. In our experiments, we set the number of partitions equal to the number of blocks as long as it does not exceed twice the number of cores. This

allows Spark to handle stragglers by assigning them to another core. Unfortunately, this does not guarantee the best performance (Figure 2) because the workload assignment is not balanced (Figure 8).

5.7 System Overhead

The computation overhead is significantly larger for Giraph and GraphX. Giraph uses Hadoop and GraphX uses Spark for resource management, scheduling, and fault tolerance. The cost of starting a job and closing it are high in Hadoop and Spark. Blogel and GraphLab use MPI for communication between machines, and therefore, do not have the overhead of an underlying infrastructure. They interact with HDFS using C++ libraries but they do not depend on the job or task tracker in Hadoop.

Although the overhead time is small in Flink Gelly, we found that the system frequently fails after running a few jobs. It turns out that Flink does not reclaim all memory used by the system in between workload executions. This causes the system to eventually fail due to out-of-memory error. Thus, we had to restart Flink after each workload.

5.8 WCC Experiments

The WCC workload needs special handling because it requires the processing system to handle edges in both directions. Therefore, Gelly, Blogel and Giraph have the overhead of pre-computing the in-neighbors before executing the algorithm. Moreover, Blogel and Giraph cannot benefit from the message combiner in this workload, because messages in the first iteration should not be combined, since they are used to discover in-neighbors, not to find the smallest vertex id in the connected component. Furthermore, computing the WCC requires more memory than other workloads, because each vertex needs to recognize its in- and out-neighbors. Giraph failed to load the UK0705 in the 16 and 32 machine clusters and failed to load the WRN in the 16 machine cluster. Giraph could not finish computation of WCC for the WRN dataset in the 32 machine cluster, but succeeded to compute the WCC in almost 24 hours using the 64 machine cluster.

Blogel-V is the only system that could compute WCC on the WRN dataset using the 16 machine cluster due to its low memory requirements. GraphLab with random partitioning failed to load UK0705 dataset in the 16 machine cluster. On the other hand, GraphLab auto partitioning significantly reduces the execution time in comparison to random partitioning. However, the loading time for auto partitioning is high when Grid and PDS algorithms are not applicable (e.g. in the 32 and 128 machines cluster).

GraphX performance for WCC using the UK0705 dataset over 128 machines was significantly worse than all other systems and also worse than GraphX performance over 64 machines. This is an example of the influence of the number of partitions (Table 5) in Spark on the GraphX performance.

Gelly successfully finished the execution of WCC for Twitter and UK0705 in all clusters. However, it failed with timeout error to compute WCC for the WRN dataset in the 16, 32, and 64 cluster. Gelly finished WCC for WRN in slightly less than 24 hours using 128 machines.

5.9 ClueWeb experiments

ClueWeb represents a web graph, and has 42.5 billion edges and almost one billion vertices. The size of this dataset is 700GB (adjacency list) and 1.2 TB (edge list). Only the 128 cluster can hold it using its total 3 TB memory.

Table 6: The time, in seconds, used by each iteration for the WRN dataset. For SSSP and WCC to finish in 24 hours, the iteration time should be 2.4 and 1.8 respectively.

	Giraph		GraphX	
	SSSP	WCC	SSSP	WCC
16	6	OOM	120	420
32	3	3.2	17	30

Table 7: Blogel-V performance on the ClueWeb dataset using a cluster of 128 machines. Numbers represent number of seconds used for each processing phase.

Workload	Read	Execute	Save	Others
PageRank	132.5	139.7	10.5	15.3
WCC	134.1	152.5	11.5	10.6
SSSP	158.3	89.3	2.2	20.7
K-hop	161.6	0.03	0.2	16.4

GraphLab could not load the dataset in memory. Although we do not know ClueWeb's replication factor (since it could not be loaded), the other web graph, UK0705, has replication factors of 3.6 and 4.5 for the 64 and 128 machine clusters, respectively. If we assume a similar replication factor for ClueWeb, the data is larger than available memory. For the same reason, Gelly and Giraph could not finish their computation. We found that total physical memory used by Giraph to process UK0705 (originally 32 GB) using the 128 machine cluster is 1322 GB. Table 8 summarizes Giraph memory consumption for all datasets.

Blogel-V, was the only system that could perform any workload on ClueWeb in the 128 cluster (Table 7). This suggests that graph processing systems should be conscious of memory requirements, despite the common assumption that memory is available and cheap and most real graphs can fit in memory of current workstations. Most graph systems are optimized to decrease processing time at the expense of larger memory, but our results suggest caution. Finally, ClueWeb results also show that Hadoop MapReduce platform and distributed out-of-core systems may have a role; they are significantly slower than in-memory systems, but they can finish the task when memory is constrained or graph size is too large.

5.10 Hadoop and HaLoop Experiments

As noted earlier, it was expected that Hadoop and HaLoop would be slower than in-memory systems. As expected, HaLoop was faster than Hadoop due to its optimizations. However, our experiments do not show the 2× speedup that was reported in the HaLoop paper. Moreover, existing HaLoop implementation has some issues. The loop management mechanism introduced by HaLoop eliminates the usability of some basic Hadoop features, such as counters. It is not possible to use custom counters during iteration management to check for convergence. Moreover, HaLoop suffers from a bug that occasionally causes mapper output to be deleted before all reducers use them, in large cluster sizes[12].

CPU utilization is better in HaLoop than Hadoop, because in Hadoop CPUs spend a long time waiting for I/O operations. Since HaLoop tries to allocate the same mapper to the same data partitions, there is not too much data shuffling. It is interesting to note that both Hadoop and HaLoop use similar average physical memory in their workers. This identifies an opportunity for HaLoop: instead of

[12]It typically fails after a few iterations in the 64 and 128 machine clusters.

Table 8: Total Giraph Memory across the cluster. All numbers are in GB; first row shows cluster size.

dataset (size in GB)	16	32	64	128
Twitter (12.5)	191.5	323.6	606.4	923.5
UK0705 (31.9)	264.0	411.8	717.6	1322.6
WRN (13.6)	363.7	475.4	683.4	1054.1

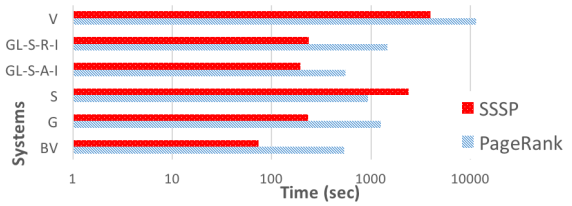


Figure 9: Computing SSSP (116 iterations) and 55 iterations of PageRank for the UK dataset using a cluster of 32 machines.

caching files on local disks, HaLoop could have utilized the available memory to further reduce execution time.

5.11 Vertica Experiments

Although Vertica supports r3.4xlarge and r3.8xlarge instances only, we ran our experiments using r3.xlarge to make these results comparable to the rest of our experiments. We use similar SQL queries to the ones described in [30]. These experiments are not as complete as others because we were allowed to use the system for a short trial period[13]. Nonetheless, we believe the results in Figures 9 and 10 present a good indication of its performance in large clusters.

Unlike previously reported results [30], Vertica is not competitive relative to native in-memory graph processing systems. As the cluster size increases, so does the gap between its performance and other systems. Previously reported experiments were conducted on only 4 machines and that may explain the competitive results.

The main reason behind Vertica's performance with large clusters is its requirement to create and delete new temporary tables during execution, because each table is partitioned across multiple machines. Moreover, self-join operation involves shuffling. The larger the cluster, the more expensive data shuffling becomes. Figure 10 supports this argument. Although Vertica footprint is small, the I/O-wait time and network cost are significant. Increasing the cluster size, significantly adds to these overheads. On the other hand, distributed graph processing systems can utilize the computing power of larger clusters without significant I/O and network overhead.

There is a Vertica extension to avoid the intermediate disk I/O, but this extension is not yet publicly available. It works on multiple cores on a single machine, but it does not support shared-nothing parallel architectures.

5.12 Scalability

LDBC [29] discusses two orthogonal types of scalability analysis: strong/weak and horizontal/vertical scalability. In a strong scalability experiment the same dataset is used with different cluster sizes (horizontal scalability) or with one machine and different number of cores (vertical scalability). In a weak scalability experiment the data load (represented in graph size) of each machine in the cluster is fixed as we change the cluster size. For example, as we double the cluster size, we also double the graph size to keep the data load per machine constant.

Our study does not include vertical scalability experiments because all our systems were introduced as parallel shared-nothing systems. We only consider real datasets whose sizes are fixed. Therefore, our scalability analysis is "strong".

Blogel, Giraph, Gelly, and GraphLab show steady performance increase as the cluster size increases. GraphX and Vertica do not show the same scalability potential. GraphX suffers from load balancing issues: the higher the number of workers, the lower the balance between machines. Vertica, on the other hand, has to shuffle more data as the cluster gets larger. That said, scalability was not noticeable during the execution of SSSP and K-hop workloads because most graph vertices do not participate in each iteration during SSSP and K-hop computation.

[13]The community edition of Vertica is restricted to 3 machines only.

Table 9: Time in seconds for a single thread algorithm (S) and best performing parallel system using 16 machines (P)

	PageRank		SSSP		WCC	
	P	S	P	S	P	S
Twitter	BV=260	490	BV=48.3	422	GL=248	452
UK0705	BV=338.7	720	BV=122.3	610	GL=492.67	632
WRN	BV=268.3	880	BV=11295	455	BV=19831	640

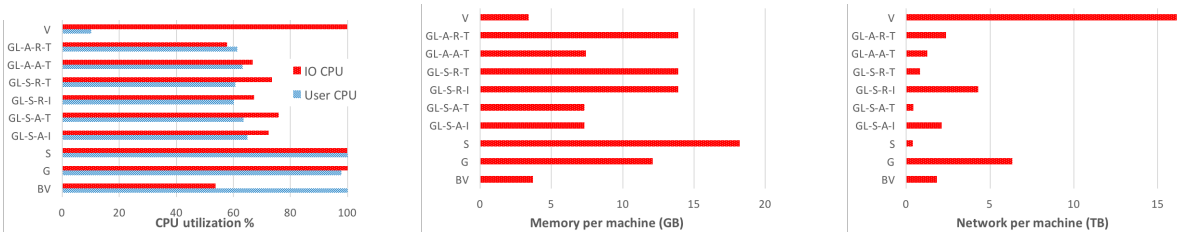
5.13 COST Experiment

COST [39] stands for Configuration that Outperforms a Single Thread, and is used in the literature to evaluate the overhead of parallel algorithms and systems. The main idea is that parallel algorithms are often not optimal due to the special design considerations for parallel processing between machines. COST factor represents the response time of single thread divided by the response time of a parallel system. In the COST experiment, we used the single thread implementation of the GAP Benchmark Suite [16] on a large machine with 512 GB memory. Table 9 summarizes the performance of a single-thread implementation (S) and the performance of the best parallel system (P) using 16 machines. Looking at Figures 5 and 6, it is clear that the performance of some systems using multiple machines is worse than their single thread performance.

Some of the algorithms used in this experiment are different than ones described in Section 3. The PR algorithm is similar to the one used by all systems. The SSSP algorithm[14] processes the workload from two directions and pre-computes each vertex degree in its initial phase [15]. The WCC algorithm implementation[15] is based on Shiloach-Vishki algorithm with further optimizations [12, 44, 33].

Although many systems, using 16 machines, have a COST < 1, meaning they perform worse than a single thread implementation, best parallel systems perform better than the single thread in most cases. However, definitive conclusions using the COST factor are hard to reach. For PR, the cost factor is between 2 and 3 which means the 16-machines cluster performs two to three times faster than the single thread. For SSSP and WCC the cost factor is 0.5 to 0.11 for power-law datasets, but it is 0.04 and 0.03 respectively for WRN. or reachability-based workloads, the best parallel system could be two orders of magnitude slower than a single thread implementation. The large number of iterations

[14]https://tinyurl.com/y7rsgg9w
[15]https://tinyurl.com/ydxhpfyx



(a) Maximum User and I/O CPU (b) Memory footprint (c) Network usage

Figure 10: Understanding how Vertica use its computing resources in comparison to other systems. All results were collected while computing 55 iterations of PageRank for the UK0705 dataset using a cluster of 64 machines.

leads to significant network overhead between machines. Of course, single-thread performance requires larger machines, e.g., running WCC on WRN using the single thread implementation uses 112GB memory – four times the memory available in the machines used in our experiments.

The surprisingly bad COST factor of many parallel systems is due to three main factors:

- Different algorithms: Parallel systems adopt simple algorithms that can scale well, while the single thread implementations include several optimizations. It is an interesting future work to study the possibility of parallelizing these optimizations.
- Replication: Parallel systems need to partition the dataset with significant replication factors (see Table 4). This adds overhead to the overall dataset sizes.
- Network: Parallel systems incur network overhead, which, of course, is absent in single thread implementation.

6. RELATED WORK

Many studies of graph processing systems focus on Pregel-like systems [27, 14, 37]. LDBC [29] includes industry-driven systems such as GraphPad [11] from Intel and PGX [28] from Oracle. In contrast, our choice of systems under study follows a systematic classification (Section 1).

LDBC is the only study that uses vertical/horizontal and strong/weak scalability. Our study does not include vertical scalability experiments for reasons discussed in Section 5.12.

All studies consider power-law real and synthetic datasets. Many systems use graph data generators, such as DataGen [21], WGB [10], RTG [8], LUBM [26], and gMark [13] to create large datasets. The main objective of these graph generators is to represent real graphs and allow the creation of graph datasets with different sizes. In this this study, we only use real graph datasets of varying sizes, some of which are larger than any other dataset in existing studies. We also include more diverse datasets (road network, social network, and web pages) than others study.

The following highlight the major findings of previous studies and where our results differ:

[37, 27] Giraph and GraphLab have similar performance when both use random partitioning. However, GraphLab has an auto mode that allows it to beat Giraph in certain cluster sizes.

[47] It is not fair to say that existing implementation of block-centric computation is faster than vertex-centric. The execution time is faster in block-centric, but the overall response time is slower due to overheads discussed in Section 5.1.

[14] It was previously reported that GraphX has the best performance across all systems [14]. Our results contradicts this assertion. GraphX paper [25] never claimed that it was more efficient than other systems, even though a special version of GraphX was used that includes multiple optimizations that are not yet available in the most recent Spark release[16]. Furthermore, the version of Spark (v 1.3.0) used in the study making this claim [14] does not compute PageRank values accurately in all cases. We suspect this causes fast convergence because we had similar experience in our earlier results.

[30] Vertica is not competitive to existing parallel graph processing systems. Section 2.6 has further details about Vertica performance. In a nutshell, Vertica I/O and network overhead is significantly larger than graph systems. This overhead increases as the cluster size increases.

Finally, a recent study [42] reports a user survey to explain how graphs are used in real life. The paper aims to understand types of graphs, computations, and software users need when processing their graphs. The focus is different and perhaps complementary to our paper.

7. CONCLUSION

In this paper we present results from extensive experiments on eight distributed graph processing systems (Hadoop, HaLoop, Vertica, Giraph, GraphLab, GraphX, Flink Gelly, and Blogel) across four workloads (PageRank, WCC, SPSS, K-hop) over four very large datasets (Twitter, World Road Network, UK 200705, and ClueWeb). We focused on scale-out performance. Our experiments evaluate a wider set of systems that follow more varied computation models, over a larger and more diverse datasets than previous studies [14, 27, 37]. Ours is the most extensive independent study of graph processing systems to date. Our results indicate that the best system varies according to workload and particular data graph that is used. In some cases our study confirms previously reported results, in other cases the results are different and the reasons for the divergence are explained.

8. ACKNOWLEDGMENTS

We thank Semih Salihoglu and Khuzaima Daudjee for their feedback on the paper. We also acknowledge the contribution of our undergraduate research assistants: Heli Wang, Runsheng Guo, and Anselme Goetschmann. This research was supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada.

[16]This was confirmed by GraphX/Spark team: `https://tinyurl.com/y9246wpp`.

9. REFERENCES

[1] Flink. https://flink.apache.org/.

[2] Gelly: Flink graph api. https://ci.apache.org/projects/flink/flink-docs-stable/.

[3] Giraph. http://giraph.apache.org.

[4] Hadoop. http://hadoop.apache.org.

[5] Timely data flow. https://github.com/frankmcsherry/timely-dataflow.

[6] Private correspondence with Blogel team., 2015.

[7] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[8] Leman Akoglu and Christos Faloutsos. Rtg: a recursive realistic graph generator using random typing. *Data Mining and Knowledge Discovery*, 19(2):194–209, 2009.

[9] Khaled Ammar and M. Tamer Özsu. Experimental analysis of distributed graph systems. *arXiv:1806.08082*, 2018.

[10] Khaled Ammar and M.Tamer Özsu. WGB: Towards a universal graph benchmark. In et. al. Rabl, Tilmann, editor, *Advancing Big Data Benchmarks*, Lecture Notes in Computer Science, pages 58–72. Springer, 2014.

[11] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *Proc. 30th Int. Parallel & Distributed Processing Symp.*, pages 313–322, 2016.

[12] David A Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. of Parallel Processing*, pages 547–556, 2005.

[13] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. gmark: schema-driven generation of graphs and queries. *IEEE transactions on knowledge and data engineering*, 29(4):856–869, 2017.

[14] Omar Batarfi, RadwaEl Shawi, AymanG. Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi, and Sherif Sakr. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[15] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, volume 21, pages 12:1–12:10, 2013.

[16] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv:1508.03619*, 2015.

[17] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. The HaLoop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190, 2012.

[18] Deepayan Chakrabarti, Christos Faloutsos, and Mary McGlohon. Graph Mining: Laws and Generators. In *Proc. Managing and Mining Graph Data*, pages 69–123, 2010.

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX Symp. on Operating System Design and Implementation*, pages 137–149, 2004.

[21] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 619–630, 2015.

[22] Martin Erwig and Fernuniversitat Hagen. The graph voronoi diagram with applications. *Networks*, 36:156–163, 2000.

[23] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *Proc. 7th Biennial Conference on Innovative Data Systems Research*, pages 1–10, 2015.

[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th USENIX Symp. on Operating System Design and Implementation*, pages 17–30, 2012.

[25] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 599–613, 2014.

[26] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[27] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[28] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx.d: a fast distributed graph processing engine. In *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[29] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB*, 9(13):1317–1328, 2016.

[30] Alekh Jindal, Samuel Madden, Malu Castellanos, and Meichun Hsu. Graph analytics using vertica relational database. In *Proc. IEEE International Conference on Big Data*, pages 1191–1200, 2015.

[31] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: a peta-scale graph mining system implementation and observations. In *Proc.*

2009 IEEE Int. Conf. on Data Mining, pages 229–238, 2009.

[32] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *Proc. 5nd ACM Symp. on Cloud Computing*, pages 18:1–18:13, 2014.

[33] Kishore Kothapalli, Jyothish Soman, and PJ Narayanan. Fast GPU algorithms for graph connectivity. In *Proc. Workshop on Large Scale Parallel Processing*, pages 66–75, 2010.

[34] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proc. 11th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 177–187, 2005.

[35] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proc. 11th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 177–187, 2005.

[36] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[37] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[38] 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 *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146, 2010.

[39] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proc of the 15th USENIX Conference on Hot Topics in Operating Systems*, 2015.

[40] Frank McSherry, Derek G Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proc. 6th Biennial Conference on Innovative Data Systems Research*, 2013.

[41] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proc. 24th ACM Symp. on Operating System Principles*, pages 439–455, 2013.

[42] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4), 2017.

[43] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. In *Proc. 25th Int. Conf. on Scientific and Statistical Database Management*, pages 1–12, 2013.

[44] Yossi Shiloach and Uzi Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[45] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[46] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB*, 10(5):493–504, 2017.

[47] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[48] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX Symp. on Networked Systems Design and Implementation*, pages 15–28, 2012.