LA3: A Scalable Link- and Locality-Aware Linear Algebra-Based Graph Analytics System

Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud
Carnegie Mellon University in Qatar
{myahmad, okhattab, aimalik, amusleh, mhhamoud}@qatar.cmu.edu

Mucahid Kutlu, Mostafa Shehata, Tamer Elsayed
Qatar University
{mucahidkutlu, mostafa.shehata, telsayed}@qu.edu.qa

ABSTRACT

This paper presents *LA3*, a scalable distributed system for graph analytics. LA3 couples a vertex-based programming model with a highly optimized linear algebra-based engine. It translates any vertex-centric program into an iteratively executed sparse matrix-vector multiplication (SpMV). To reduce communication and enhance scalability, the adjacency matrix representing an input graph is partitioned into locality-aware 2D tiles distributed across multiple processes. Alongside, three major optimizations are incorporated to preclude redundant computations and minimize communication. First, the link-based structure of the input graph is exploited to classify vertices into different types. Afterwards, vertices of special types are factored out of the main loop of the graph application to avoid superfluous computations. We refer to this novel optimization as *computation filtering*. Second, a *communication filtering* mechanism is involved to optimize for the high sparsity of the input matrix due to power-law distributions, common in real-world graphs. This optimization ensures that each process receives only the messages that pertain to non-zero entries in its tiles, substantially reducing communication traffic since most tiles are highly sparse. Lastly, a *pseudo-asynchronous computation and communication* optimization is proposed, whereby processes progress and communicate asynchronously, consume messages as soon as they become available, and block otherwise. We implemented and extensively tested LA3 on private and public clouds. Results show that LA3 outperforms six related state-of-the-art and popular distributed graph analytics systems by an average of 10×.

PVLDB Reference Format:
Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, Tamer Elsayed. LA3: A Scalable Link- and Locality-Aware Linear Algebra-Based Graph Analytics System. *PVLDB*, 11(8): 920-933, 2018.
DOI: https://doi.org/10.14778/3204028.3204035

Proceedings of the VLDB Endowment, Vol. 11, No. 8
Copyright 2018 VLDB Endowment 2150-8097/18/04... $ 10.00.
DOI: https://doi.org/10.14778/3204028.3204035

1. INTRODUCTION

Modern-day networks, such as the Web, online social networks, and wireless sensor networks, continue to grow and generate a wealth of valuable information. The link-based structures of these networks are usually modeled as graphs, enabling domain experts to mine them using a variety of graph analytics algorithms, including PageRank, Connected Components, and Shortest Paths, to name a few. To effectively implement and execute these algorithms, modern graph analytics platforms like Giraph [1], PowerGraph [19], Gemini [42], and GraphPad [11], among others, are typically utilized. These platforms can be classified into three major categories, namely, *vertex-centric*, *linear-algebra-centric*, and *hybrid*, depending on the adopted programming abstraction and the underlying execution model.

Most graph analytics platforms (e.g., Giraph and PowerGraph) lie under the vertex-centric category, wherein they provide a vertex-based programming abstraction on top of an iterative graph-based processing engine. In particular, the application programmer writes sequential code that is executed concurrently over all vertices, and the resulting interactions between connected vertices are translated into messages. Iterative execution continues until the states of the vertices converge or a pre-defined number of iterations is reached. This vertex-centric category has become very popular due to its simple, yet flexible programming abstraction and highly parallelizable execution model [15, 3].

Aside from vertex-centric systems, much work has been done by the high-performance scientific computing (HPC) community to develop linear-algebra-centric graph analytics platforms. In these platforms, a formalized linear algebra-based programming abstraction is provided over of a linear algebra-based processing engine [22, 12]. The application programmer writes iterative code assuming an underlying adjacency matrix, which represents the input graph. The code calls specialized linear algebra primitives that correspond to well-defined graph algorithms. For instance, edge traversals from a set of vertices can be translated into sparse matrix-vector multiplication (SpMV) calls on the graph adjacency matrix (or its transpose) [38]. In fact, most graph analytics algorithms can be implemented via overloading a generalized version of SpMV [23]. For example, the Single-Source Shortest Paths algorithm can be implemented via replacing the *multiply* and *add* operations of SpMV with *add* and *min* operations, respectively [23, 38, 22].

Recently, a third breed of platforms has emerged with systems like GraphMat [38] and GraphPad [11], where a vertex-based abstraction is layered on top of a linear algebra-based engine. Specifically, the application is still written using a vertex-based programming interface, but gets automatically translated into iterative calls to linear algebra routines. Clearly, these *hybrid* platforms aim at reaping the benefits of both the vertex- and linear-algebra-centric paradigms, namely, the wide familiarity and flexibility of the vertex-based programming abstraction, alongside the scalable parallelism and low-level optimizations of the linear algebra-based execution model. In this paper, we present *LA3*, a scalable distributed system for graph analytics that falls under this new category of hybrid platforms.

Akin to some existing hybrid systems, LA3 partitions the adjacency matrix, which represents the input graph, into equal-sized *tiles* via a two-dimension (2D) partitioning algorithm. However, in contrast to these systems, LA3 employs a unique 2D-Staggered tile placement strategy, which exploits *locality* upon distributing tiles across processes and thereby reduces communication and expedites computation. Furthermore, LA3 adopts three other major optimizations: *computation filtering*, *communication filtering*, and *pseudo-asynchronous computation and communication.*

Computation filtering capitalizes on the observation that graph vertices are of four different types: (1) *source* vertices with only outgoing edges, (2) *sink* vertices with only incoming edges, (3) *isolated* vertices with no incoming or outgoing edges, and (4) *regular* vertices with incoming and outgoing edges. We show that non-regular (i.e., source, sink, and isolated) vertices can generally be safely factored out from the main execution loop of a graph program. Consequently, we suggest a vertex classification mechanism that classifies vertices into these four types. Afterwards, computation filtering factors out non-regular vertices from the main loop, saving thereby many redundant computations. We show that an adjacency matrix representation of a graph greatly facilitates both the vertex classification pre-processing step and the computation filtering optimization.

Besides, communication filtering optimizes for the high degree of sparsity in the input matrix due to the power-law distributions observed in real-world graphs [19]. In particular, applying a 2D partitioning algorithm over the input matrix results in most tiles becoming hyper-sparse [23]. As a way to leverage this hyper-sparsity, LA3 employs a point-to-point communication layer with a special communication filter per process. This filtering ensures that each process receives only messages corresponding to the non-zero entries (edges) in its tiles. Since most tiles are extremely sparse, our communication filtering optimization substantially reduces inter-process communication. Moreover, applying it in conjunction with computation filtering, which reduces the number of non-zero entries in tiles by factoring out non-regular vertices, further boosts its advantage and differentiates LA3 from other linear-algebra-centric and hybrid systems.

Finally, pseudo-asynchronous computation and communication interleaves computation with communication, both within each iteration and across successive iterations. Specifically, during normal execution each process still receives and sends messages asynchronously. In other words, a process consumes and processes received messages in any order, assuming computations can be pursued incrementally, which is naturally afforded by SpMV operations. Additionally, it dispatches messages as soon as they are produced. To this end, a process only blocks when it does not have more work to do, yet still expects to receive more messages. However, if a process is fully done with its work under the current iteration, it can immediately start with the subsequent one, making our computational model effectively asynchronous. Unlike LA3, other linear-algebra-centric and hybrid systems employ asynchrony for communication but not computation.

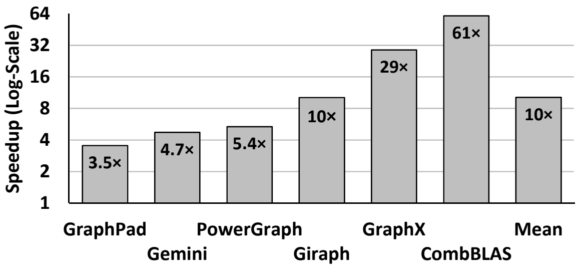


Figure 1: LA3's speedup versus other systems averaged over various standard applications and datasets. Mean speedup is 10× over all systems.

In short, this paper makes the following contributions:

- We present LA3, a scalable distributed system for graph analytics that provides a familiar vertex-based programming interface on top of a highly optimized linear algebra-based execution engine. To encourage reproducibility and extensibility, we make LA3 open source[1].
- We demonstrate the value of *computation filtering* for factoring out redundant computations and communication (when augmented with *communication filtering*) for iterative graph processing.
- We promote *pseudo-asynchronous computation and communication*, in which processes communicate asynchronously, consume messages as soon as they become available, and only block otherwise.
- We suggest a locality-aware 2D-Staggered tile placement strategy, which enables this pseudo-asynchronous optimization, both within each iteration and across successive iterations.
- We provide a comprehensive performance evaluation of LA3, comparing it against the state-of-the-art graph analytics systems. Results show that LA3 offers significant performance and scalability advantages over those systems (see Figure 1).

The rest of the paper is organized as follows. We provide a background on graph analytics in Section 2. Details of LA3 are presented in Section 3. In Section 4, we discuss our experimentation results. Lastly, related work and conclusions are given in Sections 5 and 6, respectively.

2. BACKGROUND

Graph analytics systems with a graph-based processing engine typically represent (directed) graphs using either an *adjacency list* or an *edge list*. The adjacency list maintains a list of entries, each with a unique vertex alongside its set of neighboring vertices (via outgoing edges). In contrast, the edge list maintains a list of records, each representing an edge with its two end vertices. On the other hand, graph

[1] https://github.com/cmuq-ccl/LA3

analytics systems with a linear algebra-based processing engine represent (directed) graphs using an *adjacency matrix*, wherein each non-zero entry (i, j) represents an (outgoing) edge between vertices i and j. Irrespective of the format, careful consideration is needed to design data structures for *big graphs* (i.e., graphs with billions of edges and vertices), which can scale effectively in terms of capacity and access requirements. For instance, a highly sparse adjacency matrix is usually stored compactly in the popular *Compressed Sparse Row (CSR)* data structure (or a variant of it), which trades access time for reduced memory footprint [23].

Graph analytics systems typically split big graphs into multiple partitions for exploiting higher parallelism and/or aggregating storage mediums (e.g., memories and/or disks across machines). Two classical partitioning strategies are commonly used to partition big graphs represented as either adjacency or edge lists. First, *edge-cut* partitioning is utilized to split an adjacency list, resulting in pairs of connected vertices potentially mapped to different partitions. In contrast, *vertex-cut* partitioning splits an edge list such that a vertex with multiple incident edges may appear at multiple partitions. In principle, when processing a big graph in a distributed setting, minimizing the *cut size* (or the number of edges/vertices across partitions) serves in reducing communication traffic, which is vital for scalability.

Likewise, an adjacency matrix may be partitioned along one (1D) or two (2D) dimensions. Clearly, 1D and 2D partitioning are conceptually analogous to edge-cut and vertex-cut partitioning, respectively. Although 2D partitioning necessitates additional metadata for communication and coordination purposes, pursuing SpMV on 2D slices (or *tiles*) requires asymptotically less communication traffic than on 1D slices (or *records* or *partitions*), especially when the number of processes/machines is scaled out [23] (more on this in Section 3.2). As such, most modern linear-algebra-centric systems optimize SpMV assuming 2D partitioning [12, 11].

In the context of graph analytics, SpMV translates into a single pass (or iteration) over an input graph, such that each vertex *gathers* the messages produced by its incoming neighbors and *applies* to them a user-defined function. Afterwards, the vertex may use the result of the function to update its private state and generate a new message to be *scattered* along its outgoing edges, yet in the subsequent iteration. More formally, SpMV multiplies the transposed adjacency matrix $\mathbf{A}^{\mathrm{T}}$ of an input graph with an input message vector $\mathbf{x}$, given a pair of additive and multiplicative *semiring* operators $(\oplus.\otimes)$ [23], and outputs a result vector $\mathbf{y} = \mathbf{A}^{\mathrm{T}} \oplus.\otimes\ \mathbf{x}$. For a given vertex, each incoming message is processed by the multiplicative operator $(\otimes)$, and the result is aggregated by the additive operator $(\oplus)$. Accordingly, various iterative graph applications can be implemented using SpMV with an appropriately defined semiring.

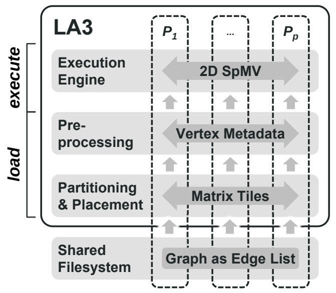
3. LA3 SYSTEM

This section presents in detail the LA3 system. First, we introduce our simple, yet flexible vertex-centric abstraction (Section 3.1). Second, we describe how LA3 loads and partitions an input graph across multiple processes (Section 3.2). Third, we discuss LA3's pre-processing phase, which collects information about the link structure of the input graph, needed for various optimizations within LA3's engine (Section 3.3). Finally, we elaborate on LA3's execution engine (Section 3.4). Figure 2 shows a high-level overview of LA3.

```
class V : Vertex<E,M,A> {
  bool init(){...}
  Acc  gather(E, M){...}
  void combine(A, &A){...}
  bool apply(A){...}
  Msg  scatter(){...}
};

Graph G("input_file");
G.load();
VProgram<V> vp(G, n);
vp.execute();
```

(a)



(b)

Figure 2: (a) A vertex-centric program V. (b) An input graph G is loaded from a shared filesystem, partitioned, distributed, and pre-processed over processes P_1 ... P_p, which then all execute V.

```
using Edge = Empty, Msg = double, Acc = double;

class VPR : Vertex<Edge, Msg, Acc> {

  const bool STATIONARY  = true;  // Default = false
  const bool OPTIMIZABLE = true;  // Default = true
  const double tolerance = 1e-5, alpha = 0.15;

  int degree; double rank;  // State

  bool init(VD d) { degree = d.value; rank = alpha; }

  Msg  scatter() { return rank / degree; }

  Acc  gather(Edge e, Msg m) { return m; }
  void combine(Acc g, Acc& a) { a += g; }

  bool apply(Acc a) {
    double r = rank;
    rank = alpha + (1 - alpha) * a;
    return fabs(rank - r) > tolerance; }
};
```

Figure 3: PageRank implemented in C++ using LA3's vertex-centric programming model. Each vertex is initialized with its out-degree (calculated by a separate vertex program, `VD`).

3.1 Programming & Computational Models

Akin to some closely related vertex-centric systems [38, 11], LA3 provides a programming model composed of five functions, namely, *init*, *gather*, *combine*, *apply*, and *scatter* (see Figure 2(a)). As shown in Figure 3, a user can define a *vertex program* via implementing these functions[2]. To execute this vertex program, the LA3 engine begins processing each *active* vertex in an iterative fashion, until the states of all vertices converge or a specified number of iterations is reached (see Algorithm 1). More precisely, each vertex is computed as follows:

Initialize. At first, each vertex state is initialized by *init*, which returns *true* or *false* to indicate whether the vertex should be activated (Algorithm 1, lines 5-7). Vertex activity is mainly relevant to *non-stationary* applications (e.g., Breadth First Search (BFS)), where not all vertices are necessarily activated at the outset and not all of them remain active over all iterations [24]. This contrasts with *stationary* applications (e.g., PageRank), wherein all vertices stay active over all iterations until completion. In LA3, a user can declare whether an application is stationary or non-stationary via a flag as illustrated in Figure 3. We discuss the performance implications of this in Section 3.4.

[2] The code in Figure 3 is simplified for clarity and brevity.

Algorithm 1 – Vertex-Centric Computational Model

```
 1: Input: Vertex program $V$
 2: Input: Partitioned graph $G$
 3: Input: Number of iterations $n$
 4:
 5: for each $v$ in $G$ do                          ▷ Initialize phase
 6:   $activate \leftarrow v$.init()
 7:   if $activate$ or $V$.isStationary() then $V$.activate($v$)
 8:
 9: for $i \leftarrow 1 \dots n$ do                  ▷ Iterative execution
10:   $msg[\,] \leftarrow \emptyset$;   $acc[\,] \leftarrow \emptyset$        ▷ Messages & accumulators
11:
12:   for each $v$ in $G$ do                        ▷ Scatter phase
13:     if $V$.isActive($v$) then $msg[v] \leftarrow v$.scatter()
14:
15:   for each $v$ in $G$ do                 ▷ Gather & combine phase
16:     for each $e_{in}$ in $v$ do               ▷ Process each in-edge
17:       $m \leftarrow msg[src(e_{in})]$          ▷ Check scattered message
18:       if $m$ is not $\emptyset$ then
19:         $g \leftarrow v$.gather($e_{in}$, $m$)         ▷ Process message
20:         $v$.combine($g$, &$acc[v]$)           ▷ Accumulate result
21:
22:   $converged \leftarrow true$
23:   if not $V$.isStationary() then $V$.deactivateAll()
24:
25:   for each $v$ in $G$ do                          ▷ Apply phase
26:     if $acc[v]$ is not $\emptyset$ then
27:       $updated \leftarrow v$.apply($acc[v]$)
28:       if $updated$ then
29:         $converged \leftarrow false$
30:         if not $V$.isStationary() then $V$.activate($v$)
31:   if $converged$ then break                            ▷ Done
```

Scatter. After a vertex is activated, *scatter* generates a new message based on its initial/updated state, which is sent along its outgoing edge(s) in the current iteration (Algorithm 1, lines 12-13).

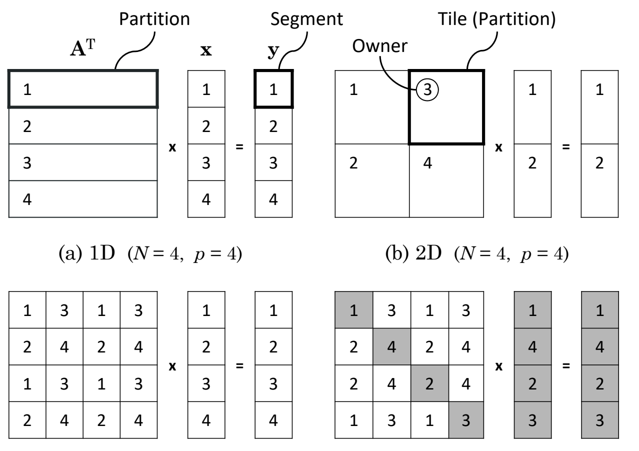
Gather & Combine. Subsequently, each vertex that receives messages along its incoming edges processes them via *gather*, and *combines* the outcomes into an accumulator (Algorithm 1, lines 15-20)[3]. In addition, *gather* may access the vertex state, which has certain performance implications as discussed in Section 3.4.

Apply. The accumulated result is then utilized by the vertex to update its state using *apply*, which returns *true* if the state changes and *false* otherwise. If false is returned, the vertex is deactivated (Algorithm 1, lines 25-30). If *all* the vertices are deactivated, the program stops and is said to converge (Algorithm 1, line 31).

This programming model enables users to implement a wide range of graph applications. However, we note that certain implementations are more amenable to benefit from some optimizations in LA3. In particular, LA3 is specially optimized for implementations where: (1) the new state of each vertex is computed *only* as a function of its accumulator value and/or current state, and (2) its outgoing messages are generated *only* as a function of its new state. We refer to such implementations as *LA3-optimizable*.

Based on the above definition, most graph applications can be implemented in an LA3-optimizable manner. For instance, BFS computes the state of a vertex (i.e., the minimum number of hops needed to reach it from some root

[3] The *gather* and *combine* functions translate into the multiplicative ($\otimes$) and additive ($\oplus$) operations of an SpMV semiring, respectively (see Section 2).



(a) 1D ($N = 4,\ p = 4$) (b) 2D ($N = 4,\ p = 4$)

(c) 2D–Cyclic ($N = 16,\ p = 4$) (d) 2D–Staggered ($N = 16,\ p = 4$)

Figure 4: Partitioning and placement strategies for distributed SpMV. Matrix partitions/tiles and vector segments numbered by owner processes. N and p are numbers of partitions (or tiles) and processes, respectively. Shaded segments in (d) have same owner as corresponding shaded tiles on diagonal.

vertex) upon receiving its first message from any of its neighboring vertices, which simply indicates that it has been reached. Consequently, BFS can be implemented via setting the state of the vertex in the *apply* function to 1 plus the first sender's state (or message). Clearly, this implementation is LA3-optimizable since it computes the state of the vertex based on its accumulator's content. Alternatively, BFS can be implemented by setting the vertex's state to the current iteration (or hop) number. Evidently, this is not an LA3-optimizable implementation since it is not a function of the vertex's accumulator value and/or current state.

A vertex program that is not LA3-optimizable results in LA3 disabling its *computation filtering* optimization (detailed in Section 3.4.1). This is because computation filtering factors non-regular vertices (i.e., source, sink, and isolated vertices) out of the main computation loop, hence, executing them only once rather than iteratively. As such, a program that uses iteration-specific information (e.g., the iteration number) to compute a vertex state will be incompatible with computation filtering because non-regular vertices will have no access to this information. To this end, users can declare whether their vertex programs are LA3-optimizable via a flag (see Figure 3). Afterwards, LA3 enables or disables computation filtering accordingly.

Finally, as depicted in Figure 2, LA3 assumes an edge list format for input graphs. Specifically, it reads the edge list as a directed graph with no self-loops or duplicate edges. In addition, it loads (by default) the graph into a transposed adjacency matrix since transposing the i^{th} matrix row (i.e., the out-edges of vertex v_i) into the i^{th} column aligns it with the i^{th} entry of the message vector (i.e., the outgoing message produced by v_i) during SpMV computation. Lastly, LA3 supports undirected graphs as well, which requires the edge list to be in a bidirectional form (i.e., each edge must have a corresponding reverse edge).

3.2 Partitioning & Locality-Aware Placement

The transposed adjacency matrix $\mathbf{A}^{\mathrm{T}}$ of an input graph may be partitioned along one or two dimensions as depicted in

Figure 4. Each *partition* is assigned to a process denoted as its *owner*. Similarly, the vectors **x** and **y** may be split into *segments*, each of which is also assigned to an owner as shown in Figure 4(a). Since **x** is normally derived from **y** after each iteration, corresponding segments of **x** and **y** should ideally share the same owner to ensure locality.

With 1D partitioning (see Figure 4(a)), splitting the matrix horizontally into N partitions is conceptually similar to a naïve edge-cut partitioning of a graph. Moreover, a naïve placement strategy might assign each partition to a different process in a round-robin or random fashion. If the number of partitions is a multiple of the number of processes p (e.g., $N = p$ in Figure 4(a)), each process can be allocated an equal number of partitions. However, this may not necessarily balance the computation, memory, and/or communication load(s) of graph processing evenly across all processes. This is especially true with real-world graphs, which usually exhibit power-law distributions, resulting in a small number of partitions containing the bulk of the edges.

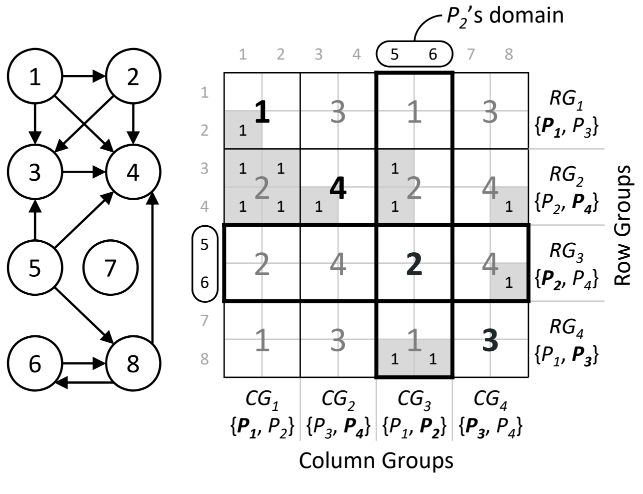
Under 1D partitioning, the SpMV product may be calculated by multiplying the i^{th} matrix partition, where $i = 1..N$, with the entire **x** vector to produce the i^{th} **y**-segment[4]. Consequently, each process must gather the **x**-segments from every other process. Hence, the total number of messages communicated is $O(p \times (p-1)) = O(p^2)$.

With 2D partitioning (see Figure 4(b)), splitting the matrix across two dimensions into N partitions (or *tiles*) is similar to applying vertex-cut partitioning on a graph. This is because the set of edges adjacent to any given vertex may get distributed across multiple partitions, necessitating that the vertex be accessible at each of these partitions. Note that a partitioning that creates equal-sized square tiles (as in our example in Figure 4(b)) implies that N is necessarily a square number. Nonetheless, as with 1D partitioning, a skew in the input graph may cause a small number of tiles to encompass most of the edges.

Under 2D partitioning, the SpMV product may be calculated by multiplying each tile in the j^{th} column and i^{th} row, where $i, j = 1..N$, with the j^{th} **x**-segment, and accumulating the resultant partial outcome into the i^{th} **y**-segment[5]. Thus, each **x**-segment is needed in *at most* $\sqrt{p}$ other processes, while calculating each **y**-segment requires the per-tile partial result from *at most* $\sqrt{p}$ other processes. Therefore, the total number of messages communicated is $O(p\times 2\sqrt{p}) = O(p\sqrt{p})$, which is a factor of $\sqrt{p}$ less than 1D partitioning. As such, 2D scales better than 1D with increasing values of p.

A 2D partitioning of a matrix is commonly accompanied by a 2D-Cyclic tile placement, where the tiles are assigned to a set of processes cyclically (see Figure 4(c)) [23]. In the simplest case, there are p tiles in each dimension, hence, $N = p^2$. While the total number of messages communicated for calculating SpMV under 2D-Cyclic is the same as 2D, Anderson et *al.* [11] showed that 2D-Cyclic outperforms 2D due to increased parallelism, which results from more refined distributions of tiles and segments. Clearly, this serves also in mitigating the load imbalance caused by skewed graphs.

LA3 employs 2D partitioning since it provides greater scalability as opposed to 1D. In particular, LA3 adopts a distributed 2D partitioner (see Figure 2(b)), which reads an input graph from a shared filesystem (e.g., NFS) and partitions it into a 2D grid of tiles. Optionally, the user may enable *vertex hashing*, which shuffles vertices around in the matrix to alleviate the effects of skew in real-world graphs[6].



(a) A graph, G **(b) $G \rightarrow \mathbf{A}^{\mathrm{T}}$ & $\boldsymbol{\mathcal{G}}$**

Figure 5: Transposed adjacency matrix $\mathbf{A}^{\mathrm{T}}$ of a directed graph G with unit edge weights (shaded). Overlaid on top, the 2D-Staggered grid $\boldsymbol{\mathcal{G}}$ (from Figure 4(d)). Group leaders are in bold.

In addition, LA3 suggests a locality-aware *2D-Staggered* tile placement in association with its 2D partitioning. Specifically, 2D-Staggered extends 2D-Cyclic placement by ensuring that each tile on the diagonal is assigned to a different process. Furthermore, each vector segment is assigned to the same owner as the corresponding tile on the diagonal (see Figure 4(d)). Thus, every process can start immediately a new iteration of SpMV computation on at least one of its tiles (i.e., the one on the diagonal) against its local **x**-segment and accumulate directly into its local **y**-segment, while it is receiving the remaining needed **x**-segments asynchronously. In contrast, a naïve 2D-Cyclic placement does not enable locality and asynchrony to be exploited in this way. We discuss this further in Section 3.4.

To materialize the above strategy, LA3 generates firstly a standard 2D-Cyclic $p \times p$ tile placement grid $\boldsymbol{\mathcal{G}}$. Afterwards, it converts $\boldsymbol{\mathcal{G}}$ into a 2D-Staggered grid by reordering the rows of $\boldsymbol{\mathcal{G}}$ such that each tile on the diagonal is owned by a different process. For example, the 2D-Staggered grid in Figure 4(d) is obtained via swapping the third and fourth rows of the 2D-Cyclic grid in Figure 4(c).

3.3 Pre-processing

Real-life graphs demonstrate a number of notable characteristics, which lend themselves to significant opportunities for optimizing iterative execution. To begin with, vertex popularities are highly skewed in various Web and social graphs [19, 20]. In addition, we observe that 42% of users in the Twitter dataset have no followers at all [20], so their popularities (as measured by PageRank) are unaffected by other users' popularities. Alongside, there are no directed paths that connect these users with other users. Hence, any computation towards calculating their popularities (or

[4] More precisely, this is referred to as SpMV *inner* product.
[5] More precisely, this is referred to as SpMV *outer* product.

[6] We observe that vertex hashing (implemented via a reversible hash function [11]) generally results in better performance on most graphs. However, a notable exception is the UK-2005 graph, as noted also by Zhu et al. [42] (see Section 4).

Table 1: Classification statistics for various datasets.

Dataset	Vertices	Reg.	Source	Sink	Iso.
LiveJournal [5]	4,847,571	81%	8%	11%	0%
Wikipedia [10]	3,566,907	68%	29%	1%	1%
Weibo [9]	58,655,849	1%	99%	0%	0%
UK-2005 [8]	39,459,925	88%	0%	12%	0%
Twitter [7]	61,578,415	55%	10%	3%	32%
CC 2012 Host [2]	101,000,000	54%	16%	18%	12%
RMAT 23 [13]	8,388,608	28%	4%	22%	46%
RMAT 26 [13]	67,108,864	24%	4%	20%	52%
RMAT 29 [13]	536,870,912	21%	4%	18%	57%

shortest paths to them, as another example) is essentially unnecessary beyond initialization. We also observe that 35% of Twitter users follow no one, so they neither affect other users' popularities nor participate in any directed paths to them. Subsequently, as long as their states depend solely on the states of their followers, they can be safely computed after all their followers' states are finalized.

To formalize the above observations, we classify vertices into four mutually exclusive types (in the context of directed graphs): *regular*, *source*, *sink*, and *isolated*, as defined in Section 1. Figure 5(a) shows a graph where vertices 1 and 5 are sources, 4 is sink, 7 is isolated, and the rest are regular.

Besides, Table 1 lists percentage-wise statistics for a variety of real-world and synthetic graphs. The proportion of non-regular vertices ranges from 12% (UK-2005) to 99% (Weibo). As pointed out in Section 3.1, these vertices can be factored out of iterative execution, yielding significant savings in computation and communication. To allow for these savings, we propose a two-step pre-processing mechanism, which firstly classifies graph vertices and subsequently packs edges into a classification-aware compressed data structure. We next elaborate on these two pre-processing steps.

Step 1: Vertex Classification. Given an adjacency matrix partitioned along two dimensions, the set of edges adjacent to a given vertex may be split across multiple tiles, which can reside on different processes. As such, determining whether this vertex has any incoming or outgoing edges may require these processes to communicate. For this sake, we introduce the concept of *row-* and *column-groups* over a 2D-Staggered tile placement grid, say $\boldsymbol{\mathcal{G}}$. Specifically, we define the i^{th} row-group, RG_i, as the set of processes that own all the tiles on the i^{th} row of $\boldsymbol{\mathcal{G}}$. To illustrate this, Figure 5(b) shows four row-groups (i.e., RG_1 to RG_4), whereby RG_3 (for example) encompasses processes P_2 and P_4 since they own all the tiles on row 3 of $\boldsymbol{\mathcal{G}}$. Besides row-groups, we define the i^{th} column-group, CG_i, as the set of processes that own all the tiles on the j^{th} column of $\boldsymbol{\mathcal{G}}$. For instance, CG_3 in Figure 5(b) consists of processes P_1 and P_2, which own all the tiles on column 3 of $\boldsymbol{\mathcal{G}}$.

Furthermore, a process in each row-group, RG_i, and a process in each column-group, CG_i, can be designated as the *leaders* of the groups, which are responsible for classifying all the vertices in RG_i and CG_i, respectively. To streamline the process, we select the leader of each group (whether row- or column-group) to be the process that owns the tile on the diagonal of the given 2D-Staggered tile placement grid, $\boldsymbol{\mathcal{G}}$. As a result, RG_i and CG_i end up having the same group leader, say GL_i. To exemplify, P_2 at RG_3 in Figure 5(b) owns the tile on the diagonal of $\boldsymbol{\mathcal{G}}$, hence, becoming the leader of RG_3. Consequently, P_2 will be responsible for classifying vertices 5 and 6, which are referred to as the *domain* of P_2. Similarly, P_2 at CG_3 in Figure 5(b) owns the tile on the diagonal of $\boldsymbol{\mathcal{G}}$, thus becoming the leader of CG_3. Therefore, P_2 is rendered the sole group leader, GL_3, of both RG_3 and CG_3. More interestingly, its domain does not expand since it remains responsible for classifying only vertices 5 and 6, which are common between RG_3 and CG_3.

To fulfill the vertex classification procedure, GL_i accumulates from each process in its column-group the set of all non-empty matrix columns (i.e., vertices with at least one outgoing edge). Let us denote this set as V_{out}. In addition, it accumulates from each process in its row-group the set of all non-empty matrix rows (i.e., vertices with at least one incoming edge). We refer to this set as V_{in}. Clearly, the intersection of the resulting sets, V_{out} and V_{in}, is the set of regular vertices in GL_i's domain. Moreover, the set-differences $(V_{out} - V_{in})$, $(V_{in} - V_{out})$, and $(V_{all} - (V_{out} \cup V_{in}))$ are the sets of source, sink, and isolated vertices, respectively, with V_{all} being the set of all the vertices in GL_i's domain. When GL_i finishes classifying its domain, it broadcasts the outcome (which we call GL_i's *metadata*) to all the processes in its row- and column-groups.

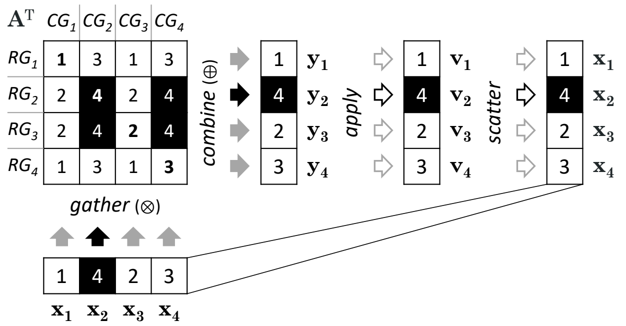
Step 2: Edge Processing. The metadata generated by our vertex classification procedure enables LA3 to exploit interesting optimizations, namely, *computation and communication filtering*. We detail these optimizations in Section 3.4. However, to lay the groundwork for this pre-processing step, we note that these optimizations leverage the types of vertices to *selectively* compute them and communicate their messages. To expedite this selective process, LA3 splits each tile into several *sub-tiles*. Each sub-tile contains a mutually exclusive subset of the non-zero elements (or edges) in its tile. These edges serve in dictating (or more precisely, *filtering*) the messages that the sub-tile's SpMV operation will receive and compute.

For instance, assume a tile's SpMV operation needs to selectively process only the messages that originated from regular vertices. Clearly, this selective (or filtered) processing will involve only the subset of edges in the tile whose sources are regular vertices. As such, we can extract this subset of edges and compactly store them at a sub-tile. Afterwards, only the messages that correspond to these edges will be received and processed at the sub-tile. Crucially, this is more efficient than performing the filtered processing at the original tile since only the needed edges will be touched, increasing thereby cache/memory locality. Alongside this, edges are also filtered based on their destination vertices, shrinking further the subset of edges at a sub-tile and making its computation even more efficient.

To this end, edge processing constructs for each tile $\mathbf{T}_{i,j}$ on the i^{th} row and j^{th} column of a 2D-Staggered grid the following four sub-tiles: (1) $\mathbf{T}_{i,j}^{src,reg}$, (2) $\mathbf{T}_{i,j}^{reg,reg}$, (3) $\mathbf{T}_{i,j}^{src,snk}$, and (4) $\mathbf{T}_{i,j}^{reg,snk}$. These four sub-tiles serve in filtering messages from: (1) source vertices to regular vertices, (2) regular vertices to regular vertices, (3) source vertices to sink vertices, and (4) regular vertices to sink vertices, respectively. Finally, each sub-tile is stored in a Doubly Compressed Sparse Column (DCSC) format [23].

3.4 Execution Engine

The core of LA3's execution engine is a distributed SpMV routine, which leverages our various partitioning, placement, and storage mechanisms discussed in the previous sections, alongside the optimizations detailed in this section. Figure 6 demonstrates a high-level overview of our engine's design.



$\mathbf{A}^T$
CG_1 CG_2 CG_3 CG_4
RG_1 1 3 1 3
RG_2 2 4 2 4
RG_3 2 4 2 4
RG_4 1 3 1 3
combine (⊕)
1 $\mathbf{y}_1$ 4 $\mathbf{y}_2$ 2 $\mathbf{y}_3$ 3 $\mathbf{y}_4$
apply
1 $\mathbf{v}_1$ 4 $\mathbf{v}_2$ 2 $\mathbf{v}_3$ 3 $\mathbf{v}_4$
scatter
1 $\mathbf{x}_1$ 4 $\mathbf{x}_2$ 2 $\mathbf{x}_3$ 3 $\mathbf{x}_4$
gather (⊗)
1 4 2 3
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$

Figure 6: 2D-Staggered SpMV at GL_2, i.e., P_4 (black). Shaded arrows denote point-to-point communication of segments along row/column-groups.

Specifically, the transposed adjacency matrix $\mathbf{A}^T$ of an input graph is initially partitioned into $p \times p$ tiles using our 2D distributed partitioner. Subsequently, the tiles are assigned to p processes based on our 2D-Staggered placement grid $\boldsymbol{\mathcal{G}}$ (see Section 3.2). As Figure 6 shows, a *state vector* $\mathbf{v}$, *message vector* $\mathbf{x}$, and *accumulator vector* $\mathbf{y}$, are each decomposed into p segments, such that the k^{th} segment of each vector is aligned with the k^{th} row of $\boldsymbol{\mathcal{G}}$ and owned by the k^{th} group leader, GL_k.

Given a user-defined vertex program (see Section 3.1), the SpMV routine is overloaded with *gather* (⊗) and *combine* (⊕) (see Section 2). Our engine iteratively executes SpMV, re-computing the result $\mathbf{y} = \mathbf{A}^T \oplus.\otimes\, \mathbf{x}$ in every iteration. Prior to triggering SpMV, $\mathbf{x}$ is computed from $\mathbf{v}$ via *scatter*. After executing SpMV, $\mathbf{v}$ is computed from $\mathbf{y}$ via *apply*.

More precisely, LA3's distributed SpMV divides the above computation into $p \times p$ partial tile computations as follows: each tile $\mathbf{T}_{i,j}$ is computed individually by its owner (say, $TO_{i,j}$) to produce the partial result $\widehat{\mathbf{y}}_i \oplus= \mathbf{T}_{i,j} \otimes \mathbf{x}_j$, which is subsequently accumulated into the final result $\mathbf{y}_i \oplus= \widehat{\mathbf{y}}_i$ at GL_i. To this end, our distributed SpMV requires that each group leader GL_k multicasts $\mathbf{x}_k$ along the k^{th} column-group at the start of each iteration. Moreover, it necessitates that each GL_k finalizes $\mathbf{y}_k$ by combining all partial results $\widehat{\mathbf{y}}_k$ along the k^{th} row-group. Lastly, once $\mathbf{y}_k$ is finalized, GL_k applies it to $\mathbf{v}_k$. To exemplify, computing tile $T_{1,2}$ at P_3 in Figure 6 requires receiving $\mathbf{x}_2$ from GL_2 and sending the partial result to GL_1, which accumulates it into $\mathbf{y}_1$. Afterwards, GL_1 applies the finalized $\mathbf{y}_1$ to $\mathbf{v}_1$.

Additionally, if the vertex program requires an access to the vertex state within *gather* (see Section 3.1), computing $\widehat{\mathbf{y}}_i \oplus= \mathbf{T}_{i,j} \otimes \mathbf{x}_j$ at $TO_{i,j}$ requires $\mathbf{v}_i$. For such programs (e.g., Triangle Counting), each $\mathbf{v}_i$ must be replicated (multicast) by its owner, GL_i, across the i^{th} row-group at the start of each iteration. Since this incurs additional communication and synchronization costs (especially if the vertex state is large), and since most applications do not need this feature, it is disabled by default. Nevertheless, it may be enabled via a flag or automatically when *gather* accesses the vertex state[7]. We now discuss the three main optimizations that LA3 incorporates within its distributed SpMV.

3.4.1 Computation Filtering

As pointed out in Section 3.3, the LA3 execution engine processes vertices selectively based on their types so as to preclude the redundant computations of source, sink, and isolated vertices during iterative execution. After the preprocessing phase, which performs vertex classification (see Section 3.3), each group leader GL_k of row-group RG_k and column-group CG_k executes the user-defined vertex program in three stages: *pre-loop*, *main-loop*, and *post-loop*, as outlined in Algorithm 2[8].

Algorithm 2 – Execution Engine (At Process GL_k)

```
1: Input: Vertex Program $V$
2: Input: 2D-Staggered Placement Grid $\boldsymbol{\mathcal{G}}$
3: Input: $GL_k$'s Tiles, $\boldsymbol{\mathcal{T}}_k$, Vector Segments, $\mathbf{v}_k$, $\mathbf{x}_k$, $\mathbf{y}_k$, $\widehat{\mathbf{y}}_k$
4: Input: Number of iterations $n$
5: $\mathbf{v}_k.init()$                                        ▷ Pre-loop
6: $\mathbf{x}_k^{src} \leftarrow \mathbf{v}_k^{src}.scatter()$            ▷ Source vertices
7: $\mathbf{x}_k^{src}$.multicast$_{async}(CG_k)$                   ▷ To col-group
8: while $n > 0$ and $V$ is not converged do                ▷ Main Loop
9:    $\mathbf{x}_k^{reg} \leftarrow \mathbf{v}_k^{reg}.scatter()$           ▷ Regular vertices
10:   $\mathbf{x}_k^{reg}$.multicast$_{async}(CG_k)$                 ▷ To col-group
11:   for each $\mathbf{x}_j$ in $\mathbf{x}_*$ do (on recv$_{async}$)
12:      for each $\mathbf{T}_{i,j}$ in $\boldsymbol{\mathcal{T}}_k$ do              ▷ Regular sub-tiles
13:         $\widehat{\mathbf{y}}_i^{reg} \oplus= (\mathbf{T}_{i,j}^{src,reg} \otimes \mathbf{x}_j^{src})$     ▷ $\mathbf{x}_j^{src}$ is cached
14:         $\widehat{\mathbf{y}}_i^{reg} \oplus= (\mathbf{T}_{i,j}^{reg,reg} \otimes \mathbf{x}_j^{reg})$     ▷ Partial accumulator
15:         $\widehat{\mathbf{y}}_i^{reg}$.send$_{async}(GL_i)$           ▷ To row-group leader
16:   for each $\widehat{\mathbf{y}}_k^{reg}$ in $\widehat{\mathbf{y}}_*$ do (on recv$_{async}$)
17:      $\mathbf{y}_k^{reg} \oplus= \widehat{\mathbf{y}}_k^{reg}$                       ▷ Final accumulator
18:   $\mathbf{v}_k^{reg}.apply(\mathbf{y}_k^{reg})$                       ▷ Regular vertices
19:   $n \leftarrow n - 1$
20: for each $\mathbf{x}_j$ in $\mathbf{x}_*$ do (on recv$_{async}$)          ▷ Post-loop
21:   for each $\mathbf{T}_{i,j}$ in $\boldsymbol{\mathcal{T}}_k$ do                ▷ Sink sub-tiles
22:      $\widehat{\mathbf{y}}_i^{snk} \oplus= (\mathbf{T}_{i,j}^{src,snk} \otimes \mathbf{x}_j^{src}) \oplus (\mathbf{T}_{i,j}^{reg,snk} \otimes \mathbf{x}_j^{reg})$
23:      $\widehat{\mathbf{y}}_i^{snk}$.send$_{async}(GL_i)$
24: for each $\widehat{\mathbf{y}}_k^{snk}$ in $\widehat{\mathbf{y}}_*$ do (on recv$_{async}$) $\mathbf{y}_k^{snk} \oplus= \widehat{\mathbf{y}}_k^{snk}$
25: $\mathbf{v}_k^{snk}.apply(\mathbf{y}_k^{snk})$                            ▷ Sink vertices
```

In the pre-loop stage (Algorithm 2: lines 5-7), all vertices (i.e., all four types) are firstly initialized. Afterwards, each *source* vertex generates a new message, which is multicast by GL_k to every other process in its column-group CG_k. Thereafter, the source vertices are deactivated, especially because they will receive no incoming messages. In addition, every receiving process in CG_k *caches* the received messages and processes them iteratively, as described below.

After the pre-loop stage, the execution enters the main-loop stage (Algorithm 2: lines 8-19). First, each *regular* vertex generates a new message, which is multicast by GL_k to every other process in its column-group CG_k. Alongside, since GL_k is also a member of other column-groups, it simultaneously receives messages from the group leaders of these column-groups. As an example, GL_2 (i.e., process P_4) in Figure 6 owns four tiles in two columns of $\boldsymbol{\mathcal{G}}$ (i.e., $\mathbf{T}_{2,2}$ and $\mathbf{T}_{3,2}$ in column 2, and $\mathbf{T}_{2,4}$ and $\mathbf{T}_{3,4}$ in column 4). Therefore, GL_2 is also a member of CG_4, hence, it receives messages from the leader of CG_4 (i.e., process P_3). Clearly, GL_2 needs these messages to process its tiles (or more precisely, its sub-tiles) in column 4.

While simultaneously receiving these messages, GL_k begins computing the SpMV product of each of its *regular* sub-tiles (i.e., $\mathbf{T}_{i,j}^{src,reg}$ and $\mathbf{T}_{i,j}^{reg,reg}$ defined in Section 3.3) against their respective source and regular messages. These SpMV computations are pursued in a column-wise fashion. Specifically, GL_k always starts with the k^{th} column since it already owns the messages needed to process its regular

[7] LA3 detects this at compile time from the user code.

[8] We note that all processes in grid $\boldsymbol{\mathcal{G}}$ are group leaders of some row- and column-groups, so they all run Algorithm 2 in parallel.

sub-tiles in column k. Afterwards, for each message segment that it receives from another column-group, it processes its regular sub-tiles in this group's column. For instance, GL_2 in Figure 6 starts immediately processing its regular sub-tiles in column 2. Then, as soon as it receives the required messages from CG_4, it processes its regular sub-tiles in column 4. To expedite performance, LA3 uses OpenMP [6] to parallelize sub-tile computations in any given column.

Furthermore, GL_k sends the outcome of each regular sub-tile it processes to the row-group leader of that sub-tile's row. Moreover, as a row-group leader itself, GL_k receives from the members (or processes) of its row-group the outcomes of their sub-tiles. Lastly, it accumulates the received outcomes with the outcomes of its own sub-tiles and uses the result to update the states of its regular vertices. For example, GL_2 in Figure 6 sends the outcomes of $\mathbf{T}_{3,2}$ and $\mathbf{T}_{3,4}$ to the leader of RG_3. And, being the leader of RG_2, it receives the outcomes of $\mathbf{T}_{2,1}$ and $\mathbf{T}_{2,3}$ in row 2. Finally, it accumulates the received outcomes with the outcomes of its own sub-tiles in row 2, namely, $\mathbf{T}_{2,2}$ and $\mathbf{T}_{2,4}$.

When the main-loop stage is done, either due to convergence or having reached a pre-defined number of iterations, execution enters the post-loop stage (Algorithm 2: lines 20-25). In this stage, GL_k computes its *sink* vertices via gathering and processing incoming messages from their neighboring source and regular vertices.

Discussion. The computation filtering optimization is most impactful with directed, stationary applications (e.g., PageRank over directed graphs), since all computations related to source and sink vertices, which remain active throughout execution, are factored out of the main-loop stage. With undirected applications, all vertices are either regular or isolated since the input graph is bidirectional, hence, only isolated vertices can be factored out. With directed, non-stationary applications, source vertices get deactivated after sending their initial messages and never get re-activated again, thus they are naturally factored out of the main loop.

Lastly, computation filtering does not apply to non-iterative applications (e.g., Triangle Counting) since its benefits only accrue over multiple iterations. In addition, it does not apply to non-LA3-optimizable applications (see Section 3.1), which may require non-regular vertices to be computed using iteration-specific information (see the BFS example in Section 3.1). As a result, LA3 disables computation filtering for non-iterative and non-LA3-optimizable applications.

3.4.2 Communication Filtering

To minimize network traffic, LA3's engine utilizes a point-to-point communication layer, which applies a *communication filtering* optimization that exploits sparsity patterns inherent in many real-world graphs. In particular, since real-life graphs are typically *sparse*, they have small average in- and out-degrees. Moreover, since they tend to follow power-law distributions, most of their vertices have in- and out-degrees lower than the average in- and out-degrees. Consequently, when the adjacency matrix of a real-world graph is partitioned using a 2D partitioning algorithm, most of its tiles are rendered *hyper-sparse* (i.e., most elements – or edges – in these tiles are zeros – or non-existent).

LA3's engine leverages hyper-sparsity in tiles via using the metadata generated during the pre-processing phase (see Section 3.3) to communicate only relevant messages between processes. Specifically, a column-group leader prepares for each process, P_i, in its group a separate outgoing $\mathbf{x}$-segment, which contains only the messages from its own $\mathbf{x}$-segment that match a non-empty column in one or more tiles of P_i. As an outcome, P_i is sent only its needed subset of messages, reducing thereby network traffic. Interestingly, when communication filtering (which reduces traffic more with sparser tiles) is utilized in conjunction with computation filtering (which makes tiles even sparser by factoring out non-regular vertices) its benefits are further magnified.

3.4.3 Pseudo-Asynchronous Computation and Communication

LA3's point-to-point communication layer allows processes to exchange vector segments asynchronously during SpMV execution. Clearly, this allows interleaving computation with communication. However, while interactions between processes are fully asynchronous, a process may still need to block if it lacks necessary $\mathbf{x}$- or partial $\mathbf{y}$-segments to proceed further, which shall be sent to it under the current iteration. Consequently, we refer to this overall optimization as *pseudo-asynchronous* computation and communication.

We note that our locality-aware 2D-Staggered tile placement strategy ensures that each process has at least one $\mathbf{x}$-segment (i.e., its own) ready to be consumed at the beginning of every iteration (see Section 3.2). Hence, each process can proceed immediately with computation under a new iteration. Furthermore, while the process is consuming and processing its own $\mathbf{x}$-segment, it will likely receive other $\mathbf{x}$-segment(s) asynchronously. Therefore, by means of locality-aware tile placement and through potentially blocking only for some segments (if any), LA3 is able to circumvent the need for expensive global barriers across iterations. This yields superior performance versus related systems as demonstrated next.

4. PERFORMANCE EVALUATION

4.1 Methodology

We ran our performance experiments on a cluster of 36 virtual machines (VMs) hosted on a private research cloud (*RC*) with 20 physical Dell PowerEdge R710 servers. Each server has dual Intel Xeon 5690 processors (12 cores total), 144 GB of memory, and dual 10 Gbps Ethernet connections. Each VM has 8 virtual cores, 30 GB of memory, and 1 Gbps of network bandwidth. Our scalability experiments were conducted on Amazon (*EC2*) with up to 128 m4.2xlarge instances. Each instance has 8 virtual cores, 31 GB of memory, and 1 Gbps of network bandwidth.

Systems. We compare LA3 against six systems listed in Table 2 and discussed in Section 5. Among these systems, Gemini and GraphPad represent the state-of-the-art distributed graph analytics systems, while the other four, namely, PowerGraph, Giraph, GraphX, and CombBLAS, are popular and widely used in academia and industry. Table 2 also shows the values assigned to some configuration parameters in the seven systems (including LA3). These values were empirically selected for each system to maximally utilize the available resources on RC and EC2, respectively.

Applications. We benchmarked each system using five popular graph analytics applications, namely, PageRank (PR), Single-Source Shortest Path (SSSP), Breadth First Search (BFS), Connected Components (CC), and Triangle Counting (TC), in line with most related works (in particular [38, 11]). These applications run over a variety of graph types as

Table 2: Systems evaluated on private cloud (RC) and Amazon (EC2) with different multi-threading and inter-process communication (IPC) modes. Parameters *p*, *ppn*, and *tpp* stand for *total number of processes*, *processes per node*, and *threads per process*.

System	Thread.	IPC	On	*p*	*ppn*	*tpp*
LA3	OpenMP	MPI	RC	144	4	2
			EC2	32-512	4	2
Gemini [42]	OpenMP	MPI	RC	36	1	8
			EC2	8-128	1	8
GraphPad [11][9]	OpenMP	MPI	RC	144	4	1
			EC2	8-128	1	8
PowerGraph [19]	OpenMP	MPI	RC	36	1	8
			EC2	8-128	1	8
Giraph [1]	JVM	Netty	RC	36	1	8
GraphX [40]	JVM	Spark	RC	36	1	8
CombBLAS [12][10]	OpenMP	MPI	RC	289	8	1

Table 3: Benchmark graph applications.

App.	Directed	Weighted	Stationary	Input Edge List
PR	Yes	No	Yes	Pairs
SSSP	Yes	Yes	No	Triples
BFS	No	No	No	Bidirectional pairs
CC	No	No	No	Bidirectional pairs
TC	No	No	No	Upper-triangular pairs

shown in Table 3. We also indicate that not every system we compared against provided implementations for all of these applications, so we coded some implementations as needed.

Datasets. We ran each benchmark on a variety of real-world and synthetic datasets representing a range of different graph types and sizes (see Table 4). The evaluations on RC included real-world datasets only. For the EC2 scalability study, in order to evaluate the impact of scaling the dataset size while keeping all other factors fixed, we utilized synthetic datasets generated using the Graph 500 Kronecker generator with its default parameter values [13, 4].

We prepared four versions of each dataset: directed pairs (unweighted), directed triples (weighted randomly between 1 and 128), bidirectional pairs, and upper-triangular pairs. All self-loops and duplicate edges were removed in the process. Moreover, we prepared each version in both binary (for LA3, Gemini, GraphPad, and PowerGraph) and text (for Giraph, GraphX, and CombBLAS) formats.

To account for performance variations we ran each experiment thrice and reported the average runtime over the three runs[11]. Moreover, any experiment that ran longer than 1 hour was stopped and assumed to have failed silently. In line with most related works, the reported runtimes do not include the ingress (i.e., loading and pre-processing) time, which is paid only once per loaded graph, and may be amortized via running as many algorithms as possible and for as many times as possible on the loaded graph[12].

4.2 Performance on Private Cloud

Let us begin with the experiments on our private cloud. Figures 7(a-e) compare LA3's performance against the six considered systems using PR, SSSP, BFS, CC (in terms of LA3's relative *speedup*), and TC (in terms of actual runtimes) on each of our six real-world datasets. Figure 7(f) shows the *overall* speedup (i.e., the geometric mean over all datasets for a given application) versus each system and across all systems. Lastly, Figure 8 plots LA3's runtime for each application on each dataset. Next, we discuss these results and our main takeaways.

[9] GraphPad's multithreading worked well on EC2 but not on RC.
[10] CombBLAS requires a square number of processes.
[11] Overall average standard deviation per experiment was 3.8%.
[12] Although not shown, LA3's ingress times were the fastest.

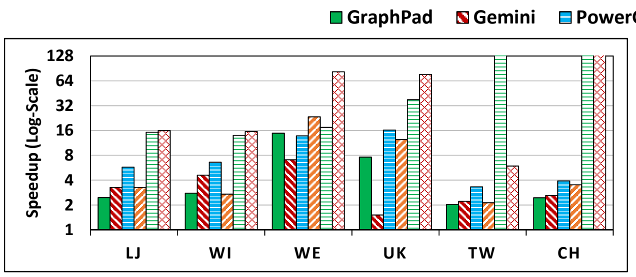
Table 4: Real-world and synthetic datasets.

Dataset	|Vertices|	|Edges|	Type
LiveJournal (LJ) [5]	4,847,571	68,993,773	Social
Wikipedia (WI) [10]	3,566,907	45,030,389	Web
Weibo (WE) [9]	58,655,849	261,321,033	Social
UK-2005 (UK) [8]	39,459,925	936,364,282	Web
Twitter (TW) [7]	61,578,415	1,468,365,182	Social
CC 2012 Host (CH) [2]	101,000,000	2,043,000,000	Web
RMAT 23 (R23)	8,388,608	134,217,728	Synthetic
RMAT 24 (R24)	16,777,216	268,435,456	Synthetic
RMAT 25 (R25)	33,554,432	536,870,912	Synthetic
RMAT 26 (R26)	67,108,864	1,073,741,824	Synthetic
RMAT 27 (R27)	134,217,728	2,147,483,648	Synthetic
RMAT 28 (R28)	268,435,456	4,294,967,296	Synthetic
RMAT 29 (R29)	536,870,912	8,589,934,592	Synthetic

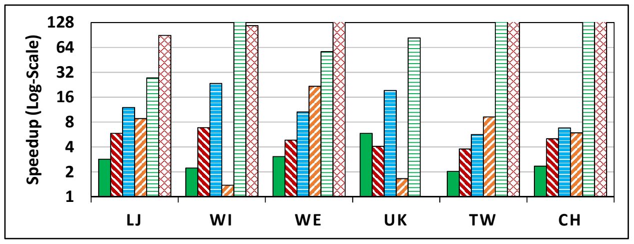
LA3 Performance. Figure 8 shows the runtime trends of each benchmark on LA3. One noteworthy application is SSSP on Weibo, which finishes very quickly. This is due to the nature of Weibo's link structure, which includes 99% source vertices and only 1% regular vertices (see Table 1). As a consequence, the main loop converges extremely quickly (using only 15 iterations), since it needs to compute only 1% of the vertices, tapping mainly into LA3's computation filtering optimization.

Another notable application is TC on UK-2005, which takes a relatively long time to complete. This is mainly for three reasons. First, the vast majority of the UK-2005's vertices are regular (i.e., 88%, which is the highest percentage among all the datasets in Table 1). Since TC only counts triangles at regular vertices, this yields a high number of computations. Second, TC is a non-iterative program, thereby missing out on many of LA3's optimizations. In particular, LA3's optimizations are geared largely towards iterative applications since their benefits accrue over multiple iterations. This applies especially to computation filtering, which factors redundant computations out of the *main loop*. Likewise, communication filtering is applied on a per-iteration basis. To a lesser extent, pseudo-asynchronous execution allows successive iterations to overlap for efficiency. Consequently, these benefits cannot be effectively reaped by non-iterative applications like TC. Third, TC runs on an upper-triangular version of UK-2005's adjacency matrix, thus failing to exploit its inherent locality. More precisely, most of the non-zeros (edges) in UK-2005's original matrix are laid out as *clusters* of highly related vertices. If these clusters are somehow maintained during partitioning, locality will be leveraged during execution, leading to less communication overhead and potentially enhanced performance. Unfortunately, this locality is lost when UK-2005's matrix is transformed into an upper-triangular one, as needed by TC.

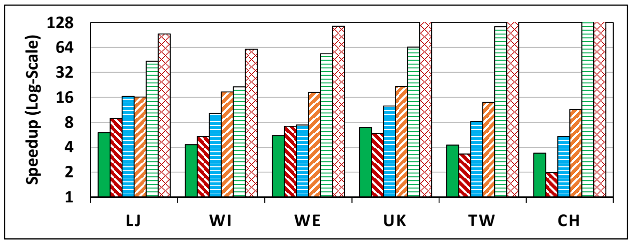
On the other hand, PR, SSSP, BFS, and CC on UK-2005 perform better than TC on UK-2005 as demonstrated in Figure 8. Clearly, this is because all these applications are iterative, thus harnessing more LA3's optimizations. Furthermore, they preserve UK-2005's inherent locality via running on its original adjacency matrix. To allow preserving this locality, we disabled vertex hashing, which shuffles vertices around in any given graph's adjacency matrix (see Section 3.2). As discussed in Section 3.2, this can be done simply via a flag (which is available in LA3's API).



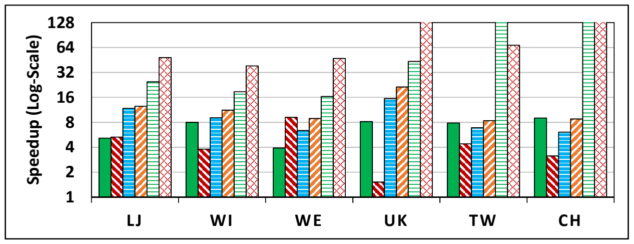
(a) PR



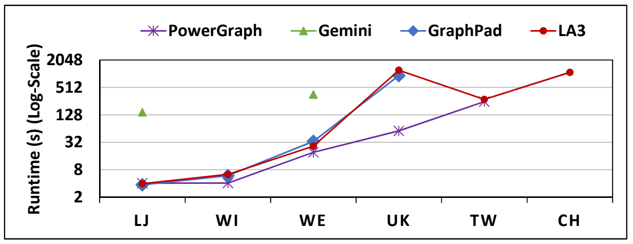
(b) SSSP



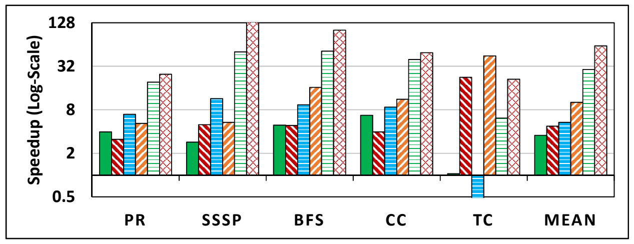
(c) BFS



(d) CC



(e) TC



(f) Application-wise & overall speedup vs. others.

Figure 7: Performance results on RC: LA3 versus other systems on PR, SSSP, BFS, CC (relative speedups), and TC (actual runtimes) on six real-world datasets. Values <1× denote slowdown. All figures are $\log_2$-scale.

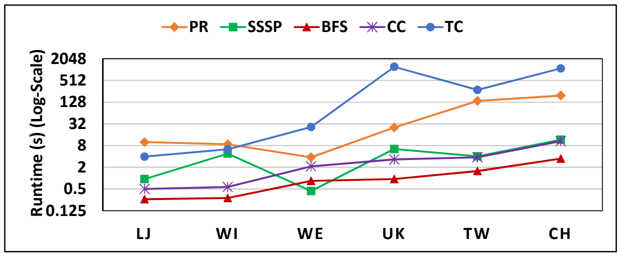


Figure 8: LA3 runtimes.

Figures 7(a-d) show that LA3 outperforms all the tested systems using PR, SSSP, BFS, and CC on all our datasets. On the flip side, as depicted in Figure 7(e), LA3 using TC performs comparably to GraphPad, especially since both are tailored specifically for iterative applications. However, LA3 using TC is slower than PowerGraph (especially, on UK-2005) for the main reasons discussed above. In addition, PowerGraph attempts to conserve by design the natural locality in input graphs (e.g., UK-2005) [19]. Nonetheless, we note that LA3 was the only system among all the tested systems that was able to successfully run TC on all our datasets. For instance, GraphPad failed on Twitter (TW) and CC 2012 Host (CH), while PowerGraph failed on CH (see Figure 7(e)) upon using TC.

Overall Speedups. Across all our benchmarks, LA3 achieved average speedups of 3.5×, 4.7×, 5.4×, 10× and 29× versus GraphPad, Gemini, PowerGraph, Giraph and GraphX, respectively (see Figure 7(f)). While GraphPad is a linear algebra-based system (akin to LA3), LA3 employs more advanced placement, computation, and communication optimizations. We discuss the performance gains of these optimizations shortly (see Table 5). Alongside, LA3 performs better than Gemini, especially at scale, as discussed in Section 4.3. As opposed to Gemini, LA3 uses 2D partitioning (which yields less communication overhead compared to 1D – see Section 3.2) in combination with a locality-aware placement strategy, let alone factoring out redundant computation and communication (unique to LA3).

In contrast, PowerGraph, Giraph, and GraphX are all non-linear algebra-based systems. Linear algebra operations are more amenable to low-level optimizations [23] (e.g., highly compressed data structures can be used to mitigate memory pressure). Alongside, Giraph and GraphX are fully synchronous and run on Java Virtual Machines (JVMs), which add various overheads like extra instruction-level and garbage collection overheads, among others. In fact, GraphX is highly memory inefficient, especially with long jobs and large-scale datasets (see Figure 7(a-d)), caused mainly by its ill-optimized data structures and Java environment. Similar observations were also reported in Kabiljo et al. [3].

Finally, LA3 achieved a 61× average speedup against CombBLAS, which is essentially an HPC library designed to run optimally on HPC platforms rather than cloud infrastructures. However, a major weakness of CombBLAS is its linear algebra programming model, which does not provide an obvious way to maintain and exploit vertex activity for non-stationary applications. Hence, all vertices are treated as

Table 5: Breakdown of optimization impacts averaged over our real-world datasets. Speedups with enabled optimizations relative to baseline (All-Off).

Application	Filter-On	Async-On	All-On
PR	1.67×	1.37×	2.14×
SSSP	2.52×	1.26×	3.01×
BFS	1.69×	1.01×	1.83×
CC	2.01×	1.38×	2.56×
Average	1.95×	1.26×	2.35×

active throughout any program, resulting in a great deal of unnecessary computations and communications (especially for non-stationary applications, which are not supposed to have all their vertices constantly active). In addition to being aware of the application type (i.e., whether stationary or non-stationary), LA3 further attempts to exclude redundant computations and communications via its respective filtering optimizations.

Optimizations. In Table 5, we illustrate the performance impacts of LA3's major optimizations for PR, SSSP, BFS, and CC, averaged across all of our real-world datasets[13]. Specifically, we measured LA3's performance in four different modes: (1) all optimizations disabled (*All-Off*); (2) only computation and communication filtering enabled (*Filter-On*); (3) only pseudo-asynchronous computation and communication enabled (*Async-On*); and (4) all optimizations enabled (*All-On*). We report the speedups under the latter three modes relative to the baseline mode (i.e., All-Off). As revealed in the table, the Filter-On and Async-On modes yield notable individual speedups. Overall, enabling all optimizations via the All-On mode results in a 2.35× speedup on average, demonstrating the value of LA3's optimizations.

4.3 Scalability on Public Cloud

We conduct two sets of experiments on Amazon EC2 to evaluate LA3's *data scalability* (*DS*) (i.e., using different synthetic dataset sizes) and *cluster scalability* (*CS*) (i.e., using different cluster sizes), respectively. We benchmark LA3 under PR, CC, and TC, representing stationary, non-stationary, and non-iterative applications, respectively. We compare LA3's scalability against the two related systems that performed best under each application in our private cloud, namely, GraphPad and Gemini for PR and CC, and GraphPad and PowerGraph for TC.

Data Scalability. In this set of experiments, we fixed the number of VMs to 32, while varying the dataset size. We tested PR and CC on R25 to R29, and TC on R23 to R27, which are all RMAT datasets (see Table 4). We selected smaller datasets for TC since it uses substantially more CPU and memory than the other two programs under any given dataset size. The runtimes of each experiment are plotted in Figure 9(a-c). As the dataset size was increased, LA3 outperformed increasingly GraphPad and Gemini under PR. This is mainly due to LA3's computation filtering, which greatly benefits directed, stationary, and iterative applications such as PR. Moreover, LA3 performed increasingly better than GraphPad (for the reasons discussed in the previous section), but comparably to Gemini under CC (at this cluster size of 32 VMs, but not at larger cluster sizes as discussed next). Finally, LA3 was slightly slower than GraphPad under TC but faster than PowerGraph, which failed to scale beyond R25. As discussed earlier, PowerGraph is a non-linear algebra-based system that does not employ highly compressed data structures. Consequently, as the dataset size was increased, it started experiencing exponentially higher memory pressure to the extent that it ran out-of-memory with R26 and R27.

Cluster Scalability. In this set of experiments, we fixed the dataset size, while varying the number of VMs from 8 to 128. In particular, we tested PR and CC on R27, and TC on R25. The runtimes of each experiment are depicted in Figure 9(d-f). Once again, LA3 greatly outperformed GraphPad and Gemini under PR. In fact, Gemini failed to scale beyond 64 VMs due to its 1D partitioning strategy, which does not scale as good as 2D partitioning approaches (see Section 2). Moreover, LA3 surpassed GraphPad under CC, which failed to fit R27 on eight VMs. However, Gemini was faster than LA3 using CC over 8-16 VMs but failed to scale beyond 64 VMs. This is because the loss from 2D partitioning (mainly, complexity) does not get offset by its gain (mainly, reduced communication traffic) at small-scale cluster sizes, and vice versa. Finally, similar to the data scalability results, LA3 was slightly slower than GraphPad but faster than PowerGraph under TC.

5. RELATED WORK

We now provide a brief overview of related centralized and distributed graph analytics systems. Table 6 classifies each system based on its architecture, graph processing paradigm, and partitioning approach.

Centralized. Most centralized graph analytics systems, including GraphLab [27], GraphChi [26], Ligra [37], MapGraph [16], G-Store [25], Mosaic [29], and GraphMat [38] offer a vertex-based abstraction. While GraphChi, G-Store, and Mosaic support out-of-core execution for processing big graphs on memory-constrained machines, GraphLab, Ligra, GraphMat, and MapGraph are in-memory systems. In contrast to the above systems, X-Stream [34] provides an edge-based abstraction with a streaming-oriented engine for out-of-core graph processing. Lastly, Galois [32] is a general-purpose, task-based system that performs competitively with specialized graph analytics systems [35].

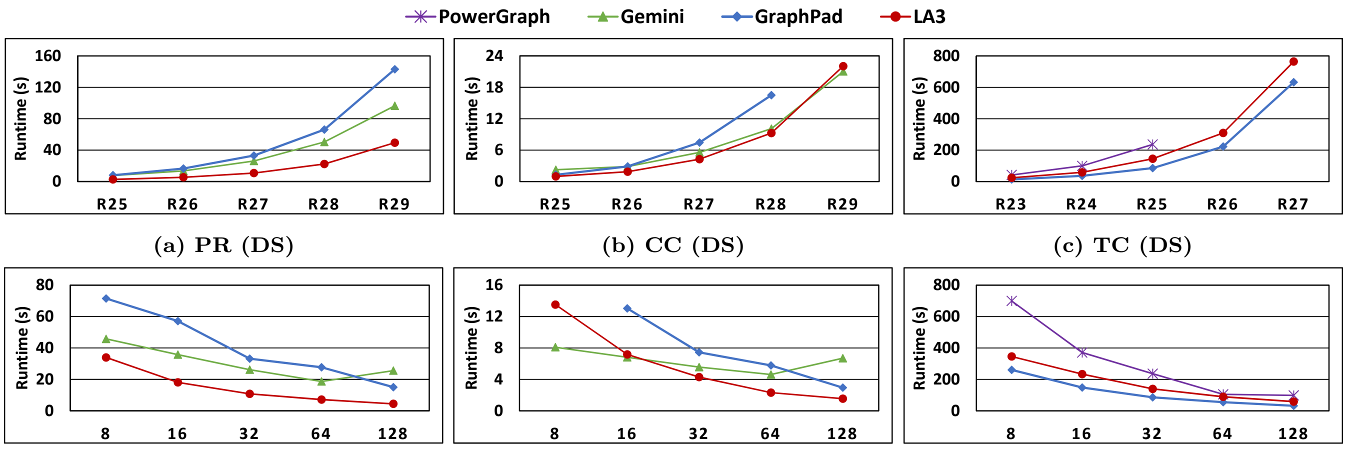
Among these centralized systems, GraphMat bears the most similarity to LA3. In particular, its execution engine translates vertex-centric code into parallelized SpMV. However, akin to all centralized systems, GraphMat cannot scale beyond the resources available on a single machine, whereas LA3 is demonstrably scalable[14].

Distributed. Most distributed graph systems can be classified as *vertex-centric*, *linear-algebra-centric*, or *hybrid.*

Vertex-Centric: Many distributed graph analytics systems adopt the vertex-centric paradigm, including Giraph [1], GraphX [40], Distributed GraphLab [28], PowerGraph [19], Gemini [42], PowerLyra [14], Mizan [24], and Chaos [33]. For instance, Giraph is an open-source implementation of Pregel [30], which partitions vertices across processes via an edge-cut strategy and implements the Bulk Synchronous Parallel (BSP) [18] execution model. Mizan is another Pregel-like system, with dynamic load balancing to alleviate computation imbalance caused by a naïve edge-cut partitioning.

[13] LA3's optimizations do not benefit non-iterative applications like TC, which by their very nature disable these optimizations.

[14] Although not included in the performance evaluation, our experiments indicate that LA3 is 1.34× slower than GraphMat on average for PR, SSSP, BFS, and CC on a single machine. This is due to the overheads associated with LA3's distributed engine.



(a) PR (DS) (b) CC (DS) (c) TC (DS)

(d) PR (CS) (e) CC (CS) (f) TC (CS)

Figure 9: Data scalability (DS) and cluster scalability (CS) results for PR, CC, and TC on EC2.

Table 6: Related graph analytics systems.

Centralized				Distributed					
Vertex-Centric		**Hybrid**	**Misc**	**Vertex-Centric**		**Linear-Algebra**		**Hybrid**	**Misc**
In-Memory	*Out-of-Core*	*In-Memory*	*Out-of-Core*	*Edge-Cut*	*Vertex-Cut*	*1D*	*2D*	*2D*	
[27, 37, 16]	[26, 25, 29]	[38]	[34, 32]	[30, 1, 24, 42, 14]	[19, 40, 33]	[39]	[12]	[11]	[36, 31, 21]
GraphLab	GraphChi	GraphMat	X-Stream	Pregel, Giraph	PowerGraph	Presto	CombBLAS	GraphPad	SociaLite
Ligra	G-Store		Galois	Mizan, Gemini	GraphX			**LA3**	Naiad
MapGraph	Mosaic			PowerLyra	Chaos				PEGASUS

GraphX is a graph processing library on top of Spark [41], thus inheriting Spark's interpreted runtime along with its scalability and fault-tolerance characteristics. It uses a sparsity-aware tabular data structure, which allows it to express various graph operations in terms of joins.

PowerGraph is an extension of Distributed GraphLab that addresses skew in real-world graphs. It decomposes each iteration of a graph program into three phases: *Gather*, *Apply*, and *Scatter* (commonly referred to as the GAS model). The GAS model can effectively emulate Pregel's canonical vertex-centric API. Gather collects information about the edges and neighbors of a given vertex (say, v), Apply executes a user-defined function on v to update its value, and Scatter uses this new value to update the data on v's edges. LA3's vertex-centric API is inspired by the GAS model.

Gemini extends Ligra's hybrid push-pull computational model [37] to a distributed setting. It dynamically switches between push- and pull-based message passing based on the graph program's level of activity. In push mode, each active vertex updates the states of its out-neighbors. In pull mode, each vertex gathers messages from its in-neighbors to update its state. Pushing is preferable when few vertices are active.

Linear-Algebra-Centric: CombBLAS [12] is a well-known and scalable linear algebra (LA) library for distributed graph analytics in the HPC domain. It adopts 2D partitioning and requires the number of processes to be square. Presto [39] is a distributed extension of R [17], an array-based language for statistical analysis and LA.

Hybrid: To the best of our knowledge, aside from LA3, GraphPad [11] is the only distributed graph analytics system under the hybrid category. It is the distributed successor of GraphMat, extending its centralized 1D SpMV-based execution engine to support distributed 2D SpMV, while retaining its GAS-like vertex-based interface. GraphPad optimizes mainly for communication. Vector segments are compressed (only when sparse) and communicated using asynchronous point-to-point MPI calls. LA3 always compresses segments, communicating them together with their bitvectors in just one message round (GraphPad uses two). Moreover, GraphPad applies post-Scatter and pre-Apply global synchronization barriers, thus losing the potential advantages of employing asynchrony, while LA3 exploits asynchrony heavily.

Finally, compared to all of the above distributed graph analytics systems, LA3 employs the hybrid paradigm, but uniquely differentiates itself via suggesting novel system optimizations informed by the link structures of big graphs and the behaviors of graph applications. In particular, its locality-aware 2D-Staggered tile placement in conjunction with its pre-processing enable several impactful optimizations within its execution engine, as discussed in Section 3.

6. CONCLUSION

We presented LA3, a scalable distributed graph analytics system, which offers a vertex-based programming interface on top of a highly-optimized LA-based engine. In particular, LA3 involves three major optimizations, namely, computation filtering for precluding redundant computations, communication filtering for reducing network traffic, and a pseudo-asynchronous paradigm for interleaving computation with communication and asynchronous message passing. Our results show that LA3 can outperform the six state-of-the-art and most popular graph analytics systems by an average of 10×. We plan to extend LA3 by including various other LA operations (e.g., sparse matrix-matrix multiplication) and exposing them via an LA-based API.

7. ACKNOWLEDGMENTS

This publication was made possible by NPRP grant #7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

8. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/, 2017.

[2] Common Crawl 2012 Host graph. http://webdatacommons.org/hyperlinkgraph/2012-08/, 2017.

[3] A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/, 2017.

[4] Graph500. http://graph500.org/, 2017.

[5] LiveJournal graph. https://snap.stanford.edu/data/soc-LiveJournal1.html, 2017.

[6] OpenMP. http://www.openmp.org/, 2017.

[7] Twitter graph. http://law.di.unimi.it/webdata/twitter-2010/, 2017.

[8] UK-2005 graph. http://networkrepository.com/web-uk-2005-all.php, 2017.

[9] Weibo graph. http://networkrepository.com/soc-sinaweibo.php, 2017.

[10] Wikipedia graph. https://www.cise.ufl.edu/research/sparse/matrices/Gleich/wikipedia-20070206.html, 2017.

[11] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. GraphPad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 313–322, 2016.

[12] A. Buluç and J. R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: a recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446, 2004.

[14] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 1:1–1:15, 2015.

[15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[16] Z. Fu, B. B. Thompson, and M. Personick. MapGraph: A high level API for fast development of high performance graph analytics on gpus. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, pages 2:1–2:6, 2014.

[17] R. Gentleman, W. Huber, and V. J. Carey. R language. In *International Encyclopedia of Statistical Science*, pages 1159–1161. 2011.

[18] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[20] W. Han, X. Zhu, Z. Zhu, W. Chen, W. Zheng, and J. Lu. Weibo, and a tale of two worlds. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2015, Paris, France, August 25 - 28, 2015*, pages 121–128, 2015.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[22] J. Kepner, P. Aaltonen, D. A. Bader, A. Buluç, F. Franchetti, J. R. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. G. Mattson, and J. E. Moreira. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–9, 2016.

[23] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, 2011.

[24] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 169–182, 2013.

[25] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 830–841, 2016.

[26] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46, 2012.

[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 340–349, 2010.

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

[29] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 527–543, 2017.

[30] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In

Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, pages 135–146, 2010.

[31] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455, 2013.

[32] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013.

[33] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 410–424, 2015.

[34] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 472–488, 2013.

[35] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 979–990, 2014.

[36] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A Datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.

[37] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146, 2013.

[38] N. Sundaram, N. Satish, M. M. A. Patwary, S. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: high performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

[39] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 197–210, 2013.

[40] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, page 2, 2013.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. 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 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[42] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 301–316, 2016.