Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation

[Technical Report]

Yi Lu, James Cheng, Da Yan, Huanhuan Wu
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{ylu, jcheng, yanda, hhwu}@cse.cuhk.edu.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 extensive experiments to evaluate the performance of existing systems on graphs with different characteristics and on algorithms with different design logic. We also study 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 [17], Giraph [3], GraphLab [16], PowerGraph [11], GraphX [29], Mizan [14], GPS [23], Giraph++ [28], Pregelix [6], Pregel+ [31], 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 [17], 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 [11] and the block-centric models in [28, 1], to address various limitations in the vertex-centric systems. However, these models also suffer from other problems such as longer graph partitioning time.

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.

In this paper, we first give a survey on existing distributed graph computing systems in Section 3. Among these systems, we conduct comprehensive experimental evaluation on Giraph [3], GraphLab/PowerGraph [16, 11], GPS [23], Pregel+ [31], and GraphChi [15]. We do not conduct experiments on other existing systems for various reasons given in Section 3.7. In Section 4, we discuss a set of graph algorithms that are popularly used to evaluate the performance of various systems in existing works, and we classify them into five categories where each category represents a different logic of distributed algorithm design. 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 study 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. We also study the effects of a few algorithmic optimizations [24].
- To test the scalability of various systems by varying the number of machines, the number of CPU cores, the number of vertices and the number of edges in graphs with different degree distributions.

Related work. Guo et al. [12] proposed a benchmarking suite to compare the performance of various systems for distributed graph computation. Their benchmark defines a comprehensive evaluation process to select representative metrics, datasets and algorithm categories. While this is a pioneering work in benchmarking distributed graph computing systems, the authors also admit that their

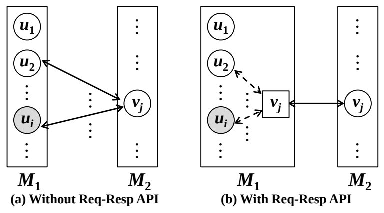


Figure 1: Illustration of Req-Resp API

	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 deletion
Pregel+	Sync	Yes	Pushing+Pulling	Mirroring + Request-respond API	No	Yes	Yes

Figure 2: Systems overview

method has limitations in terms of selection of metrics and algorithmic coverage. Satish et al. [25] evaluated the performance of a number of systems, both distributed and single-machine, and identified the potential performance gap between these systems and the hand-optimized baseline code (whose performance is close to hardware limits). The results were then used to provide insights on how the systems may be improved to better utilize hardware capacity (e.g., memory/network bandwidth). Compared with [12] and [25], we focus on the performance evaluation of vertex-centric distributed graph computing systems, among which, only GraphLab and Giraph were evaluated in [12] and [25]. We categorize graph algorithms differently (i.e., by their design logic), which leads to different conclusions on the performance of the systems. We also use more algorithms and among them, only two were used in [12] and [25]. We use more large real graphs (while only 1 large real graph in [12] and [25] has over a billion edges), and analyze the performance of the systems on four different graph characteristics. We study the effects of different system optimization techniques, as well as algorithmic optimizations [24], on the performance of the systems, which are not studied in [12] and [25]. Thus, we believe that our work is substantially different from [12] and [25], and we have made sufficient contributions in analyzing the performance of vertex-centric distributed graph computing systems.

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 [17] is implemented in C/C++ and 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 of execution, 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.

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. Giraph is implemented in Java. 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 with the number of processors can be observed by multi-threading 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 [23] is another open source Java 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+ [31] 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. Mirroring is similar to LALP in GPS; but Pregel+ integrates both mirroring and message combiner, and the Pregel+ system selects vertices for mirroring based on a cost model that analyzes the tradeoff between mirroring and message combining. Thus, the integration of mirroring and message combiner in Pregel+ leads to significantly more effective message reduction than applying combiner alone. Pregel+ also supports an additional API that allows mirroring to be used in applications where message values depend on the edge fields (e.g., single-source shortest path computation), which is not supported by LALP in GPS.

Request-Respond API. This API allows a vertex u to request another vertex v for a value, $a(v)$, 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 (e.g., as Figure 1 shows, requests from all u_i in machine M_1 are merged into one request sent to v_j in machine M_2). Suppose that there are x vertices requesting $a(v)$ from a vertex v, then the request-respond paradigm reduces the total number of messages from $2x$ (x messages for requests and x messages 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

GraphLab 2.2. (which includes PowerGraph) is implemented in C/C++. Unlike Pregel's *synchronous data-pushing model* and *message passing paradigm*, GraphLab [16] 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 [11], 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 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 vertex set, 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 GraphChi

GraphChi [15] is implemented in C/C++, which is a single-machine system that can process massive graphs from secondary storage. In addition to the vertex-centric model, GraphChi introduces two new techniques to process large graphs in a single PC.

Out-of-core computation. An innovative out-of-core data structure is used to reduce the amount of random access to secondary storage. The parallel sliding windows algorithm partitions the input graph into subgraphs, called shards. In each shard, edges are sorted by the source IDs and loaded into memory sequentially.

Selective scheduling. GraphChi supports selective scheduling in order to converge faster on some parts of the graph, particularly when the change on values is significant. Each vertex in the *update*() function (similar to *apply*() in GraphLab) can add its neighbors to the scheduler and conduct selective computation.

3.7 Other Systems

In this paper, we conduct experimental evaluation on Giraph [3], GraphLab/PowerGraph [16, 11], GPS [23], and Pregel+ [31], 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 [14] 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 [6], using Hyracks[5] 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 [29] is a graph parallel system, and it can support GraphLab and Pregel abstractions. Since GraphX is built upon the more general data parallel Spark system [18], the end-to-end performance of pipelined jobs can be superior when they are all implemented in Spark. Consider the task of first extracting the link graph from Wikipedia, and then computing PageRank on the link graph. Compared with implementing and running both jobs in Spark (with the second job done by GraphX), the traditional method of running the first (second) job by Hadoop (by any vertex-centric system that we evaluate) is more costly since it requires that the output of the first job be dumped to HDFS and reloaded by the second job. However, if only graph computation time is considered, GraphX is generally slower than GraphLab as reported in [29]. Since we focus on the performance of graph computing, we do not include GraphX in our experiments.

Trinity [26] 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 [28, 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 [3, 23, 11, 31]. 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* [20] in synchronous computation (e.g., in Pregel-like systems) and *Diameter Estimation* [13].

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* [24] 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* [22] 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* [2], there are four supersteps to simulate a three-way handshake in each phase. The *SV* [27] 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 current graph iteratively is an algorithm in this category. As a result, platforms which do not support edge deletion, such as GraphLab and PowerGraph, cannot straightforwardly support these algorithms. We choose *Graph Coloring* [24] 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* [20], *Diameter Estimation* [13], *Single-Source Shortest Paths (SSSP)* [8], *HashMin* [22] and *Shiloach-Vishkin's algorithm (SV)* [27] for computing *Connected Components (CC)*, *Bipartite Maximal Matching (BMM)* [2], and *Graph Coloring (GC)* [24].

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 [17] 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 a 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 [9], which was also used in the ANF algorithm [21].

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 [22] 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 [27] can be translated into a Pregel algorithm which requires $O(\log n)$ supersteps [30], 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 [17] 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 [10]. 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 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.

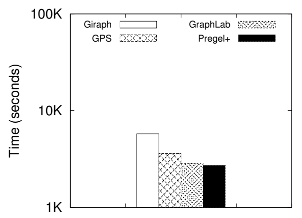
4.2 Algorithmic Optimizations

Apart from algorithm categorization, we also describe three algorithmic optimizations [24] here that can improve the performance of distributed graph computing systems on certain algorithms.

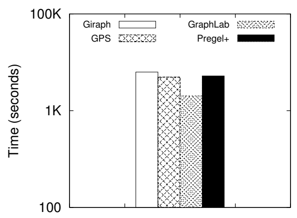
Finishing Computations Serially (FCS). Some algorithms may need a large number of supersteps to converge, even though the later supersteps are merely executing on a small fraction of the graph, called the *active-subgraph*. FCS monitors the size of the active-subgraph and sends it to the master for serial computation as soon as the size is below a threshold (5M edges by default [24]), so as to terminate the computation earlier without going a prolonged number of supersteps. The results computed in the master are then sent back to the workers. FCS can be applied to algorithms in which an inactive vertex will not be activated again in later process. Among the algorithms we discussed, only BMM and GC have this property.

Edge Cleaning On Demand (ECOD). Edge cleaning in an algorithm removes edges from the graph. ECOD delays the operation of edge cleaning and regards the edges as *stale edges* until they are involved in later computation where they are demanded to be removed.

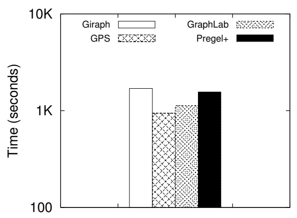
Single Pivot (SP). SP is a heuristic for speeding up the computation of connected components (CC). SP first samples a vertex v called the *pivot*, and runs the cheaper BFS algorithm instead of the CC algorithm from v. Then, a standard CC algorithm is run on the graph excluding the CC that contains v. Theoretically, v has a higher probability to be in the giant CC of the graph, and the graph excluding the giant CC can be much smaller.



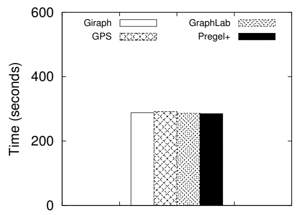
(a) PageRank on WebUK



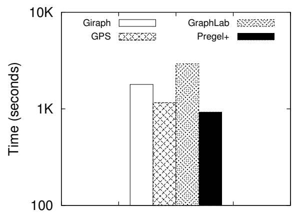
(b) PageRank on Friendster



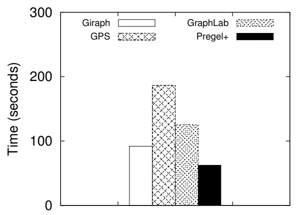
(c) PageRank on Twitter



(d) PageRank on LJ

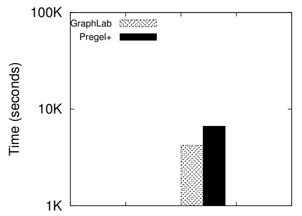


(e) PageRank on BTC

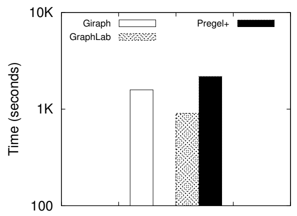


(f) PageRank on USA

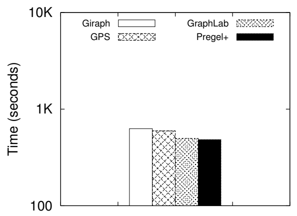
Figure 3: Performance of PageRank on Different Graphs using Giraph, GPS, GraphLab, and Pregel+



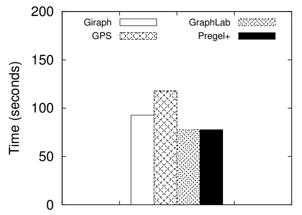
(a) Diameter Est on WebUK



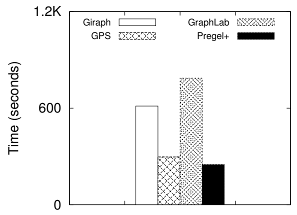
(b) Diameter Est on Friendster



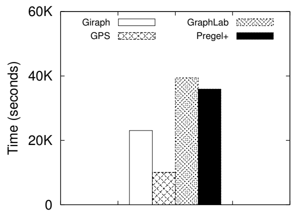
(c) Diameter Est on Twitter



(d) Diameter Est on LJ

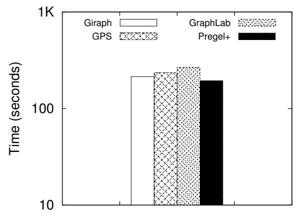


(e) Diameter Est on BTC

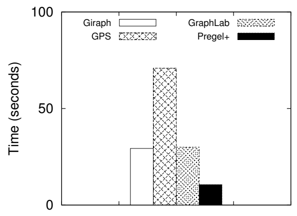


(f) Diameter Est on USA

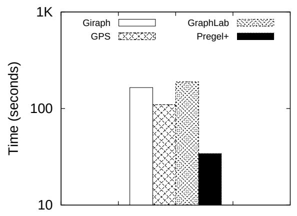
Figure 4: Performance of Diameter Estimation on Different Graphs using Giraph, GPS, GraphLab, and Pregel+



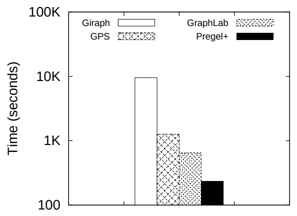
(a) HashMin on Friendster



(b) HashMin on LJ



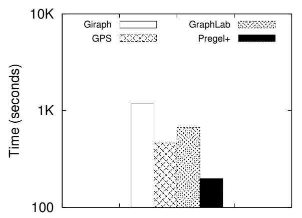
(c) HashMin on BTC



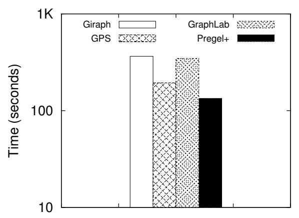
(d) HashMin on USA

Figure 5: Performance of HashMin on Different Graphs using Giraph, GPS, GraphLab, and Pregel+

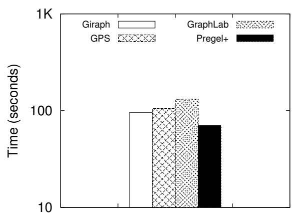
.
.



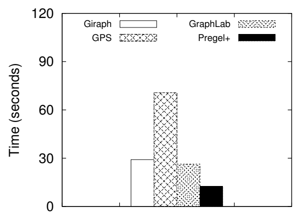
(a) SSSP on WebUK



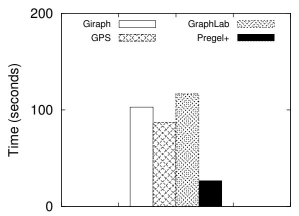
(b) SSSP on Friendster



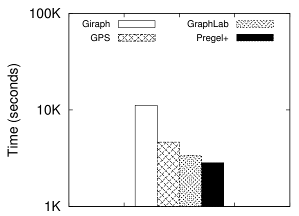
(c) SSSP on Twitter



(d) SSSP on LJ

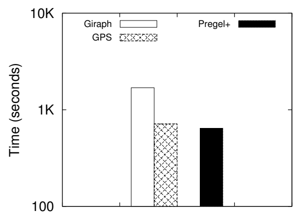


(e) SSSP on BTC

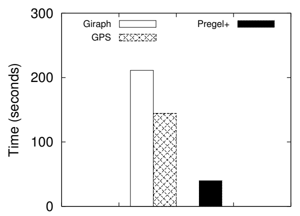


(f) SSSP on USA

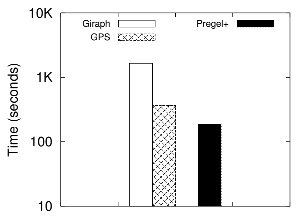
Figure 6: Performance of SSSP on Different Graphs using Giraph, GPS, GraphLab, and Pregel+



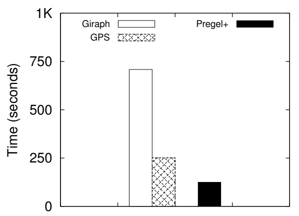
(a) SV on Friendster



(b) SV on LJ



(c) SV on BTC



(d) SV on USA

Figure 7: Performance of SV on Different Graphs using Giraph, GPS, and Pregel+

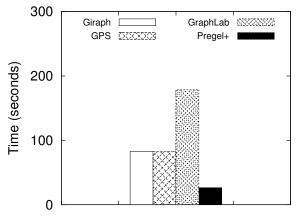
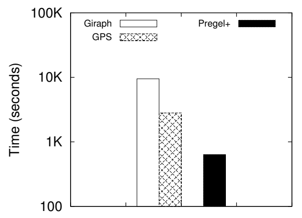
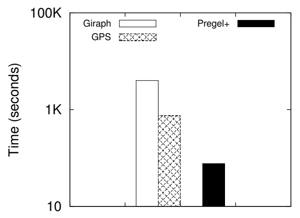


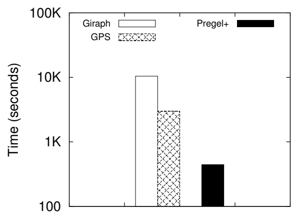
Figure 8: Performance of BMM on LiveJournal using Giraph, GPS, GraphLab, and Pregel+



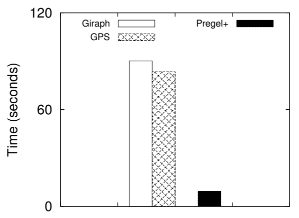
(a) Coloring on Friendster



(b) Coloring on LJ



(c) Coloring on BTC



(d) Coloring on USA

Figure 9: Performance of Graph Coloring on Different Graphs using Giraph, GPS, and Pregel+

. .

5. EXPERIMENTAL EVALUATION

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

Datasets. We used six large real-world datasets, which are from four different domains as shown in Figure 10: (1)web graph: 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, Twitter and BTC have a small diameter.

We also used synthetic datasets for scalability tests, where we generate power law graphs using Recursive Matrix (R-MAT) model [4] and random graphs using PreZER algorithm [19].

	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 10: Datasets

Experimental settings. We ran our experiments on a cluster of 15 machines, each with 48 GB DDR3-1,333 RAM, two 2.0GHz Intel(R) Xeon(R) E5-2620 CPU, a SATA disk(6Gb/s, 10k rpm, 64MB cache) and 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+, GraphChi (2014.5.19) and GraphLab 2.2 are compiled using GCC 4.4.7 with -O2 option enabled, and MPICH 3.0.4 is used.

Unless otherwise stated, we use all 15 machines for all distributed systems for all experiments; and we use 8 cores in each machine for GraphChi, Giraph and GraphLab which run with multithreading, and 8 processes (also using 8 cores) in each machine for GPS and Pregel+. There is no limit set on the amount of memory each system can use, i.e., all the systems have access to all the available memory (48GB) in each machine. GPS's polling time is set to 10 ms[7] in order to reduce the fixed overhead in each superstep (note that the default setting can lead to an overhead hundreds of times larger), while the threshold for LALP is set to 100[8]. The fault tolerance mechanisms of all the systems are off by default. All other settings, if any, of the systems are as their default. All running time reported is the wall-clock time elapsed during loading and computing, but not including dumping since this is identical for all the distributed systems.

Objectives. 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 Overview of Different Systems

We first evaluate (1)the performance of the various systems w.r.t. different algorithm categories (discussed in Section 4), (2)the performance of the various systems on graphs with different characteristics, and (3)the performance of a distributed system compared with that of a single-machine system. The results to be presented in Subsections 5.1.1–5.1.3 give readers an overview on the performance of the various systems.

In the experiments in Sections 5.1.2–5.1.3, we enable the techniques of the various systems that give the best performance, while in Sections 5.2–5.5 we analyze the effects of each individual technique.

We run the systems on every algorithm-graph combination that makes sense (except for HashMin, SV, and Coloring which are applicable on undirected graphs only, and Bipartite Maximal Matching which runs on bipartite graphs only). We also do not report all system-algorithm combinations, since it is not clear how pointer

[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`

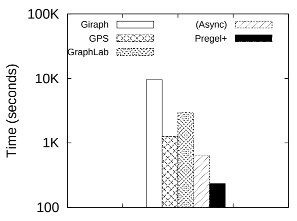
[4]`http://konect.uni-koblenz.de/networks/twitter_mpi`

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

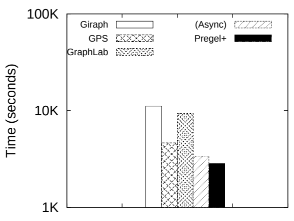
[6]`http://www.dis.uniroma1.it/challenge9/download.shtml`

[7]In GPS, each worker has a thread that periodically checks whether all of the messages in the current superstep have been received, in order to decide whether to begin the next superstep or to terminate. The period of this checking is called the polling time. If the polling time is too small, then the checking will take up too much CPU resource. If the polling time is too long, then GPS will have a large fixed overhead per superstep. The default polling time in GPS is set to 1000ms, which is really too large since most graph applications can finish the job of a superstep in less than 10ms (if it takes longer, setting a smaller polling time, e.g., 10ms, also will not waste too much resource). We also remark that this polling time should be a parameter set by the system administrator instead of a user who does not know such system details. In our opinion, the default value should be better fixed at 10ms, which was also confirmed with the author.

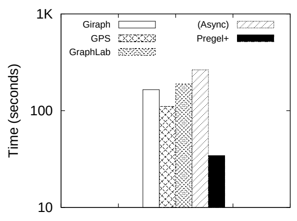
[8]It is verified in the technical report of GPS [23], `http://infolab.stanford.edu/gps/gps_tr.pdf`, that 100 is a good value for LALP.



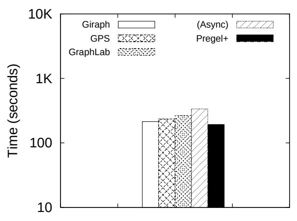
(a) HashMin on USA



(b) SSSP on USA



(c) HashMin on BTC



(d) HashMin on Friendster

Figure 11: Performance of asynchronous computing (in GraphLab) and synchronous computing

jumping in SV and edge deletion in Graph Coloring can be implemented in GraphLab. We report the results in Figures 3–9, which will be analyzed in both Sections 5.1.1 and 5.1.2.

5.1.1 Performance on Different Algorithms

We first analyze the performance of the various systems w.r.t. different algorithm categories.

Performance on always-active algorithms. Figures 3 and 4 report the performance of the systems on two representative always-active algorithms, i.e., synchronous PageRank and Diameter Estimation. It is difficult to draw a clear conclusion on which system has the best performance. Overall, GPS and Pregel+ have better performance in most cases. Between Giraph and GraphLab, GraphLab has the best performance in some cases while Giraph ran out of memory for diameter estimation on WebUK.

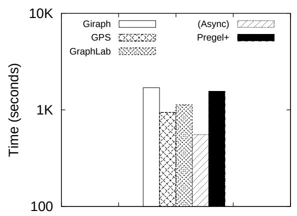
Performance on graph-traversal algorithms. Figures 5 and 6 report the performance of the systems on two representative graph-traversal algorithms, i.e., HashMin and Single-Source Shortest Paths (SSSP). The results show that Pregel+ clearly outperforms all the other systems. GPS has better performance than Giraph and GraphLab in most cases. Giraph and GraphLab have comparable performance.

Performance on multi-phase algorithms. Figures 7 and 8 report the performance of the systems on two representative multi-phase algorithms, i.e., Bipartite Maximal Matching (BMM) and the SV algorithm. Pregel+ always has the best performance for this category of algorithms, while GPS is faster than Giraph and GraphLab.

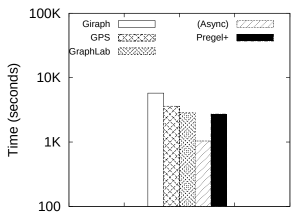
Performance on graph mutation. To test the performance on graph mutation, we use the graph coloring algorithm that first finds a maximum independent set, colors the set to a new color, and removes the induced subgraph from the graph. These procedures are repeated until all vertices are assigned colors.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

Figure 9 shows that Pregel+ is much faster than both GPS and Giraph for graph coloring, 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 mode. GraphLab supports both synchronous and asynchronous execution and we evaluate the performance as follows. First, for a graph with a large diameter, e.g., USA road network, Figures 11(a)–11(b) show that asynchronous execution is significantly faster than synchronous execution. This is because changes made to each vertex and edge during the *apply* phase in asynchronous mode are committed immediately and visible to subsequent computation; while in synchronous mode, the change commits are delayed till the end of each superstep, leading to slower convergence. However, for processing graphs with a small diameter as shown in Figures 11(c)–11(d), the overhead of locking/unlocking is not paid off by the faster convergence of asynchronous execution.



(a) PageRank on Twitter



(b) PageRank on WebUK

Figure 12: Asynchronous PageRank in GraphLab

However, for some algorithms, asynchronous execution can lead to faster convergence even for small-diameter graphs. 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. Figures 12(a)–12(b) show that PageRank takes only 554.4 seconds on the small-diameter Twitter graph using 508,251,513 updates, and takes 1,037.9 seconds on the large-diameter WebUK graph using 847,312,369 updates. However, the synchronous PageRank uses 4,679,591,698 and 11,893,340,560 updates, respectively, and are also much slower.

Overall performance. Figure 13 gives a ranking on the overall performance of the systems for different categories of algorithms. In summary, the results show that for most categories of algorithms, including graph-traversal algorithms, multi-phase algorithms, and graph mutation, Pregel+ has much superior performance than the other systems. GPS also has reasonably good performance on many of the algorithms and is generally faster than Giraph and GraphLab. An exception is that GraphLab's asynchronous mode can outperform its synchronous counterpart as well as GPS and Giraph, for processing graphs with a large diameter. In addition, GraphLab's

		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 13: Overall performance on different algorithms

asynchronous mode allows much faster convergence for algorithms like PageRank than synchronous systems.

Pregel+ and GPS have superior performance for always-active algorithms, because those algorithms generate a lot of messages, which is effectively addressed by Pregel+ and GPS's message reduction techniques (see Section 5.3). Graph-traversal algorithms also generate a large number of messages from active vertices (especially high-degree ones), usually in the preceding supersteps. Pregel+ has better performance than GPS because of its integration of mirroring and combiner, which is more effective than LALP alone in GPS. Although vertex replica in GraphLab is similar to mirroring, such replica is constructed for every vertex instead of just high-degree vertices and so the extra overhead is not paid off. For SV, Pregel+'s request-response technique is effective (see Section 5.4). But in general, no system has any specific technique to improve its performance on multi-phase algorithms, though the message reduction techniques still help to improve the performance. For graph coloring, GPS and Giraph require users 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 with vertices, which simplifies the API and enables faster edge addition/deletion.

C++ v.s. Java. Among the systems, Giraph and GPS are implemented in Java, and GraphLab and Pregel+ are implemented in C++. It is difficult to tell from our results which language, C++ or Java, leads to better performance, since in many cases the Java-based GPS has better performance than the C++-based GraphLab while in other cases a C++-based system is better than a Java-based system. Moreover, no conclusion can really be drawn based on their performance because the designs of these systems are different, and so the performance benefits may come from the design instead of the implementation language. Currently, none of the distributed systems has an implementation in both Java and C++, and hence we cannot make a comparison (based on two implementations of the same system) that gives a clear conclusion. But Java-based systems incur extra (de)serialization cost for processing objects in binary representation in memory. Moreover, the single-machine system GraphChi was implemented in both Java and C++, and [15] remarks that the Java implementation ran 2-3 times slower than the C++ implementation. For memory consumption, our result, as shown in Figure 4(a), shows that the Java-based systems ran out of memory on the large graph WebUK. GPS also ran out of memory for diameter estimation on Friendster as shown in Figure 4(b). This is mainly because a Java object takes more space than a C++ object [7]; moreover, Java uses Garbage Collector to automatically handle object deletion, which cannot keep the pool of objects small in an optimal manner and hence often leads to larger memory usage.

5.1.2 Performance on Different Graphs

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

Figure 14: Overall performance on different graphs

Next we analyze the performance of the various systems on graphs with different characteristics.

Performance on graphs with skewed degree distribution. Figures 3(a), 3(c), 3(d), 3(e), 4(a), 4(c), 4(d), 4(e), 5(b), 5(c), 6(a), 6(c), 6(d), 6(e), 7(b), 7(c), 8, 9(b) and 9(c) report the performance of the systems on graphs with skewed degree distribution (i.e., WebUK, LJ, BTC, and Twitter). The results show that Pregel+ (with the help of its mirroring 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 good performance in all the cases tested.

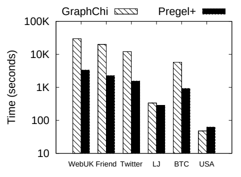
Performance on graphs with a large diameter. Figures 3(a), 3(f), 4(a), 4(f), 5(d), 6(a), 6(f), 7(d), and 9(d) report the performance of the systems on graphs with a large diameter (e.g., WebUK and USA Road). Again, each system has better performance in some cases, but overall Pregel+ has the best performance in more cases. GPS and GraphLab beat each other in roughly equal number of cases, while Giraph has the worst performance in most cases.

Performance on graphs with a small diameter. Figures 3(b), 3(c), 3(e), 4(b), 4(c), 4(e), 5(a), 5(c), 6(b), 6(c), 6(e), 7(a), 7(c), 9(a) and 9(c) report the performance of the systems on graphs with a small diameter, e.g., Friendster, Twitter and BTC. Pregel+ has the best performance in most cases, while GPS also has good performance in most cases. Giraph has poorer performance than GPS, but is better than GraphLab overall.

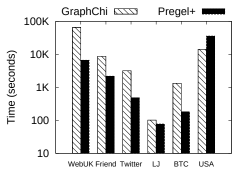
Performance on graphs with high average degree. Figures 3(a), 3(b), 3(c), 4(a), 4(b), 4(c), 5(a), 6(a), 6(b), 6(c), 7(a), and 9(a) report the performance of the systems on graphs with a relatively high average degree (e.g., WebUK, Friendster and Twitter). Pregel+ has the best performance in most cases. GPS is generally faster than GraphLab, while both of them are faster than Giraph in most cases.

Overall performance. Figure 14 gives a ranking on the overall performance of the systems for different types of graphs. According to our experiments, in most cases Pregel+ has the best performance, while GPS also has good performance on different graphs. There is rarely a case that both Pregel+ and GPS have poor performance; thus, we believe that these two systems are good choices for processing a wide range of real-world graphs. The superior performance of Pregel+ and GPS is mainly due to the use of some effective message reduction and load balancing techniques, which we analyze in Sections 5.3–5.4.

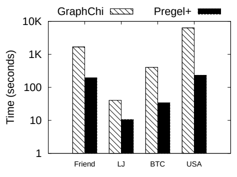
Pregel+ has the best performance in most cases mainly because of its integration of mirroring and combiner, which effectively addresses load balancing in skewed-degree graphs and reduces mes-



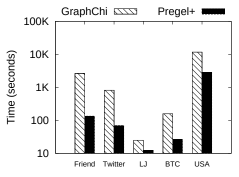
(a) PageRank



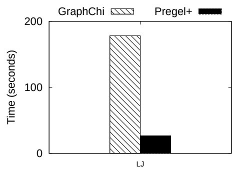
(b) Diameter Est.



(c) HashMin



(d) SSSP



(e) BMM

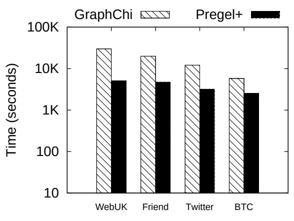
Figure 15: Performance of GraphChi v.s. Pregel+ on PageRank, Diameter Est., HashMin, SSSP and BMM

	WebUK	Friendster	Twitter	LiveJournal	BTC	USA Road
Shard #	9	4	2	1	2	1
Time	4773s	3694s	2417s	174s	774s	44s

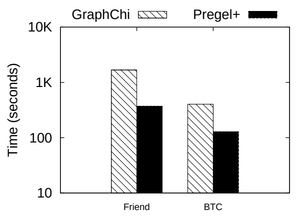
Figure 16: # of shards and preprocessing time of Graphchi

	WebUK	Friendster	Twitter	BTC
Machine #	4	3	2	2

Figure 17: Minimum number of machines required by Pregel+ to process different graphs in main memory



(a) PageRank



(b) HashMin

Figure 18: Performance of Pregel+ (with minimum number of machines) v.s. GraphChi

sages in graphs with high average degree. Similarly, GPS's LALP is also effective in addressing load balancing and in message reduction. GraphLab's vertex-cut partitioning effectively addresses load balancing, but it has an overhead of locking/unlocking (which is required even in synchronous mode, e.g., to prevent more than one vertices from scattering values to the same vertex), and hence its overall performance is not much better than Giraph (also because Giraph supports message combining while GraphLab does not). For handling large-diameter graphs which usually require a large number of supersteps, our results reveal that systems like Giraph, which have a large constant overhead per superstep, can be very slow. Pregel+ has a very small constant overhead per superstep, while GPS and GraphLab have a slightly larger constant overhead per superstep than that of Pregel+, but Giraph's is significantly larger (hundreds of ms). Finally, small diameter usually leads to faster convergence and hence for systems like GraphLab, which has a larger start-up overhead in constructing vertex replica, can be slower than other systems. We note that Pregel+ also has a smaller start-up overhead than other systems.

5.1.3 Performance Comparison with GraphChi

We also compare with a single-machine baseline, GraphChi [15]. Due to space limitation, we only report the performance of Pregel+ and GraphChi in Figures 15(a)–15(e). Performance of other systems can be compared by referring to the performance of Pregel+ and other systems in Figures 3–9.

GraphChi needs to pre-sort the graph into shards before beginning the execution of an algorithm. The number of shards for different graphs is calculated by GraphChi automatically. We report the number of shards and the preprocessing time in Figure 16, since each graph only needs to be preprocessed once.

We do not run SV and GC since it is not clear to us how pointer jumping in SV and edge deletion in GC can be implemented in GraphChi. We also note that GraphChi took much longer to run SSSP on WebUK and we killed the job after its running time is three orders of magnitude longer than Pregel+'s.

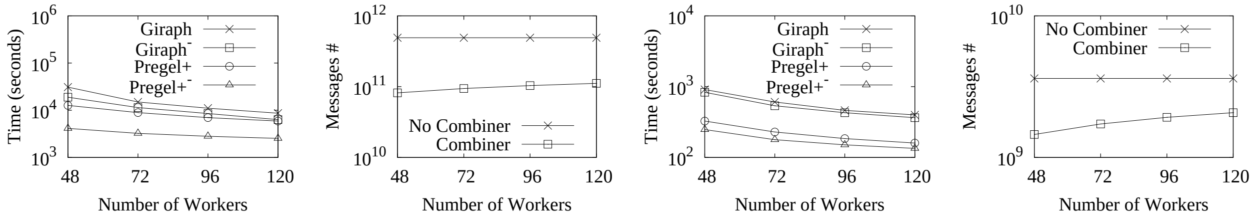
The results show that Pregel+ is about 10 times faster than GraphChi when processing the large graphs, WebUK, Friendster, Twitter and BTC. But for the two smaller graphs, LJ and USA, which can fit in the memory of a single machine, GraphChi uses fully main-memory mode and its running time is closer to that of Pregel+ (and even faster in two cases on USA). Thus, GraphChi is a reasonable choice for moderate-sized graphs. But when the graph is large and sufficient computing resources are available, a distributed system can achieve much better performance than a single-machine system. As shown in Figure 17, Pregel+ requires 4, 3, 2 and 2 machines to process WebUK, Friendster, Twitter and BTC in memory, respectively; and given such number of machines, Pregel+ is already much faster than GraphChi as shown in Figure 18. We note that Java-based systems such as Giraph and GPS may require more machines as they use more memory.

5.2 Effects of Message Combiner

We now study the effect of 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 [23]. 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 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 $1 \le j \le n/M$. Let us focus on a specific vertex v_j on M_i, and infer under what condition a combiner should be used.



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

Figure 19: Effects of combiner in Giraph and Pregel+ with different number of workers

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,

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

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

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

For large graphs, we have

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

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. □

According to Theorem 1, if a large number of machines are available and the average degree is small, then indeed applying combiner may not improve the performance much as claimed by the authors of GPS [23]. 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 thousands of machines. When M is smaller, 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 for each system, Giraph vs Giraph$^-$ and Pregel+ vs Pregel+$^-$, where the superscript '$^-$' indicates that combiner is not applied. As shown in Figures 19(b) and 19(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 19(a) and 19(c) further show that the running time of both systems with combiner is shorter than that without combiner. In conclusion, applying combiner can always reduce the total number of messages and shorten the running time. Although the improvement is not so obvious in some cases, there also exist cases where the improvement is quite significant, e.g., running PageRank in Pregel+$^-$ on WebUK. This conclusion has also been verified on many other algorithms on the datasets we used, and we have not found a case where applying combiner leads to worse performance than without combiner.

5.3 Effects of 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 20 and for running HashMin on BTC in Figure 21. The results show that applying LALP/mirroring reduces the running time in both cases. The improved performance can be explained by Figures 22(a)–22(b) and Figures 23(a)–23(b), which show the total number of messages sent by each of the 120 workers of GPS and Pregel+, respectively. The shorter red bars indicate the number of messages when LALP/mirroring is enabled and the longer blue bars represent the number of message sent by each worker of GPS/Pregel+ without LALP/mirroring.

We can observe that in both cases, the skewed distribution in the number of messages sent by the workers is evened by applying LALP/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 the number of supersteps that involve a large number of redundant messages is only 4, and so message reduction is significantly smaller than in Diameter Estimation where many supersteps involves a large number of redundant messages.

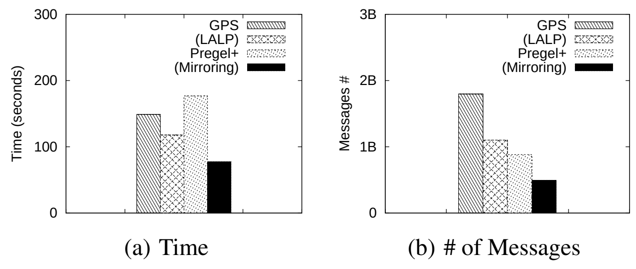


Figure 20: Diameter Est on LJ (LALP and mirroring)

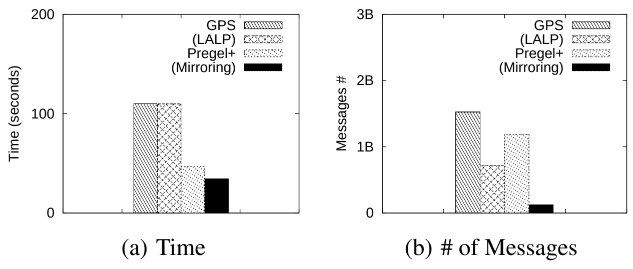


Figure 21: HashMin on BTC (LALP and mirroring)

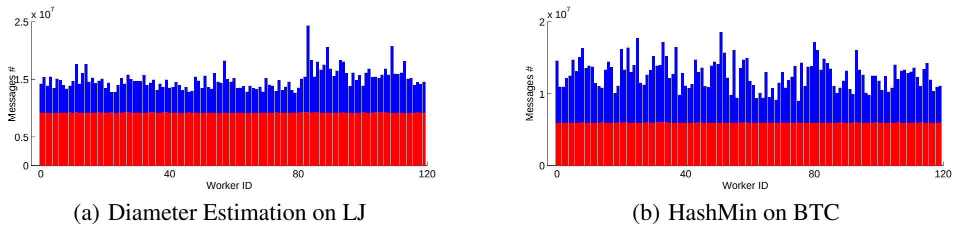


Figure 22: Total number of messages sent by each worker of GPS (with/without LALP)

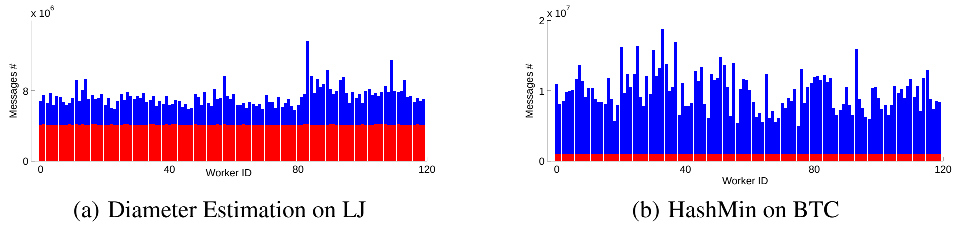


Figure 23: Total number of messages sent by each worker of Pregel+ (with/without mirroring)

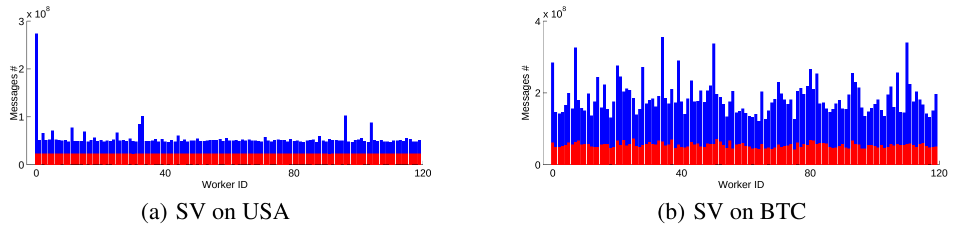


Figure 24: Total number of messages sent by each worker of Pregel+, running SV (with/without request-respond) on USA and BTC

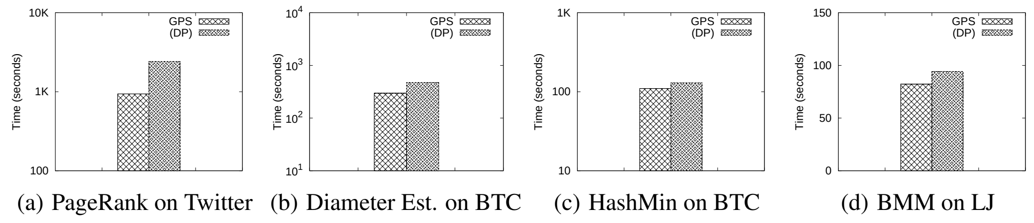
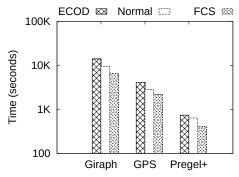


Figure 26: Effects of dynamic repartitioning

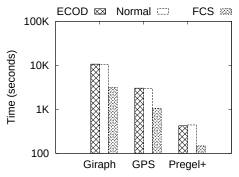
5.4 Effects of Request-Respond API

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.

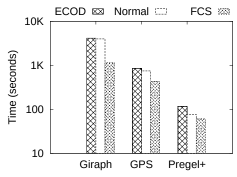
We test SV on the USA road network and the BTC graph using



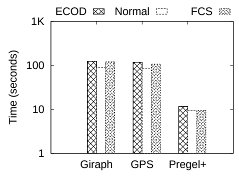
(a) GC on Friendster



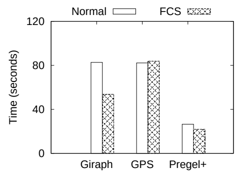
(b) GC on BTC



(c) GC on LJ

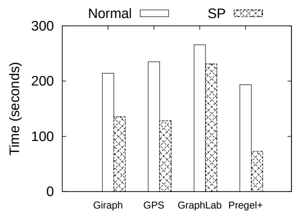


(d) GC on USA

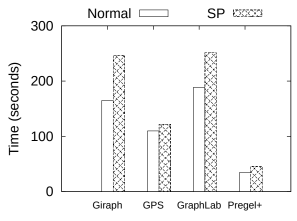


(e) BMM on LJ

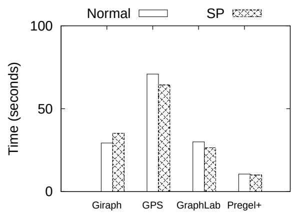
Figure 27: Effects of ECOD and FCS



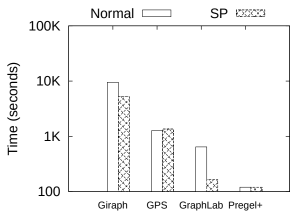
(a) CC on Friendster



(b) CC on BTC

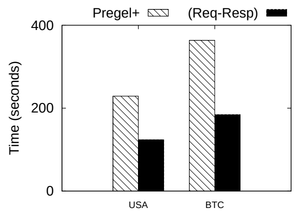


(c) CC on LJ

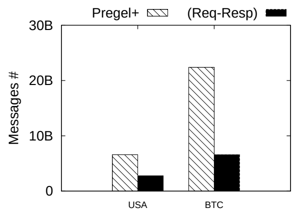


(d) CC on USA

Figure 28: Effects of SP



(a) Time



(b) # of Messages

Figure 25: Effects of request-respond API in Pregel+

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 24(a) and 24(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 24(a) and 24(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.

Figure 25(b) shows that the total number of messages is significantly reduced by applying the request-respond technique, and Figure 25(a) further shows that the running time of Pregel+ is almost halved for running the SV algorithm on both the USA road network and the BTC graph.

5.5 Effects of 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 26(a)–26(d). In all cases, DP does not obtain a good graph partition in the entire process, and therefore, the performances degrades due to the computational overhead incurred by the technique. As pointed out in [23], the benefit of DP can only be observed in very limited settings, e.g., very large number of supersteps of running PageRank computation. Therefore, it is difficult for DP to gain performance benefit in general, and we have tested many cases and have not found a case where DP can improve the performance.

5.6 Effects of Algorithmic Optimizations

We next study the effects of the algorithmic optimizations described in Section 4.2.

Figures 27(a)–27(e) report the effects of applying FCS and ECOD in Giraph, GPS and Pregel+. Note that FCS and ECOD cannot be applied in GraphLab since it does not support edge deletion and Pregel-like aggregator. The results show that FCS considerably improves the (normal) performance of Graph Coloring (GC) in all the systems in all cases except on USA, which is because GC on USA only takes hundreds of supersteps to converge and hence the overhead to send the active subgraph to the master machine exceeds the benefit gained by FCS, while GC takes thousands of supersteps to converge on other graphs and hence FCS at the master machine can significantly reduce the number of supersteps. Figure 27(e) shows the effects of FCS on BMM, which shows that the performance of both Giraph and Pregel+ is improved. On the other hand, the results show that ECOD degrades the performance. This is mainly because GC removes every stale edge in later computation and hence ECOD does not reduce the total workload. Moreover, ECOD incurs additional overhead. Thus, ECOD is only effective in algorithms in which many stale edges will not be touched again after they are decided to be removed [24].

Figures 28(a)–28(d) report the effects of applying SP in different systems for computing connected components (CC), where we use HashMin to compute CCs in the remaining graph after BFS from the pivot. The performance of the systems on Friendster is significantly improved, but degrades on BTC. The reason is that, all vertices in Friendster constitutes a single giant CC, and so no matter which vertex is chosen as the pivot, the computation will terminate after running BFS from the pivot. In other words, less costly BFS is ran on the whole graph instead of HashMin. However, the largest component of BTC consists of only around 3% of all the vertices. Thus, SP can only label a small fraction of the graph and the overhead exceeds the gain obtained by SP. Figures 28(c)–28(d) show the effects of SP for processing the LJ and USA graphs, in which the performance of the systems is improved in most cases since the largest connected component in LJ consists of more than 98% vertices and there is only one connected component in USA.

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 PageRank 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 Effects of Number of Machines/CPU-Cores

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 processes).

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

We first consider PageRank on WebUK, Figure 29(a) shows that only Giraph scales linearly with the number of machines, though it is significantly slower than the other systems. Pregel+ scales almost linearly (note that Figures 29(a)–29(d) are in logarithmic scale) and it is the fastest system in all settings. For GraphLab, we can only obtain the results when there are at least 12 machines in the cluster, as the total aggregate memory of the cluster 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 29(b). For processing the BTC graph, Giraph, GraphLab, and Pregel+ all scale linearly with the number of machines, but GPS's running time does not change much as the number of machines increases.

Effects of CPU core number. We vary the number of CPU cores in each machine from 4 to 16 in this experiment, and fix the number of machines in the cluster to 15. The number of processes in GPS and Pregel+ is the same as the number of CPU cores.

For running PageRank on WebUK, Figure 29(c) shows that only GraphLab scales sub-linearly with the number of CPU cores. This is because GraphLab can take advantage of multithreading. Giraph also uses multithreading but the effect is not obvious. The running time of GPS and Pregel+ decreases considerably (about 1.5 times) when the number of processes in each machine increases from 4 to 8, but further increasing the number of processes does not improve the performance since the overhead of network communication also increases with the number of processes.

For processing BTC, Figure 29(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 processes in each machine doubles from 4 to 8, while the performance of GPS even degrades when the number of processes in each machine increases.

5.7.2 Effects of Graph Size

We now report the performance of the systems by varying the number of vertices and edges using synthetically generated random graphs [19] and power-law graphs [4]. We set the number of machines in the cluster to be 15 and the number of CPU cores in each machine to be 8.

Effects of vertex number. We vary the number of vertices in the synthetic graphs from 50M to 400M, while we fix the average vertex degree of each graph to 20. As Figures 30(a)–30(d) show, the running time of all the systems increases approximately linearly with the number of vertices. However, GPS ran out of memory when running PageRank on the two largest graphs, while GraphLab ran out of memory when running both PageRank and HashMin on the two largest graphs. Giraph and Pregel+ can run on all graphs, but Giraph has poorer scalability than Pregel+ when the graph size becomes larger.

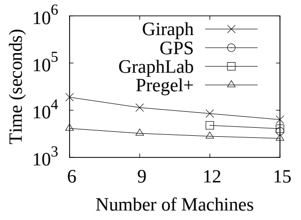
Effects of edge number. We fix the number of vertices in each graph to 100M, and vary the average vertex degree in the synthetic graphs from 10 to 80 (i.e., the number of edges changes from 1 billion to 8 billion). For this set of experiments, Figures 31(a)–31(d) show that the running time of Giraph, GPS, and Pregel+ increases sub-linearly with 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). GraphLab has the best scalability among all systems for running PageRank, but it exhibits the worse scalability for running HashMin.

6. CONCLUSIONS

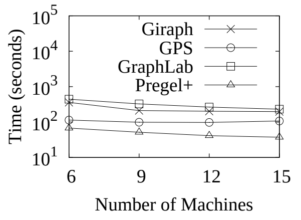
We conducted extensive experiments to evaluate the performance of Giraph [3], GraphLab/PowerGraph [16, 11], GraphChi [15], GPS [23], and Pregel+ [31] with respect to various graph characteristics, algorithm categories, various optimization techniques, and system scalability. While we observed that there 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 14) and for running certain categories of algorithms (see Figure 13). 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.

Pregel+ has better performance mainly thanks to its combination of mirroring, message combining, and request-respond techniques. GPS also benefits significantly from its LALP technique for load balancing and message reduction, but GPS does not employ message combining while we showed that combiner can always reduce messages. GraphLab uses vertex-cut partitioning for load balancing, but it incurs extra overhead in locking/unlocking, which can lead to inferior performance than other systems. We, however, remark that GraphLab's asynchronous mode can outperform its synchronous counterpart as well as GPS and Giraph. Giraph generally has the poorest performance because it does not have any specific technique for handling skewed workload and only relies on combiner to reduce messages. Finally, we also believe that the efficiency of Pregel+ also comes from its C++ implementation, which at least uses less memory, and according to [15] can be 2-3 times faster, than a Java implementation.

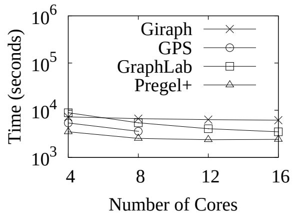
We will release all codes and information of our empirical study, and hope our results can provide useful insights for users, researchers



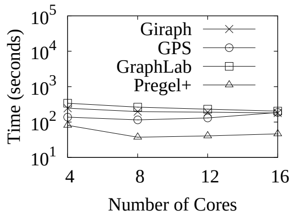
(a) PR: Effect of Machine#



(b) CC: Effect of Machine#

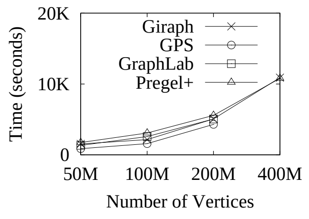


(c) PR: Effect of Core #

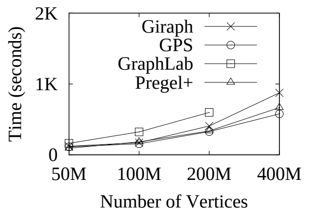


(d) CC: Effect of Core #

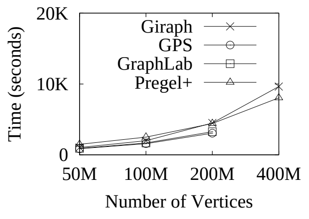
Figure 29: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of machines or CPU cores



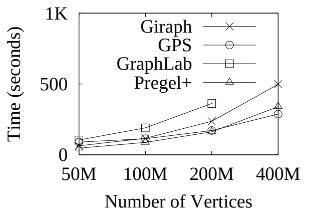
(a) PR: Random Graphs



(b) CC: Random Graphs

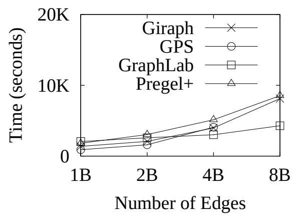


(c) PR: Power-law Graphs

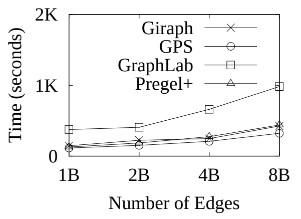


(d) CC: Power-law Graphs

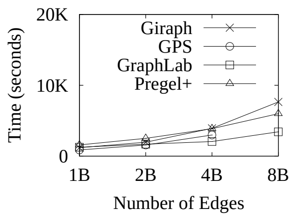
Figure 30: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of vertices



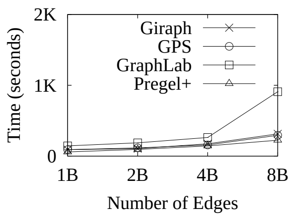
(a) PR: Random Graphs



(b) CC: Random Graphs



(c) PR: Power-law Graphs



(d) CC: Power-law Graphs

Figure 31: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of edges

and system developers in using existing systems and designing new systems.

7. REFERENCES

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

[2] 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.

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

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

[5] 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.

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

[7] Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 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] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

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

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[12] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *IPDPS*, 2014.

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

[14] 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.

[15] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[16] 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.

[17] 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.

[18] 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.

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

[20] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. 1999.

[21] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: a fast and scalable tool for data mining in massive graphs. In *KDD*, 2002.

[22] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma.

Finding connected components in MapReduce in logarithmic rounds. In *ICDE*, pages 50–61, 2013.

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

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

[25] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD Conference*, pages 979–990, 2014.

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

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

[28] 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.

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

[30] D. Yan, J. Cheng, Y. Lu, and W. Ng. Practical Pregel algorithms. *(http://www.cse.cuhk.edu.hk/pregelplus/papers/ppa.pdf)*, 2013.

[31] D. Yan, J. Cheng, Y. Lu, and W. Ng. Pregel+: Technical report. *(http://www.cse.cuhk.edu.hk/pregelplus/pregelplus.pdf)*, 2014.