Scaling PageRank to 100 Billion Pages

Stergios Stergiou*
utopcell@google.com

ABSTRACT

Distributed graph processing frameworks formulate tasks as sequences of supersteps within which communication is performed asynchronously by sending messages over the graph edges. PageRank's communication pattern is identical across all its supersteps since each vertex sends messages to all its edges. We exploit this pattern to develop a new communication paradigm that allows us to exchange messages that include only edge payloads, dramatically reducing bandwidth requirements. Experiments on a web graph of 38 billion vertices and 3.1 trillion edges yield execution times of 34.4 seconds per iteration, suggesting more than an order of magnitude improvement over the state-of-the-art.

CCS CONCEPTS

• **Information systems → Graph-based database models**; **Computing platforms**.

KEYWORDS

graph processing, implicit targets, pagerank

ACM Reference Format:
Stergios Stergiou. 2020. Scaling PageRank to 100 Billion Pages. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3366423.3380035

1 INTRODUCTION

The advent of web search and the popularity of social networks has led to the creation of very large graphs. Facebook reports 2 billion monthly active users [5] and at least 1 trillion connections between them [12], while Google reports indexing hundreds of billions of web pages [6]. Executing PageRank at this scale requires efficient graph processing systems. Several such systems have been proposed in the recent past, including Pegasus [19], Pregel [24], HaLoop [9], Giraph [4], comb. BLAS [10], PowerGraph [14], Giraph++ [32], Naiad [25], GPS [29], Mizan [20], Spark/GraphX [15], Blogel [33], PowerLyra [11], BIDMat/Kylix [35], X-Stream [28], Chaos [27] and GraphChi [21]. Extensive comparisons between state-of-the-art frameworks can be found in [16] and [34].

One of the main tasks of a large-scale graph processing framework is to efficiently implement the following primitive: iterate over the graph's vertices, generate messages for each of the edges' targets, and deliver all generated messages to the appropriate targets. To achieve this, the input graph is first split into a number of partitions. Partitioning can either be user-driven, the output of a graph balancing algorithm or random, but typically graph vertices are randomly placed in partitions. A source partition transmits messages towards a target partition in the form of {message target id, message payload} tuples. Target identifiers are necessary so that the receiving partition is able to deliver the message to the appropriate vertex upon reception. However, vertex ids are large hashes whose size can dominate the size of the tuple (see Section 3.1). This is especially true for PageRank, where payloads are typically only 32-bits long. While some graph processing systems map vertex ids to a contiguous range of integers [29], thereby reducing their size, relabeling vertex ids is expensive for web-scale graphs.

Orthogonally, even though random placement generates well-balanced partitions, any partitioning scheme will lead to inlinks imbalance when there exist large in-degree vertices. To address this, graph processing systems introduced partitioning schemes where individual edges (instead of vertices) are randomly partitioned [14]. This approach indeed leads to partitions that are also inlink-balanced, but imposes a significant overhead as the outlinks of any given vertex no longer reside in the same partition. As a consequence, multiple partitions need to maintain state information of a given vertex and additional communication is necessary to keep this information synchronized. Alternatively, source-side aggregation has been used to address the problem, where messages towards the same target are combined before sent. This is effective because there will be a large number of edges on every partition pointing to a large in-degree vertex. However, source-side aggregation requires maintaining a mapping from unique edge targets to aggregated payloads, which is impractical since the number of unique edge targets per partition is usually the same order of magnitude as the number of all edges in the partition.

In this paper, we introduce a new communication paradigm for distributed graph processing systems, specifically optimized for PageRank-like communication patterns, which allows us to: (1) exchange messages between partitions by only transmitting message payloads; and (2) perform partial source-side message aggregation, only for frequent message targets. First, we impose an order on the messages exchanged between two arbitrary partitions within each superstep. This allows us to memorize the order of incoming target ids per {source, target} partitions tuple. Consequently, partitions are able to transmit just message payloads: the corresponding target partitions are expected to pair the incoming payloads with the correct target vertex identifiers upon receiving them (see Figure 1). Second, we observe that after incoming id orders have been obtained, source partitions no longer need to maintain target ids for each edge and therefore, only the target *partition* id is maintained for each edge. This allows us to accelerate the local iterations over each partition's edges by transforming vertex identifiers into a more compact form that maintains just enough information for the system to be able to send the corresponding messages (see Section 3.3). Third, we establish a fast, deterministic method for selecting

*Work done while author was at Yahoo! Research. Author is now at Google.

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380035

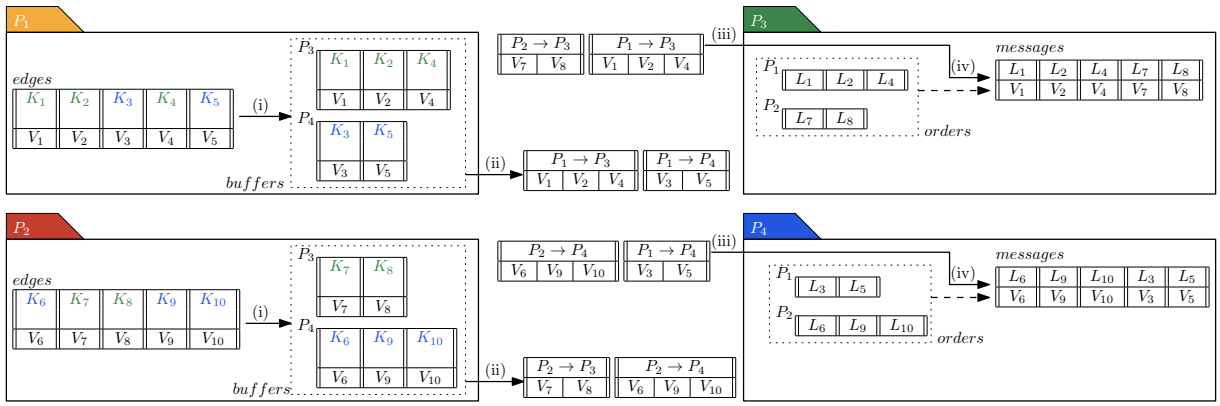

Figure 1: Implicit Targeting. Edge target identifiers are labeled as K_i. Edge payloads are labeled as V_i. Partitions P_1 and P_2 are sending messages to partitions P_3 and P_4. (i) each partition iterates over the edges assigned to it and buffers messages per target partition; (ii) buffered messages are sent to their respective target partitions; (iii) packets arrive at target partitions; (iv) incoming messages containing payloads are paired with pre-computed target vertex identifiers (labeled as L_j) and delivered to the application. L_j are local vertex identifiers, only known to their partitions, corresponding to the global K_j identifiers.

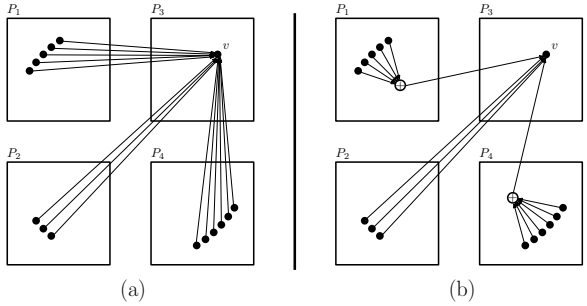

Figure 2: Source-side Aggregation. Partitions P_1, P_2 and P_4 contain 5, 3 and 6 edges respectively towards vertex v in partition P_3. (a) aggregation is not performed; (b) source-side aggregation is performed with frequency threshold equal to 4. Messages from partitions P_1 and P_4 are aggregated, while messages from partition P_2 are sent directly to v as they are below the frequency threshold.

a small subset of target vertices in each partition for which we perform source-side payload aggregation. Specifically, we deduce a set of target ids per partition whose elements are the target of at least f_{TH} edges in the partition, where f_{TH} is a system-generated local frequency threshold (see Figure 2). This set includes all large in-degree vertices w.h.p., is small compared to the number of unique target ids, and is obtained via a lightweight streaming algorithm that requires memory only proportional to the size of the set.

We implemented our new communication paradigm, aptly named *implicit targeting* (as target ids are implicit in messages), on *Hronos*, a Yahoo! graph processing framework that has previously been used for ML [31] [18] and other [30] distributed workloads. PageRank [26], the classic web graph algorithm that simulates the surfing behavior of a random user [22], has been extensively used for evaluating the performance of graph processing frameworks. We evaluated PageRank on graphs with up to 96.3 billion vertices. Specifically, for a graph of ~38 billion vertices and ~3.1 trillion edges, we obtained execution times of 34.4 seconds per iteration. We also compared our system against Giraph on smaller public graphs and demonstrated a 30X mean speedup.

The rest of the paper is organized as follows. In Section 2, we briefly describe PageRank and the online algorithm used to detect frequent elements. In Section 3 we describe our implicit targeting communication pattern. In Section 4 we analyze the performance of our source-side message aggregation algorithm. In Section 5 we report experimental results.

2 PRELIMINARIES

2.1 PageRank

Let C be the set of pages in a web corpus, $d(v)$ be the number of outlinks of page v and W be a column stochastic matrix that represents the connections between pages such that $W[u, v] = 1/d(v)$ if there exists a link from page v to page u and 0 otherwise. PageRank imposes a total order on C assuming ties are broken arbitrarily. It denotes the probability that a user (called the random surfer) would end up at a particular page if they started from an arbitrary page and either visited one of the outlinks from that page, or with a small probability α jumped to a pre-specified set of pages (named the teleportation set T). For pages without any outlinks the user would jump directly to a page in T.

For each graph vertex v we maintain the current iteration PageRank ($rank(v)$) and the next iteration PageRank ($next\_rank(v)$) values. $rank(v)$ is initialized to either 1.0 or to a value computed at a previous sync point. In the main computation phase, before processing any edges, all local $next\_rank$ values are copied into $rank$, just before they are initialized to 0. Each process iterates over all edges present in its graph partition. For each edge (u, v) it sends a ΔPR message to target vertex v, where $\Delta\text{PR} = (1 - \alpha) \cdot rank(u)/d(u)$. It also adds $\alpha \cdot rank(u)$ to a local teleportation variable $teleportation$. If u has no outlinks, it adds the complete $rank(u)$ to $teleportation$. Concurrently, upon reception of a message the process adds the

Algorithm 1 FrequentItems< H >

```
1:  function Process(value)
2:      if value ∈ items.keys() then
3:          items[value] ← items[value] + 1;
4:      else if items.size() < H then
5:          items[value] = 1;
6:      else
7:          for item ∈ items.keys() do
8:              items[item] ← items[item] − 1
9:              if items[item] = 0 then
10:                 delete items[item];
```

corresponding ΔPR to the $next\_rank(v)$ of the graph vertex v associated with the message. If v is not one of the vertices that the process is maintaining, then it redirects the incoming ΔPR to *teleportation*. This can happen if edge (u, v) points to an uncrawled page. At the end of the iteration, the master process collects all local *teleportation* values from its peers, aggregates them, and distributes the sum to its peers and itself.

2.2 Frequent Items

Following, we present an algorithm that detects all target ids of frequency $f > n/(H+1)$, of which there can be at most H, where n is the number of all target ids in the partition. Let us first describe the algorithm via an example for the base case of $H = 1$ [8]. Given an array A of n elements, we would like to detect the majority element if one exists. That is, we would like to detect an element that appears more than $n/2$ times in the array. The algorithm works in a streaming mode: it does not know n in advance and only examines each element once. It maintains a *counter* and a *candidate* variable. Initially, $counter = 1$ and $candidate = A[1]$. It examines each of the array elements $A[2], \ldots, A[n]$. If the current element is the same as the one in *candidate*, then *counter* is incremented. Otherwise, *counter* is decremented. If *counter* becomes negative, then *candidate* is set to the current element and *counter* is reset to 1. In the end, if a majority element exists, it will be present in *candidate*. Intuitively, this holds because the number of all non-majority elements in A is smaller than the number of times the majority element appears in A and therefore they are not enough to displace it from *candidate* after all elements have been processed. The algorithm is generalized in a straightforward manner for arbitrary H [13]. It maintains a map (*items*) between each *candidate* and its corresponding *counter*. It has $O(n)$ amortized time complexity and $O(H)$ space complexity. It is depicted on Algorithm 1.

3 IMPLICIT TARGETING

Implicit targeting communication assumes that the target partition understands how to deliver messages to a target vertex without actually receiving the graph vertex identifier along with the message it corresponds to. The underlying idea is the following: During each superstep, a target partition j receives sets of messages from its peers without any specific order of arrival. However, sets of messages arriving from a particular source partition i will be delivered in-order to j. We guarantee this by adopting unidirectional message queues for point-to-point communication between i and j. So long as the source partition i iterates over its edges in the same order, the subset of the edges corresponding to target partition j will also be processed in the same order and therefore, the corresponding messages will be sent out, and be received by partition j in the same order.

3.1 Vertex Identifiers

For large graphs, we can see that identifiers that are smaller than 128-bits are impractical via a birthday paradox argument: Let H be the number of unique values of a vertex id and $q(n, H) = 1 - p(n, H)$ be the probability that no two values are the same in a set of n values randomly drawn from $[1..H]$ with repetition. Then

$$\begin{aligned} q(n,H) &= 1 \cdot \left(1 - \frac{1}{H}\right) \cdot \left(1 - \frac{2}{H}\right) \cdots \left(1 - \frac{n-1}{H}\right) \\ &\approx e^{-1/H} \cdot e^{-2/H} \cdots e^{-(n-1)/H} \\ &\approx e^{-n^2/(2H)} \end{aligned}$$

Solving for n, we obtain the size of graph $n(p, H)$ for which there will be collisions between its vertex identifiers with probability at least p:

$$n(p,H) \approx \sqrt{2H \ln \frac{1}{1-p}}$$

Assuming 64-bit vertex identifiers and $p = 10^{-6}$, there will be collisions between ids with graphs of as few as 6.1 million vertices. Arguably, such graphs are not the target of a large-scale graph framework. Moreover, for public graphs such as the Common Crawl Graph (see Table 1), the collision probability is 26%. On the other hand, 128-bit ids allow for graphs with 820 billion vertices before collisions for $p = 10^{-15}$.

3.2 Target Learning Phase

In order for each partition to learn the graph's vertex identifiers, a *target-learning* phase is executed during graph loading: After a partition has loaded its subset of the graph, it knows the *partition out-degrees* for each of its peer partitions, i.e. the number of edges that point to graph vertices in each of the other partitions. In the first part of the target-learning phase, partitions also collect *partition in-degrees*, i.e. the number of edges they should be expecting from each peer partition during each superstep. In the second part of the target-learning phase, each partition scans its subset of the graph and sends out messages that contain the edges' primary target identifiers. The partition can quickly compute the target for each edge as it knows the exact range of identifiers each partition is responsible for. At the same time, each partition receives sets of messages from its peers. For each message, it obtains the mapping from the primary vertex identifier to the local vertex identifier, which it then places at the end of the incoming order vector for the corresponding source partition. After sending out all messages, each partition sends termination messages to each peer.

3.3 Partition Learning Phase

Being able to send payloads without target ids significantly reduces communication requirements and surfaces a new opportunity for optimization. First, we note that it is no longer necessary for each source partition to know the primary vertex identifiers of its edges'

targets. This information would be necessary for a traditional communication pattern, as the target partition needs to associate the primary id to a local id in order to process each payload appropriately. However, in the implicit targeting communication pattern, it is sufficient for the source partition to know just the target partition id, and for the target partition to know the originating partition for each *set* of incoming payloads. Therefore, each partition iterates over its edges, deduces their target partitions, and stores only target partition identifiers for each of the edges. This typically reduces the footprint for each partition's edges eight-fold (since 128-bit identifiers for vertices are replaced with 16-bit identifiers for the partitions they belong to) and speeds up iterating over a partition's edges. The *partition learning phase* occurs concurrently with the target learning phase. We note that the time required to establish the implicit targeting communication pattern is always less than the execution time of a single superstep that iterates over all graph edges in a traditional communication pattern, and is therefore very efficient.

4 MESSAGE AGGREGATION

Maintaining in-order vectors per partition introduces another challenge to the system: As each target partition may access its $K-1$ incoming-message orders non-sequentially, it would be inefficient to store them on secondary storage and are therefore maintained in memory, which is a limited resource. While graph partitions are typically well-balanced, inlinks per partition are not. In fact, we have observed inlink count imbalances of five-fold or more. One way to address this challenge would be to rebalance the graph as a preprocessing step. However, graph partitioning algorithms are expensive to execute on large-scale graphs, so that any benefit is typically not enough to offset the delay they introduce to the overall execution time except for very long-running computations. An alternative to graph rebalancing would be to combine messages with the same target primary identifier on the source side. This approach would indeed address the inlinks imbalance. However, in order to support source-side aggregation, one needs to maintain a structure that is $O(p_u)$, where p_u is the number of unique edge targets in the partition. Hash-based graph partitioning implies that this is $p_u = O(p_s)$, where p_s is the number of edge targets in the partition. Therefore, this approach simply moves the memory bottleneck to a different component of the system.

A natural way to refine this approach would be to selectively perform source-side aggregation only for the very frequent target primary ids (target ids with high inlink counts within the partition). This optimization indeed removes the memory overhead of source-side aggregation. Nevertheless it introduces another challenge: It is not possible to deterministically compute the frequency of the elements in an array of size n unless $\Omega(n)$ memory is used [23]. Although this would be a one-time operation, it is still inefficient for very large graphs. We observe the following: We do not really need to know the exact frequency of all target ids in each partition. We only need to identify those that cross a given frequency threshold. This greatly simplifies the time and space complexity of the problem and allows us to adopt the algorithm in Section 2.2. In this context, n is equal to p_s.

The algorithm is executed while loading the graph partition. Each primary target identifier is examined via Process() (Algorithm 1). It is essentially executed at no cost, as graph loading is heavily I/O bound and the processing of each vertex is pipelined with the loading of the next vertex. We note that the elements in the set FI of the resulting H target identifiers are not guaranteed to appear more than $n/(H+1)$ times. Rather, it is only guaranteed that the target identifiers whose frequency is larger than $n/(H+1)$ will be part of the resulting set of H ids.

Let us now examine the performance of a system that adopts source-side aggregation. Let v be a vertex with large in-degree d_v, K be the number of partitions and $\mu = d_v/K$. Conditioned on v, the sources of all edges (s_i, v) are evenly distributed across all partitions K. Let us consider applying aggregation on a specific partition with frequency $f = \mu$. Then the probability that v is in FI is 50%. As a result, we expect $d_v/2 + K$ messages to be sent across all partitions corresponding to the (s_i, v) edges. This approach is very inefficient, as it only discards half of the messages towards vertex v. Fortunately, we will subsequently show that almost no edge escapes source-side aggregation when $f = \mu/2$.

Lemma 4.1. *The probability that a specific partition P has at most k edges towards vertex v with in-degree d_v is:*

$$\sum_{i=0}^{k} \binom{\mu K}{i} \frac{1}{K}^{i} \left(1 - \frac{1}{K}\right)^{\mu K - i} \tag{1}$$

Proof. Let X_i be the indicator variable that denotes whether the i-th edge has been placed in P. Then X_i is a Bernoulli RV with success probability $1/K$. Therefore $\sum_i X_i$ follows the binomial distribution characterized by d_v trials and $1/K$ success probability. □

We will use Hoeffding's inequality to bound Eq. (1).

Theorem 4.2 (Hoeffding [17]). *Let X_i be d_v IID Bernoulli RVs with success probability $1/K$. Then it holds that:*

$$\mathbb{P}\left(\sum_i X_i \leq \mu - \epsilon d_v\right) \leq \mathrm{e}^{-2\epsilon^2 d_v} \tag{2}$$

We can now bound the probability that a specific partition does not apply source-side aggregation to the edges it is responsible for that are targeting vertex v.

Theorem 4.3. *The probability that a specific partition P has at most $\mu/2$ edges towards vertex v is bounded by $\mathrm{e}^{-\mu/2K}$.*

Proof. Let $\epsilon = 1/2K$. Then from Eq. (2) we obtain:

$$\mathbb{P}\left(\sum_i X_i \leq \mu/2\right) \leq \mathrm{e}^{-2\epsilon^2 d_v} = \mathrm{e}^{-\mu/2K}.$$

□

We can now compute the expected number of messages that will be transmitted towards vertex v from all partitions K.

Theorem 4.4. *The expected number of messages that are targeting vertex v is at most:*

$$K + \frac{\mu K}{2} \mathrm{e}^{-\mu/2K} \tag{3}$$

Table 1: Public Datasets

Dataset	Vertices	Edges
Common Crawl [1, 3]	3,315,452,339	128,736,952,320
WebUK 2006-2007 [7]	133,633,040	5,507,679,822
Twitter [2]	52,579,682	1,963,263,821

Table 2: Web Graph Properties

Property	Value
URLs	37.73 Billion
Total Outlinks	3.076 Trillion
Graph Storage Footprint	99.2 TB

Proof. There are at most K partitions on which source-side aggregation can be applied, therefore at most K messages will be emitted as a result of such aggregations. On the partitions where source-side aggregation is not applied, at most $\mu/2$ edges will be targeting vertex v. By linearity of expectation, at most $\frac{\mu K}{2} e^{-\mu/2K}$ non-aggregated edges will reach vertex v. □

By Eq. 3 we see that this simple, coordination-free scheme applies an exponentially small dampening factor to d_v. In Section 5, we will see in practice that source-side aggregation almost completely eliminates inlink imbalance even for small values of H.

5 EVALUATION

We explore the performance of PageRank on *Hronos* and compare it with Giraph [12]. Giraph is a production-hardened framework that has been shown to scale to 1 trillion edge graphs [12]. A recent comparison between Pregel+, GPS, Giraph and GraphLab on the WebUK[7] and Twitter[2] datasets for PageRank can be found in [34]. We provide results for publicly available graphs (Table 1) and large proprietary web graphs (Table 2). We collected results on a 3,000-node cluster of the following node configuration: 64GB RAM, 2x Intel Xeon E5-2620, 10Gbps Ethernet.

5.1 Public Datasets

We first provide results for *Hronos* and Giraph on the public datasets (see Table 3). We observe mean speedups of more than 15X for 64 threads, and more than 30X for 5,000 threads. We observe that Giraph appears to favor fewer workers, each of which using multiple threads, rather than single-thread workers. For each experiment, we show the best Workers×Threads combination that we could find via a parameter sweep. Better results could not be obtained for any of the experiments even when we allowed for up to 40,000 threads. For all Giraph experiments, workers were given the full memory of the nodes.

5.2 Web Graphs

We provide results on a web graph of 37.7 billion vertices and 3.076 trillion edges (see Table 2). Loading and pre-processing of the graph partitions consumes approximately 9' for the slowest mapper. Service discovery requires negligible time using a single-node ZooKeeper. We first examine how well balanced graph splits

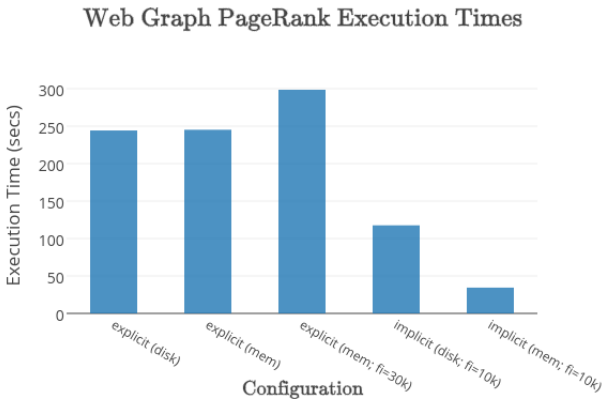

Figure 3: Absolute execution times for a single PageRank iteration. Results shown a web graph of 37.7 billion vertices and 3076 billion edges for explicit pattern (graph on disk), explicit pattern (graph in memory), explicit pattern (graph in memory; H = 30000), implicit pattern (graph on disk; H = 10000), implicit pattern (graph in memory; H = 10000).

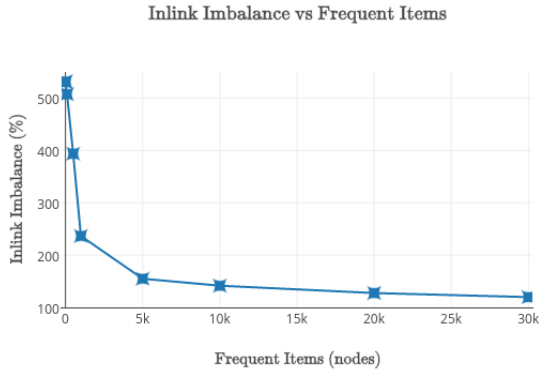

Figure 4: Maximum / mean memory required vs Frequent Items H. Memory imbalance is 19.68% for H = 30000.

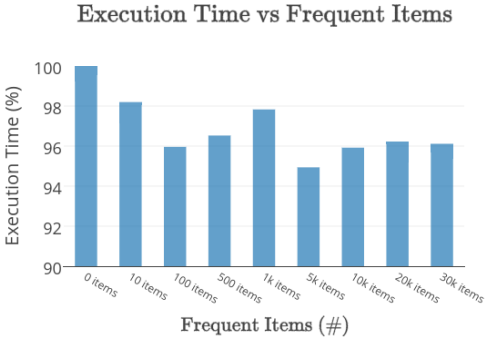

Figure 5: Relative execution times for implicit pattern for Frequent Items H. For H = 30000 execution time is 96.1% compared to the case where H = 0. Overall messages sent are 97% of the total using source-side aggregation with H = 30000.

Table 3: *Hronos* Performance for a single PageRank Iteration on Public Datasets

Dataset	*Hronos*		
	P=64	P=384	P=5000
Common Crawl 2012	128.19"	7.63"	2.45"
WebUK 2006	3.64"	0.53"	< 0.1"
Twitter	1.06"	0.21"	< 0.1"
	Giraph		
	P=64	P=384	P=5000
Common Crawl 2012	-	-	67.51"
WebUK 2006	51.6"	9.54"	9.50"
Twitter	17.57"	6.05"	9.21"
	Hronos Speedup over Giraph		
	P=64	P=384	P=5000
Common Crawl 2012	-	-	**27.55X**
WebUK 2006	**14.17X**	**18.00X**	**95.00X+**
Twitter	**16.58X**	**28.81X**	**92.10X+**

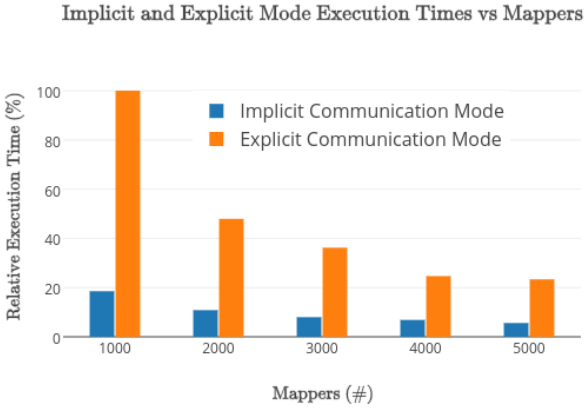

Figure 6: Relative execution times for one PageRank iteration for implicit pattern (H =1k) and explicit pattern (graph in memory) for 1000, 2000, 3000, 4000 and 5000 graph splits.

are. As expected, partitions become more imbalanced as the split count increases. Nevertheless, in the worst case, the largest partition on the 5,000 graph split is only 12.24% larger than average.

In Figure 3 we show absolute execution times for the various communication patterns. In the implicit targeting communication pattern, the system is able to complete one PageRank iteration in 34.4 seconds. We observe a speedup of 7.2X of the implicit pattern over the explicit pattern. The overhead of scanning the graph from disk rather than from memory is negligible in the explicit pattern, as the bottleneck in this case is communication. The opposite is true for the implicit pattern, where scanning the graph from disk is ~30% slower. In addition to our main web graph we collected performance results on the largest web graph we were granted access to, of 96.3 billion vertices and 5.874 trillion edges. For this we used implicit pattern, $H = 10000$, 64-bit payloads, and measured 134-second iterations.

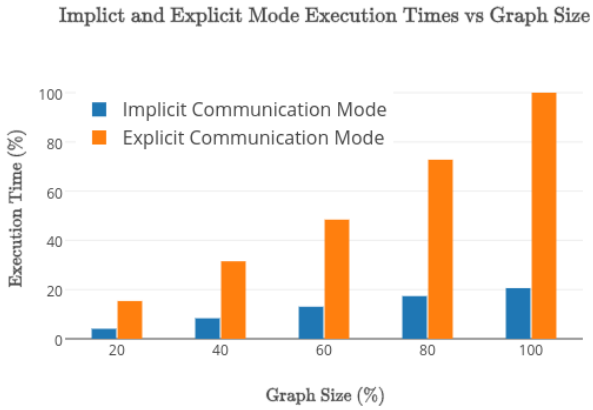

Figure 7: Relative execution times for a single PageRank iteration for implicit pattern (H = 10000) and explicit pattern (graph in memory) for 5000 graph splits and 20%, 40%, 60%, 80% graph sub-sampling.

We then examine the number of inlinks to a partition when source-side aggregation is used for different values of H. First, we observe that inlink imbalance is indeed a significant problem. Specifically, the largest inlink count is more than five times larger than average (Figure 4). Inlink count dictates the size of the in-order vectors for the implicit pattern and therefore has a direct impact on memory requirements. Source-side aggregation, through the detection of high-frequency inlinks, resolves the imbalance even for relatively small numbers of H. For instance, for $H = 30000$, the worst-case imbalance is 19.68%. We also measure the performance of the algorithm for various values of H and observe there is no overhead when using aggregation. In fact, we observe a performance increase in accordance with the reduction in overall messages sent over the network. For instance, when $H = 30000$, the messages sent over the network are 97% of all messages, while execution time improves to 96% compared to the baseline case (see Figure 5).

Finaly, *Hronos* exhibits strong and weak scalability properties: (1) we observe that performance improves almost linearly with the number of graph partitions for both the implicit and explicit pattern (Figure 6;) and (2) we sub-sample the web graph in order to inspect performance as the graph size increases while keeping a fixed split count and observe that the system scales linearly with graph size (Figure 7).

6 CONCLUSION

In this work, we introduced a novel communication paradigm for graph processing frameworks that is optimized for PageRank-like workloads. We evaluated its performance on graphs of up to 96 billion vertices and 5.9 trillion edges and demonstrated more than an order of magnitude improvements over the state-of-the-art.

ACKNOWLEDGMENTS

We would like to thank Dipen Rughwani for collecting experimental results on Giraph and Kostas Tsioutsiouliklis for insightful discussions and proof-reading earlier versions of this work.

REFERENCES

[1] 2016. Common Crawl. http://commoncrawl.org/.
[2] 2016. Twitter User Graph. http://konect.uni-koblenz.de/networks/twitter_mpi.
[3] 2016. Web Data Commons. http://webdatacommons.org/.
[4] 2017. Apache Giraph. http://giraph.apache.org/.
[5] 2020. Facebook Company Info. http://newsroom.fb.com/company-info/.
[6] 2020. How Search organizes information. https://www.google.com/search/howsearchworks/crawling-indexing/.
[7] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.
[8] RobertS. Boyer and J.Strother Moore. 1991. MJRTY-A Fast Majority Vote Algorithm. In *Automated Reasoning*, RobertS. Boyer (Ed.). Automated Reasoning Series, Vol. 1. Springer Netherlands, 105–117.
[9] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 285–296.
[10] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.
[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.
[12] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proceedings of the VLDB Endowment* 8, 12 (2015).
[13] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.
[14] JE Gonzalez, Y Low, H Gu, D Bickson, and C Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *OSDI* (2012).
[15] JE Gonzalez, RS Xin, and A Dave. [n.d.]. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI'14 Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*.
[16] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 7, 12 (Aug. 2014), 1047–1058. https://doi.org/10.14778/2732977.2732980
[17] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association* 58, 301 (1963), 13–30.
[18] Shan Jiang, Yuening Hu, Changsung Kang, Tim Daly Jr, Dawei Yin, Yi Chang, and Chengxiang Zhai. 2016. Learning Query and Document Relevance from a Web-scale Click Graph. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 185–194.
[19] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. 2009. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 229–238.
[20] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 169–182.
[21] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin, et al. 2012. GraphChi: Large-Scale Graph Computation on Just a PC.. In *OSDI*, Vol. 12. 31–46.
[22] AN Langville and CD Meyer. 2011. *Google's PageRank and beyond: The science of search engine rankings*.
[23] Yibei Ling and Wei Sun. 1992. A Supplement to Sampling-based Methods for Query Size Estimation in a Database System. *SIGMOD Record* 21, 4 (Dec. 1992), 12–15.
[24] G Malewicz, MH Austern, and AJC Bik. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*.
[25] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.
[26] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford InfoLab* (1999).
[27] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 410–424.
[28] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.
[29] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management - SSDBM*. ACM Press, 1. https://doi.org/10.1145/2484838.2484843
[30] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 540–546.
[31] Stergios Stergiou, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis. [n.d.]. Distributed Negative Sampling For Word Embeddings. In *AAAI'17 Thirty-First AAAI Conference on Artificial Intelligence*.
[32] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.
[33] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.
[34] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. 1307–1317.
[35] H Zhao and J Canny. 2014. Kylix: A Sparse Allreduce for Commodity Clusters. In *43rd International Conference on Parallel Processing (ICPP)*.