LFGraph: Simple and Fast Distributed Graph Analytics*

Imranul Hoque†
VMware, Inc.
ihoque@vmware.com

Indranil Gupta
University of Illinois, Urbana-Champaign
indy@illinois.edu

Abstract

Distributed graph analytics frameworks must offer low and balanced communication and computation, low preprocessing overhead, low memory footprint, and scalability. We present LFGraph, a fast, scalable, distributed, in-memory graph analytics engine intended primarily for directed graphs. LFGraph is the first system to satisfy all of the above requirements. It does so by relying on cheap hash-based graph partitioning, while making iterations faster by using publish-subscribe information flow along directed edges, fetch-once communication, single-pass computation, and in-neighbor storage. Our analytical and experimental results show that when applied to real-life graphs, LFGraph is faster than the best graph analytics frameworks by factors of 1x–5x when ignoring partitioning time and by 1x–560x when including partitioning time.

1 Introduction

Distributed graph processing frameworks are being increasingly used to perform analytics on the enormous graphs that surround us today. A large number of these graphs are directed graphs, such as follower graphs in online social networks, the Web graph, recommendation graphs, financial networks, and others. These graphs may contain millions to billions of vertices, and hundreds of millions to billions of edges.

Systems like Pregel [30], GraphLab [29], GraphChi [27], and PowerGraph [20] are used to compute metrics such as PageRank and shortest path, and to perform operations such as clustering and matching. These frameworks are vertex-centric and the processing is iterative. In each iteration (called a *superstep* in some systems) each vertex executes the same code and then communicates with its graph neighbors. Thus, an iteration consists of a mix of computation and communication.

A distributed graph analytics engine running in a cluster must pay heed to five essential aspects:

1. *Computation:* The computation overhead must be low and load-balanced across servers. This determines per-iteration time and thus overall job completion time. It is affected by the number and distribution of vertices and edges across servers.

2. *Communication:* Communication overhead must be low and load-balanced across servers. This also determines per-iteration time and thus overall job completion time. It is affected by the quantity and distribution of data exchanged among vertices across servers.

3. *Pre-Processing*: Prior to the first iteration, the graph needs to be partitioned across servers. This partitioning time must be low since it represents upfront cost and is included in job completion time.

4. *Memory*: The memory footprint per server must be low. This ensures that fewer servers can be used for processing large graphs, e.g., when resources are limited.

5. *Scalability*: Smaller clusters must be able to load and process large graphs. As the cluster size is grown, communication and computation must become cheaper, and the entire job must run faster.

Each of today's graph processing frameworks falls short in at least one of the above categories. We will elaborate later in Section 2.3, and also experimentally compare our approach against existing systems. For now, Table 1 summarizes a qualitative comparison, and we

*This work was supported in part by AFOSR/AFRL grant FA8750-11-2-0084 and in part by NSF grant CCF 0964471.

†Work done while the author was at University of Illinois, Urbana-Champaign.

TRIOS'13, November 03, 2013, Farmington, PA, USA.

http://dx.doi.org/10.1145/2524211.2524218

Goal	Pregel	GraphLab	PowerGraph	LFGraph
Computation	2 passes, combiners	2 passes	2 passes	1 pass
Communication	$\propto$ #edge-cuts	$\propto$ #vertex ghosts	$\propto$ #vertex mirrors	$\propto$ #external in-neighbors
Pre-processing	Cheap (Hash)	Cheap (Hash)	Expensive (Intelligent)	Cheap (Hash)
Memory	High (store out-edges + buffered messages)	High (store in- and out-edges + ghost values)	High (store in- and out-edges + mirror values)	Low (store in-edges + remote values)
Scalability	Good but needs min #servers	Good but needs min #servers	Good but needs min #servers	Good and runs with small #servers

Table 1: *LFGraph vs. existing systems: a qualitative comparison*

briefly discuss. GraphChi [27] is a disk-based single-server framework and is slower than distributed frameworks. Pregel [30] was the first vertex-centric distributed graph processing framework. It suffers from both high memory footprint and high communication overhead. GraphLab [29] and PowerGraph [20] have lower communication overhead compared to Pregel, and PowerGraph also balances computation. They are both faster than Pregel. However, these latter systems store in-links and out-links for each vertex, hence increasing their memory footprint. They are thus unable to process large graphs on small clusters.

The fastest of these systems, PowerGraph, uses intelligent partitioning of vertices across servers. While this pre-processing reduces per iteration runtime, it is an expensive step by itself. For instance, we found that when running PageRank on PowerGraph with 8 servers and 30 iterations (a value that Pregel uses [30]), the intelligent partitioning step constituted 80% of the total job runtime. This upfront cost might make sense if it is amortized over multiple analytics jobs on the same graph. However, as we show in the paper, cheaper partitioning approaches do not preclude faster iterations.

This paper presents LFGraph[1], the first system to satisfy the five requirements outlined earlier. LFGraph is a fast, scalable, distributed, in-memory graph analytics framework. It is primarily intended for directed graphs, however it can be adapted for undirected graphs. The unique design choices in our system are:

- **Cheap Partitioning**: We rely merely on hash-based partitioning of vertices across servers, helping us balance computation and communication. This approach lowers pre-processing overhead and system complexity.
- **Decoupling Computation from Communication:** This allows us to optimize communication and computation independent of each other. It also leads to modular code.
- **Publish-Subscribe Mechanism**: Most graph computations involve information flow along its directed edges. LFGraph leverages this for efficiency by using a publish-subscribe mechanism across different servers. After each iteration, vertex values are fetched exactly once and they are batched – we call this *fetch-once behavior*. This leads to significant savings, e.g., compared to PowerGraph [20], LFGraph reduces network traffic by 4x.
- **Single-pass Computation**: The per-iteration computation at each server is done in one pass, resulting in low computation overhead. Each of Pregel, PowerGraph, and GraphLab uses multiple passes. Pregel incurs the additional overhead of message combiners. LFGraph is simpler and yet its individual iterations are faster than in existing systems.
- **No Locking:** LFGraph eliminates locking by decoupling reads and writes to a vertex's value.
- **In-neighbor Storage**: LFGraph maintains for each vertex only its in-neighbors. Compared to existing systems which maintain both in- and out-neighbors, LFGraph lowers memory footprint and is thus able to run large graphs even on small clusters. We also extend LFGraph to undirected graphs by treating each edge as two directed edges.

This paper presents the design of LFGraph, analytical results comparing it against existing systems, and a cluster deployment of our implementation comparing it to the best system, PowerGraph. Our experiments used both synthetic graphs with a billion vertices, as well as several real graphs: Twitter, a Web graph, and an Amazon recommendation graph. LFGraph is faster than existing systems by 2x–5x for PageRank, by 1x–2x for Single-Source Shortest Path, and by 2x for Triangle Count, when ignoring the expensive pre-processing stage. However, when including the pre-processing stage, LFGraph outperforms existing systems by 5x–380x for PageRank and by 1x–560x for Single-Source Shortest Path.

Further, our experiments reveal that subtle differences between real-world graphs and ideal power-law graphs make it sub-optimal (e.g., in PowerGraph) to specifically optimize for the latter. One key takeaway is that hash-based partitioning suffices for real-world power-law-like graphs while intelligent partitioning schemes yield little benefit in practice. Our work also shows that paying

[1]This stands for Laissez-Faire Graph Processing System.

careful attention to design choices and their interactions is a graph processing system can greatly improve performance. For instance, compared to PowerGraph, LFGraph improves memory footprint by 8x–12x, communication overhead by 4x–4.8x, and eliminates the intelligent placement phase, which in PowerGraph consumes 90%–99% of overall runtime.

2 Computation Model

This section presents the assumptions LFGraph makes, the LFGraph abstraction, and a qualitative comparison with existing systems. Then we present LFGraph's API and sample graph processing applications using this API.

2.1 Assumptions

- LFGraph performs computations on the graph itself rather than performing data mining operations on graph properties such as user profile information.
- LFGraph framework is intended for value propagation algorithms. Values propagate along the direction of the edges. Algorithms that fall in this category include PageRank, Single-Souce Shortest Path, Triangle Count, Matching, Clustering, Graph Coloring, etc.
- LFGraph assumes that the number of high degree vertices is much larger than the number of servers. This is necessary to achieve load balance (see Section 4.2) and to reduce communication overhead.

2.2 LFGraph Abstraction

An LFGraph server stores each graph vertex as a tuple ⟨vertex ID, user-defined value⟩. The type of the user-defined value is programmer-specified, e.g,. in PageRank it is a floating point, for Single-Source Shortest Path (SSSP) it is an integer, and for Triangle Count it is a list. For each vertex a list of *incoming* edges is maintained. An edge is also associated with a user defined value that is static, e.g., the edge weight.

Abstraction 1 LFGraph

```
1: function LFGRAPH(Vertex v)
2:     val[v] ← f(val[u] : u ∈ in_neighbor(v))
3: end function
```

LFGraph uses the programming model shown in Abstraction 1. The programmer writes a vertex program $f()$. This program runs in iterations, akin to supersteps in existing systems [20, 29, 30]. Each vertex is assigned to one server. The start of each iteration is synchronized across servers. During an iteration, the vertex program for vertex v reads the values of its incoming neighbors, performs the computation specified by $f()$, and updates its own value. If v's value changes during an iteration, it is marked as active, otherwise it is marked as inactive. The framework transmits active values to the servers containing neighboring vertices. The computation terminates either at the first iteration when all vertices are inactive (e.g., in SSSP), or after a pre-specified number of iterations (e.g., in PageRank).

2.3 Qualitative Comparison

The abstractions employed by Pregel, GraphLab, and PowerGraph are depicted respectively in Abstraction 2, 3, and 4. To contrast with LFGraph we first discuss each of these systems and then summarize LFGraph. We use a running example below (Figure 1). Table 1 summarizes this discussion.

Abstraction 2 Pregel

```
1: function PREGEL(Vertex v)
2:     val[v] ← f(msg), sender(msg_i) ∈ in_neighbor(v)
3:     send_message(val[v], u), u ∈ out_neighbor(v)
4: end function
```

Abstraction 3 GraphLab

```
1: function GRAPHLAB(Vertex v)
2:     val[v] ← f(val[u]), u ∈ in_neighbor(v)
3:     if updated(val[v]) then
4:         activate(u), u ∈ out_neighbor(v)
5:     end if
6: end function
```

Abstraction 4 PowerGraph

```
1: function POWERGRAPH(Vertex v_i)
2:     val[v_i] ← f(val[u]), u ∈ in_neighbor(v_i)
3:     val[v] ← sync(v_i), v_i ∈ replica(v)
4:     if updated(val[v]) then
5:         activate(u), u ∈ out_neighbor(v_i)
6:     end if
7: end function
```

Pregel: Pregel assigns each vertex to one server. Per iteration, v's vertex program uses its received neighbor values to update the vertex value, and then sends this new value back out to servers where v's neighbors are located.

Consider the sliver of the graph depicted in Figure 1(a). We focus on the vertex program for A only, and our example cluster contains two servers $S1$ and $S2$. Figure 1(b) shows that Pregel's communication overhead (dashed arrows) is proportional to the number of edges crossing server boundaries – A's value is sent twice from $S1$ to $S2$, once for each neighbor. Pregel does allow programmers to write combiners to optimize communication, but this increases computation complexity by requiring an additional pass over the outgoing messages.

Besides, some analytics programs do not lend themselves easily to combiners.

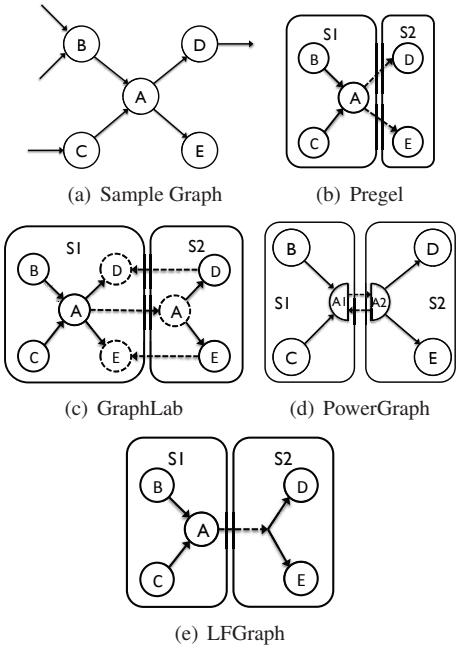


Figure 1: *Communication overhead*

GraphLab: GraphLab first assigns each vertex (say A) to one server ($S1$). Then for each of A's in- and out-neighbors not assigned to $S1$, it creates *ghost* vertices, shown as dashed circles in Figure 1(c). A is assigned to $S1$ but is ghosted at $S2$ since its out-neighbor D is there. This allows all edge communication to avoid the network, but at the end of the iteration all the ghosts of A need to be sent its new value from A's main server ($S1$). This means that GraphLab's communication overhead is proportional to the number of ghosts. However, the number of ghosts can be very large – it is bounded by min(cluster size, total number of in- and out-neighbors). Section 4 shows that this leads to high communication overhead when processing real graphs with high degree vertices.

If A's value at a server is updated during an iteration, GraphLab activates its outgoing neighbors (lines 3–5 in Abstraction 3). This requires GraphLab to store both in- and out- neighbor lists, increasing memory footprint. Further, per vertex, two passes are needed over its in- and out- neighbor lists. The first pass updates its value, and the second activates the out-neighbors.

PowerGraph: In order to target power-law graphs, PowerGraph places each *edge* at one server. This means that vertex A may have its edges placed at different servers. Thus PowerGraph creates *mirrors* for A at $S1$ and $S2$, as shown in Figure 1(d). The mirrors avoid edge communication from crossing the network. However, the mirrors need to aggregate their values during the iteration. PowerGraph does this by designating one of the mirrors as a master. In the middle of the iteration (line 3 of Abstraction 4), all A's mirrors send their values to its master ($A1$), which then aggregates them and sends them back. Thus, communication overhead is proportional to twice the number of vertex mirrors, which can be very large and is bounded by min(cluster size, total number of in- and out-neighbors). We show in Section 4 that PowerGraph incurs high communication overhead for real graphs.

Function	Description
getInLinks()	returns a list of in-edges
getUpdatedInLinks()	returns a list of in-edges whose source vertices updated in the previous iteration
int getOutLinkCount()	returns the count of out-edges
getValue(int vertexID)	returns the value associated with vertexID
putValue(VertexValue value)	writes updated value
int getStep()	get iteration count

Table 2: *LFGraph API: Vertex class methods*

LFGraph: As depicted in Figure 1(e), LFGraph assigns each vertex exactly to one server (A at $S1$). LFGraph makes a single pass over the in-neighbor list of A – this reduces computation. $S1$ stores only a publish list of servers where A's out-neighbors are placed (only $S2$ here), and uses this to forward A's updated value. This leads to the fetch-once behavior at $S2$.

In comparison, Pregel does not have fetch-once communication. In GraphLab and PowerGraph values are propagated only once among ghosts/mirrors. However, communication overhead is high in these systems due to the large number of ghosts/mirrors. Concretely, the publish list of LFGraph is upper-bounded by min(cluster size, total number of out-neighbors), which is smaller than the number of ghosts or mirrors in GraphLab and PowerGraph respectively – thus LFGraph's memory footprint is smaller, communication overhead is lower, and it works even in small clusters. Section 3 elaborates further on the design, and we analyze it in Section 4.

LFGraph trades off computation for reduced storage – in an iteration, it needs to run through all the vertices to check if any of them is in fact active. In contrast, PowerGraph and GraphLab have activate/deactivate triggers which can enable/disable the execution of a neighboring vertex in the succeeding iteration.

2.4 LFGraph API

The programmer writes an LFGraph program which uses LFGraph's Vertex class. The exported methods of the

Vertex class (simplified) are depicted in Table 2. We show how these methods can be used to write three graph analytics program: PageRank [31], SSSP (Single-Source Shortest Path), and Triangle Count [38].

PageRank Vertex Program

```
1:  if getStep() = 0 then
2:      putValue(1)
3:  else if getStep() < 30 then
4:      total ← 0
5:      for e ∈ getInLinks() do
6:          v ← e.getSource()
7:          total ← total + getValue(v)
8:      end for
9:      pagerank ← (0.15 + 0.85 × total)
10:     putValue(pagerank/getOutLinkCount())
11: end if
```

SSSP Vertex Program

```
1:  if getStep() = 0 then
2:      if vertexID = srcID then
3:          putValue(0)
4:      else
5:          putValue(∞)
6:      end if
7:  else
8:      min_dist ← ∞
9:      for e ∈ getUpdatedInLinks() do
10:         v ← e.getSource()
11:         dist ← getValue(v) + e.getValue()
12:         min_dist ← min(min_dist, dist)
13:     end for
14:     if getValue(vertexID) > min_dist then
15:         putValue(min_dist)
16:     end if
17: end if
```

TriangleCount Vertex Program

```
1:  if getStep() = 0 then
2:      putValue(getInLinks())
3:  else
4:      count ← 0
5:      s1 ← getValue(vertexID)
6:      for e ∈ getInLinks() do
7:          v ← e.getSource()
8:          s2 ← getvalue(v)
9:          count ← count + set_intersect(s1, s2)
10:     end for
11:     count ← count/2
12: end if
```

PageRank Each vertex sets its initial PageRank to 1 (line 1–2). In subsequent iterations each vertex obtains its in-neighbors' values via *getValue*() (line 5–8), calculates its new PageRank (line 9), and updates its value using *putValue*() (line 10). The LFGraph system is responsible for transferring the values to the appropriate servers.

SSSP In the first iteration, only the source vertex sets its value (distance) to 0 while all others set their value to ∞ (line 2–6). During subsequent iterations a vertex reads the value of its *updated* in-neighbors, calculates the minimum distance to the source through all of its in-neighbors (line 9–13), and updates its value if the minimum distance is lower than its current value (line 14–16). LFGraph only transfers a vertex's value if it was updated during the previous iteration.

Triangle Count This works on an undirected graph, so *getInLinks*() returns all neighbors of a vertex. In the first iteration, each vertex initializes its *value* to the list of its neighbors (line 1–2) . In the second iteration, a vertex calculates, for each of its neighbors, the number of their common neighbors (line 6–10). The final answer is obtained by dividing the count by 2, since triangles are double-counted (line 11).

3 System Design

The LFGraph system consists of three components:

1. *Front-end*: The front-end stores the vertex program and configuration information. The only coordination it performs is related to fault tolerance (Section 3.4).
2. *JobServers*: A single JobServer runs at each server machine. A JobServer is responsible for loading and storing the part of the graph assigned to that server, and launching the vertex-program. The JobServer is composed of four modules, which subsequent sections detail: i) Graph Loader, ii) Storage Engine, iii) Computation Workers, and iv) Communication Workers. Each iteration at a JobServer consists of a *computation phase* run by several computation workers, followed by a decoupled *communication phase* performed by several communication workers.
3. *BarrierServer:* This performs distributed barrier synchronization among JobServers at three points: i) after loading the graph, ii) during each iteration in between the computation and communication phases, and iii) at the end of each iteration.

Figure 2 shows an example execution of an LFGraph iteration. Steps 1 – 3 comprise the pre-processing iteration (graph loading), and steps 4 – 6 are repeated for each iteration. We elaborate on each step:

1. The front-end stores the vertex program and a configuration file containing the following pieces of information: graph data location (e.g., an NFS path), number of JobServers, IP addresses of JobServers, IP address of BarrierServer, number of computation

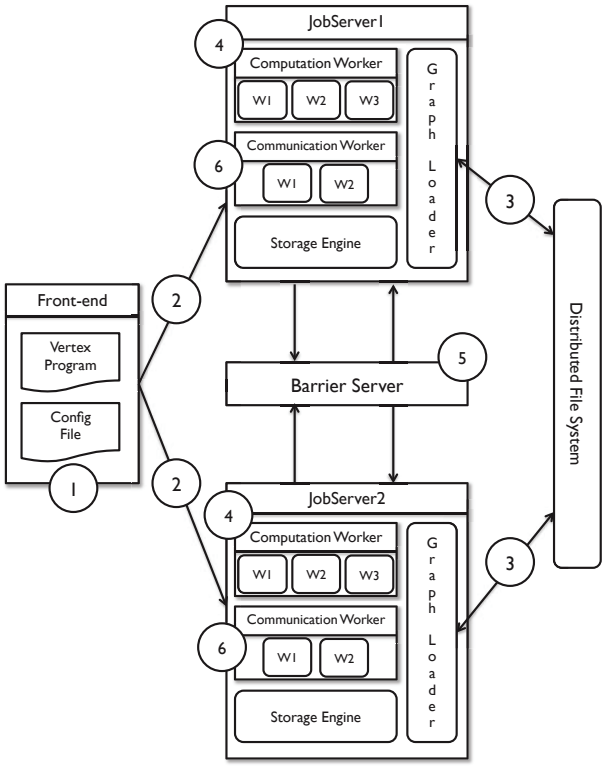


Figure 2: *LFGraph system: the life of a graph computation job*

and communication workers per JobServer, and (if applicable) maximum number of iterations desired.

2. The front-end sends the vertex program and configuration file to all JobServers.
3. The Graph Loaders collectively load the graph (Section 3.1) and split them across the Storage Engines (Section 3.2). The JobServer then signals *next* to the BarrierServer.

   Each iteration repeats the following steps 4 – 6.

4. When the BarrierServer signals back that the barrier is completed, a JobServer starts the next iteration. It does so by spawning multiple local computation worker threads (Section 3.3) to start the computation phase, and sending each worker a sub-shard of its vertices.
5. When all computation workers finish, the JobServer signals the BarrierServer. The signal is one of two types: *next* or *halt*. The latter is signaled only if the termination conditions are met, e.g., maximum number of iterations desired has been reached (e.g., in PageRank) or no local vertices have updated their value in the computation phase (e.g., in SSSP). The BarrierServer terminates the job only if all JobServers signal *halt*.
6. If any of the JobServers signaled *next*, the BarrierServer signals back when the barrier is reached. Then the JobServer starts the communication phase by spawning communication worker threads (Section 3.5). Communication workers are responsible for sending vertex values to remote JobServers. Then it signals *next* to the BarrierServer to start the next iteration.

We next detail the four modules inside the JobServer.

3.1 Graph Loader

Graph Loaders are collectively responsible for loading and partitioning vertex data from the distributed file system (e.g., NFS) and sending them to JobServers. LFGraph can accept a variety of input formats, e.g., edge list, out-neighbor adjacency list, in-neighbor adjacency list, etc. The data is sharded across Graph Loaders. A Graph Loader first uses the path provided by the front-end to load its assigned set of vertices. For each line it reads, it uses a consistent hash function on the vertex ID to calculate that vertex's JobServer, and transmits the line to that JobServer. For efficiency, the Graph Loader batches several vertices together before sending them over to a JobServer.

3.2 Storage Engine

This component of the JobServer stores the graph data, uses a publish-subscribe mechanism to enable efficient communication, and stores the vertex values. It maintains the following data structures.

Graph Storage: This stores the list of in-neighbors for each vertex assigned to this JobServer. *getInLinks*() in Table 2 accesses this list.

Subscribe Lists: This maintains a short-lived list per remote JobServer. Each such list contains the vertices to be fetched from that specific JobServer. The list is built only once – at the pre-processing iteration while loading the graph. Consider our running example from Figure 1(e). $S2$'s subscribe list for $S1$ consists of $\{A\}$. The subscribe list is short-lived because it is garbage-collected after the preprocessing iteration, thus reducing memory usage.

Publish Lists: A JobServer maintains a publish list for each remote JobServer, containing the vertices to be sent to that JobServer. Publish lists are intended to ensure that each external vertex data is fetched exactly once. In the pre-processing iteration, JobServers exchange subscribe lists and use them to create publish lists. In our example, JobServer $S2$ sends to $S1$ its subscribe list for JobServer $S1$. Then the publish list at $S1$ for $S2$ contains those vertices assigned to $S1$ which have out-neighbors assigned to $S2$, i.e., the set $\{A\}$.

Local Value Store: For each vertex assigned to this JobServer (call this a local vertex), this component stores the value for that vertex. We use a two-version system for each value – a *real version* from the previous iteration and a *shadow version* for the next iteration. Writes

are always done to the shadow value and reads always occur from the real value. At the end of the iteration, the real value is set to the shadow value, and the latter is uninitialized. The shadow is needed because computation workers at a JobServer share the local value store. Thus a vertex D at JobServer $S2$ may update its local value, but later another local vertex E also at $S2$ needs to read D's value. Further, this two-version approach decouples reading and writing, thus eliminating the need for locking.

Each value in the local value store is also tagged with an $updated_{A,S1}$ flag, which is reset at the start of an iteration. Whenever the shadow value is written, this flag is set. The communication worker component (which we describe later) uses this flag.

Remote Value Store: This stores the values for each in-neighbor of a vertex assigned to this JobServer, e.g., at JobServer $S2$, the remote value store contains an entry for remote vertex A. There is only one version here since it is only used for reading. This value is also tagged with a flag $updated_{A,S2}$ which is reset at the start of the communication phase – the flag is set only if $S2$ receives an updated value for A during the current communication phase, otherwise it is left unset. This information is used to skip vertices, whose values did not update, in the upcoming iteration. The $getUpdatedInLinks()$ function (in Table 2) of the vertex uses the update flags to return the list of neighbors whose values were updated.

We briefly discuss memory implications of the above five stores of the Storage Engine. Any graph processing framework will need to store the graph and the local value store. The subscribe list is garbage collected. Thus LFGraph's additional overheads are only the publish list and the remote value store. The latter of these dominates, but it stays small in size even for large graphs. For a log-normal graph with 1 billion vertices and 128 billion edges in a cluster of 12 machines running the SSSP benchmark, the per-JobServer remote value store was smaller than 3.5 GB.

3.3 Computation Worker

A computation worker is responsible for running the front-end-specified vertex program sequentially for the sub-shard of vertices assigned to it. Our implementation uses a thread pool for implementing the workers at a JobServer. The number of computation workers per JobServer is a user-specified parameter. For homogeneous clusters, we recommend setting this value to the number of cores at a server.

The computation worker accesses its JobServer's local value store and remote value store. Yet, no locking is required because of the sub-sharding and the two-versioned values. For each vertex this worker reads its in-neighbors' data from either the remote or local value store (real versions only), calculates the new value and writes its updated value into the local value store (shadow version).

3.4 Fault Tolerance

Currently, LFGraph restarts the computation when it encounters a failure. However, it could be optimized to continue the computation after reassigning lost vertices to other servers. Further, lost vertices with at least one remote neighbor are naturally replicated at other non-faulty servers, and those values could be used after the failure.

3.5 Communication Worker

A communication worker runs the decoupled communication phase. It does not rely on locking. The worker runs in three phases. First, each worker is assigned a sub-shard of remote value stores. It resets the update flags in this sub-shard. Second, the worker is assigned a sub-shard of remote JobServers. For each assigned remote JobServer, the worker looks up its publish list, and then sends the requisite vertex values. It uses shadow values from the local value store, skipping vertices whose update flags are false. When a JobServer receives a value from a remote JobServer, it forwards this to the appropriate local worker, which in turn updates the remote value store and sets the update flags. Third and finally, the worker is assigned a sub-shard of the local vertices, for which it moves the shadow value to the real value.

4 Communication Analysis

We first present mathematical analysis for the communication overhead of LFGraph and existing graph processing frameworks (Section 4.1). Then we use three real-world graphs (Twitter, a Web graph, and an Amazon recommendation graph) to measure the realistic impact of this analysis (Section 4.2) and compare these systems. Although we will later present experimental results from our deployment (Section 6), the analysis in this section is the most appropriate way to compare the fundamental *techniques* employed by different systems. This analysis is thus independent of implementation choices (e.g., C++ vs. Java), optimizations, and myriad possible system configurations.

4.1 Mathematical Analysis

Define the communication overhead of a given vertex v as the expected number of values of v sent over the network in a given iteration. We assume random placement of vertices on servers for all systems. We also assume all vertex values have changed, thus the metric is an upper bound on the actual average per-vertex communication. Then, define the communication overhead of an algorithm as the average of the communication overheads across all vertices. We assume the directed graph has $|V|$

vertices, and the cluster contains N servers ($V >> N$). We denote the out-neighbor set of a vertex v as $D_{out}[v]$ and its in-neighbor set as $D_{in}[v]$. $|D_{out}[v] \cup D_{in}[v]| \neq 0$, i.e., no vertex is completely disconnected. We also assume that values are propagated in one direction and values are of fixed sizes.

4.1.1 Pregel

In a default (combiner-less) Pregel setting, each vertex is assigned to one server. Thus values are sent along all edges. An edge contributes to the communication overhead if its adjacent vertices are on different servers. An out-neighbor of v is on a different server than v with probability $\left(1-\frac{1}{N}\right)$. The *expected* communication overhead of v is thus:

$$E[C_P(v)] = |D_{out}[v]| \times \left(1-\frac{1}{N}\right) \tag{1}$$

Therefore, Pregel's communication overhead is:

$$E[C_P] = \frac{\sum_{v \in V}\left(|D_{out}[v]| \times \left(1-\frac{1}{N}\right)\right)}{|V|} \tag{2}$$

4.1.2 GraphLab

In GraphLab, each vertex is assigned to a server. However, the vertex has multiple ghosts, one at each remote server. A ghost is created at remote server S if at least one of v's in- or out-neighbors is assigned to S. The main server where v is assigned then collects all the updated values from its ghosts. v has no neighbors at a given remote server with probability $\left(1-\frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}$. Thus the probability that v has at least one of its neighbors at that remote server is: $\left(1-(1-\frac{1}{N})^{|D_{out}[v] \cup D_{in}[v]|}\right)$. Hence the *expected* communication overhead of v is:

$$E[C_{GL}(v)] = (N-1) \times \left(1-\left(1-\frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}\right) \tag{3}$$

The communication overhead of GraphLab is:

$$E[C_{GL}] = \frac{\sum_{v \in V}\left((N-1) \times \left(1-\left(1-\frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}\right)\right)}{|V|} \tag{4}$$

4.1.3 PowerGraph

In PowerGraph, each vertex is replicated (mirrored) at several servers – let r_v denote the number of replicas of vertex v. One of the replicas is designated as the master. The master receives updated values from the other $(r_v - 1)$ replicas, calculates the combined value, and sends it back out to the replicas. Thus, the communication overhead of v is $2 \times (r_v - 1)$. Plugging in the value of r_v from [20], the *expected* communication overhead of PowerGraph is:

$$\begin{aligned} E[C_{PG}] &= \frac{2 \cdot \sum_{v \in V}(r_v - 1)}{|V|} \\ &= \frac{2 \cdot \sum_{v \in V}\left(N \times \left(1-\left(1-\frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}\right)-1\right)}{|V|} \end{aligned} \tag{5}$$

4.1.4 LFGraph

In LFGraph, a vertex v is assigned to one server. Its value is fetched *by* a remote server S if at least one of v's *out*-neighbors is assigned to S. v has at least one out-neighbor at S with probability $\left(1-(1-\frac{1}{N})^{|D_{out}[v]|}\right)$. The *expected* communication overhead of v is:

$$E[C_{LFG}(v)] = (N-1) \times \left(1-\left(1-\frac{1}{N}\right)^{|D_{out}[v]|}\right) \tag{6}$$

Therefore, LFGraph's communication overhead is:

$$E[C_{LFG}] = \frac{\sum_{v \in V}\left((N-1) \times \left(1-\left(1-\frac{1}{N}\right)^{|D_{out}[v]|}\right)\right)}{|V|} \tag{7}$$

4.1.5 Discussion

Our analysis yields the following observations:

- The overheads of Pregel and LFGraph depend on $|D_{out}[v]|$ only, while those of GraphLab and PowerGraph depend on $|D_{out}[v] \cup D_{in}[v]|$.
- For an undirected graph $|D_{out}[v] \cup D_{in}[v]| = |D_{out}[v]| = |D_{in}[v]|$, so communication overhead of LFGraph and GraphLab are similar for such graphs.
- PowerGraph is a factor of 2 worse in communication overhead compared to GraphLab.
- LFGraph has its lowest relative communication overhead when $|D_{out}[v] \cup D_{in}[v]| \gg |D_{out}[v]|$, i.e., when out-neighbor and in-neighbor sets are more disjoint from each other, and the in-degree is larger than the out-degree.

4.2 Real-World Graphs

We now study the impact of the previous section's analysis on several real-world directed graphs: 1) Twitter, 2) a graph of websites in UK from 2007, and 3) a recommendation graph from Amazon's online book store. The characteristics of these are summarized in Table 3. The traces contain a list of vertices and out-neighbor adjacency lists per vertex.

We calculate the equations of Section 4.1 for each graph by considering each of its vertices and the neighbors of those vertices. This denotes the communication

Graph	Description	$\|V\|$	$\|E\|$
Twitter	Twitter follower network [26]	41.65M	1.47B
UK-2007	UK Web graph [14]	105.9M	3.74B
Amazon	Similarity among books in Amazon store [12, 13]	0.74M	5.16M

Table 3: *Real-world graphs*

overhead, i.e., number of values sent over the network per iteration per vertex. Figure 3 plots this quantity for different cluster sizes ranging from 2 to 256.

First we observe that in all three plots, among all compared approaches, LFGraph has the lowest communication overhead. This is largely due to its fetch-once behavior.

Second, Pregel plateaus out quickly. In fact, the knee of its curve occurs (for each of the graphs) around the region where the x-axis value is in the ballpark of the graph's average out-degree. Up to that inflection point, there is a high probability that a randomly selected vertex will have neighbors on almost all the servers in the system. Beyond the knee, this probability drops.

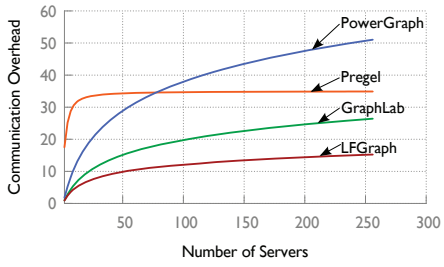
Third, LFGraph's improvement over GraphLab is higher for the Twitter workload than for the other two workloads. This is because the in-neighbor and out-neighbor sets are more disjoint in the Twitter graph than they are in the UK-2007 and Amazon graphs.

Fourth, in Figure 3(c) (Amazon workload), when cluster size goes beyond 10, GraphLab's overhead is higher than Pregel's. This is because on an Amazon book webpage, there is a cap on the number of recommendations (out-neighbors). Thus out-degree is capped, but in-degree is unrestricted. Further, average value of $|D_{out}[v] \cup D_{in}[v]|$ is lower in the Amazon workload (9.58) than in Twitter (57.74) and UK-2007 (62.56). Finally, as the cluster size increases, GraphLab's communication overhead plateaus, with the knee at around the value of $|D_{out}[v] \cup D_{in}[v]|$ – this is because when $N \gg |D_{out}[v] \cup D_{in}[v]|$ in eq. 3, $C_{GL}(v) \approx |D_{out}[v] \cup D_{in}[v]|$. For Twitter and UK-2007, N is never large enough for GraphLab's overhead to reach its cap. Hence, GraphLab's overhead stays lower than Pregel's for those two workloads.

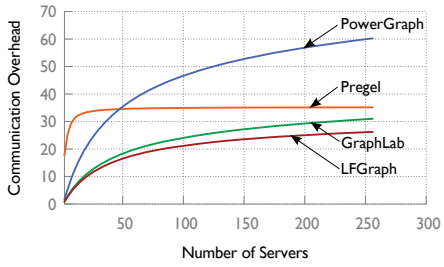
We conclude that in real-world directed graphs, LFGraph's hash-based partitioning and fetch-once communication suffices to achieve a lower communication overhead than existing approaches.

5 Load Balance in Real Graphs

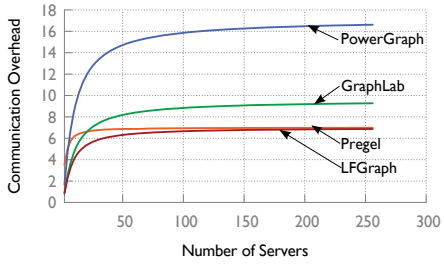
We use synthetic power-law graphs and the three real-world graphs from Table 3 to analyze the computation balancing and communication balancing in LFGraph.



(a) Twitter



(b) UK-2007 Web Graph



(c) Amazon Recommendation Graph

Figure 3: *Communication overhead for real-world graphs*

5.1 Computation Balance

The completion time of an iteration is influenced by imbalance across computation workers. This is because tail workers that take longer than others will cause the entire iteration to finish later.

Prior works [20, 25, 38] have hypothesized that power-law graphs cause substantial computation imbalance, and since real-world graphs are similar to power-law graphs, they do too. In fact, the community has treated this hypothesis as the motivation for intelligent placement schemes, e.g., edge placement [20]. We now debunk this hypothesis by showing that when using random partitioning, while ideal power-law graphs do cause computation imbalance, real-world power-law-like directed graphs in fact do not. The primary reason is subtle differences between the in-degree distributions of these two types of graphs.

First we examine synthetic power-law graphs to see why they do exhibit computation imbalance. A power-law graph has degree d with probability proportional to $d^{-\alpha}$. Lower α means a denser graph with more high

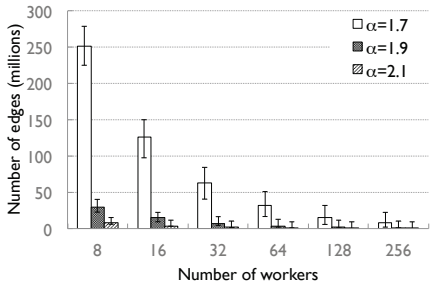


Figure 4: *Computation overhead for synthetic power-law graphs*

degree vertices. We created three synthetic graphs with 10 million vertices each with $\alpha = \{1.7, 1.9, 2.1\}$. We first selected the in-degree of each vertex using the power-law distribution. We then assigned vertices to servers using hash-based partitioning. For simplicity the rest of this section assumes only one computation worker per server.

In LFGraph the runtime of a computation worker during an iteration is proportional to the total number of in-edges processed by that worker. We thus use the latter as a measure for the load at a computation worker. Figure 4 plots, for different cluster sizes, LFGraph's average computation overhead along with error bars that show the maximum and minimum loaded workers. As expected, the average computation load falls inversely with increasing cluster size. However, the error bars show that all synthetic graphs suffer from computation imbalance. This is especially prominent in large clusters – with 256 servers, the maximum loaded worker is 35x slower than the average worker. In fact these slowest workers were the ones assigned the highest degree vertices.

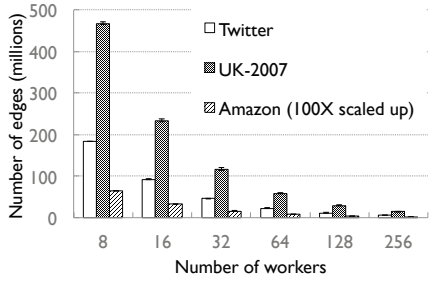


Figure 5: *Computation overhead for real-world graphs*

Next, we repeat the same experiment on the three real-world graphs from Table 3. Figure 5 depicts, for the three real-world graphs, average computation load along with error bars for maximum and minimum computation load. We see that unlike in Figure 4, the bars are much smaller here. Across all the experiments, the most imbalanced worker is only 7% slower than the average worker in its iteration.

The primary reason for this difference in behavior between power-law and real-world graphs is shown in Figure 5, which plots the in-degree distributions of the

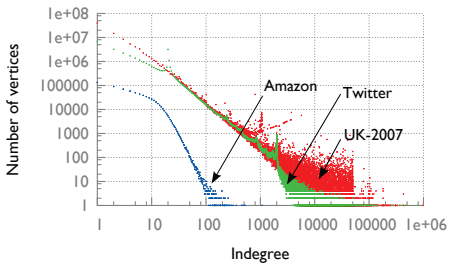


Figure 6: *In-degree distribution for real-world graphs*

three real graphs. While ideal power-law graphs have a straight tail in their log-log plot, each of the real graphs has a *funnel* at its tail. Other real graphs have also been shown to exhibit this pattern [15]. This indicates that in real graphs there are lots of the highest degree vertices – so they spread out and become load-balanced across servers, because the servers are far fewer in number than vertices. On the other hand, in an idealized power-law graph, there is only one of the very highest degree vertex and this causes unbalanced load. We conclude that hash-based partitioning suffices for real power-law-like graphs, and that intelligent placement schemes will yield little benefit in practice.

5.2 Communication Balance

Communication imbalance is important because it can lead to increased processing time. During the communication phase each server receives vertex data from all other servers. Since the transfers are in parallel, if a server *Si* receives more data from *Sj* than from *Sk*, the overall data transfer time will increase even if the average communication overhead stays low.

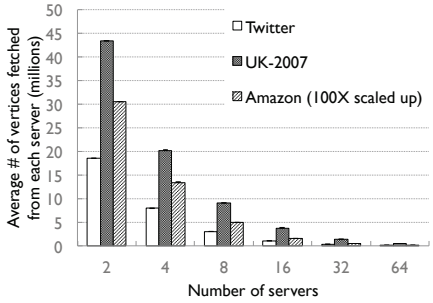


Figure 7: *Communication overhead for real-world graphs*

For the three real-world graphs we measure the vertex data transferred between every pair of servers. Figure 7 plots the average along with bars for maximum and minimum. The small bars indicate that LFGraph balances communication load well. Specifically, across all the experiments, the most imbalanced worker transfers only 10% more data than the average worker in its iteration.

6 Experimental Evaluation

We experimentally evaluate our LFGraph system (implemented in C++) from the viewpoint of runtime, memory footprint, as well as overhead and balancing w.r.t. both computation and communication. While doing so we also compare LFGraph against the best-known graph processing system, i.e., PowerGraph [20], using its open-source version 2.1 [7].[2]

PowerGraph [7, 20] offers three partitioning schemes. In increasing order of complexity, these variants are: i) Random, ii) Batch (greedy partitioning without global coordination), and iii) Oblivious (greedy partitioning with global coordination). We use the synchronous mode of PowerGraph without enabling delta caching. We performed experiments with the asynchronous mode of PowerGraph but found that the runtime did not change much from the synchronous mode. Optimizations such as delta caching can be used orthogonally even in LFGraph, thus yielding similar benefit as in PowerGraph.

Our target cluster is Emulab and our setup consists of 32 servers each with a quad-core Intel Xeon E5530, 12 GB of RAM, and connected via a 1 GigE network. We present results from: 1) the Twitter graph containing 41M vertices and 1.4B edges (Table 3), and 2) larger synthetic graphs with log-normal degree distribution, containing 1B vertices and up to 128B edges.

We study three main benchmarks: i) PageRank, ii) Single-Source Shortest Path (SSSP), and iii) Triangle Count. These applications are chosen because they exhibit different computation and communication patterns: in PageRank all vertices are active in all iterations, while in SSSP the number of active vertices rises in early iterations and falls in later iterations. Thus PageRank is more communication-intensive while SSSP exhibits communication heterogeneity across iterations. The Triangle Count benchmark allows us to evaluate LFGraph on an undirected graph. Further unlike PageRank and SSSP, in Triangle Count the value sizes vary across vertices.

We summarize our key conclusions:

- For communication-heavy analytics such as PageRank, when including the partitioning overhead, LFGraph exhibits 5x to 380x improvement in runtime compared to PowerGraph, while lowering memory footprint by 8x to 12x.
- When ignoring the partitioning overhead, LFGraph is 2x–5x faster than the PowerGraph variants for communication-heavy analytics such as PageRank and Triangle Count. However, for less communication-intensive analytics such as SSSP, LFGraph is 1x–2x faster than PowerGraph (depending on the PowerGraph variant) at large enough cluster sizes.
- LFGraph's hash-based placement scheme achieves both computation and communication balance across workers, and lowers overall runtime.
- For undirected graphs with Triangle Count, LFGraph is 2x faster than PowerGraph, ignoring partition overhead.

[2] Although GraphLab has lower communication overhead, its implementation is slower than PowerGraph.

6.1 PageRank Benchmark

Runtime: We ran the PageRank benchmark with 10 iterations[3] on the Twitter graph. Figure 8 compares LFGraph against the three PowerGraph variants. This plot depicts runtime *ignoring the partitioning iteration* for PowerGraph's Oblivious and Batch variants. Each datapoint is the median over 5 trials.

The reader will notice missing data points for PowerGraph at cluster sizes of 4 servers and fewer. This is because PowerGraph could not load the graph at these small cluster sizes – this is explained by the fact that it stores both in-links and out-links for each vertex, as well as book-keeping information, e.g., mirror locations.

Among the PowerGraph variants, random partitioning is the slowest compared to the intelligent partitioning approaches – this is as expected, since partitioning makes iterations faster. However, LFGraph is 2x faster than the best PowerGraph variant. Thus, even on a per-iteration basis, LFGraph's one-pass compute and fetch-once behavior yields more benefit than PowerGraph's intelligent partitioning.

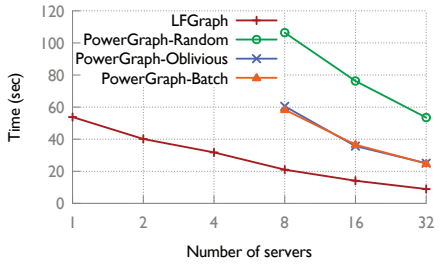


Figure 8: *PageRank runtime comparison for Twitter graph (10 iterations), ignoring partition time.*

Next, Figure 9 includes the partitioning overhead in the runtime, and shows runtime improvement of LFGraph. In a small cluster with 8 servers, LFGraph is between 4x to 100x faster than the PowerGraph variants. In a large cluster with 32 servers the improvement grows to 5x–380x. The improvement is the most over the intelligent partitioning variants of PowerGraph because LFGraph avoids expensive partitioning. We observed that PowerGraph's intelligent partitioning schemes contributed 90% –99% to the overall runtime for the PageRank benchmark.

[3] Our conclusions hold even with larger number of iterations.

LFGraph's improvement increases with cluster size. This indicates intelligent partitioning is prohibitive at all cluster sizes. In a small cluster, distributed graph processing is compute-heavy thus intelligent partitioning (e.g., in PowerGraph) has little effect. In a large cluster, intelligent partitioning can speed up iterations – however, the partitioning cost itself is directly proportional to cluster size and contributes sizably to runtime.

Figure 9: *PageRank runtime improvement for Twitter graph (10 iterations), including partition time.*

Memory Footprint: While PowerGraph stores in- and out-links and other book-keeping information, LFGraph relies only on in-links and publish-lists (Section 3). We used the `smem` tool to obtain LFGraph's memory footprint. For PowerGraph we used the heap space reported in the debug logs. Figure 10 shows that LFGraph uses 8x to 12x less memory than PowerGraph.

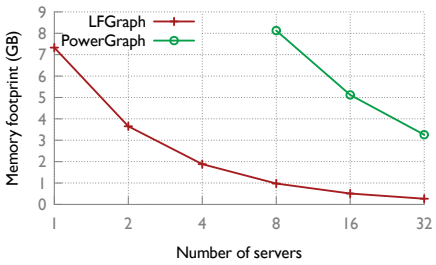


Figure 10: *Memory footprint of LFGraph and PowerGraph*

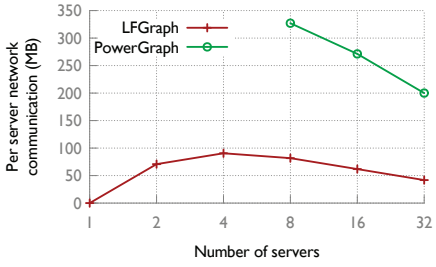
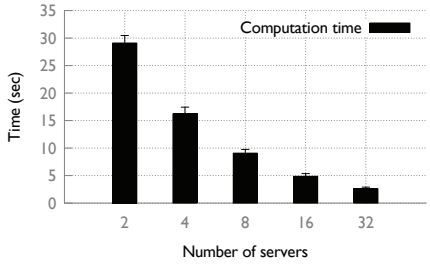


Figure 11: *Network communication for LFGraph and PowerGraph*

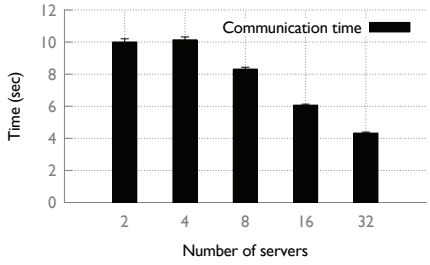
Communication Overhead: Figure 11 shows that LFGraph transfers about 4x less data per server than PowerGraph – this is consistent with our analysis in Section 4. We also notice that the LFGraph's communication reaches a peak at about 4 servers. This is because the per-server overhead equals the total communication overhead divided by the number of servers. As the cluster size is increased, there is first a quick rise in the total communication overhead (see Section 4 and Figure 3). Thus the per-server overhead rises at first. However as the total communication overhead plateaus out, the cluster size increase takes over, thus dropping the per-server overhead. This creates the peak in between.

Finally, we observe that although the total communication overhead rises with cluster size (Figure 3), in the real deployment the per-iteration time in fact falls (Figure 8). This is because of two factors: i) communication workers schedule network transfers in parallel, and ii) Emulab offers full bisection bandwidth offered between every server pair. Since (ii) is becoming a norm, our conclusions generalize to many datacenters.

Computation and Communication Balance: As a counterpart to Section 5, Figure 12 shows, for different cluster sizes, the average overhead at a server (measured in time units) along with the standard deviation bars. The small bars indicate good load balance in LFGraph. The communication bars are smaller than the computation bars primarily due to the stability of Emulab's VLAN.



(a) Computation Balance



(b) Communication Balance

Figure 12: *Computation and communication balance in LFGraph*

Computation vs. Communication: Figure 13 shows the split between computation and communication at different cluster sizes. First, computation time decreases with increasing number of servers, as expected from the increasing compute power. Second, the communication time curve mirrors the per-server network overhead from Figure 11. Third, compute dominates communication in small clusters. However, at 16 servers and beyond, the

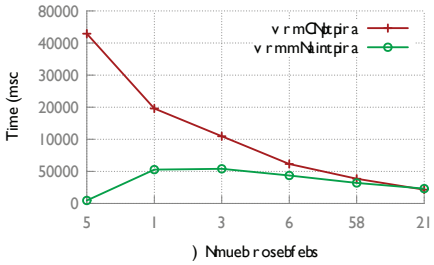


Figure 13: *Communication and computation split of PageRank computation*

two phases take about the same time. This indicates that the importance of balancing computation and communication load in order to achieve the best runtime. LFGraph achieves this balance.

Breaking Down LFGraph's Improvement: In order to better understand the sources of improvement in LFGraph we plot the runtime of 10 iterations of PageRank along with the time spent in the communication phase, for LFGraph and PowerGraph, ignoring partition time, in Figure 14. While LFGraph's communication and computation phases are separable, PowerGraph's phases overlap, making it difficult to evaluate each phase. We use the data transferred and network bandwidth to compute PowerGraph's communication overhead. The plot indicates that LFGraph's performance improvement is due to shorter communication and computation phases. First, during the iterations, LFGraph transfers less data. So, the communication phase is shorter. Second, LFGraph processes only incoming edges. So, its computation phase is shorter compared to that of PowerGraph.

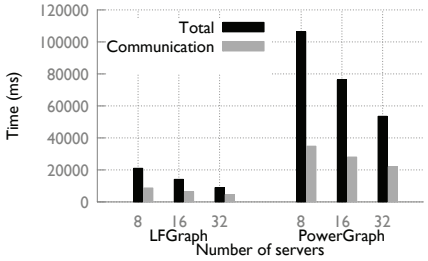


Figure 14: *Breaking down the performance gain of LFGraph compared to PowerGraph, ignoring partition time*

One important observation from the plot is that although we expected the computation phase to be 2x faster in LFGraph compared to PowerGraph (because LFGraph processes incoming edges, while PowerGraph processes both incoming and outgoing edges), the improvement actually ranges from 5x to 7x. This is because communication and computation overlap in PowerGraph. As a result, the increased communication overhead negatively affects the computation time in PowerGraph. In addition, because communication and computation are disjoint in LFGraph, we can optimize communication by batching data transfers. On the other hand, PowerGraph is unable to perform such optimizations.

PageRank on Undirected Graph: We evaluate LFGraph on an undirected graph using PageRank on the Twitter graph. This is motivated by two reasons. First, we showed in Section 4.1 that LFGraph's improvement over PowerGraph is the largest when the incoming and outgoing edge lists of vertices are disjoint. The improvement is the lowest when the incoming and outgoing edge lists of vertices overlap completely, i.e., the graph is undirected. Second, it is important to compare the memory overhead of LFGraph and PowerGraph for undirected graphs because PowerGraph already stores both incoming and outgoing edges. For an undirected graph we expect that PowerGraph's memory overhead for storing the edge list would be similar to that of LFGraph's.

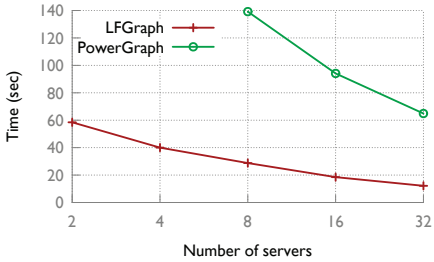


Figure 15: *PageRank runtime comparison for undirected Twitter graph (10 iterations), ignoring partition time*

Figure 15 shows PageRank's runtime on the undirected version of the Twitter graph for 10 iterations, ignoring partition time. As before, PowerGraph was unable to load the graph on less than 8 servers. Interestingly, LFGraph was unable to run the benchmark on a single server, because of the increased size of the undirected version of the graph. Overall, even for undirected graphs LFGraph runs faster than PowerGraph by 5x (as opposed to the 6x observed for the directed graph case).

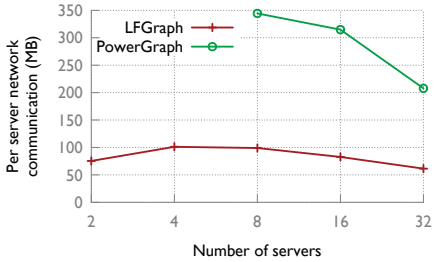


Figure 16: *Network communication for LFGraph and PowerGraph on undirected Twitter graph (PageRank benchmark)*

The reduction in improvement is classified by Figure 16, where we plot the communication overheads of LFGraph and PowerGraph. For undirected graphs, LFGraph's communication overhead is up to 3.4x lower than PowerGraph's. Recall that for directed graphs LFGraph exhibited 4.8x lower communication overhead

compared to PowerGraph. This behavior conforms to our analysis in Section 4.1.

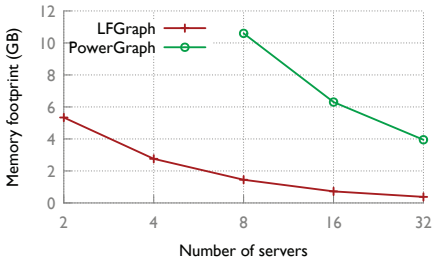


Figure 17: *Memory footprint of LFGraph and PowerGraph for the undirected Twitter graph (PageRank benchmark)*

Finally, in Figure 17, we show the memory overhead of LFGraph and PowerGraph for the undirected Twitter Graph. Contrary to our speculation, we observe that LFGraph's memory overhead is 7x–10x lower compared to PowerGraph's memory overhead. This is due to two reasons. First, PowerGraph has to maintain location information of the mirrors, which requires additional memory. Second, PowerGraph maintains two lists even for undirected graphs when the incoming and outgoing lists are identical. Therefore, we conclude that LFGraph performs better than PowerGraph even for undirected graphs on the PageRank benchmark.

6.2 SSSP Benchmark

We ran the SSSP benchmark on the Twitter graph. The benchmark ran for 13 iterations. Figure 18 shows the comparison between LFGraph and the three PowerGraph variants, ignoring the partitioning time for PowerGraph's Oblivious and Batch strategies.

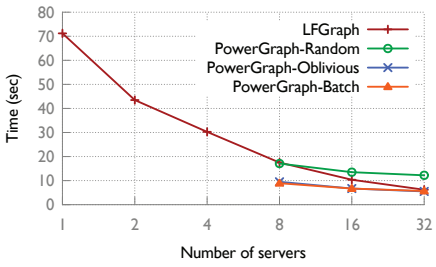


Figure 18: *SSSP runtime comparison for Twitter graph, ignoring partition time*

First, we observe that LFGraph successfully ran the benchmark on a small cluster (4 servers and less), while PowerGraph could not, due to its memory overhead. Second, unlike in PageRank (Section 6.1), LFGraph and PowerGraph are comparable. At 8 servers, LFGraph's performance is similar to that of PowerGraph's random placement but worse than PowerGraph's intelligent placement strategies. This is due to two factors: i) SSSP incurs less communication than PageRank, especially in later iterations, and ii) LFGraph does not store out-links, thus unlike PowerGraph it cannot activate/deactivate vertices for the next iteration. Recall that in LFGraph, a vertex has to iterate through all of its in-neighbors to check which were updated in the previous iteration.

At 16 servers and beyond, LFGraph is better than PowerGraph's random placement (for e.g., 2x better at 32 servers). At 32 servers LFGraph exhibits similar runtime as the Oblivious and Batch strategies. This is because communication starts to dominate computation at these cluster sizes.

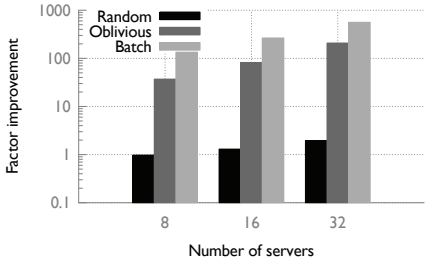


Figure 19: *SSSP runtime improvement for Twitter graph, including partition time*

Finally, Figure 19 shows LFGraph's improvement over the PowerGraph variants, when including the partitioning time. We observe up to a 560x improvement. This is because PowerGraph spent 98.5%-99.8% of the time for intelligent placement purpose.

We conclude that for SSSP-like analytics LFGraph is almost always preferable to PowerGraph, with the only exception being the corner-case scenario where the graph is loaded once and processed multiple times and in a cluster that is medium-sized (8–16 servers in Figure 18).

6.3 Undirected Triangle Count

We further validate the performance on undirected graphs. We present results from Undirected Triangle Count benchmark on the undirected version of the Twitter graph. The values associated with the vertices are the edge-lists. Thus, the sizes of values are variable and large compared to PageRank (a single floating point) and SSSP (a single integer). This benchmark captures communication that is both variable across vertices and much larger in volume than our previous experiments.

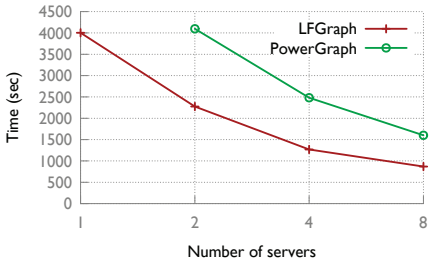


Figure 20: *Triangle Count on the undirected Twitter Graph, ignoring partition time*

Due to the large resource requirement of Triangle

Count computation, we performed this experiment on a beefy cluster – an 8 server Emulab cluster with each server containing four 2.2 GHz E5-4620 Sandy Bridge processor, 128 GB RAM, and connected via a 10 GigE network. We make two important observations here. First, LFGraph continues to outperform PowerGraph in terms of runtime – the improvement is about 2x. Second, the experiment could not be run on a single machine in case of PowerGraph due to its extensive memory requirement. We conclude that LFGraph's benefits persist in communication-intensive undirected graph applications.

6.4 Larger Graphs

We create 10 synthetic graphs varying in number of vertices from 100M to 1B, and with up to 128B edges. We run the SSSP benchmark on it. These graphs are generated by choosing out-degrees from a log-normal distribution with $\mu = 4$ and $\sigma = 1.3$, with out-neighbors selected at random. To avoid the network overhead for graph creation, we cap the in-degree of each vertex at 128 and choose in-neighbors at random such that the probability of choosing a vertex as an in-neighbor follows the aforesaid log-normal distribution. Other papers [30] have used similar graphs for evaluating their systems.

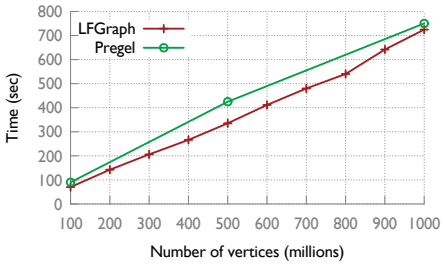


Figure 21: *SSSP runtime for a synthetic graph*

Due to the graph size we performed this experiment on a beefy cluster – a 12 server Emulab cluster with each server containing four 2.2 GHz E5-4620 Sandy Bridge processors, 128 GB RAM, and connected via a 10 GigE network.

Figure 21 shows the results for LFGraph. We juxtapose the Pregel performance numbers from [30] – those Pregel experiments used 300 servers with 800 workers. In comparison, our LFGraph deployment with only 12 servers with 96 workers uses around 10x less compute power. Even so we observe an overall improvement in runtime. The cores per server ratio is higher in the LFGraph setting – we believe this is in keeping with current architecture and pricing trends. Thus we conclude that LFGraph can perform comparably to industrial-scale systems while using only a fraction of the resources.

7 Related Work

Single-Server Systems: Single-server graph processing systems are limited by memory and cannot process large graphs [22]. GraphChi [27] leverages the disk for large graph processing on a single server, making it slower than distributed frameworks. Grace [33] relies on machines with many cores and large RAM. It has two drawbacks – parallelizability is limited by the number of cores, and real-world large graphs cannot be stored in a single machine's memory.

Distributed Graph Processing Frameworks: Pregel has been the inspiration for several distributed graph processing systems, including Giraph [1], GoldenOrb [6], Phoebus [10], etc. These systems suffer from unscalability at small cluster sizes. For instance, we found that Giraph was unable to load a graph with 10M vertices and 1B edges on a 64 node cluster (16 GB memory each). Others reported similar experiences [20].

GraphLab [28, 29] and PowerGraph [20] also support asynchronous computations. Asynchronous models do not have barriers – so, fast workers do not have to wait for slow workers. However, asynchrony makes it difficult to reason about and debug such programs.

Distributed-matrix models have been used for graph processing [3, 39]. These are harder to code in as they do not follow the more intuitive 'think like a vertex' paradigm (i.e., vertex programs) that Pregel, GraphLab, and PowerGraph do.

Finally, Piccolo [32] and Trinity [36] realize a distributed in-memory key-value store for graph processing. Trinity also supports online query processing on graphs and is known to outperform MPI-based implementations such as PBGL [21]. However, both Piccolo and Trinity require locking for concurrency control. We performed experiments with Trinity and observed that LFGraph achieves a 1.6x improvement.

General-purpose Data Processing Frameworks: General-purpose data processing frameworks such as MapReduce [18] and its variants [2], Spark [40], etc. can be used for graph analytics [4, 16, 19, 23]. However, they are not targeted at graph computations, thus they are difficult to program graph algorithms in due to the model mismatch. Further their performance is not competitive with graph processing frameworks [20].

Graph Databases: Graph databases such as FlockDB [5], InfiniteGraph [8], and Neo4j [9] are increasingly being used to store graph-structured data. Although these databases support efficient query and traversal on graph-structured data, they are not designed for graph analytics operations.

Performance Optimization: Various techniques have been designed to optimize performance of graph-based computations. These techniques include multilevel partitioning [11, 24], greedy partitioning [37], join partitioning and replication strategies [34]. Based on our results, we believe that such complex partitioning schemes can be avoided while still improving performance. GPS [35]

uses dynamic repartitioning schemes for runtime optimization. Mizan [25] uses dynamic load balancing for fast processing, Surfer [17] uses bandwidth-aware placement techniques to minimize cross-partition communication. These dynamic techniques can be applied orthogonally inside LFGraph.

8 Summary

Fast and distributed processing of large real-world graphs requires low and balanced computation and communication, low pre-processing overhead, low memory footprint, and good scalability. We have presented LFGraph, a system for fast, scalable, distributed, in-memory graph analytics. To satisfy the above requirements, LFGraph uses cheap hash-based graph partitioning, publish-subscribe information flow, fetch-once communication, single-pass computation, and in-neighbor storage. We have shown analytically that these techniques incur lower communication overhead than existing systems, and exhibit good load balance. Ignoring the expensive pre-processing stage, LFGraph is faster than the best existing system by 2x–5x for PageRank, by 1x–2x for SSSP, and by 2x for Triangle Count. When including the pre-processing stage, LFGraph is 5x–380x and 1x–560x faster for PageRank and SSSP respectively. We have also shown, both analytically and experimentally, that intelligent graph partitioning schemes yield little benefit and are prohibitive.

Acknowledgements: We thank our shepherd Benjamin Wester and anonymous reviewers for their insightful comments and suggestions.

References

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

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache Hama. http://hama.apache.org.

[4] Apache Mahout. http://mahout.apache.org/.

[5] FlockDB. https://github.com/twitter/flockdb.

[6] GoldenOrb version 0.1.1. http://goldenorbos.org/.

[7] GraphLab version 2.1. http://graphlab.org.

[8] InfiniteGraph. http://www.objectivity.com.

[9] Neo4j. http://www.neo4j.org.

[10] Phoebus. https://github.com/xslogic/phoebus.

[11] ABOU-RJEILI, A., AND KARYPIS, G. Multilevel Algorithms for Partitioning Power-Law Graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS '06)* (2006).

[12] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11* (2011), pp. 587–596.

[13] BOLDI, P., AND VIGNA, S. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW '04)* (2004), pp. 595–601.

[14] BORDINO, I., BOLDI, P., DONATO, D., SANTINI, M., AND VIGNA, S. Temporal Evolution of the UK Web. In *Proceedings of the 1st International Workshop on Analysis of Dynamic Networks (ICDM-ADN '08)* (2008), pp. 909–918.

[15] BRODER, A., KUMAR, R., MAGHOUL, F., RAGHAVAN, P., RAJAGOPALAN, S., STATA, R., TOMKINS, A., AND WIENER, J. Graph Structure in the Web. In *Proceedings of the 9th International World Wide Web Conference (WWW '00)* (2000), pp. 309–320.

[16] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 285–296.

[17] CHEN, R., YANG, M., WENG, X., CHOI, B., HE, B., AND LI, X. Improving Large Graph Processing on Partitioned Graphs in the Cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)* (2012), pp. 1–13.

[18] DEAN, J., AND GHEMAWA, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI '04)* (2004), pp. 137–149.

[19] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)* (2010), pp. 810–818.

[20] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation(OSDI '12)* (2012), pp. 17–30.

[21] GREGOR, D., AND LUMSDAINE, A. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *Proceedings of the 4th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing* (2005).

[22] HAGBERG, A., SCHULT, D., AND SWART, P. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference* (2008), pp. 11–15.

[23] KANG, U., AND FALOUTSOS, C. E. T. C. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09)* (2009), pp. 229–238.

[24] KARYPIS, G., AND KUMAR, V. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing 48* (1998), 96–129.

[25] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys '13)* (2013).

[26] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)* (2010), pp. 591–600.

[27] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)* (2012), pp. 31–46.

[28] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. GraphLab: A New Parallel Framework for Machine Learning. In *Proceeding of the 26th Conference on Uncertainty in Artificial Intelligence (UAI '10)* (2010), pp. 340–349.

[29] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[30] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD '10)* (2010), pp. 135–146.

[31] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, 1999.

[32] POWER, R., AND LI, J. Piccolo: Building Fast and Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)* (2010), pp. 1–14.

[33] PRABHAKARAN, V., WU, M., WENG, X., MCSHERRY, F., ZHOU, L., AND HARIDASAN, M. Managing Large Graphs on Multi-Cores with Graph Awareness. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12)* (2012), pp. 41–52.

[34] PUJOL, J. M., ERRAMILLI, V., SIGANOS, G., YANG, X., LAOUTARIS, N., CHHABRA, P., AND RODRIGUEZ, P. The Little Engine(s) That Could: Scaling Online Social Networks. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)* (2010), pp. 375–386.

[35] SALIHOGLU, S., AND WIDOM, J. GPS: A Graph Processing System. Technical Report, Stanford University, 2012.

[36] SHAO, B., WANG, H., AND LI, Y. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '13)* (2013).

[37] STANTON, I., AND KLIOT, G. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)* (2012), pp. 1222–1230.

[38] SURI, S., AND VASSILVITSKII, S. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the 20th international Conference on World Wide Web (WWW '11)* (2011), pp. 607–614.

[39] VENKATARAMAN, S., BODZSAR, E., ROY, I., AUYOUNG, A., AND SCHREIBER, R. S. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys '13)* (2013).

[40] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)* (2012).