# An Experimental Comparison of Pregel-like Graph Processing Systems[*]

Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, Tianqi Jin
David R. Cheriton School of Computer Science, University of Waterloo
{m25han, kdaudjee, kammar, tozsu, x36wang, t6jin}@uwaterloo.ca

## ABSTRACT

The introduction of Google's Pregel generated much interest in the field of large-scale graph data processing, inspiring the development of Pregel-like systems such as Apache Giraph, GPS, Mizan, and GraphLab, all of which have appeared in the past two years. To gain an understanding of how Pregel-like systems perform, we conduct a study to experimentally compare Giraph, GPS, Mizan, and GraphLab on equal ground by considering graph and algorithm agnostic optimizations and by using several metrics. The systems are compared with four different algorithms (PageRank, single source shortest path, weakly connected components, and distributed minimum spanning tree) on up to 128 Amazon EC2 machines. We find that the system optimizations present in Giraph and GraphLab allow them to perform well. Our evaluation also shows Giraph 1.0.0's considerable improvement since Giraph 0.1 and identifies areas of improvement for all systems.

## 1. INTRODUCTION

With the advancement and popularity of social networking and scientific computation technologies, graph data has become ubiquitous. More and more interesting problems require processing graph data for real-world applications, including business intelligence, analytics, data mining, and online machine learning. For example, Google's PageRank algorithm must determine influential vertices for more than 1 trillion indexed webpages [4]. In the case of social graphs, Facebook needs to calculate popularity or personalized ranking, determine shared connections, find communities, and perform advertisement propagation for over 900 million active users [6]. In communication and road networks, determining the maximum flow and routing transportation, respectively, both require processing large graphs [26]. Recently, scientists have also been leveraging biology graphs to understand protein interactions and pathology graphs to identify anomalies [30].

Traditionally, processing graph data is considered computationally hard because of the irregular nature of natural graphs. The Bulk Synchronous Parallel (BSP) model [37] provides a means to design parallel processing algorithms that scale with more workers. Google's Pregel [29] is a BSP implementation that provides a native API specifically for writing algorithms that process graph data. Pregel is still a simple model with many opportunities for improvement, leading to the emergence of several different graph processing frameworks. Some prominent ones include Apache Hama [2], Apache Giraph [1], Catch the Wind (CatchW) [34], GPS [32], GraphLab [28] (which now incorporates PowerGraph [18]), and Mizan [23].

The relative performance characteristics of such systems are unclear. Although each one reports performance results, they are not easily comparable. In this paper we address this issue by focusing on exposing the behaviour of different systems that are attributable to both *built-in* and *optional system optimizations*, optimizations intrinsic to the system and agnostic to the graph or algorithm used. Thus, this study exposes the intrinsic capabilities of different systems.

Since these systems vary in infrastructure, we narrow our study to open source *Pregel-like* systems: systems that use Pregel's vertex-centric approach. Hence, this excludes Hama, which is a generalized BSP framework that is neither optimized nor specialized for graphs, and CatchW, as it is not open source and its code was not available upon request. Additionally, we exclude graph database systems, as their goal is to persistently manage graph data, while graph processing frameworks address in-memory batch processing of large graphs [12].

We therefore compare Giraph, GPS, GraphLab, and Mizan. The first three are all popular systems that the research and industrial community are currently using and building upon. For example, Giraph has incorporated several optimizations from GPS, has a rapidly growing user base, and has been scaled by Facebook to graphs with a trillion edges [10]. Mizan has emerged as a competitor to these systems. Given its recency and similar graph processing objectives, it is valuable to also understand how Mizan performs in the same setting as Giraph, GPS, and GraphLab.

Our primary goals and contributions are (i) a systematic and fair comparison of four open-source Pregel-like systems (Giraph, GPS, Mizan, and GraphLab) to study how their built-in and optional system optimizations impact performance, (ii) the use of running time, memory usage, and

[*]This work was supported by an education research grant from Amazon Web Services.

*Proceedings of the VLDB Endowment,* Vol. 7, No. 12
Copyright 2014 VLDB Endowment 2150-8097/14/08.

network traffic as metrics, together with real-world datasets and diverse algorithms, to explain the performance of each system, and (iii) a discussion of the advantages and disadvantages of each system with respect to graph data processing to provide system designers and users with information that can promote the use and development of such systems.

We are aware of two existing experimental studies of Pregel-like systems [16, 19] but both are lacking in scale, in terms of dataset sizes and number of machines used. Furthermore, [16] has poor algorithm diversity and only considers one experimental metric, while [19] lacks comprehensive experimental results for all combinations of algorithms, datasets, and machines. Finally, neither study consider some of the systems that we include, such as GPS and Mizan.

This paper is organized as follows. In Section 2, we give some background on BSP and Pregel. In Section 3, we describe the systems that we test and, in Section 4, the four algorithms that we use as workloads. We detail our evaluation methodology, including datasets and metrics, in Section 5, and analyze our results in Section 6. We highlight our experiences with each system in Section 7 before concluding in Section 8.

## 2. BACKGROUND ON BSP AND PREGEL

Bulk Synchronous Parallel (BSP) is a parallel programming model with a message passing interface (MPI), developed to address the problem of parallelizing jobs across multiple workers for scalability [37]. In contrast to distributed shared memory, using MPI allows message batching, which ensures no remote reads with high latency, and avoids heavy locking, as MPI is inherently free of deadlocks and race conditions. BSP is essentially a vertex state machine where each vertex can be active or inactive at each state. The computation consists of a sequence of supersteps, with synchronization between workers occurring at superstep barriers (Figure 1). An inherent limitation of this synchronous approach is that workers can encounter the straggler problem, where fast workers must wait for the slow ones. However, BSP provides nearly full coverage over matrix computation, machine learning, and graph processing problems.

Pregel [29] is one of the first BSP implementations that provides a native API specifically for programming graph algorithms, while also abstracting away the underlying communication details. The fundamental computing paradigm Pregel employs can be characterized as "think like a vertex". Graph computations are specified in terms of what each vertex has to compute; edges are communication channels for transmitting computation results from one vertex to another, and do not participate in the computation. The computation is split into *supersteps*. At each superstep, a vertex can execute a user-defined function, send or receive messages to its neighbours (or any other vertex with a known ID), and change its state from active to inactive. Supersteps end with a synchronization barrier (Figure 1), ensuring that messages sent from one superstep are delivered at the beginning of the following superstep. A vertex may vote to halt at any superstep and is woken up when it receives a message. Pregel terminates when all vertices are inactive and no more messages are in transit.

To avoid communication overheads, Pregel preserves data locality by ensuring computation is performed on locally stored data. The input graph is loaded once at the start

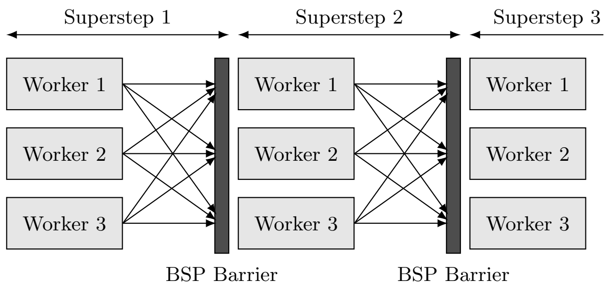

**Figure 1: Basic computation model of Pregel, illustrated with three supersteps and three workers [23].**

of a program and all computations are executed in-memory. As a result, Pregel supports only graphs that fit in memory.

Pregel uses a master/workers model. One machine acts as the *master*, while the others become *workers*. The master partitions the input graph into subgraphs and assigns each partition to a worker. The master is responsible for coordinating synchronization at the superstep barriers and for instructing workers to perform checkpointing (a means of fault-tolerance). Each worker independently invokes a `compute()` function on the vertices in its portion of the graph. Workers also maintain a message queue to receive messages from the vertices of other workers. Additionally, user-defined *combiners* can be used to combine messages (at the sender or receiver) to improve performance. For global coordination, *aggregators* can be used. Each vertex can contribute a value to an aggregator in one superstep and obtain a globally reduced value in the next superstep [29].

## 3. SYSTEMS TESTED

In this section, we detail the four systems tested, our reasons for selecting them, and our configuration choices. Specific technical details of our setup are given in Section 5.1.

Table 1 summarizes the basic properties of Pregel and the tested systems. For each system, items in bold are optional optimizations that we experiment with and without. Graph storage is the data structure used to hold the input graph in memory. In particular, CSR and CSC are the compressed sparse row and column formats respectively [25]. `master.compute()` allows sequential computations at the master. GraphLab effectively provides this functionality through its blocking and non-blocking aggregators, which can collect vertex or edge data to a central location. Finally, all systems feature combiners, aggregators, and, except for Pregel, use the Hadoop Distributed File System (HDFS).

An important issue for all systems is that of graph partitioning. To perform efficient distributed computation, the input graph must be partitioned across workers in a time-efficient manner that (1) minimizes inter-worker communication and (2) ensures a balanced workload at each worker (to minimize stragglers). This is the balanced graph partitioning problem, which is known to be NP-complete [5]. While there are approximation algorithms, such as those in METIS [21], they can take several hours to find a good partitioning for large graphs and are not time-efficient. Hence, for all systems, we use random hash partitioning. Despite being naive, it is the fastest graph partitioning method. GraphLab has other partitioning methods, but the number of machines we use do not always satisfy their required constraints.

Table 1: Properties and features of Pregel and tested systems.

| System | Language | Computation Model | Comm. Library | Graph Storage | `master.compute()` | Other Optimizations |
|---|---|---|---|---|---|---|
| Pregel | C++ | BSP | N/A | N/A | No | none |
| Giraph | Java | BSP | Netty | **Byte array, hash map** | Yes | none |
| GPS | Java | BSP | MINA | Arrays | Yes | **LALP, dynamic migration** |
| Mizan | C++ | BSP | MPICH | C++ vectors | No | dynamic migration |
| GraphLab | C++ | GAS | MPICH | CSR/CSC [25] | Yes* | **async computation** |

## 3.1 Giraph

Apache Giraph [1] is an open-source alternative to the proprietary Pregel. Giraph runs workers as map-only jobs on Hadoop and uses HDFS for data input and output. Giraph also uses Apache ZooKeeper for coordination, checkpointing, and failure recovery schemes.

We pick Giraph 1.0.0 due to its large developer and user base, which includes Facebook [10]. Giraph 1.0.0 has also undergone multiple optimizations since Giraph 0.1. These include sharded aggregators, which avoid bottlenecks at the master, and serialization of vertices, edges, and messages into byte arrays to yield substantial memory savings and reduced Java garbage collection overheads [10].

Giraph supports different data structures for vertex adjacency lists. By default, byte array is used as it is space efficient and, as our results will show, leads to faster input loading times. However, byte array edges are inefficient for graph mutations—the addition or removal of vertices or edges—because all edges are deserialized whenever an edge is removed. Hash map edges, on the other hand, are less efficient for memory but very efficient for mutations.

As distributed minimum spanning tree (DMST), one of our workloads, performs graph mutations, we test Giraph with both byte array and hash map edges. Finally, checkpointing is disabled as other systems do not perform it.

## 3.2 GPS

GPS is another open-source Pregel implementation from Stanford InfoLab [32]. GPS was reported to be 12× faster than Giraph 0.1 but, as our results will show, this gap has since narrowed due in part to Giraph 1.0.0's adoption of some of GPS's optimizations [32].

We include GPS because it is a relatively full-featured experimental system. There are many graph algorithms implemented, providing confidence in the system's correctness. GPS also features many built-in system optimizations, such as single *canonical* vertex and message objects to reduce the cost of allocating multiple Java objects [32]. Other optimizations like using per-worker rather than per-vertex message buffers (which improves network usage) and reducing thread synchronization all help improve performance [31].

GPS offers Large Adjacency List Partitioning (LALP), an optional performance optimization for algorithms that send to all of its neighbours the same message [32]. LALP works by partitioning the adjacency lists of high-degree vertices across different workers. It is beneficial for algorithms like PageRank, weakly connected components (WCC), and single source shortest path (SSSP) with unit edge weights (Section 4) but does not work for algorithms like DMST or when aggregators are involved. LALP also requires a threshold parameter whose optimal value depends on both the input graph and the number of workers [32]. As it is impractical to tune this parameter for every combination of graph, algorithm, and workers, we choose a value of 100 that we determined experimentally to work well overall.

GPS also features an optional *dynamic migration* scheme. In most systems, graph partitioning is done before, but never during, a computation. Dynamic migration repartitions the graph during computation by migrating vertices between workers, to improve workload balance and network usage. In particular, the scheme in GPS exchanges vertices between workers based on the amount of data sent by each vertex. The scheme locates migrated vertices by relabelling their vertex IDs and updating the adjacency lists in which they appear, meaning it does not work for algorithms such as DMST that send messages directly to specified vertex IDs. The scheme decreases network I/O but does not always improve computation time [32]. We experiment with dynamic migration to independently verify these findings.

Finally, GPS provides several algorithm-specific optimizations, many of which use tunable parameters that are dependent on the input graph. Not only are such optimizations dependent on the particular graph and algorithm, but they can also reduce the performance gap between systems, making it more difficult to observe differences due to built-in system optimizations. Hence, we do not use any algorithm-specific optimizations.

We run GPS with no optional optimizations, with only LALP, and with only dynamic migration.

## 3.3 Mizan

Mizan is an open-source project developed by KAUST, in collaboration with IBM Research [23]. Similar to GPS, Mizan was reported to be 2× faster than Giraph 0.1 when running in its *static* mode, where no dynamic migration is used [23]. Our study considers a more up-to-date version of both Mizan and Giraph in this same mode.

We include Mizan because it has the same graph data processing objectives as Giraph, GPS, and GraphLab. There are few algorithms implemented for Mizan, providing a good opportunity to test both the system's performance and correctness. In fact, we were able to identify and correct several bugs in the system through the course of implementing our algorithms, such as incorrect aggregator values and halted workers not waking up upon receiving a message.

Mizan also offers an optional dynamic migration scheme that is more complex than that of GPS, but Mizan does not function correctly with this scheme enabled [22]. Hence, we run Mizan with its default configuration in its static mode. Unlike the other systems, Mizan partitions the graph separately from the algorithm execution, which is an issue we discuss in Section 6.4.

## 3.4 GraphLab

GraphLab is an open-source project started at CMU [28] and now supported by GraphLab Inc. We use the latest version of GraphLab 2.2 [3], which supports distributed computation and incorporates the features and improvements of PowerGraph [18]. We include GraphLab because of its popularity and maturity for graph analytic tasks.

Unlike the previous systems, GraphLab uses the GAS decomposition (Gather, Apply, Scatter), which is similar to, but fundamentally different from, the BSP model. In the GAS model, a vertex accumulates information about its neighbourhood in the Gather phase, applies the accumulated value in the Apply phase, and updates its adjacent vertices and edges and activates its neighbouring vertices in the Scatter phase. In particular, vertices can directly pull their neighbours' data (via Gather), without having to explicitly receive messages from those neighbours. In contrast, a vertex under the BSP model can learn its neighbours' values only via the messages that its neighbours push to it. The GAS model also enables completely asynchronous execution without the need for communication barriers.

Another key difference is that GraphLab partitions graphs using vertex cuts rather than edge cuts. Consequently, each edge is assigned to a unique machine, while vertices are replicated in the caches of remote machines. This is a feature incorporated from PowerGraph, as vertex cuts allow high degree vertices to be partitioned across multiple machines, yielding better balanced workloads for graphs with skewed degree distributions [18]. In contrast, Giraph, GPS, and Mizan all perform edge cuts and do not replicate vertices.

Lastly, unlike Giraph, GPS, and Mizan, GraphLab does not fully support graph mutations. It supports adding edges, but not removal of vertices or edges.

GraphLab offers two execution modes: synchronous and asynchronous. Like BSP, the synchronous mode uses communication barriers. However, because it follows the GAS model, synchronous mode is also *adaptive*: a vertex that has converged no longer needs to participate in the computation since its neighbours can pull its last value via Gather. This avoids wasting CPU and network resources. In contrast, a converged vertex must stay active in BSP to send its neighbours messages containing its last value. The asynchronous mode, or distributed locking engine in [28], is fully asynchronous and has no notion of communication barriers or supersteps. It uses distributed locking to avoid conflicts and to maintain serializability (equivalence to some sequential execution). We use the asynchronous mode as an additional comparison point. Hence, we test GraphLab with both synchronous and asynchronous modes[1].

## 4. ALGORITHMS

We consider four categories of graph algorithms: random walk, sequential traversal, parallel traversal, and graph mutation (Table 2) [35].

*Random walk* algorithms perform computations on all vertices based on the random walk model. *Sequential traversal* algorithms find one or more paths in a graph according to some optimization criteria without updating or mutating the graph structure. Such algorithms typically start at one source vertex and, at superstep $i$, process vertices that are distance $i$ from the source. Unlike sequential traversal, *parallel traversal* algorithms start by considering all vertices at once. Multiple vertices participate in each superstep until they acquire some common information. Finally, *graph mutation* encompasses algorithms that change the graph structure during computation.

For our experiments, we select one representative algorithm from each category: PageRank for random walk, SSSP for sequential traversal, WCC for parallel traversal, and DMST for graph mutation. Table 2 summarizes how heavily these algorithms utilize CPU, memory, and network resources. We describe each algorithm next. Additional technical details can be found in Section 5.3.

## 4.1 PageRank

PageRank is an algorithm used by Google to rank webpages, based on the idea that more important websites likely receive more links from other websites. We use PageRank with a 0.85 damping factor, the same value used in [29]. This factor means that given a webpage (a vertex) the user is browsing, there is an 85% likelihood of jumping to a random webpage from the outgoing links (out-edges) of the current page and a 15% likelihood of jumping to a random webpage chosen from the entire web (the input graph).

Specifically, all vertices start with a value of 1.0. At each superstep, a vertex $u$ updates its value to $p = 0.15 + 0.85x$, where $x$ is the sum of values received from its in-edges, and sends $p/\deg^{+}(u)$ along its out-edges, where $\deg^{+}(u)$ is $u$'s outdegree. This gives the expectation value. Dividing it by the number of vertices gives the probability value.

We include PageRank as it is a popular algorithm provided in many systems. PageRank's simplicity provides a straightforward test of core system components and performance. As all vertices remain active throughout the computation, communication is also stable across supersteps.

## 4.2 SSSP

Single-source shortest path (SSSP) finds the shortest path between a given source vertex and all other vertices in its connected component. A path in an undirected graph is a sequence of adjacent vertices and is shortest if the sum of the constituent edge weights is minimized.

We use a parallel variant of the Bellman-Ford algorithm [13], which starts by setting the distance of the source vertex to 0 and all other vertices to $\infty$. Only the source vertex is active in the first superstep. Each active vertex $u$ sends all neighbours $v$ its current minimum distance plus the weight of the edge $(u, v)$. In the next superstep, vertices that receive a message become active, and each active vertex takes the minimum of its received distances. If a smaller distance is found, the vertex propagates the finding to its neighbours. The number of supersteps taken by SSSP is limited by the graph's longest shortest path.

Being a traversal algorithm, SSSP's network usage is variable: the amount of communication increases, peaks, then decreases with increasing supersteps. Hence, SSSP is the simplest algorithm that tests how well a system handles dynamically changing communication. In particular, unlike PageRank, vertices always vote to halt and are woken only by received messages. System bottlenecks are also easier to identify because communication is small in the initial supersteps and gradually increases.

[1]The chromatic engine is no longer part of GraphLab 2.2 [27].

Table 2: Properties of tested algorithms.

| Algorithm | Category | Similar Algorithms | CPU | Memory | Network |
|---|---|---|---|---|---|
| PageRank | Random walk | HITS [24] | Medium | Medium | High |
| SSSP | Sequential traversal | BFS, DFS, reachability | Low | Low | Low |
| WCC | Parallel traversal | label propagation, graph clustering | Low | Medium | Medium |
| DMST | Graph mutation | graph coarsening, graph aggregation [36] | High | High | Medium |

## 4.3 WCC

Weakly connected components (WCC) is an algorithm that finds the maximal weakly connected components of a graph. A component is weakly connected if every pair of vertices is mutually reachable when ignoring edge directions.

WCC is implemented using the HCC algorithm [20]. All vertices are initially active. Each vertex starts as its own component by setting its component ID to its vertex ID. When a vertex receives a smaller component ID, it updates its vertex value with the received ID and propagates that ID to its neighbours. Like SSSP, WCC takes at most a number of supersteps equal to a graph's longest shortest path.

As a parallel traversal algorithm, WCC helps corroborate results from SSSP under a heavier network load, without changing the input graph. Furthermore, unlike SSSP, all vertices are initially active and, unlike PageRank, vertices can halt before others. This results in network communication that decreases monotonically with increasing supersteps, allowing us to observe system performance under another distinct communication pattern.

## 4.4 DMST

Distributed minimum spanning tree (DMST) finds the minimum spanning tree (MST) of an undirected, weighted graph. This tree is unique if all edge weights are distinct. We use the parallel Boruvka algorithm [11, 33]. The input graph must be undirected but need not be connected. For an unconnected graph, DMST outputs the minimum spanning forest, a union of MSTs.

This algorithm is interesting because of its complexity. For example, the implementation in Giraph is about 1300 lines of code, compared to 100 lines for SSSP. The algorithm requires custom vertex, edge, and message data types and sends three different types of messages.

The algorithm proceeds in four phases. In the first phase, each vertex finds a minimum weight out-edge. In the second phase, vertices perform the pointer-jumping algorithm [11], by using question and answer messages to determine their *supervertex*, which represents a connected component of the graph to which its children belong. This phase can repeat for an indeterminate number of supersteps so it is synchronized using summation aggregators. In the third phase, vertices perform edge cleaning. A vertex first removes out-edges to neighbours with the same supervertex. It then modifies each remaining out-edge to point to the supervertex of the edge's destination vertex. Finally, in the fourth phase, vertices send their adjacency lists to their supervertex, which merges them according to minimum weight. Non-supervertices vote to halt, while supervertices return to the first phase as regular vertices. The graph gets smaller and smaller, with the algorithm terminating when only unconnected vertices remain.

DMST is CPU intensive and exercises a far less predictable message pattern. Communication peaks during the third and fourth phases, but remains small in the first and second phases. This peak also decreases over time as fewer vertices remain active. Since the second phase is repeated multiple times, it is also difficult to predict when such peaks will occur. Additionally, sending edges to the supervertex is many-to-one communication, which can turn supervertices into a bottleneck. DMST is also a good system stress test because it makes extensive use of aggregators and graph mutations. Both are important components of a Pregel-like system that are not tested by the other algorithms.

## 5. EVALUATION METHODOLOGY

In this section, we describe our system and algorithm configurations, datasets, and metrics.

## 5.1 System Setup

We run experiments on setups of 16, 32, 64, and 128 machines, with a separate master for each setup. All machines are m1.xlarge Amazon EC2 spot instances, located in us-west-2c. Each instance has four virtual CPUs, equivalent to eight 1.7 GHz Xeon processors, and 15GB of memory. We choose m1.xlarge because it strikes a good balance between supplying sufficient memory for nearly all systems to execute all datasets, while allowing performance differences to remain discernible for the smaller datasets. All machines run Ubuntu 12.04.1 with Linux kernel 3.2.0-36-virtual.

We use Giraph 1.0.0, GPS rev 110, Mizan 0.1bu1, and GraphLab 2.2 (2a063b3829). We apply patches to Mizan that fix the bugs mentioned in Section 3.3, as well as a bug fix to GPS to correctly enable more than 128 workers. The systems run with Hadoop 1.0.4, jdk1.6.0_30, MPICH 3.0.2, and read input and write output to HDFS. Giraph and GPS use a maximum per-machine JVM heap size of 14.5GB.

To best utilize the available CPU cores of each instance, it is necessary to consider multithreading. First, we distinguish between a *worker* and a *machine*: a machine is a single EC2 instance, on which multiple workers may run. Giraph and GraphLab both support multiple compute threads per worker, so additional performance can be gained while maintaining one worker per machine. This is desirable as adding workers consumes more memory than adding threads due to the duplication of core data structures. Workers also require their own communication threads, which can increase contention. We found that 2 compute and 2 input/output threads on Giraph gave the best performance. We leave GraphLab in its default configuration of 2 compute threads. GPS and Mizan do not support multiple compute threads, so additional performance can be attained only by using more workers. GPS has a sweet spot at 2 workers per machine: adding more workers degrades performance due to contention. Mizan suffers a substantial memory blow up

**Table 3: Directed datasets. Values in parentheses are for the undirected versions used in DMST.**

| Graph | $\lvert V\rvert$ | $\lvert E\rvert$ | Avg In/Outdegree | Max In/Outdegree | Largest SCC |
|---|---|---|---|---|---|
| soc-LiveJournal1 **(LJ)** | 4.8M | 68M (86M) | 14 / 14 (18) | 13.9K / 20K (20K) | 3.8M (79%) |
| com-Orkut **(OR)** | 3.0M | 117M (234M) | 38 / 38 (76) | 3.4K / 33K (33K) | 3.0M (100%) |
| arabic-2005 **(AR)** | 22.7M | 639M (1.11B) | 28 / 28 (49) | 575K / 9.9K (575K) | 15.1M (66.7%) |
| twitter-2010 **(TW)** | 41.6M | 1.46B (2.40B) | 35 / 35 (58) | 770K / 2.9M (2.9M) | 33.4M (80.3%) |
| uk-2007-05 **(UK)** | 105M | 3.73B (6.62B) | 35 / 35 (63) | 975K / 15K (975K) | 68.5M (64.7%) |

(more than 2×) with 4 workers, so we use 2 workers both as a suitable trade-off between time and memory usage and to match the parallelism present in the other systems.

## 5.2 Datasets

Table 3 shows the datasets we obtained from SNAP[2] (Stanford Network Analysis Project) and LAW[3] (Laboratory for Web Algorithmics) [9, 8, 7]. All graphs are real-world datasets with millions to billions of edges. We use $|V|$ and $|E|$ to denote the number of vertices and directed edges (arcs) respectively. `LJ`, `OR`, and `TW` are social network graphs, while `AR` and `UK` are both webgraphs. All datasets are stored on HDFS as uncompressed ASCII text files. In addition, for GraphLab, we manually split each file into contiguous parts equal to the number of machines. This ensures that, like the other systems, GraphLab performs parallel input loading.

The datasets can be characterized by several properties. The simplest is density: a graph is dense if $|E| = O(|V|^2)$, so all of our datasets are sparse. Similarly, the average indegrees and outdegrees of the graph give a sense of how large $|E|$ is relative to $|V|$. As real-world datasets tend to follow a power law degree distribution, the maximum indegree and outdegree give a good sense of how skewed the degree distribution is. The three largest graphs contain vertices with very high degrees, which may cause communication or computation bottlenecks due to workload imbalances. In particular, `TW`'s maximum degree is nearly three times that of `UK`. Lastly, the largest strongly connected component (SCC) gives a sense of how tightly clustered or connected a graph is. The social network graphs exhibit the phenomenon of a unique giant component [14], while the size of the webgraphs' largest SCC indicates a fairly sizable core [15].

For DMST, we require undirected graphs with weights. As all graphs are directed, we add the missing edges to produce an undirected graph. Unique weights are then assigned to every undirected edge (i.e., both the in-edge and out-edge) randomly using a 34-bit maximal-length linear feedback shift register (LFSR) [38].

On all systems, we use `LJ`, `OR`, `AR`, and `TW` for 16 and 32 machines and all five datasets for 64 and 128 machines. Since DMST requires undirected weighted graphs, whose file sizes are three times that of their directed counterparts, we are able to run only `LJ`, `OR`, and `AR` for 16 and 32 machines. For 64 and 128 machines, we extend this to `TW` and `UK`.

## 5.3 Algorithms

For Giraph, we use the existing PageRank, SSSP, and WCC implementations and our own DMST as it was not previously implemented. Like GPS, our DMST uses Giraph's `master.compute()` to track the computation phases.

[2]`http://snap.stanford.edu/data/`

[3]`http://law.di.unimi.it/datasets.php`

For GPS, all necessary algorithms are provided. GPS has three variations of DMST. We use the basic unoptimized variant that matches the implementations in Giraph and Mizan. While PageRank, SSSP, and WCC can all run with and without LALP and dynamic migration, DMST can only run with the optional optimizations disabled (Section 3.2).

For Mizan, we use the existing PageRank algorithm and our own SSSP, WCC, and DMST implementations. SSSP and WCC were previously unimplemented, while the DMST algorithm used in [23] is the GHS algorithm [17] rather than parallel Boruvka. Furthermore, the GHS implementation does not rely only on Mizan's provided API: it is partly built into Mizan itself. Unfortunately, our DMST implementation unearthed a bug in the graph mutation component of Mizan. At the time of writing, this is not yet fixed [22], so we are unable to run DMST in Mizan.

For GraphLab, all algorithms are provided. PageRank and SSSP both support asynchronous mode while WCC does not. Additionally, DMST cannot be implemented efficiently because GraphLab does not fully support graph mutations (Section 3.4).

On Giraph, GPS, and Mizan, PageRank is terminated after 30 supersteps. On GraphLab, PageRank is terminated based on error threshold: vertices halt if their value changes less than the error threshold. This is necessary as superstep termination conditions are impossible for GraphLab's asynchronous mode and this also ensures that synchronous mode computes PageRank to the same accuracy. Because GraphLab's synchronous engine is adaptive (Section 3.4), vertices in PageRank can halt before others. In contrast, vertices in the BSP model must stay active to send its neighbours messages. Otherwise, neighbours will compute incorrect values, leading to errors that can prevent convergence. The error threshold is determined in Giraph by computing the maximum change in the PageRank value across all vertices at superstep 30.

Lastly, for all systems, SSSP, WCC, and DMST execute to completion. SSSP uses unit edge weights and starts at the same source vertex for all systems. This ensures all systems perform the same amount of work.

## 5.4 Evaluation Metrics

We measure performance through the following metrics: total time, setup time, computation time, CPU utilization, memory usage, and network usage (network I/O).

*Total time* is the total running time from start to finish. It can be divided into *setup time*, the time taken to load and partition the input graph (as well as write the output), and *computation time*, which includes local vertex computation, barrier synchronization, and communication. Computation time, in combination with CPU utilization, can help identify message processing overheads, especially when algorithms have short per-vertex computations.

For memory, we define per-machine memory usage as the difference between the maximum and minimum memory used by a single machine during an experiment. This excludes the background usage of Hadoop and OS processes, which is typically between 200 to 300MB. We focus on *maximum memory usage*, the maximum per-machine usage across all worker machines. This gives the minimum memory resources all machines need to run an experiment without failure.

Similarly, per-machine network usage is the total number of bytes sent or received by a single machine for an experiment. We focus on *total network usage*, the sum of per-machine usage across all worker machines, as it best enables a high-level comparison of each system's network I/O. Additionally, we distinguish between total outgoing (sent) and total incoming (received) network usage.

Lastly, using different datasets and number of machines enables us to investigate the *scalability* of each system: how performance scales with the same graph on more machines or with larger graphs on the same number of machines.

To track these metrics, we use the total and setup times reported by all systems. For network usage, some systems have built-in message counters but they all differ. Message count is also a poor measure of network traffic, as messages can be of different sizes (such as in DMST) and header sizes are ignored. Further, the notion of messages is ill-defined for GraphLab's asynchronous mode. For these reasons, we use `/proc/net/dev` to track the total bytes sent and received at each machine, before and after every experiment.

For more fine-grained statistics, and to compute memory usage, we rely on one-second interval reports from `sar` and `free`. These are started locally on both the master and worker machines to record CPU, network, and memory utilization in a decentralized manner. They are started before and killed after each experiment to ensure a small but constant overhead across all experiments. Finally, Giraph 1.0.0 has a bug that prevents proper clean up of Java processes after a successful run. We solve this by killing the offending processes after each run.

For each experiment, we perform five runs and report both the mean and 95% confidence intervals for computation time, setup time, maximum memory usage, and total incoming network usage. We show only incoming network I/O as it is similar to total outgoing network I/O and exhibits identical patterns.

## 6. EXPERIMENTAL RESULTS

In this section, we present and analyze our experimental results. Due to space constraints and in the interests of readability, we present only a subset of all our experimental data. In particular, we do not show data for the master, since its memory and network usage are smaller and less revealing of system performance. All data and plots, along with our code and EC2 images, are available online.[4]

For all plots, each bar represents a system tested. Bars are grouped by dataset and number of machines. Time plots are split into computation time (coloured) and setup time (light gray), with Mizan's separate graph partitioner in dark gray (Figure 2). In the legend, GPS (none) denotes GPS with no optional optimizations, while GPS (LALP) and GPS (dynamic) denote GPS with only LALP or only dynamic migration respectively. We summarize our observations next.

[4]`http://cs.uwaterloo.ca/~kdaudjee/graph-processing`

**Table 4: Performance for `AR`, `TW`, `UK`.**

| | Computation | Setup | Total Time | Memory | Network |
|---|---|---|---|---|---|
| | | | No Mutations | | |
| Gi-BA | ★★★☆ | ★★★★ | ★★★★ | ★★★☆ | ★★★★ |
| Gi-HM | ★★★☆ | ★★★☆ | ★★★☆ | ★★☆☆ | ★★★★ |
| GPS | ★★★☆ | ★★☆☆ | ★★☆☆ | ★★★★ | ★★☆☆ |
| Mizan | ★☆☆☆ | ★☆☆☆ | ★☆☆☆ | ★☆☆☆ | ★★★☆ |
| GL-S | ★★★★ | ★★★☆ | ★★★★ | ★★★☆ | ★★★☆ |
| GL-A | ★☆☆☆ | ★★★☆ | ★☆☆☆ | ★★☆☆ | ★☆☆☆ |
| | | | With Mutations (DMST) | | |
| Gi-BA | ★☆☆☆ | ★★★★ | ★☆☆☆ | ★★☆☆ | ★★★★ |
| Gi-HM | ★★★★ | ★★★★ | ★★★★ | ★★☆☆ | ★★★★ |
| GPS | ★★★★ | ★☆☆☆ | ★★☆☆ | ★★★★ | ★☆☆☆ |

### 6.1 Summary of Results

A relative ranking of system performance, for computation, setup, and total time, maximum memory usage, and total network I/O, are presented for non-mutation and mutation (DMST) algorithms in Table 4. System names are abbreviated as Gi-BA and Gi-HM for Giraph byte array and hash map respectively; GPS for GPS with no optional optimizations; and GL-S and GL-A for GraphLab's synchronous and asynchronous modes. We exclude GPS's other modes as they provide little performance benefits. We focus on the larger graphs (`AR`, `TW`, and `UK`) and summarize our findings for `LJ` and `OR` in Section 6.6.

For each performance attribute, we provide a 4 star relative ranking, with 1 for poor performance, 2 for suboptimal, 3 for good, and 4 for excellent. The ranking of a system is roughly an average of its performance across algorithms.

Overall, for the non-mutation algorithms tested, Giraph and GraphLab are close: we recommend Giraph byte array over GraphLab for very large graphs on a limited number of machines, but GraphLab's synchronous mode over Giraph otherwise. In each case, the recommended system has the best all-around performance. When mutations are required, we recommend Giraph hash map over byte array if memory is not a constraint. If memory is the primary constraint, then GPS is the best option for both mutation and non-mutation algorithms.

Next, we discuss results for each system, focusing on the larger graphs, before highlighting our findings for the `LJ` and `OR` graphs.

### 6.2 Giraph

Compared to GPS, Giraph's computation times are longer for PageRank and comparable or shorter for SSSP and WCC (Figure 2). Since Giraph's setup times are much shorter than GPS's, especially on 64 and 128 machines, Giraph is faster overall: up to 3× shorter total time on PageRank, 8× on SSSP, and 5.5× on WCC. This demonstrates Giraph 1.0.0's substantial improvements since version 0.1, when GPS was reported to be 12× faster for PageRank [32]. Giraph has longer computation times than GraphLab's synchronous mode but comparable or shorter setup times. Consequently, Giraph's total time is comparable to GraphLab for SSSP and WCC, but is up to 3.5× longer on PageRank due to GraphLab's much shorter computation times.

Giraph generally has the lowest network usage for SSSP and WCC, but receives more data than GraphLab's synchronous mode for PageRank, since synchronous mode is adaptive (Figure 4). Overall, Giraph byte array's maximum memory usage is higher than GPS (Figure 3). Compared to GraphLab, Giraph's memory usage tends to scale better for larger graphs on the same number of machines, while GraphLab scales better for the same graph on more machines. However, Giraph's maximum memory usage is much higher than the average usage across machines, indicating suboptimal workload balance. Hence, there is an opportunity to further improve partitioning via, for example, GraphLab's vertex cut approach.

For PageRank, SSSP, and WCC, Giraph byte array's performance is better than Giraph hash map in computation time, setup time, and maximum memory usage. Computation time is up to 2.5× shorter for SSSP, while setup times are 4× shorter on UK (Figure 2b). Byte array is more memory efficient, using up to half the maximum memory of hash map (Figure 3). Finally, as edge types do not affect network I/O, they both have identical network usage (Figure 4).

For DMST, Giraph byte array is considerably slower than hash map: 31× slower on AR (Figure 2d). Furthermore, Giraph byte array does not complete on TW even after 24 hours, meaning that it is over 100× slower than hash map. Byte array's poor performance for DMST is due to its computational inefficiency for graph mutations (Section 3.1): 99% of the computation time is spent performing mutations. Workload imbalance is also critically important, as machines handling high-degree vertices are very CPU-bound and become stragglers that hold back the entire computation. For example, on UK, 98% of the time is spent in the first superstep that performs mutations, as that is when all vertices are active (Section 4.4). Although TW is smaller than UK in size, it takes far longer because it has more high-degree vertices.

Giraph hash map and GPS are both faster because neither hash maps nor arrays suffer deserialization overheads. Compared to GPS, Giraph hash map achieves better computation and setup times as well as lower network I/O (Figure 4d). However, it can consume up to twice the memory used by GPS (Figure 3d). Indeed, this is why hash map fails for UK on 128 machines. Therefore, there is a need for more memory efficient adjacency list data structures to handle graph mutations in Giraph.

## 6.3 GPS

We found that GPS's setup time increases when using more machines (Figure 2). For example, setup time on 128 machines is up to 3× slower than on 32 machines for AR. This coincides with a large increase in total network I/O, especially evident under SSSP due to its low network usage during computation (Figure 4b). Both issues occur because HDFS splits an input file into multiple data blocks, which are then distributed and stored on different machines. Each GPS worker loads a part of the input file by reading a contiguous range of bytes rather than by favouring HDFS data blocks local to its machine. This results in expensive remote reads and the observed overheads. Hence, setup times in GPS can be shortened by exploiting data locality to reduce network overheads.

Additionally, a superstep in GPS has a fixed overhead of 2 seconds. In contrast, a superstep in Giraph can be as short as 0.2 seconds. This overhead is because GPS's communication threads poll for messages, with a default interval of 1 second. This can negatively impact performance when there

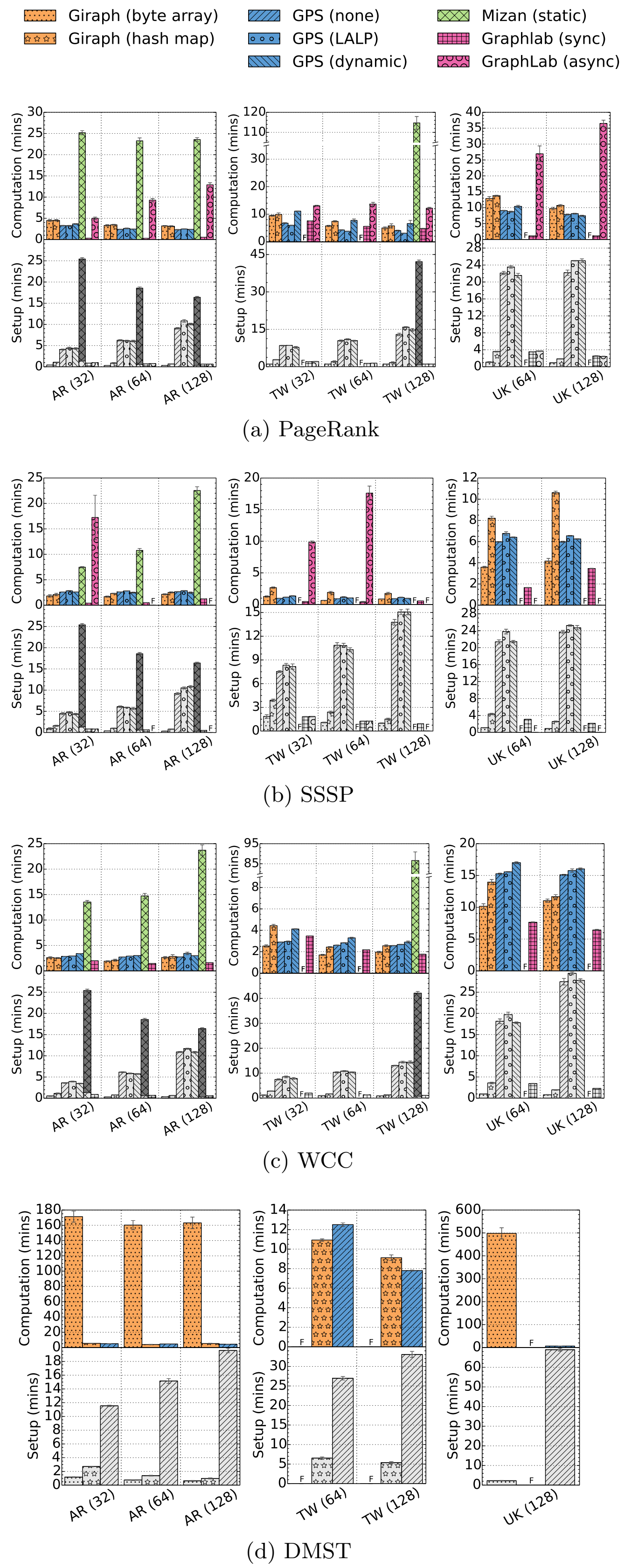

**Figure 2: Setup and computation times. Missing bars labelled with 'F' indicate unsuccessful runs.**

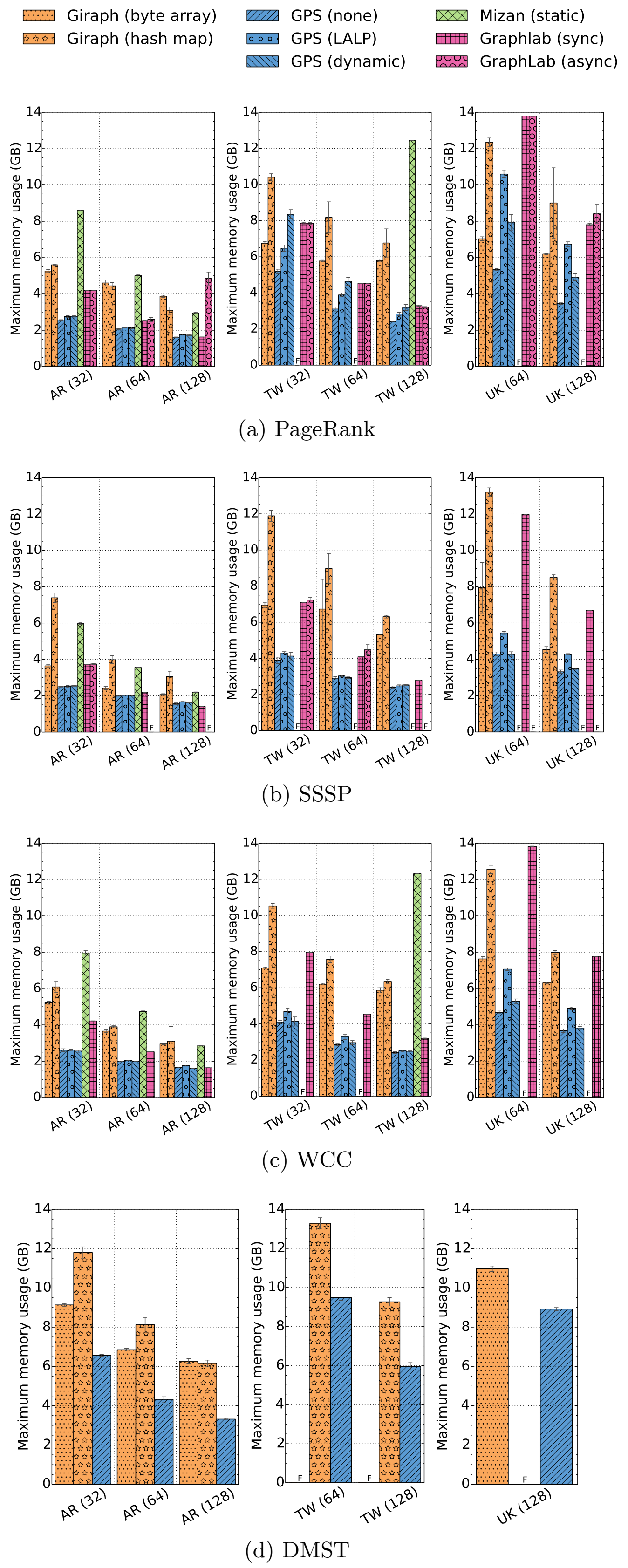

**Figure 3: Maximum memory usage. Missing bars labelled with 'F' indicate unsuccessful runs.**

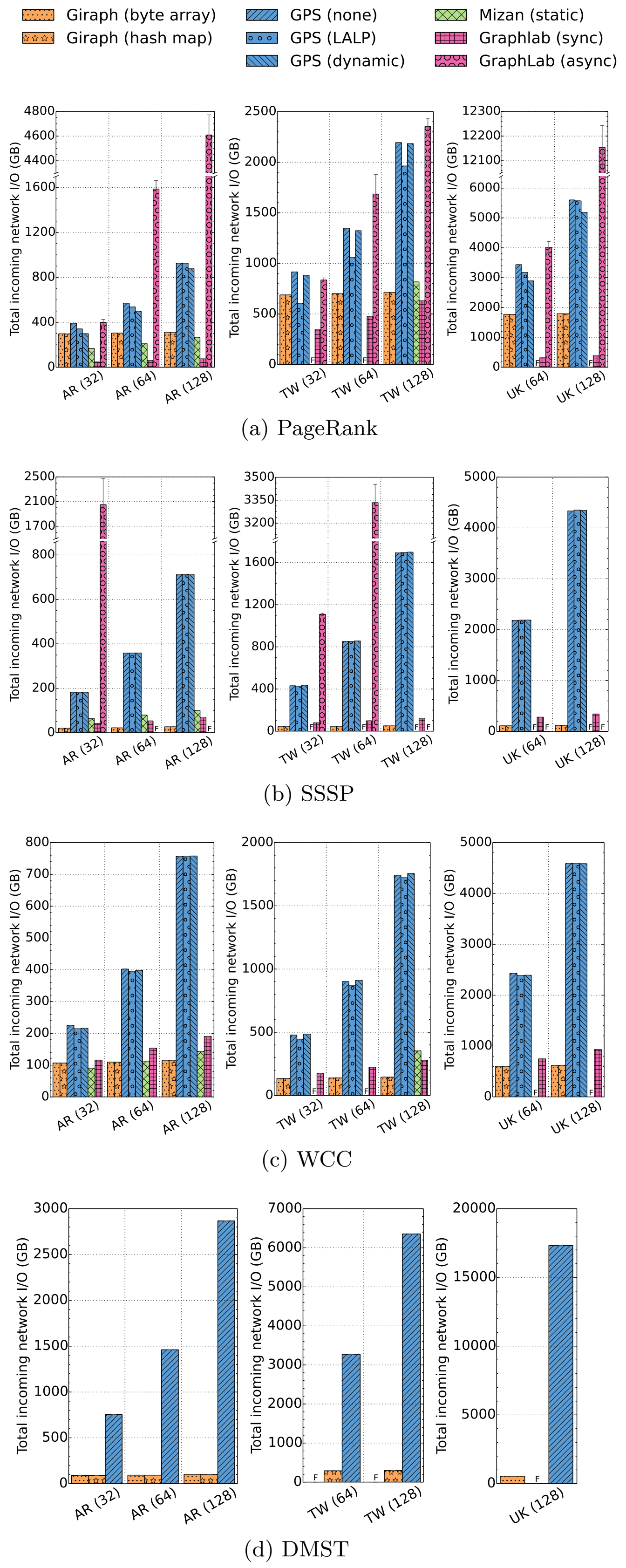

**Figure 4: Total incoming network I/O. Missing bars labelled with 'F' indicate unsuccessful runs.**

are multiple short supersteps, such as in the second phase of DMST. Hence, a more efficient means of obtaining messages without polling is another optimization opportunity.

GPS with no optional optimizations has excellent memory efficiency across all experiments (Figure 3). This is largely due to system optimizations such as canonical objects (Section 3.2), although the use of 32-bit vertex IDs does play a minor role (all other systems use 64-bit IDs). While switching Giraph's vertex IDs from longs to integers does improve memory usage, it is not as low as GPS.

For its optional optimizations, LALP and dynamic migration provide limited benefits: when computation time is reduced, it is largely offset by overheads in setup time (Figure 2a). Both optimizations incur higher memory usage and provide only slightly reduced network I/O compared to no optimizations (Figures 3 and 4). An exception is PageRank on TW: due to TW's greater number of high-degree vertices, network I/O is reduced by 1.8× with LALP. In contrast, dynamic migration increases computation time for highly skewed graphs such as TW on PageRank (Figure 2a) due to the overheads in relabelling adjacency lists. Our dynamic migration results agree with [32] regarding computation time, but do not show a 2× improvement for network I/O. In particular, while UK has noticeably reduced network I/O under dynamic migration, TW does not (Figure 4a).

## 6.4 Mizan

Mizan performs poorly across all experiments. Setup time alone is longer than the total time of the other systems, except for GraphLab's asynchronous mode (Figure 2). Mizan's graph partitioning is done separately from computation and is slow because it processes the graph to track both in-edges and out-edges of every vertex and performs multiple large reads and writes to HDFS. Neither Giraph nor GPS store in-edges explicitly, while GraphLab does so efficiently using CSC [25]. Loading the partitioned data is faster with more machines but graph partitioning times scale poorly.

For computation time, Mizan is over 20× slower than Giraph, GPS, and GraphLab's synchronous mode for PageRank on TW (Figure 2a). This agrees with [23], as Mizan's 2× performance gain over Giraph 0.1 in PageRank suggests that it should be 6× slower than GPS, and hence Giraph 1.0.0. Mizan's poor performance is due to a lack of built-in system optimizations and an inefficient implementation of message processing with respect to details such as coarseness of locking, data structure efficiency, synchronization between threads, and so on (Section 3.2). For example, a superstep in Mizan takes 2 minutes on PageRank with TW, compared to 10 seconds in Giraph and GPS.

For PageRank and WCC, Mizan also spends a large portion of its time in the first superstep: nearly 1 hour for TW. This coincides with a large amount of prolonged network traffic and a sudden sustained increase of memory usage (up to double of what was previously used). A similar issue occurs with SSSP when nearing its peak network usage. We suspect this slowdown is related to the setup costs of communication data structures and/or suboptimal use of temporary data structures. For SSSP and WCC, Mizan's computation time also increases when more than 32 machines are used: SSSP with AR on 128 machines is 3× slower than on 32 machines (Figure 2b). The scalability bottleneck is because message processing times more than double when using over 32 machines.

Mizan's memory usage scales well when adding more machines but poorly when larger graphs are used. For example, Mizan can only run PageRank on TW with 128 machines and still consumes 12GB of maximum memory (Figure 3a). This is again due to its lack of system and communication optimizations. Lastly, Mizan achieves low network usage due to the use of byte compression (Figure 4), but that does not improve its computation times.

The fact that Giraph, GPS, and GraphLab (synchronous) all outperform Mizan suggests that system designers should first focus on improving built-in system optimizations before adding features like dynamic migration. The negative results for GPS's dynamic migration also support the idea that optimizing message processing may be more fruitful.

## 6.5 GraphLab

GraphLab's synchronous mode achieves the shortest computation times due to a combination of better balanced workloads and the ability to perform adaptive computation. Relative to Giraph, computation time is up to 19× shorter on PageRank, 4× on SSSP, and 1.8× on WCC. GraphLab's gains for SSSP and WCC are lower because adaptive computation is less advantageous.

In contrast, GraphLab's asynchronous mode is a mixed bag. While async always has longer computation times than synchronous mode, it can perform comparably to Giraph and GPS in some cases. However, async does not scale well and its performance rapidly deteriorates beyond a number of machines that varies with the input graph (Figure 2b), as async is highly graph dependent. In other words, a graph can be too small for a given number of machines. This degradation is accompanied by a significant increase in network I/O: over 11× for 128 machines compared to 32 machines, on PageRank with AR (Figure 4a). In some cases, async also suffers a large increase in memory usage (>14GB), which causes the failures in SSSP (Figure 2b).

Async's poor performance is due to lock contention, the lack of message batching, and communication overheads in distributed locking. Specifically, some machines, having completed execution on their portion of the graph, will initiate a termination detection algorithm only to find other machines that are not done. After a while, they reinitiate the algorithm, only to cancel again. This repeated cycling between starting and cancelling termination is expensive and can result in substantial overheads. Async's network I/O is also highly variable due to its non-deterministic execution.

Given GraphLab's excellent synchronous performance, there is little reason to run PageRank or SSSP asynchronously for the datasets we have tested: while async can provide faster convergence on a single machine [28], the overheads, poor scalability, and difficulty in tuning distributed locking prevent a net performance gain. Despite its overheads, async may perform better in specific cases where BSP is inefficient, such as SSSP on graphs with very large diameters. Finally, async's poor network scalability indicates that it can achieve better performance on a small number of powerful machines rather than a large cluster of weaker machines.

## 6.6 Results for LJ and OR Datasets

The smaller LJ and OR graphs share similar trends with the larger graphs. For GPS, its poor setup time performance is more evident: 128 machines take 10× longer than 16 machines on LJ. Furthermore, GPS's optional optimizations

Table 5: Our experiences with each system.

| | Giraph | GPS | Mizan | GraphLab |
|---|---|---|---|---|
| Dev. Support | ★★★★ | ★★★☆ | ★☆☆☆ | ★★★★ |
| Usability | ★★★★ | ★★☆☆ | ★★★☆ | ★★★★ |

still provide little benefit on `LJ` and `OR`, while for DMST, GPS's superstep overheads result in computation times that are up to twice as long as Giraph hash map. Similarly, Mizan's scalability issues remain evident on SSSP and WCC. Lastly, GraphLab's asynchronous mode fails on 128 machines for both `LJ` and `OR`, on both PageRank and SSSP, due to a substantial increase in memory usage. However, this is easy to avoid as `LJ` and `OR` run fine on fewer machines.

## 7. EXPERIENCES

In this section we highlight our experiences with each system, particularly from an end-user's point of view. We consider two aspects: *development support* (dev. support), which is how user-friendly a system is to algorithm developers in terms of API, documentation, and so forth; and *usability*, how easy it is to use and monitor the system and its progress, and how effectively errors or crashes are displayed. A high-level ranking is provided in Table 5.

### 7.1 Giraph

Given its large developer base and rigorous coding style requirements, Giraph is the easiest system to understand and code for. In contrast, neither GPS nor Mizan are nearly as well-documented, requiring us to contact the authors for help in understanding certain system components. Giraph's well-documented API allowed us to identify potential pitfalls without needing to understand the implementation details. Depending on the use case, Giraph's use of Java and its tight integration with Hadoop, HIVE, and so on can be another advantage.

Structurally, Giraph is designed in a modular manner, allowing custom input, output, and data type formats. These data types must be specified as command-line arguments, which can initially be daunting. However, this setup provides unparallelled flexibility, especially compared to the limited input and output options of GPS and Mizan. Changing the input format is also easy in GraphLab but, from our limited experience, the implementation is less modular.

Finally, Giraph has excellent usability due to its use of the Hadoop web monitoring interface. When an experiment runs into trouble, it is evident in both the console output and on the web interface. Additionally, Giraph outputs log files, which are useful for debugging and monitoring progress.

### 7.2 GPS

GPS has a similar API to Giraph, making it easy to understand and to port code over. While we did not develop as extensively in GPS or have the opportunity to work with its global memory map (GPS's version of aggregators), we found its API documentation to be lacking compared to Giraph and GraphLab. Finally, unlike Giraph, custom data types in GPS are hard coded as a job configuration class for each algorithm. While not as flexible as Giraph's command-line approach, it provides the same modularity.

For usability, GPS provides an excellent, albeit poorly documented, web interface that tracks many detailed statistics. However, GPS does have a few issues that make experimentation difficult. First, communication teardown is not robust: GPS workers and master do not free their ports immediately, forcing a minimum wait time of a minute between runs. Without this, connections will fail to establish on the next run, as ports are hard-coded in a configuration file. Second, failures or errors are not prominently displayed on the web interface and do not appear at all in the console. In particular, failed computations never timeout.

### 7.3 Mizan

Mizan[5] is still very much an experimental system compared to Giraph and GraphLab. It has many core features implemented but still has a few bugs and is missing several useful features. For example, it currently assumes that vertex values and edge weights are of the same type. This makes DMST consume more memory as every edge must store 5 fields rather than 3. The inability to read in edge weights from an input graph is also a barrier to testing more complex inputs, such as graphs with randomly generated unique edge weights used for DMST. Finally, Mizan does not yet support `master.compute()`, which is also useful for DMST.

As Mizan is written in C++, there is less run-time checking than Java. In combination with Mizan's API design, algorithm implementations are more susceptible to memory errors than in Giraph, GPS, and GraphLab. Sparse API documentation means identifying major pitfalls require understanding the internal code. For example, deleting or changing a vertex value immediately frees the previous underlying object, despite the original allocation occurring in the user's algorithm code. In contrast, both Giraph and GraphLab clearly document unsafe operations. Having such warnings, or a more robust API, would make development much easier. Lastly, differing coding styles in Mizan mean strange bugs can arise due to subtle inconsistencies. These bugs make it difficult to debug Mizan's system on our own, and are exacerbated by the fact that Mizan does not perform logging.

For usability, Mizan provides only console output but it contains sufficient information about where the computation is. However, the console output suffers from serialization issues, so text gets clobbered when using many workers. Finally, unlike GPS, failures are immediate and do not hang.

### 7.4 GraphLab

While we did not develop algorithms in GraphLab, we found that its source code and API are well-documented and easy to understand. GraphLab makes good use of C++ features and, unlike Mizan, there is less confusion about memory allocation and deallocation. Like Giraph, GraphLab also has an active developer and user community.

Similar to Mizan, GraphLab outputs only to the console and failures do not hang. While it does not log to files like Giraph and GPS, it can output different levels of information for debugging. Unlike Giraph and GPS, GraphLab provides parallel input loading only when the input graph file is manually split into disjoint parts. Supporting parallel loading for a single input file would help improve usability. Finally, like GPS and Mizan, GraphLab is not as tightly integrated with Hadoop as Giraph.

[5]We thank Mizan's authors for providing several bug fixes.

## 8. CONCLUSION

Graph processing systems are increasingly important as more and more problems require dealing with graphs. To this end, we presented a thorough comparison of four recent graph processing systems, Giraph, GPS, Mizan, and GraphLab, on five datasets and four different algorithms: PageRank, SSSP, WCC, and DMST. We used 16, 32, 64, and 128 Amazon EC2 instances with computation time, setup time, memory usage, and network I/O as our metrics. We found that Giraph and GraphLab's synchronous mode have good all-around performance, while GPS excels at memory efficiency. We also found that Giraph, GPS, and GraphLab's synchronous mode outperform Mizan in all experiments.

We identified Giraph hash map as a better choice than Giraph byte array for graph mutations. We found that GPS's LALP and dynamic migration optimizations provide little performance benefit, and that GraphLab's asynchronous mode has poor scalability and performance due to communication overheads. Finally, for each system, we identified potential areas of improvement: for Giraph, better workload balancing to reduce maximum memory usage, and a need for adjacency list data structures that are both mutation and memory efficient; for GPS, exploiting data locality to improve the scalability of setup times, and avoiding message polling to minimize superstep overheads; for Mizan, adding system and message processing optimizations to improve performance and scalability; and, for GraphLab, reducing communication overheads for its asynchronous mode.

## 9. REFERENCES

[1] Apache Giraph. `http://giraph.apache.org`.

[2] Apache Hama. `http://hama.apache.org`.

[3] GraphLab. `http://graphlab.org`.

[4] J. Alpert and N. Hajaj. We knew the web was big... `http://googleblog.blogspot.ca/2008/07/we-knew-web-was-big.html`, 2008.

[5] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA '04*, pages 120–124, 2004.

[6] S. Austin. Facebook Passes the 900 Million Monthly Users Barrier. `http://blogs.wsj.com/digits/2012/04/23/facebook-passes`, 2012.

[7] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[8] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW '11*, pages 587–596, 2011.

[9] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW '04*, pages 595–602, 2004.

[10] A. Ching. Scaling Apache Giraph to a trillion edges. `http://www.facebook.com/10151617006153920`, 2013.

[11] S. Chung and A. Condon. Parallel Implementation of Borvka's Minimum Spanning Tree Algorithm. In *IPPS '96*, pages 302–308, 1996.

[12] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking Traversal Operations over Graph Databases. In *ICDEW 2012*, pages 186–189, 2012.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[14] E. David and K. Jon. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[15] D. Donato, L. Laura, S. Leonardi, and S. Millozzi. Simulating the Webgraph: A Comparative Analysis of Models. *Computing in Science and Engineering*, 6(6):84–89, 2004.

[16] B. Elser and A. Montresor. An evaluation study of BigData frameworks for graph processing. In *IEEE Big Data 2013*, pages 60–67, 2013.

[17] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, pages 17–30, 2012.

[19] 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*, 2014.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM '09*, pages 229–238, 2009.

[21] Karypis Lab. METIS and ParMETIS. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[22] Z. Khayyat. Personal correspondence, Apr 2014.

[23] 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 '13*, pages 169–182, 2013.

[24] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *J. ACM*, 46(5):604–632, 1999.

[25] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *OSDI '12*, pages 31–46, 2012.

[26] G. Laporte. The Vehicle Routing Problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345–358, 1992.

[27] Y. Low. Is Chromatic Engine still supported. `http://groups.google.com/d/topic/graphlab-kdd/2T0CjLodHFc/`, 2013.

[28] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD/PODS '10*, pages 135–146, 2010.

[30] N. Przulj. Protein-protein interactions: Making sense of networks via graph-theoretic modeling. *BioEssays*, 33(2):115–123, 2011.

[31] S. Salihoglu and J. Widom. GPS: A Graph Processing System. Technical report, Stanford, 2012.

[32] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM '13*, pages 22:1–22:12, 2013.

[33] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. Technical report, Stanford, 2013.

[34] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE '13*, pages 553–564, 2013.

[35] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *VLDB '13*, 7(3):193–204, 2013.

[36] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD/PODS '08*, pages 567–580, 2008.

[37] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[38] R. Ward and T. Molteno. Table of Linear Feedback Shift Registers. `http://www.eej.ulst.ac.uk/~ian/modules/EEE515/files/old_files/lfsr/lfsr_table.pdf`, 2007.