Large-Scale Distributed Graph Computing Systems: A Survey

Yi Lu[#1], James Cheng[#2], Da Yan[*3], Huanhuan Wu[#4]
[#]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*
{[1]ylu, [2]jcheng, [4]hhwu}@cse.cuhk.edu.hk
[*]*Department of Computer Science and Engineering, The Hong Kong University of Science and Technology*
[3]yanda@cse.ust.hk

ABSTRACT

With the prevalence of graph data in real-world applications (e.g., social networks, mobile phone networks, web graphs, etc.) and their ever-increasing size, many distributed graph computing systems have been developed in recent years to process and analyze massive graphs. Most of these systems adopt Pregel's vertex-centric computing model, while various techniques have been proposed to address the limitations in the Pregel framework. However, there is a lack of comprehensive comparative analysis to evaluate the performance of various systems and their techniques, making it difficult for users to choose the best system for their applications. We conduct a survey which extensively evaluates the performance of existing systems on graphs with different characteristics and on algorithms with different design logic. We also assess the effectiveness of various techniques adopted in existing systems, and the scalability of the systems. The results of our study reveal the strengths and limitations of existing systems, and provide valuable insights for users, researchers and system developers.

1. INTRODUCTION

Many distributed graph computing systems have been proposed to conduct all kinds of data processing and data analytics in massive graphs, including Pregel [15], Giraph [4], GraphLab [14], PowerGraph [9], GraphX [27], Mizan [13], GPS [21], Giraph++ [25], Pregelix [7], Pregel+ [2], and Blogel [1]. These systems are all built on top of a shared-nothing architecture, which makes big data analytics flexible even on a cluster of low-cost commodity PCs.

The majority of the systems adopt a "think like a vertex" vertex-centric computing model [15], where each vertex in a graph receives messages from its incoming neighbors, executes the user-specified computation and updates its value, and then sends messages to its outgoing neighbors. The vertex-centric computing model makes the design and implementation of scalable distributed algorithms simple for ordinary users, while the system handles all the low-level details. There are also a few extensions to the vertex-centric model, e.g., the edge-centric model in PowerGraph [9] and the block-centric models in [25, 1], to address various limitations in the vertex-centric systems. However, these models also suffer from other problems as a tradeoff (see details in Section 3).

While many distributed graph computing systems have emerged recently, it is unclear to most users and researchers what the strengths and limitations of each system are and there is a lack of overview on how these systems compare with each other. Thus, it is difficult for users to decide which system is better for their applications, or for researchers and system developers to further improve the performance of existing systems or design new systems.

So far, we have only seen performance evaluations conducted by the respective authors of various systems in their research papers and the focus of these evaluations is mostly on their own system. In particular, there is a lack of study on the weaknesses of their own systems and detailed comparative analysis concerning various systems. Thus, we conduct an extensive study on the performance of various distributed graph computing systems.

We first give a detailed survey on existing distributed graph computed systems in Section 3. Among these systems, we conduct comprehensive experimental evaluation on Giraph [4], GraphLab/ PowerGraph [14, 9], GPS [21], and Pregel+ [2]. We do not conduct experiments on other existing systems for various reasons given in Section 3.6. We also classify a set of graph algorithms, many of which have been popularly used to evaluate the performance of various systems in existing works, into five categories of algorithms, where each category represents a different logic of distributed algorithm design (see details in Section 4). Then, in Section 5, we conduct a comprehensive analysis on the performance of various systems focusing on the following key objectives:

- To evaluate the performance of various systems in processing large graphs with different characteristics, including skewed (e.g., power-law) degree distribution, small diameter (e.g., small-world), large diameter, (relatively) high average degree, and random graphs.
- To evaluate the performance of various systems with respect to different categories of algorithms presented in Section 4.
- To assess the effects of individual techniques used in various systems on their performance, hence analyzing the strengths and limitations of each system. The techniques to be examined include message combiner, mirroring, dynamic repartitioning, request-respond API, asynchronous and synchronous computation, and support for graph mutation.
- To test the scalability of various systems by varying the number of machines, the number of workers, the number of vertices and the number of edges in graphs with different degree distributions.

To our best knowledge, no existing work has conducted an extensive study on any one set of the above four sets of performance

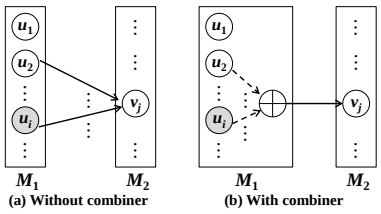


Figure 1: Illustration of combiner

	Computing mode	Message passing	Data pushing/pulling	Skewed workload balancing	Multi-threading	Combiner support	Graph mutation
Giraph	Sync	Yes	Pushing	–	Yes	Yes	Yes
GPS	Sync	Yes	Pushing	Large adjacency list partitioning + Dynamic repartitioning	No	No	Only edges
GraphLab	Sync/Async	Only neighbors	Pushing/Pulling	Vertex cut	Yes	No	No
Pregel+	Sync	Yes	Pushing+Pulling	Mirroring + Request-respond API	No	Yes	Yes

Figure 2: Systems overview

evaluations. Some works [9, 21] may have considered power-law graphs but largely ignored large-diameter graphs, while most works do not provide any insight on the performance of their systems on specific algorithm categories. There is no single work that has analyzed the effect of various techniques on the performance of different systems. For scalability test, existing works only tested by varying the number of machines (only Pregel also varied the graph size), while some even did not conduct any scalability test.

We give detailed comparative analysis on all the four sets of evaluation criteria listed above. We also provide insights on the performance of each system w.r.t. graph characteristics, algorithm categories, and system scalability, which can be used to match with the requirements of specific applications, so that users can select the best system(s) or researchers/developers can explore new ideas/techniques to address the system limitations.

2. PRELIMINARY

We first define some basic graph notations. Let $G = (V, E)$ be a graph, where V and E are the sets of vertices and edges of G. If G is undirected, we denote the set of neighbors of a vertex $v \in V$ by $\Gamma(v)$. If G is directed, we denote the set of in-neighbors (out-neighbors) of a vertex v by $\Gamma_{in}(v)$ ($\Gamma_{out}(v)$). Each vertex $v \in V$ has a unique integer ID, denoted by $id(v)$. The diameter of G is denoted by $\delta(G)$, or simply δ when G is clear from the context.

The distributed graph computing systems evaluated in this paper are all based on a shared-nothing architecture, where data are stored in a *distributed file system (DFS)*, e.g., *Hadoop's DFS*. The input graph is stored as a distributed file in a DFS, where each line records a vertex and its adjacency list. A distributed graph computing system consists of a cluster of k workers, where each worker w_i keeps and processes a batch of vertices in its main memory. Here, "worker" is a general term for a computing unit, and a machine can have multiple workers in the form of threads/processes.

A job is processed by a graph computing system in three phases as follows. **(1)Loading**: each worker loads a portion of vertices from the DFS into its main-memory; then workers exchange vertices through the network (e.g., by hashing on vertex ID as in Pregel) so that each worker w_i finally keeps all and only those vertices that are assigned (i.e., hashed) to w_i. **(2)Iterative computing**: in each iteration, each worker processes its own portion of vertices sequentially, while different workers run in parallel and exchange messages. **(3)Dumping**: each worker writes the output of all its processed vertices to the DFS. Most existing graph-parallel systems follow the above three-phase procedure.

3. A SURVEY ON EXISTING SYSTEMS

We briefly discuss existing distributed graph computing systems, and highlight their distinguished features.

3.1 Pregel

Pregel [15] is designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to different machines in a cluster, where each vertex v is associated with its adjacency list (i.e., the set of v's neighbors). A program in Pregel implements a user-defined *compute*() function and proceeds in iterations (called *supersteps*). In each superstep, the program calls *compute*() for each active vertex. The *compute*() function performs the user-specified task for a vertex v, such as processing v's incoming messages (sent in the previous superstep), sending messages to other vertices (to be received in the next superstep), and making v vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Pregel numbers the supersteps in the order when they are executed, so that a user may use the current superstep number when implementing the algorithm logic in the *compute*() function. As a result, a Pregel algorithm can perform different operations in different supersteps by branching on the current superstep number (e.g., perform one operation in superstep i, and perform a different operation in superstep $(i + 1)$).

Message Combiner. If x messages are to be sent from a machine M_i to a vertex v in a machine M_j, and some commutative and associative operation is to be applied to the x messages in *v.compute*() when they are received by v, then these x messages can be first combined into a single message which is then sent from M_i to v in M_j. To achieve this goal, Pregel allows users to implement a *combine*() function to specify how to combine messages that are sent from machine M_i to the same vertex v in machine M_j.

Figure 1 illustrates the idea of combiner, where x messages are sent by vertices in machine M_i to the same target vertex v in machine M_j, and assume that the "sum" operator is to be applied to the x messages. In Figure 1(a) combiner is not applied, and x messages are sent to v in M_j, and then the messages are summed up when the *compute*() function is applied to v in M_j. In Figure 1(b) combiner is applied, and the x messages are first combined into their sum, and then the sum (i.e., a single message) is sent v in M_j.

Aggregator. Pregel also supports aggregator, which works as follows. Each vertex can provide a value to an aggregator in *compute*() in a superstep. Then the system aggregates those values and makes the aggregated result available to all vertices in the next superstep. Aggregator can be used for global communication.

3.2 Giraph

Since Google's Pregel is proprietary, Yahoo! initiated the development of the Giraph system as an open source implementation of Google's Pregel. Later, Facebook built its Graph Search services upon Giraph, and further improved the performance of Giraph by introducing the following optimizations:

Multi-threading. Facebook adds multithreading to graph loading, dumping, and computation. In CPU bound applications, a speedup near-linear to the number of processors can be observed by multithreading the application code.

Memory Optimization. The initial release of Giraph by Yahoo! requires high memory consumption, since all data types are stored as separate Java objects. A large number of Java objects greatly

degrades the performance of Java Virtual Machine (JVM). Since object deletion is handled by JAVA's Garbage Collector (GC), if a machine maintains a large number of vertex/edge objects in main memory, GC needs to track a lot of objects and the overhead can severely degrade the system performance. To decrease the number of objects being maintained, JAVA-based systems maintain vertices in main memory in their binary representation. Thus, the later Giraph system organizes vertices and messages as main memory pages, where each page is simply a byte array object that holds the binary representation of many vertices.

3.3 GPS

GPS [21] is another open source implementation of Google's Pregel, with additional features. GPS extends the Pregel API to include an additional function, *master.compute*(), which provides the ability to access to all of the global aggregated values, and store global values which are transparent to the vertices. The global aggregated values can be updated before they are broadcast to the workers. Furthermore, GPS also introduces the following two techniques to boost system performance:

Large Adjacency-List Partitioning (LALP). When LALP is applied, adjacency lists of high-degree vertices are not stored in a single worker, but they are rather partitioned across workers. For each partition of the adjacency list of a high-degree vertex, a mirror of the vertex is created in the worker that keeps the partition. When a high-degree vertex broadcasts a message to its neighbors, at most one message is sent to its mirror at each machine. Then, the message is forwarded to all its neighbors in the partition of the adjacency list of the high-degree vertex.

Dynamic Repartitioning. This optimization technique repartitions the graph dynamically according to the workload during the execution process, in order to balance the workload among all workers and reduce the number of messages sent over the network. However, dynamic repartitioning also introduces extra network workload to reassign vertices among workers. Sometimes the overhead of dynamic repartitioning exceeds the benefits gained.

3.4 Pregel+

Pregel+ [2] is implemented in C/C++ and each worker is simply an MPI process. In addition to the basic techniques provided in existing Pregel-like systems, Pregel+ introduces two new techniques to further reduce the number of messages.

Mirroring. The mirroring technique is designed to eliminate communication bottlenecks caused by high vertex degree. Given a high-degree vertex v, a mirror of v is constructed in a worker if some of v's neighbors reside in the worker. When v sends a message, $a(v)$, to its neighbors, v first sends $a(v)$ to all its mirrors and then each mirror forwards $a(v)$ to the neighbors of v that reside in the local worker of the mirror. The local message forwarding does not involve any message passing and hence the number of messages is bounded to the number of workers regardless of how large the degree of a vertex may be.

Request-Respond API. This API allows a vertex u to request another vertex v for its attribute $a(v)$, and the requested value will be available to u in the next iteration. The request-respond programming paradigm simplifies the coding of many Pregel algorithms, as otherwise at least three iterations are required to explicitly code each request and response process. The request-respond paradigm can also effectively reduce the number of messages passed, since all requests from a machine to the same target vertex v are merged into one request. Suppose that there are x vertices requesting a vertex v's attribute $a(v)$, then the request-respond paradigm reduces the total number of messages from $2x$ (x for requests and x for responses) in Pregel's message passing paradigm to at most twice the number of workers, even though x can be much larger.

3.5 GraphLab and PowerGraph

Unlike Pregel's *synchronous data-pushing model* and *message passing paradigm*, GraphLab [14] adopts an *Gather, Apply, Scatter* (*GAS*) *data-pulling model* and shared memory abstraction. A program in GraphLab implements a user-defined *GAS* function for each vertex.

To avoid the imbalanced workload caused by high-degree vertices in power-law graphs, a recent version of GraphLab, called PowerGraph [9], introduces a new graph partition scheme to handle the challenges of power-law graphs as follows.

In the *Gather* phase, each active vertex collects information from adjacent vertices and edges, and performs a generalized sum operation over them. This generalized sum operation must be commutative and associative, ranging from a numerical sum to the union of the collected information. In the *Apply* phase, each active vertex can update its value based on the result of the generalized sum and its old value. Finally, in the *Scatter* phase, each active vertex can activate the adjacent vertices. However, unlike Pregel's message passing paradigm, GraphLab can only gather information from adjacent edges and scatter information to them, which limits the functionality of the *GAS* model. For example, the S-V algorithm to be described in section 4.1 is hard to be implemented in GraphLab.

GraphLab maintains a global scheduler, and workers fetch vertices from the scheduler for processing, possibly adding the neighbors of these vertices into the scheduler. The GraphLab engine executes the user-defined *GAS* function on each active vertex until no vertex remains in the scheduler. The GraphLab scheduler determines the order to activate vertices, which enables GraphLab to provide with both synchronous and asynchronous scheduling.

Asynchronous Execution. Unlike the behaviors in a synchronous model, changes made to each vertex and edge during the *Apply* phase are committed immediately and visible to subsequent computation. Asynchronous execution can accelerate the convergence of some algorithms. For example, the *PageRank* algorithm (see section 4.1.1) can converge much faster with asynchronous execution. However, asynchronous execution may incur extra cost due to locking/unlocking and intertwined computation/communication.

Synchronous Execution. GraphLab also provides a synchronous scheduler, which executes the *Gather*, *Apply*, and *Scatter* phases in order as an iteration. The *GAS* function of each active vertex runs synchronously with a barrier at the end of each iteration. The function can know the iteration number, which is similar to the superstep number in Pregel. Changes made to the vertex value is committed at the end of each iteration. Vertices activated in each iteration are executed in the subsequent iteration. However, the frequent barriers and inability to operate on the most recent data may lead to an inefficient distributed execution and slow convergence for some algorithms.

Vertex-Cut Partitioning. Natural graphs usually exhibit skewed degree distribution (e.g. power-law). If the graph is partitioned by hashing the vertex ID as in many Pregel-like systems, it will lead to imbalanced workload: high-degree vertices will send and/or receive more messages than others. Therefore, PowerGraph partitions the input graph by cutting the vertices, so that the edges of a high-degree vertex are handled by multiple workers. As a tradeoff, vertices are replicated across workers, and communication among all workers are required to guarantee that the vertex value on each replica remains consistent.

3.6 Other Systems

In this paper, we conduct experimental evaluation on Giraph [4], GraphLab/PowerGraph [14, 9], GPS [21], and Pregel+ [2], which we have discussed in Sections 3.2-3.5 and we also give an overview of various features supported by these systems in Figure 2. There are also a number of other systems that we do not evaluate experimentally, which we explain in this subsection.

Mizan [13] is a C++ optimized Pregel system that supports dynamic load balancing and efficient vertex migration, based on runtime monitoring of vertices to optimize the end-to-end computation. However, Mizan performs pre-partitioning separately which takes much longer compared with Giraph and GPS, and the overhead of pre-partitioning can exceed the benefits. For this reason, we could not run Mizan on many large graphs used in this paper and hence we do not include it in our experimental evaluation.

Pregelix [7], using Hyracks[6] as the runtime execution engine, expresses Pregel's semantics as database-style data flows and executes them on a general-purpose data-parallel engine using classical parallel query evaluation techniques. We do not evaluate Pregelix experimentally in this paper since it imposes a size limit on the adjacency list of each vertex, making it unable to load real-world graphs with high-degree vertices.

GraphX [27], based on Spark [16], is both a data parallel and graph parallel system. GraphX can provide with GraphLab and Pregel abstractions, but GraphX is consistently outperformed by GraphLab (the same conclusion can also be drawn from the results shown in [26]) and thus we exclude GraphX in our experimental evaluation results.

Trinity [23] is a graph-parallel system based on distributed key-value store that supports efficient online query processing and offline analytics on large graphs. However, Trinity is not open source and hence we cannot evaluate its performance experimentally.

The block-centric computing model recently proposed by [25, 1] considers a subgraph as a computing block (instead of a vertex or an edge). Each block receives messages from other blocks, makes some computation and updates the vertices values in the block, then send messages to other blocks. The block-centric computing model can help to decrease the number of iterations in a vertex-centric algorithm and reduce the number of messages to be transmitted through the network, but the trade-off is that it generally requires a longer pre-processing or partitioning time. We focus on the vertex-centric systems in this paper and do not evaluate the block-centric systems (also due to space limit).

4. ALGORITHMS

To evaluate the performance of different distributed graph computing systems, we use seven graph algorithms, most of them are also used in the performance evaluation of existing systems [4, 21, 9, 2]. We divide the graph algorithms into five categories based on the behaviors of the *compute* function in Pregel-like systems and the *GAS* function in Graphlab and PowerGraph, as follows.

Always-Active. An algorithm is Always-Active if every vertex in every superstep sends messages to all its neighbors. In such an algorithm, the distribution of messages sent and received by all active vertices is the same across supersteps, i.e., the communication workload of every machine remains the same. Typical algorithms in this category include *PageRank* [18] in synchronous computation and *Diameter Estimation* [12] (e.g., in Pregel-like systems).

GraphLab's Async Algorithms. This category is specifically for algorithms that are designed to run on GraphLab's (also PowerGraph's) asynchronous computation model. Such algorithms add vertices to the scheduler, and then workers fetch vertices from the scheduler and pull data from neighboring edges and vertices for processing. The asynchronous execution can accelerate the convergence of algorithms such as the asynchronous *PageRank* and *Graph Coloring* [22] algorithms for GraphLab and PowerGraph.

Graph Traversal. Graph traversal is a category of graph algorithms for which there is a set of starting vertices, and other vertices are involved in the computation based on whether they receive messages from their in-neighbors. Attribute values of vertices are updated and messages are propagated along the edges as the algorithm traverses the graph. Algorithms such as *HashMin* [20] and *Single-Source Shortest Paths* [8] are in this category.

Multi-Phase. For this category of algorithms, the entire computation can be divided into a number of phases, and each phase consists of some supersteps. For example, in *Bipartite Maximal Matching* [3], there are four supersteps to simulate a three-way handshake in each phase. The *SV* [24] algorithm also falls into this category, since it simulates tree hooking and star hooking in each phase.

Graph Mutation. Algorithms in this category need to change the topological structure of the input graph through edges and/or vertices addition and/or deletion. For example, the version of the graph coloring algorithm that removes a maximal independent set from the existing graph iteratively is an algorithm in this category. As a result, platforms which do not support graph mutation, such as GraphLab and PowerGraph, cannot straightforwardly support these algorithms. We choose *Graph Coloring* [22] as a presentative graph mutation algorithm, but note that there is another version of *Graph Coloring* designed for GraphLab and PowerGraph that does not require graph mutation.

4.1 Algorithm Description

We introduce seven typical graph algorithms here: *PageRank* [18], *Diameter Estimation* [12], *Single-Source Shortest Paths (SSSP)* [8], *HashMin* [20] and *Shiloach-Vishkin's algorithm (SV)* [24] for computing *Connected Components (CC)*, *Bipartite Maximal Matching (BMM)* [3], and *Graph Coloring (GC)* [22].

4.1.1 PageRank

Given a directed web graph $G = (V, E)$, where each vertex (page) v links to a list of pages $\Gamma_{out}(v)$, the problem is to compute the PageRank value of each vertex in V. Let $pr(v)$ be the PageRank value of a vertex v.

The typical PageRank algorithm [15] for Pregel-like systems works as follows. Each vertex v keeps two fields: $pr(v)$ and $\Gamma_{out}(v)$. In superstep 0, each vertex v initializes $pr(v) = 1$ and sends each out-neighbor of v a message with a value of $pr(v)/|\Gamma_{out}(v)|$. In superstep i ($i > 0$), each vertex v sums up the received PageRank values, denoted by sum, and computes $pr(v) = 0.15 + 0.85 \times sum$. It then distributes $pr(v)/|\Gamma_{out}(v)|$ to each of its out-neighbors. This process terminates after a fixed number of supersteps or the PageRank distribution converges.

The asynchronous version of PageRank algorithm for GraphLab and PowerGraph, named as *Async-PageRank*, works as follows. Each vertex v keeps three fields: $pr(v)$, $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, where $pr(v)$ is initialized as 1. We define the generalized sum in the *GAS* function as a numerical sum. Then for each active vertex v fetched from the scheduler, in the *Gather* phase, the values $pr(u)/|\Gamma_{out}(u)|$ for all neighbors $u \in \Gamma_{in}(v)$ are gathered and summed up (let sum be this sum). In the *Apply* phase, we update $pr(v) = 0.15 + 0.85 \times sum$. In the *Scatter* phase, if the change in value of $pr(v)$ is greater than ϵ (e.g., a typical value of ϵ is 0.01), we add each vertex $u \in \Gamma_{out}(v)$ to the scheduler. This process terminates after a fixed number of supersteps.

4.1.2 Diameter Estimation

Given an undirected graph $G = (V, E)$, we denote the distance between u and v in G by $d(u, v)$. We define the neighborhood function $N(h)$ for $h = 0, 1, \ldots, \infty$ as the number of pairs of vertices that can reach each other in h hops or less. $N(0)$ is thus equal to the number of vertices in G.

Each vertex v keeps two fields: $N[h; v]$ and $\Gamma_{out}(v)$, where $N[h; v]$ indicates the set of vertices v can reach in h hops. In superstep 0, each vertex v sets $N[0; v]$ to $\{v\}$, and broadcasts $N[0; v]$ to each $u \in \Gamma(v)$. In superstep i ($i > 0$), each vertex v receives messages from its neighbors and set the value of $N[i; v]$ as the union of $N[i-1; v]$ and $N[i-1; u]$ for all $u \in \Gamma(v)$. A global aggregator is used to compute the total pairs of vertices, denoted by $N(i)$, that can be reached from each other after superstep i. The algorithm terminates if the following stop condition is true: in superstep i, $N(i)$ is less than or equal to $(1 + \epsilon) * N(i-1)$.

To handle the large volume of each vertex's neighborhood information, i.e., $N[h; v]$, the algorithm applies the idea of Flajolet-Martin [10], which was also used in the ANF algorithm [19].

4.1.3 Single-Source Shortest Paths (SSSP)

Let G=(V, E) be a weighted graph, where each edge $(u, v)$$\in$$E$ has length $\ell(u, v)$. The length of a path P is equal to the sum of the length of all the edges on P. Given a source $s \in V$, the SSSP algorithm computes a shortest path from s to every other vertex $v \in V$, denoted by $SP(s, v)$, as follows. Each vertex v keeps two fields: $\langle prev(v), dist(v)\rangle$ and $\Gamma_{out}(v)$, where $prev(v)$ is the vertex preceding v on $SP(s, v)$ and $dist(v)$ is the length of $SP(s, v)$. Each out-neighbor $u \in \Gamma_{out}(v)$ is also associated with $\ell(v, u)$.

Initially, only s is active with $dist(s) = 0$, and $dist(v) = \infty$ for any other vertex v. In superstep 0, s sends a message $\langle s, dist(s) + \ell(s, u)\rangle$ to each $u \in \Gamma_{out}(s)$, and votes to halt. In superstep i ($i > 0$), if a vertex v receives messages $\langle w, d(w)\rangle$ from any of v's in-neighbor w, then v finds the in-neighbor w^* such that $d(w^*)$ is the smallest among all $d(w)$ received. If $d(w^*) < dist(v)$, v updates $\langle prev(v), dist(v)\rangle = \langle w^*, d(w^*)\rangle$, and sends a message $\langle v, dist(v) + \ell(v, u)\rangle$ to each out-neighbor $u \in \Gamma_{out}(v)$. Finally, v votes to halt.

4.1.4 HashMin

Assume each CC C in an undirected graph G has a unique ID, and for each vertex v in C, let $cc(v)$ be the ID of C. Given G, HashMin [20] computes $cc(v)$ for each v in G, and hence all CCs for G, as all vertices with the same $cc(v)$ form a CC.

Each vertex v keeps two fields: $min(v)$ and $\Gamma(v)$, where $min(v)$ is initialized as the ID of the vertex itself. HashMin broadcasts the smallest vertex ID seen so far by each vertex v as follows. In superstep 0, each vertex v sets $min(v)$ to be the smallest ID among $id(v)$ and $id(u)$ of all $u \in \Gamma(v)$, broadcasts $min(v)$ to all its neighbors, and votes to halt. In superstep i ($i > 0$), each vertex v receives messages from its neighbors; let min^* be the smallest ID received, if $min^* < min(v)$, v sets $min(v) = min^*$ and broadcasts min^* to its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, $min(v) = cc(v)$ for all v.

4.1.5 Shiloach-Vishkin's Algorithm (SV)

The HashMin algorithm requires $O(\delta)$ supersteps for computing CCs, which is too slow for graphs with a large diameter, such as spatial networks where $\delta \approx O(\sqrt{n})$. The SV algorithm [24] can be translated into a Pregel algorithm which requires $O(\log n)$ supersteps [28], which can be much more efficient than HashMin for computing CCs in general graphs.

The SV algorithm groups vertices into a forest of trees, so that all vertices in each tree belong to the same CC. The tree here is relaxed to allow the root to have a self-loop. Each vertex v keeps two fields: $D[u]$ and $\Gamma_{out}(u)$, where $D[u]$ points to the parent of u in the tree and is initialized as u (i.e., forming a self loop at u).

The SV algorithm proceeds in phases, and in each phase, the pointers are updated in the following three steps: (1)for each edge (u, v), if u's parent w is the root, set w as a child of $D[v]$, which merges the tree rooted at w into v's tree; (2)for each edge (u, v), if u is in a star, set u's parent as a child of $D[v]$; (3)for each vertex v, set $D[v] = D[D[v]]$. We perform Steps (1) and (2) only if $D[v] < D[u]$, so that if u's tree is merged into v's tree due to edge (u, v), then edge (v, u) will not cause v's tree to be merged into u's tree again. The algorithm ends when every vertex is in a star.

4.1.6 Bipartite Maximal Matching (BMM)

Given a bipartite graph $G = (V, E)$, this algorithm computes a BMM, i.e., a matching to which no additional edge can be added without sharing an end vertex. The algorithm [15] proceeds in phases, and in each phase, a three-way handshake is simulated.

Each vertex v keeps three fields: $S[v]$, $M[v]$ and $\Gamma_{out}(v)$, where $S[v]$ indicates which set the vertex is in (L or R) and $M[v]$ is the name of its matched vertex (initialized as -1 to indicate that v is not yet matched).

The algorithm computes a three-way handshake in four supersteps: (1)each vertex v, where $S[v] = L$ and $M[v] = -1$, sends a message to each of its neighbors $u \in \Gamma_{out}(v)$ to request a match; (2)each vertex v, where $S[v] = R$ and $M[v] = -1$, randomly chooses one of the messages w it receives, sends a message to w granting its request for match, and sends messages to other requestors $w' \neq w$ denying its request; (3)each vertex v, where $S[v] = L$ and $M[v] = -1$, chooses one of the grantors w it receives and sends an acceptance message to w; (4)each vertex v, where $S[v] = R$ and $M[v] = -1$, receives at most one acceptance message, and then changes $M[v]$ to the acceptance message's value. All vertices vote to halt at the end of a superstep.

4.1.7 Graph Coloring (GC)

Given an undirected graph $G = (V, E)$, GC computes a color for every vertex $v \in V$, denoted by $color(v)$, such that if $(u, v) \in E$, $color(u) \neq color(v)$.

The GC algorithm for Pregel-like systems normally adopts the greedy GC algorithm from [11]. The algorithm iteratively finds a maximal independent set (MIS) from the set of active vertices, assigns the vertices in the MIS a new color, and then removes them from the graph, until no vertices are left in the graph. Each iterative phase is processed as follows, where all vertices in the same MIS are assigned the same color c: (1)each vertex $v \in V$ is selected as a tentative vertex in the MIS with a probability $1/(2 * |\Gamma(v)|)$; if a vertex has no neighbor (i.e. an isolated vertex or becoming isolated after graph mutation), it is a trivial MIS; each tentative vertex v then broadcasts $id(v)$ to all its neighbors; (2)each tentative vertex v receives messages from its tentative neighbors; let min^* be the smallest ID received, if $min^* > id(v)$, then v is included in the MIS and $color(v) = c$, and $id(v)$ is broadcast to its neighbors; (3)if a vertex u receives messages from its neighbors (that have been included in the MIS in superstep (2)), then for each such neighbor v, delete v from $\Gamma(u)$.

Since GraphLab and PowerGraph do not support graph mutation, we use an alternative greedy GC algorithm, which executes in asynchronous mode as follows. The algorithm colors each vertex v with a color $c \in C(v) = \{0, 1, \ldots, |\Gamma(v)|\}$, where $color[v]$ is initialized as v's ID. The generalized sum in the *GAS* function

	Data	Type	|V|	|E|	AVG Deg	Max Deg
Web graphs	WebUK	directed	133,633,040	5,507,679,822	41.21	22,429
Social networks	Friendster	undirected	65,608,366	3,612,134,270	55.06	5,214
	Twitter	directed	52,579,682	1,963,263,821	37.33	779958
	LiveJournal	undirected	10,690,276	224,614,770	21.01	1,053,676
RDF	BTC	undirected	164,732,473	772,822,094	4.69	1,637,619
Spatial networks	USA Road	undirected	23,947,347	58,333,344	2.44	9

Figure 3: Datasets

is defined as the union of all the colors from each vertex's neighbors. In the *Gather* phase, for each vertex all colors are collected from its neighbors, we denote the union as S. In the *Apply* phase, we set the color of each vertex v to a minimum color c, where $c \in C(v) \setminus S$. In the *Scatter* phase, for each adjacent edge (u, v), if $color(u) = color(v)$, we add u to the scheduler.

5. EXPERIMENTAL EVALUATION

We now evaluate experimentally the performance of *Giraph*, *GPS*, *Pregel+*, and *GraphLab* (we use GraphLab 2.2 which includes all the features of PowerGraph). All the source codes of the algorithms used in our evaluation can be downloaded from `https://github.com/graphsystems/evaluation`, while the source codes of the different systems can be found in their own websites.

System settings and datasets. We ran our experiments on a cluster of 15 machines, each has 48 GB DDR3-1,333 RAM and two 2.0GHz Intel(R) Xeon(R) E5-2620 CPU with a Broadcom Gigabit Ethernet BCM5720 network adapter, running 64-bit CentOS 6.5 with Linux kernel 2.6.32. Giraph 1.0.0 and GPS (rev. 112) are built on JDK 1.7.0 Update 45, the hadoop DFS is built on Apache Hadoop 1.2.1. Pregel+, and GraphLab 2.2 are compiled using GCC 4.4.7 with -O2 option enabled, and MPICH 3.0.4 is used.

We used six large real-world datasets, which are from four different domains as shown in Figure 3: (1)web graphs: *WebUK*[1]; (2)social networks: *Friendster*[2], *LiveJournal* (LJ)[3] and *Twitter*[4]; (3)RDF graph: *BTC*[5] ; (4)road networks: *USA*[6]. Among them, WebUK, LJ, Twitter and BTC have skewed degree distribution; WebUK, Friendster and Twitter have average degree relatively higher than other large real-world graphs; USA and WebUK have a large diameter, while Friendster and Twitter have a small diameter.

We also used synthetic datasets for scalability tests. We generate power law graphs using Recursive Matrix (R-MAT) model [5] and random graphs using PreZER algorithm [17].

We use 8 cores in each machine for Giraph and GraphLab which run with multithreading, and 8 workers (using 8 cores) in each machine for GPS[7], Pregel+ and Blogel. All running time reported is the wall-clock time elapsed during loading and computing, but not including dumping since it is identical for all systems.

Objectives of experimental evaluation. We evaluate the systems following the key objectives listed in Section 1, and provide detailed comparative analysis on each of the evaluation criteria.

5.1 Performance on Different Graphs

We first evaluate the performance of the various systems on graphs with different characteristics. The results of this experiment (and the next experiment to be discussed in Section 5.2) also give readers an overview on the performance of the systems. In the experiments in Sections 5.1- 5.2, we enable all the techniques of various systems that give the best performance, while in Sections 5.3- 5.6 we assess the effects of each individual techniques.

We run the different categories of algorithms discussed in Section 4 with different systems. The results are reported in Figures 4(a)-4(l), which will be analyzed in both Section 5.1 and Section 5.2. Note that not all algorithms are suitable for all datasets (e.g., it does not make much sense to run PageRank on the USA road graph), while not all systems support all algorithms (e.g., it is not clear how SV and graph mutation in coloring can be implemented in GraphLab and PowerGraph).

Performance on graphs with skewed degree distribution. Figures 4(a), 4(b), 4(c), 4(d), 4(i), 4(j), and 4(k) report the performance of various systems on graphs with skewed degree distribution (i.e., WebUK, LJ, BTC, and Twitter). The results show that Pregel+ (with the help of its mirror and request-respond techniques) has the best performance in most cases, while GPS (with the help of LALP) also has good performance in most of the cases. GraphLab is faster than Giraph in about half of the cases but is slower in other cases. Overall, there is no system that is always better than the others in processing graphs with skewed degree distribution, but Pregel+ and GPS are the better choices as either one of them has the best performance in all the cases tested.

Performance on graphs with a large diameter. Figures 4(a), 4(g) and 4(h) report the performance of various systems on graphs with a large diameter (e.g., WebUK and USA Road). The results are more clear for handling such graphs as we can obtain a ranking according to their running time, i.e., Pregel+ has the best performance, and then followed by GraphLab, GPS, and Giraph. It can also be observed that Giraph performs rather poorly in handling both WebUK and USA Road.

Performance on graphs with a small diameter. Figures 4(b), 4(e), 4(f) and 4(l) report the performance of various systems on graphs with a small diameter, e.g., Friendster and Twitter, which are also small-world graphs. Pregel+ records the best performance on Friendster but does not perform so well on Twitter. GPS has the best performance on Twitter and also has good performance on Friendster. GraphLab has good performance on Twitter, but performs poorly on Friendster. Giraph has the worst performance on Twitter, and also does not perform well on Friendster.

Performance on graphs with high average degree. Figures 4(a), 4(b), 4(e), 4(f) and 4(l) report the performance of various systems on graphs with a relatively high average degree (e.g., WebUK, Friendster and Twitter). Pregel+ has the best performance in all the cases except PageRank on Twitter. GPS is the fastest on Twitter and also has good performance on the other two graphs. GraphLab also has good performance on both WebUK and Twitter, but is rather slow on Friendster. Giraph has the worst performance overall.

Conclusions. Figure 5 gives a ranking on the overall performance of various systems on different types of graphs. Note that the ranking is only a rough estimation based on our experimental results. In most case, Pregel+ has the best performance, while in the case when Pregel+ does not perform well, GPS often has the best performance. Thus, we believe that these two systems are good choices for processing a wide range of real-world graphs.

5.2 Performance on Different Algorithms

We next evaluate the performance of the various systems w.r.t. different categories of algorithms presented in Section 4. The re-

[1] `http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05`

[2] `http://snap.stanford.edu/data/com-Friendster.html`

[3] `http://konect.uni-koblenz.de/networks/livejournal-groupmemberships`

[4] `http://konect.uni-koblenz.de/networks/twitter_mpi`

[5] `http://km.aifb.kit.edu/projects/btc-2009/`

[6] `http://www.dis.uniroma1.it/challenge9/download.shtml`

[7] The polling time is set to 10 msec, in order to reduce the fixed overhead in each superstep.

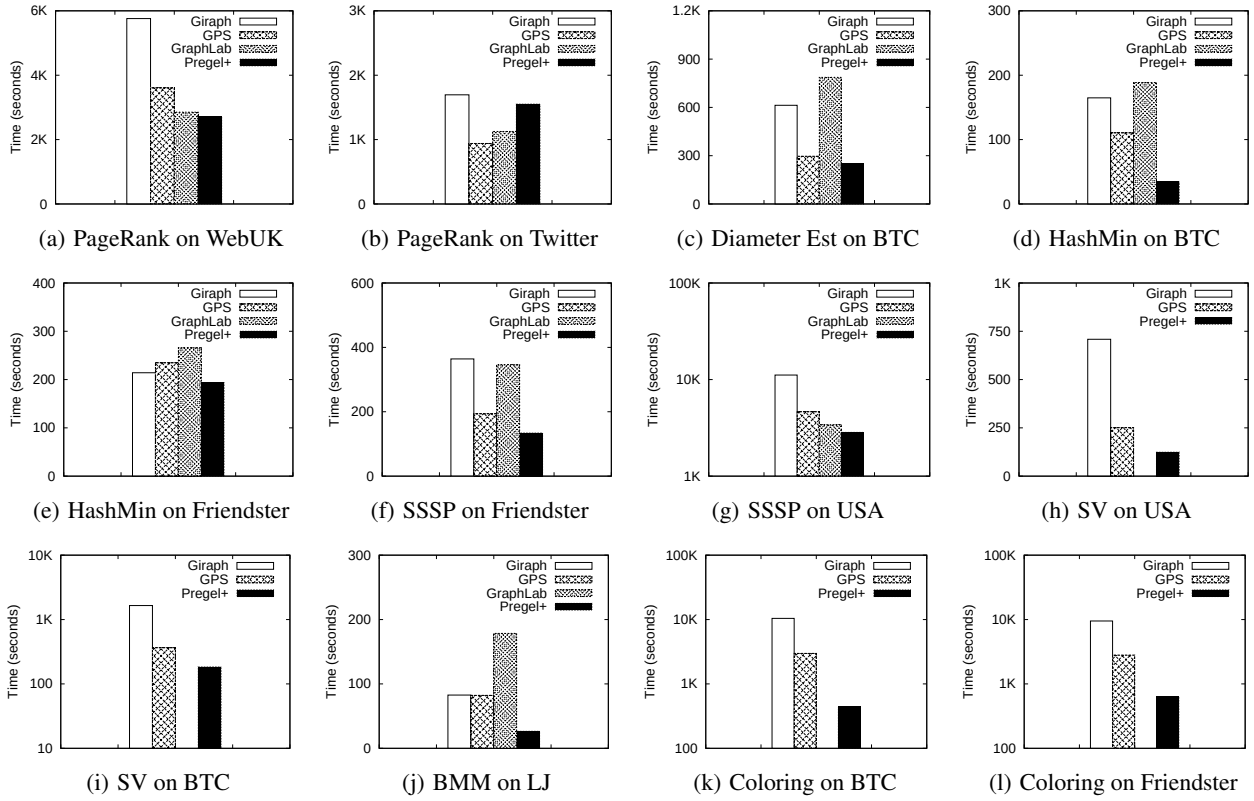


Figure 4: Performance overview on Giraph, GPS, GraphLab, and Pregel+

	Overall performance ranking (1: best)			
	1	**2**	**3**	**4**
Skewed degree	Pregel+	GPS	GraphLab / Giraph	-
Large diameter	Pregel+	GraphLab	GPS	Giraph
Small diameter	Pregel+ / GPS	GraphLab	Giraph	-
High average degree	Pregel+	GPS	GraphLab	Giraph

Figure 5: Overall performance on different graphs

sults of our experiments are also partly reported in Figures 4(a)-4(l), and some more specific results are reported in Figures 6(a)-6(d), and Figures 7(a)- 7(b).

Performance on always-active algorithms. Figures 4(a), 4(b), and 4(c) report the performance of various systems on two representative always-active algorithms, i.e., synchronous PageRank and Diameter Estimation. Pregel+ is the fastest for PageRank on WebUK and for Diameter Estimation on BTC, but has poor performance for PageRank on Twitter. GPS has the best performance for PageRank on Twitter, but is considerably slower than Pregel+ for PageRank on WebUK. Thus, it is difficult to draw a clear conclusion which system has the best performance. The results of these algorithms on other datasets (not reported due to space limit) also show similar patterns. Overall, Pregel+ and GPS are the two systems with better performance, while Giraph has the worst performance in most cases.

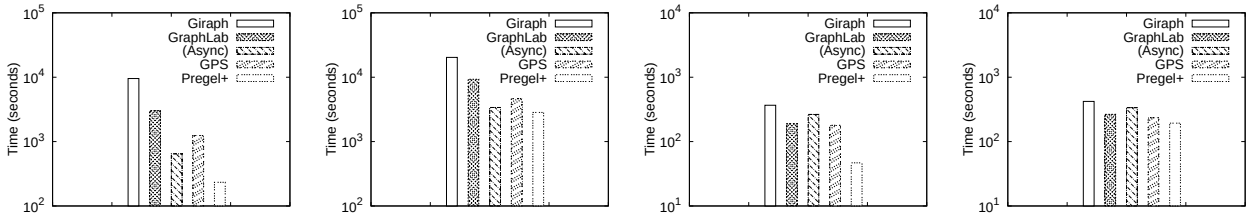
Performance on graph-traversal algorithms. Figures 4(d), 4(e), 4(f), and 4(g) report the performance of various systems on two representative graph-traversal algorithms, i.e., HashMin and Single-Source Shortest Paths (SSSP). For this type of algorithms, the results show that Pregel+ clearly outperforms all the other systems, while GPS also has good performance in most cases. Giraph and GraphLab have comparable performance in this set of experiments.

Performance on multi-phase algorithms. Figures 4(h), 4(i), and 4(j) report the performance of various systems on two representative graph-traversal algorithms, i.e., Bipartite Maximal Matching (BMM) and the SV algorithm. The results show that Pregel+ has the best performance for this type of algorithms, while GPS is faster than Giraph and GraphLab.

Performance on graph mutation. To test the performance on graph mutation, we use the graph coloring algorithm that first finds a maximum independent set, colors the set to a new color, and removes the induced subgraph from the graph. These procedures are repeated until all vertices are assigned colors. Since edges are removed during the computation, we only report the performance of Giraph, GPS and Pregel+ as GraphLab does not support graph mutation. Note that although GPS supports edge addtion/deletion, it cannot add/remove vertices after loading the graph. Giraph and Pregel+ support all the functionality on graph mutation.

Figures 4(k) and 4(l) report the running time of the systems for graph coloring in BTC and Friendster. The results show that Pregel+ is much faster than both GPS and Giraph, which can be explained as follows. In GPS and Giraph, users need to subclass a separate Edge Class during the graph loading phase and edge deletion requests must be made in the *compute*() function; while in Pregel+, the edge information of a graph is stored on vertices, which simplifies the API and enables faster edge addtion/deletion.

Performance on GraphLab's async algorithms. To show the performance difference between synchronous and asynchronous algorithms, we first test asynchronous and synchronous versions of



(a) HashMin on USA (b) SSSP on USA (c) HashMin on BTC (d) HashMin on Friendster

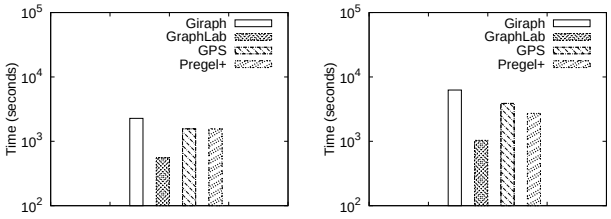
Figure 6: Performance of asynchronous computing (in GraphLab) and synchronous computing

HashMin and SSSP on the USA road network. The results, as reported in Figures 6(a) and 6(b), show that the asynchronous GraphLab algorithms are significantly faster than the synchronous counterparts. To give more details, in running HashMin, the *apply*() function is called 7,120,626,977 times in synchronous execution of GraphLab, while it is only called 1,787,580,028 times in asynchronous execution, which results in 4.6 times shorter elapsed running time. Similarly, in running SSSP, the *apply*() function is called 42,807,001,388 times in synchronous execution of GraphLab, but is only called 9,711,860,625 times in asynchronous execution, which results in 2.7 times shorter elapsed running time.

In the figures, we also show the running time of the synchronous systems, i.e., Giraph, GPS and Pregel+, as a reference, which shows that the asynchronous GraphLab has better performance than GPS and Giraph, though not as fast as Pregel+. However, these results are obtained from processing a graph with a large diameter, for which asynchronous execution can be more effective since changes made to each vertex and edge during the *apply* phase are committed immediately and visible to subsequent computation. However, for processing graphs with a small diameter, asynchronous execution may not be as effective which we show as follows.

We test HashMin on BTC and Friendster, both of which are graphs with a small diameter. The asynchronous execution calls the *apply*() function 476,490,831 and 201,912,097 times, respectively, while the synchronous counterpart calls the *apply*() function 959,566,514 and 465,376,922 times, respectively. However, the synchronous execution is 1.4 times and 1.3 times faster than the asynchronous execution, as reported in Figures 6(c) and 6(d). This is because, although the asynchronous execution call the *apply*() function for fewer times than its synchronous counterpart, the asynchronous global scheduler needs to maintain more vertices, which will introduce an extra cost of locking/unlocking that leads to an overhead exceeding the benefits gained by asynchronous execution.

However, for some algorithm, asynchronous execution can help the algorithms converge faster. For example, for asynchronous PageRank, most vertices can converge after only a small number of updates. On the contrary, in synchronous execution, all vertices need to update their PageRank values and distribute their new values to neighbors. In each superstep, there are $O(n)$ updates made and $O(m)$ messages transmitted. In asynchronous execution, the global scheduler only maintains the vertices that need to be updated. If there is a significant change in some vertex's PageRank value, then it activates its neighbors and puts them into the global scheduler. In Figures 7(a) and 7(b), we can observe that the PageRank algorithm runs only 554.4 seconds on Twitter using 508,251,513 updates and 1,037.9 seconds on WebUK using 847,312,369 updates. However, the synchronous PageRank uses 4,679,591,698 and 11,893,340,560 updates, respectively, and are much slower.



(a) PageRank on Twitter (b) PageRank on WebUK

Figure 7: Asynchronous PageRank in GraphLab

		Overall performance ranking (1: best)		
		1	**2**	**3**
Always active		Pregel+/GPS	GraphLab	Giraph
Graph traversal		Pregel+	GPS	GraphLab/Giraph
Multi-phase	SV	Pregel+	GPS	Giraph
	BMM	Pregel+	GPS/Giraph	GraphLab
Graph mutation		Pregel+	GPS	Giraph

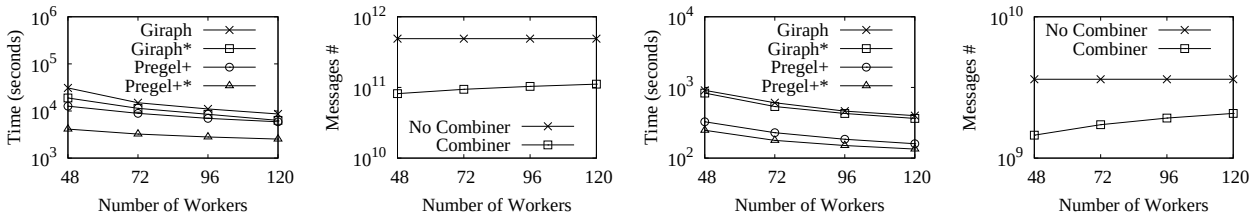
Figure 8: Overall performance on different algorithms

Conclusions. Figure 8 gives a rough ranking on the overall performance of different systems on different types of algorithms. In summary, the results of this experiment show that for most types of algorithms, including graph-traversal algorithms, multi-phase algorithms, and graph mutation, Pregel+ has much superior performance than the other existing systems. GPS also has reasonably good performance on these types of algorithms and is generally faster than Giraph and GraphLab. However, we note that the asynchronous GraphLab can outperform its synchronous counterpart as well as GPS and Giraph, especially for processing graphs with a large diameter, but its performance can be worse than the synchronous version for processing graphs with a small diameter. Nevertheless, asynchronous GraphLab allows much faster convergence for algorithms like PageRank than synchronous systems.

5.3 Effect of Message Combiner

We first assess the effect of using message combiner, which is a technique to help reduce the total number of messages that transmit through the network, when commutative and associative operations are to be applied to the messages. However, GPS does not perform sender-side message combining, as the authors claim that very small performance difference can be observed whether combiner is used or not [21]. To verify whether this claim is valid, we first analyze how many messages can be combined by applying a message combiner as follows.

THEOREM 1. *Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we assume that the vertex set is evenly partitioned among M machines (i.e., each machine holds n/M*



(a) PageRank on WebUK (b) PageRank on WebUK (c) SSSP on Friendster (d) SSSP on Friendster

Figure 9: Effect of combiner in Giraph, and Pregel+ with different number of workers

vertices). We further assume that the neighbors of a vertex in G are randomly chosen among V, and the average degree $deg_{avg} = m/n$ is a constant. Then, at least $(1 - \exp\left\{-\frac{deg_{avg}}{M}\right\})$ fraction of messages can be combined using a combiner in expectation.

PROOF. Consider a machine M_i that contains a set of n/M vertices, $V_i = \{v_1, v_2, \ldots, v_{n/M}\}$, where each vertex v_j has ℓ_j neighbors for $1 \leq j \leq n/M$. Let us focus on a specific vertex v_j on M_i, and infer under what condition a combiner should be used.

Consider an application where all vertices send messages to all their neighbors in each superstep, such as in PageRank computation. Let $u \in \Gamma_{out}(v_j)$ be a neighbor of v_j. Then, if another vertex $v_k \in V_i \backslash \{v_j\}$ sends messages through the network and v_k also has u as its neighbor, then v_j's message to u is wasted since v_j's message can be combined with v_k's message to u as v_k is also in M_i. Since the neighbors of a vertex in G are randomly chosen among V, we have

$$\Pr\{u \in \Gamma_{out}(v_k)\} = \ell_k/n,$$

and therefore,

$$\begin{aligned} &\Pr\{v_j\text{'s message to } u \text{ is not combined in } M_i\} \\ = &\prod_{v_k \in V_i \backslash \{v_j\}} \Pr\{u \notin \Gamma_{out}(v_k)\} = \prod_{v_k \in V_i \backslash \{v_j\}} \left(1 - \frac{\ell_k}{n}\right). \end{aligned}$$

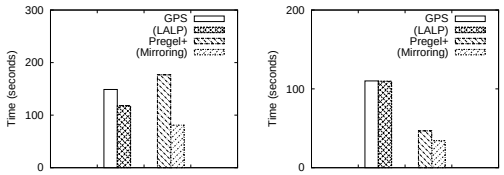
We regard each ℓ_k as a random variable whose value is chosen independently from a degree distribution with expectation $E[\ell_k] = m/n = deg_{avg}$. Then, the expectation of the above equation is given by

$$\begin{aligned} &E\left[\prod_{v_k \in V_i \backslash \{v_j\}} \left(1 - \frac{\ell_k}{n}\right)\right] = \prod_{v_k \in V_i \backslash \{v_j\}} E\left[1 - \frac{\ell_k}{n}\right] \\ = &\prod_{v_k \in V_i \backslash \{v_j\}} \left(1 - \frac{E[\ell_k]}{n}\right) = \prod_{v_k \in V_i \backslash \{v_j\}} \left(1 - \frac{deg_{avg}}{n}\right) \\ \geq &\prod_{v_k \in V_i} \left(1 - \frac{deg_{avg}}{n}\right) = \left(1 - \frac{deg_{avg}}{n}\right)^{n/M}. \end{aligned}$$

For large graphs, we have

$$\begin{aligned} &\Pr\{v_j\text{'s message to } u \text{ is not combined in } M_i\} \\ \approx &\lim_{n \to \infty} \left(1 - \frac{deg_{avg}}{n}\right)^{n/M} = \exp\left\{-\frac{deg_{avg}}{M}\right\}, \end{aligned}$$

where the last step is derived from the fact that $\lim_{n\to\infty}(1 - 1/n)^n = e^{-1}$. Therefore, the fraction of messages that can be combined using a combiner is given by $(1 - \exp\left\{-\frac{deg_{avg}}{M}\right\})$ in expectation. □



(a) Diameter Est on LJ (b) HashMin on BTC

Figure 10: Effect of LALP and mirroring

According to Theorem 1, if a large number of machines is available and the average degree is small, then indeed applying combiner may not improve the performance much as claimed by the authors of GPS [21]. For example, if $M = 1000$ and $deg_{avg} = 10$, then only 1% of the messages can be combined. However, in many applications and for many datasets (e.g., for all the algorithms we discuss in this paper and graphs with more than 5 billions of edges we used here), one may not require or use so many machines. In those situations, combiners can effectively reduce the number of messages to be sent over the network and hence improve the performance of the systems, which we verify as follows.

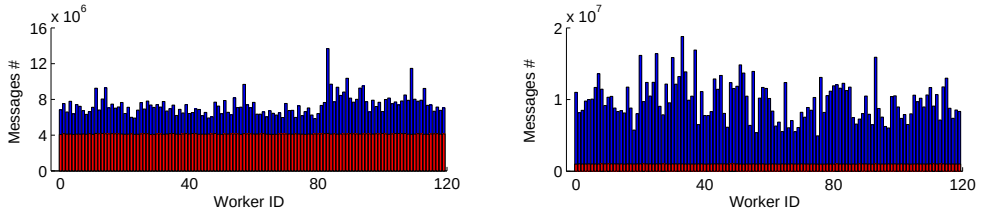
We assess the effect of combiner by testing the two systems, Giraph and Pregel+, that support combiner. We use two versions of each system, Giraph vs Giraph* and Pregel+ vs Pregel+*, where the superscript '*' indicates that combiner is applied. As shown in Figures 9(b) and 9(d), there is an obvious reduction on the total number of messages sent over the network when combiner is applied. As the number of machines increases, less messages are combined but the number is still considerably smaller than that without combiner.

Figures 9(a) and 9(c) further show that the running time of both systems by applying combiner is shorter than without combiner. In conclusion, applying combiner can always reduce the total number of messages and shorten the running time, and although the improvement may not be so obvious in some cases, there are cases the improvement is quite significant, e.g., running PageRank in Pregel+* on WebUK. This conclusion has also been verified on many other algorithms on most datasets we used, and we have not found a case that applying combiner obtains worse results than without combiner (interesting readers can compile the source codes of our algorithms for different systems and verify the results).

5.4 LALP and Mirroring

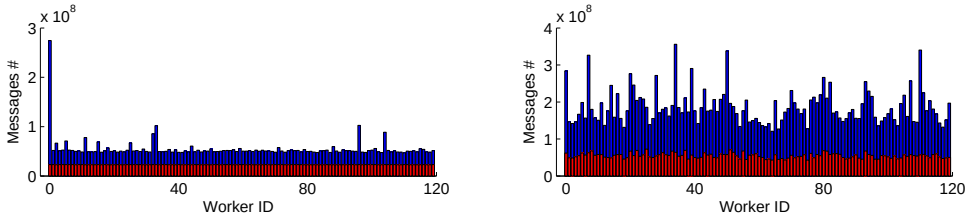
LALP in GPS and mirroring in Pregel+ (see details in Section 3) are techniques that are specifically designed to address imbalanced workload caused by high degree vertices in power-law graphs, as well as to reduce the number of messages sent over the network.

We report the performance of GPS and Pregel+, with and without LALP/mirroring, for running Diameter Estimation on LJ in Figure 10(a) and for running HashMin on BTC in Figure 10(b). The re-



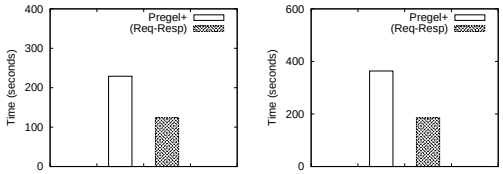
(a) Diameter Estimation on LJ (b) HashMin on BTC

Figure 11: Total number of messages sent by each worker (with/without mirroring)



(a) SV on USA (b) SV on BTC

Figure 12: Total number of messages sent by each worker, running SV (with/without request-respond) on USA and BTC



(a) SV on USA (b) SV on BTC

Figure 13: Effect of request-respond technique

sults show that applying LALP/mirroring reduces the running time in both cases. The improved performance can be explained by Figures 11(a) and 11(b), which show the total number of messages sent by each of the 120 workers of Pregel+ (note that we are not able to obtain this information for GPS but expect similar results due to the similarity in the LALP and mirroring techniques). The shorter red bars indicate the number of messages when mirroring is enabled and the longer blue bars represent the number of message sent by each worker of Pregel+ without mirroring.

We can observe that in both cases, the skewed distribution in the number of messages sent by the workers is evened by applying mirroring. In addition, there is also a significant reduction in the number of messages sent by each worker. As a result, for running Diameter Estimation on LJ, using LALP in GPS is 1.3 times faster and using mirroring in Pregel+ is 2.2 times faster than without using the techniques. For running HashMin on BTC, the reduction in running time is not as significant, which is because in the process there are only 4 supersteps that involve a large amount of redundant messages.

5.5 Request-Respond API

Apart from skewed degree distribution, imbalanced workload can also be created by algorithm logic. One example of such an algorithm is the SV algorithm for computing CCs, in which the field $D[v]$ for every vertex v in the same CC has the same value. Thus, during the computation, a vertex u may communicate with many vertices $\{v_1, v_2, \ldots, v_\ell\}$ in its CC if $u = D[v_i]$ for $1 \leq i \leq \ell$. In this case, u sends many messages which causes the performance bottleneck.

We test SV on the USA road network and the BTC graph using 120 workers (Workers 0–119). We can observe highly imbalanced communication workload among different workers, as represented by the longer blue bars shown in Figures 12(a) and 12(b), which indicate the total number of messages sent by each worker during the entire computation of SV. By applying the request-respond technique, the number of messages is significantly reduced for both datasets and the message distribution is also evened among the workers, as shown by the short red bars in Figures 12(a) and 12(b). We remark that the imbalanced communication workload is caused by the logic of SV instead of skewed vertex degree, since the largest vertex degree of the USA road network is merely 9 but the message distribution is highly skewed at worker 0.

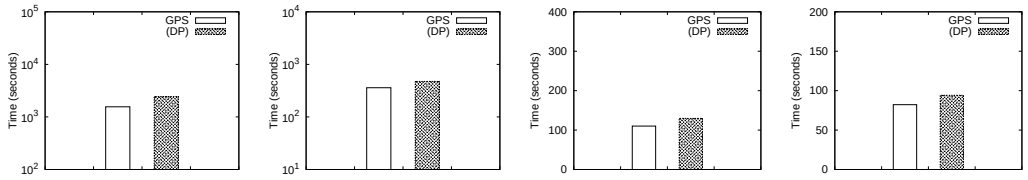
Figures 13(a) and 13(b) further show that due to the balanced workload and the reduction in the number of messages, the running time is also significantly reduced by applying the request-respond technique.

5.6 Dynamic Repartitioning

GPS also adopts a dynamic repartitioning (DP) technique to redistribute vertices across workers. The DP technique can be applied in all algorithms and on all graph types. We report the performances of PageRank on Twitter, Diameter Estimation on BTC, HashMin on BTC, and BMM on LiveJournal in Figures 14(a)-14(d). However, in all cases, DP does not obtain a good graph partition in the entire process, therefore the performances degrades due to the computational overhead of adopting the technique. As pointed in [21], the benefit of DP can only be seen on very limited settings, e.g., very large number of supersteps to run PageRank computation. Therefore, it is difficult for DP to gain performance benefit in general cases, and we also tested many other cases and failed to find a case where DP can improve the performance.

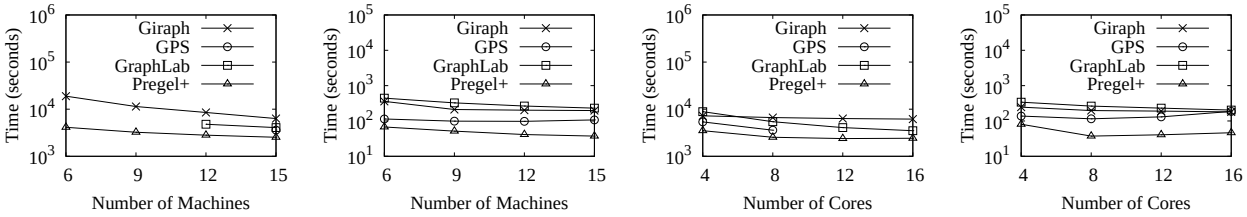
5.7 Scalability of Various Systems

We evaluate the scalability of the systems on both real-world graphs and synthetic graphs: (1) real-world graphs: we run PageRank on WebUK and HashMin on BTC, by scaling the number of machines while fixing the number of CPU cores in each machine and by scaling the number of CPU cores in each machine while fixing the number of machines; (2) synthetic graphs: we run PageR-



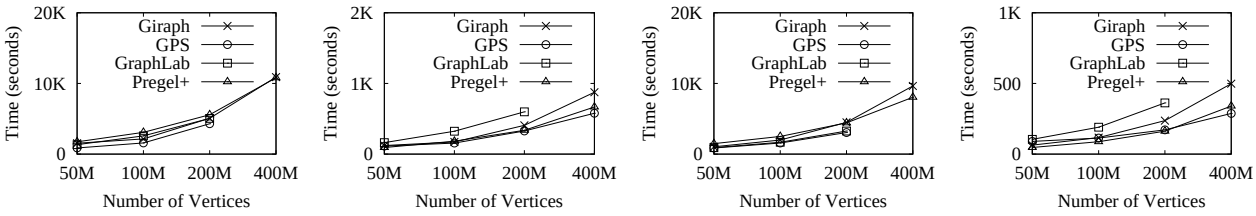
(a) PageRank on Twitter (b) Diameter Est. on BTC (c) HashMin on BTC (d) BMM on LJ

Figure 14: Effect of dynamic repartitioning



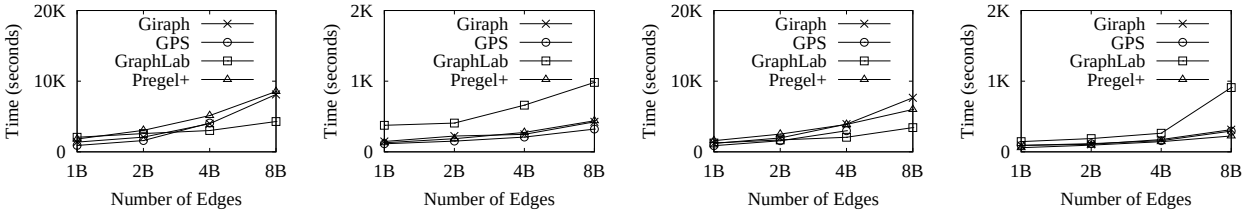
(a) PR: Effect of Machine# (b) CC: Effect of Machine# (c) PR: Effect of Core # (d) CC: Effect of Core #

Figure 15: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of machines or workers



(a) PR: Random Graphs (b) CC: Random Graphs (c) PR: Power-law Graphs (d) CC: Power-law Graphs

Figure 16: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of vertices



(a) PR: Random Graphs (b) CC: Random Graphs (c) PR: Power-law Graphs (d) CC: Power-law Graphs

Figure 17: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of edges

ank and HashMin on synthetic datasets by scaling the number of vertices while fixing the graph density and by scaling the number of edges while fixing the number of vertices.

5.7.1 Results of Different Number of Machines/Workers

We first report the performance of the systems by varying the number of machines or CPU cores (note that the number of cores is the same as the number of workers).

Scaling the number of machines. We vary the number of machines from 6 to 15 in this experiment, by setting the number of CPU cores in each machine to 8. GPS and Pregel+ run 8 workers on each machine, while Giraph and GraphLab can take advantages of all the computing resource from the 8 cores by multithreading.

For running PageRank on WebUK, Figure 15(a) shows that only Giraph scales linearly in the increase in the number of machines, though Giraph's running time is significantly longer than that of other systems. Pregel+ scales almost linearly (note that the figure is in logarithmic scale) and its running time is the shortest in all settings. For GraphLab, we can only obtain the result when there are at least 12 machines in the cluster, as the aggregate memory is not sufficient for running GraphLab on this large web graph when there are less than 12 machines. The situation is similar for GPS, but for the smaller BTC graph, we obtain their results for all cases as reported in Figure 15(b). For processing BTC, Giraph, GraphLab, and Pregel+ all scale linearly in the number of machines, but GPS's running time does not change much as the number of machines increases.

Scaling the number of CPU cores. We vary the number of CPU cores in each machine from 4 to 16 in this experiment, by setting the number of machines in the cluster to 15. The number of workers

for GPS and Pregel+ is the same as the number of CPU cores.

For running PageRank on WebUK, Figure 15(c) shows that only GraphLab scales sub-linearly in the increase in the number of CPU cores, as GraphLab can take advantage of multithreading. Giraph also uses multithreading but the effect does not seem to be obvious. The running time of GPS and Pregel+ decreases considerably (about 1.5 times) when the number of workers in each machine increases from 4 to 8, but further increasing the number of workers does not help the performance since the overhead of network communication also increases with the number of workers.

For processing BTC, Figure 15(d) shows that multithreading in Giraph and GraphLab becomes even less effective since the dataset is much smaller. Pregel+ scales only linearly when the number of workers in each machine doubles from 4 to 8, while the performance of GPS even degrades when the number of workers in each machine increases.

5.7.2 Results of Different Number of Vertices/Edges

We now report the performance of the systems by varying the number of vertices and edges using synthetic random graphs [17] and power-law graphs [5]. We set the number of machines in the cluster to be 15 and the number of CPU cores in each machine to be 8.

Scaling the number of vertices. We vary the number of vertices in the synthetic graphs from 50M to 400M, by fixing the average vertex degree of each graph to 20. As shown in Figures 16(a)-16(d), the running time of all the systems increases approximately linearly as the number of vertices increases linearly. However, GPS ran out of memory for running PageRank on the two largest graphs, while GraphLab ran out of memory for running both PageRank and HashMin on the two largest graphs. Giraph and Pregel+ can run on all graphs, but Giraph scales more poorly than Pregel+ when the graph becomes larger.

Scaling the number of edges. We fix the number of vertices in each graph to 100M, and vary the average vertex degree in the synthetic graphs from 10 to 80, i.e., the number of edges changes from 1 billion to 8 billion. For this set of experiments, Figures 17(a)-17(d) show that the running time of Giraph, GPS, and Pregel+ increases slower than the increase in the number of edges, which indicates that the systems have good scalability (except for GPS which ran out of memory for running PageRank on the two largest graphs). For GraphLab, it has the best scalability among all systems for running PageRank, but it also records the worse scalability for running HashMin.

6. CONCLUSIONS

We conducted extensive experiments to evaluate the performance of Giraph [4], GraphLab/PowerGraph [14, 9], GPS [21], and Pregel+ [2] with respect to various graph characteristics, algorithm categories, various optimization techniques, and system scalability. While we observed that there is no single system which has superior performance in all cases, our results do suggest a preference of systems for processing specific graphs (see Figure 5) and for running certain categories of algorithms (see Figure 8). We also found that combiner can always improve performance, LALP/mirroring and request-respond techniques are effective in workload balancing and in message reduction, while dynamic repartitioning degrades performance due to large overhead. In addition, our results also reveal that different systems scale differently w.r.t. the increase in the number of machines, workers, vertices and edges in different graphs. We will release all codes and information of our empirical study, and hope our results can provide useful insights for users, researchers and system developers in using existing systems and designing new systems.

7. REFERENCES

[1] Blogel. `http://www.cse.cuhk.edu.hk/blogel/`.

[2] Pregel+. `http://www.cse.cuhk.edu.hk/pregelplus/`.

[3] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High speed switch scheduling for local area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, 1993.

[4] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[5] D. A. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[6] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[7] Y. Bu. Pregelix: dataflow-based big graph analytics. In *SoCC*, 2013.

[8] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *SODA*, pages 516–525, 1994.

[9] G. J. E, L. Yucheng, G. Haijie, B. Danny, and G. Carlos. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[10] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[11] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency - Practice and Experience*, 12(12):1131–1146, 2000.

[12] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011.

[13] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

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

[15] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[16] Z. Matei, C. Mosharaf, D. Tathagata, D. Ankur, M. Justin, M. Murphy, F. M. J, S. Scott, and S. Ion. In *NSDI*, pages 2–2, 2012.

[17] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT*, pages 331–342, 2011.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[19] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: a fast and scalable tool for data mining in massive graphs. In *KDD*, pages 81–90, 2002.

[20] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. In *ICDE*, pages 50–61, 2013.

[21] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, page 22, 2013.

[22] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[23] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD Conference*, pages 505–516, 2013.

[24] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[25] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

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

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: a resilient distributed graph system on spark. In *GRADES*, 2013.

[28] D. Yan, J. Cheng, Y. Lu, and W. Ng. Practical pregel algorithms for massive graphs. *Technical Report (http://www.cse.cuhk.edu.hk/pregelplus/papers/ppa.pdf)*, 2013.