# Efficient Distributed Graph Analytics using Triply Compressed Sparse Format

Mohammad Hasanzadeh Mofrad, Rami Melhem
*University of Pittsburgh*
Pittsburgh, USA
{moh18, melhem}@pitt.edu

Yousuf Ahmad and Mohammad Hammoud
*Carnegie Mellon University in Qatar*
Doha, Qatar
{myahmad, mhhamoud}@cmu.edu

***Abstract*—This paper presents Triply Compressed Sparse Column (TCSC), a novel compression technique designed specifically for matrix-vector operations where the matrix as well as the input and output vectors are sparse. We refer to these operations as SpMSpV$^2$. TCSC compresses the nonzero columns and rows of a highly sparse input matrix representing a large real-world graph. During this compression process, it encodes the sparsity patterns of the input and output vectors within the compressed representation of the sparse matrix itself. Consequently, it aligns the compressed indices of the input and output vectors with those of the compressed matrix columns and rows, thus eliminating the need for extra indirections when SpMSpV$^2$ operations access the vectors. This results in fewer cache misses, greater space efficiency and faster execution times. We evaluate TCSC's performance and show that it is more space and time efficient compared to CSC and DCSC, with up to $11\times$ speedup. We integrate TCSC into GraphTap, our suggested linear algebra-based distributed graph analytics system. We compare GraphTap against GraphPad and LA3, two state-of-the-art linear algebra-based distributed graph analytics systems, using different dataset scales and numbers of processes. We demonstrate that GraphTap is up to $7\times$ faster than these systems due to TCSC and the resulting communication efficiency.**

***Index Terms*—Triply compressed sparse column, TCSC, sparse matrix, sparse vector, SpMV, SpMSpV$^2$, big graphs, distributed graph analytics**

## I. Introduction

Scalable systems for big data analytics are invaluable tools for efficiently extracting insights from vast volumes of data generated mainly by billions of Internet-connected users and devices. Big data domains that focus on the relationships between data points (e.g., the Web, social networks, recommendation systems, and road networks, to name just a few) typically model such data as graphs. Most traditional graph analytics systems employ vertex-centric computation systems [1]–[10]. However, many recent systems have opted alternatively for linear algebra-based computation systems, leveraging decades of work by the HPC community on optimizing the performance and scalability of basic linear algebra operations [11]–[20].

In the language of linear algebra, a graph is usually represented as an adjacency matrix and most common graph operations can be executed atop this matrix using a handful of generalized basic linear algebra primitives [12]. Also, since big real-world graphs tend to produce highly sparse matrices, the *data structures* and *algorithms* associated with these operations need to be highly optimized for sparsity. Often, such optimizations are pursued independently, resulting in various algorithms that do not inherently exploit certain common data structural optimizations and vice-versa. In this paper, we show that tightly coupling specific algorithmic and data structural optimizations can yield significant performance and scalability benefits in both centralized and distributed settings.

To elaborate, given an input graph, $G$, with $n$ vertices, most graph algorithms on $G$ can be translated to an iteratively executed Sparse Matrix-Vector (SpMV) operation, $y = A \oplus\cdot\otimes x$, where $A$ is the transpose of $G$'s $n \times n$ adjacency matrix, $x$ and $y$ are input and output vectors of length $n$, and $\oplus\cdot\otimes$ is a pair of overridable additive and multiplicative operations [12]. The algorithms would then iteratively apply the results from $y$ back to $x$, looping until they converge or stopped after certain numbers of iterations. The sparse matrix, $A$, is commonly stored using some variant of Compressed Sparse Column (CSC), which essentially compresses its nonzero elements into an array [12], [21]. As for $x$ and $y$, they may be stored in either a dense or a sparse vector representation [12], [22]. The dense representation stores an uncompressed array of length $n$. However, real-world graph matrices tend to have significantly many empty columns and rows, rendering the corresponding $x$ and $y$ elements irrelevant during SpMV [19]. Therefore, a sparse format maintains relevant elements only, either as a compressed list of (index, value) pairs, or as an uncompressed array paired with a bitvector that marks the relevant entries only. While the compressed form is more space efficient, it does not allow direct index accesses, for which purpose additional index mapping metadata must be maintained.

One of our key observations is that indirect accesses and cache misses on compressed $x$ and $y$ vectors incur substantial performance penalties during SpMV execution. Motivated by this, we present *Triply Compressed Sparse Column* (*TCSC*), a new technique for co-compressing the sparse matrix together with the sparse input and output vectors in a tightly-coupled fashion. TCSC implicitly encodes the sparsity of the vectors within the compressed sparse matrix data structure itself, while being more space efficient overall compared to the popular and asymptotically efficient Doubly Compressed Sparse Column (DCSC) format [21], [23]. As such, not only does TCSC enables direct index accesses on compressed $x$ and $y$ vectors,

it also does so without requiring any additional bitvectors or index mapping metadata. With this in mind, we carefully co-design the SpMV algorithm to take full advantage of TCSC's features and refer to this optimized operation as *Sparse Matrix - Sparse input and output Vectors* (*SpMSpV*$^2$). In short, TCSC working in tandem with SpMSpV$^2$ results in faster execution times, fewer cache misses, and efficient space utilization.

We show that TCSC provides promising performance and space efficiency on a single machine. Nonetheless, for handling truly big graphs, TCSC needs to scale out to a distributed setting. To this end, we introduce *GraphTap*, a new distributed graph analytics system built around TCSC, which taps into the sparsity of the matrix and the input and output vectors offering thereby fast executions of SpMSpV$^2$ kernels. Alongside efficient computation, TCSC allows GraphTap to reduce communication in a distributed setting via precluding the need for exchanging index mapping metadata across machines. As a result, GraphTap scales better in terms of both data and cluster sizes when compared against the state-of-the-art distributed linear algebra-based graph analytics systems, namely, GraphPad [18] and LA3 [19]. In summary, we demonstrate that GraphTap is up to $4\times$ faster than GraphPad and $7\times$ faster than LA3 using a range of standard graph analytics algorithms on top of real-world and synthetic datasets.

The rest of the paper is organized as follows. In Section II, we provide some background on related work. Section III describes the motivation of the current work. In Section IV, we elaborate on TCSC. GraphTap is introduced in Section V. In Section VI, we report our experimental results and we conclude in Section VII.

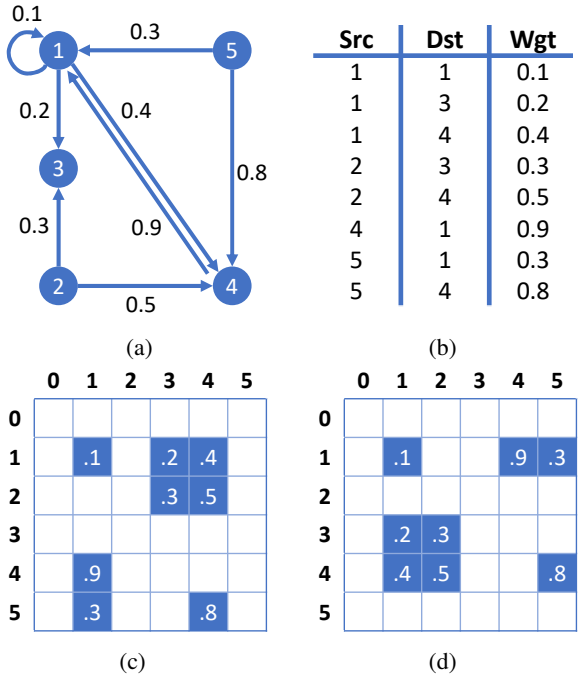
## II. Background

### A. Linear Algebra-based Distributed Graph Analytics Systems

As pointed out in Section I, graph algorithms can be translated into iterative linear algebra primitives (e.g., SpMV operations). For instance, Pegasus [11], which is implemented atop Hadoop, is one of the first distributed graph mining systems that supports SpMV operations. Alongside, CombBLAS [13] is an edge-centric distributed graph analytics system that offers a rich set of primitives via its API including SpMV and Sparse Matrix-Matrix (SpMM) operations, among others. To represent sparse matrices, CombBLAS uses DCSC [21].

GraphMat [17] is a multi-core graph analytics system, which fills the gap between the performance and productivity of graph analytics platforms. It abstracts a vertex program through a generalized iterative SpMV operation. It uses DCSC for representing sparse matrices and utilizes lists of (index, value) pairs for representing sparse vectors. GraphPad [18] (distributed GraphMat) uses OpenMP for scaling up (intra-node scalability) and MPI for scaling out (inter-node scalability). For this sake, it adopts a 2D tiling strategy [24], [25] to distribute the adjacency matrix of a graph among machines.

Akin to GraphPad [18], LA3 [19] is a distributed linear algebra-based graph analytics system, which partitions the adjacency matrix of an input graph into a 2D grid of tiles and stores each tile in a DCSC data structure. LA3 incorporates




| Src | Dst | Wgt |
|---|---|---|
| 1 | 1 | 0.1 |
| 1 | 3 | 0.2 |
| 1 | 4 | 0.4 |
| 2 | 3 | 0.3 |
| 2 | 4 | 0.5 |
| 4 | 1 | 0.9 |
| 5 | 1 | 0.3 |
| 5 | 4 | 0.8 |

Fig. 1: (a) The input graph with 6 vertices and 8 edges. (b) The edge list of the input graph. Each entry is an edge from the source endpoint (Src) to the destination endpoint (Dst) with a weight (Wgt). (c) The adjacency matrix of the graph. (d) The transpose of the adjacency matrix denoted by $A$.

three optimizations, namely: 1) computation filtering, which excludes subsets of trivial vertices out of the main loop of a running graph application, 2) communication filtering, which ensures that each vertex receives only the information that are necessary for accurate calculations within the graph application, and 3) pseudo-asynchronous computation and communication, which overlaps communication and computation to expedite performance.

### B. Column Compressed Sparse Formats

Graphs are highly sparse structures. Many linear-algebra based graph processing systems use CSC or DCSC to store the adjacency matrix of a graph since they are both space efficient and fast to traverse [13], [17]–[19][1]. We next delve deeper into CSC and DCSC to set the stage for our proposed TCSC. We use a running example of an adjacency matrix from Figure 1 to explain the CSC and DCSC formats.

*1) CSC Format:* Figure 2 shows the CSC representation of matrix $A$ from Figure 1. In CSC, $JA$ is an array of column pointers, $IA$ is an array of row numbers, and $VA$ is an array that contains the nonzero values (or weights) in $A$. As such, $|JA| = n + 1$, $|IA| = nnz$, and $|VA| = nnz$, where $n$ is the number of vertices and $nnz$ is the number of edges. The space requirement of CSC (without considering the space required for storing vectors) is $n + 2\ nnz + 1$.

[1]This is especially the case if the goal is to access the nonzero elements of the matrix in column order. If, however, the goal is to access these nonzero elements in row order, similar formats, namely, Compressed Sparse Row (CSR) or Doubly Compressed Sparse Row (DCSR) [23] are typically used.

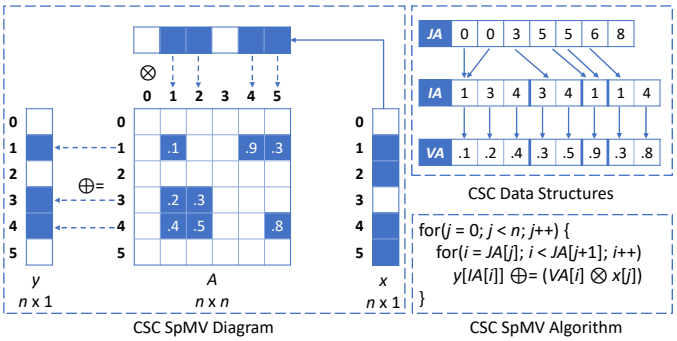



Fig. 2: CSC format for Figure 1d.

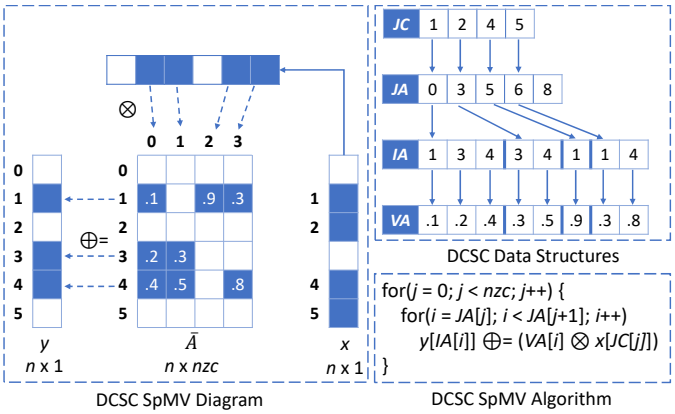



Fig. 3: DCSC format for Figure 1d.

The SpMV operation $y = A \oplus\cdot\otimes x$ is a widely used linear algebra operation. In this operation, $A$ is highly sparse, and $x$ and $y$ vectors are uncompressed. For many applications, this operation is repeated multiple times with changes in input vector $x$. Although CSC is a common way of compressing $A$, it fundamentally lacks direct indexing of sparse input and output vectors. Figure 2 shows how an SpMV kernel runs on a CSC data structure. From this figure, the row and column indices retrieved by CSC essentially belong to the original number of rows and columns, $n$. With the presence of compressed vectors, CSC requires mappings from uncompressed to compressed vectors for converting $JA$ and $IA$ indices.

*2) DCSC Format:* DCSC [21] is an extension of CSC, whereby it further compresses matrix $A$ by removing the zero (empty) columns avoiding thereby repeated elements in array $JA$. Since zero columns are removed, a level of indirection is required to index the retained nonzero columns. To this end, DCSC introduces an array for column indices, $JC$, which provides constant time access to nonzero columns (see Figure 3). In DCSC, $|JC| = nzc$, $|JA| = nzc + 1$, $|IA| = nnz$, and $|VA| = nnz$, where $nzc$ is the number of nonzero columns. Subsequently, the space requirement of DCSC is $2\ nzc + \ 2nnz + 1$, without considering the space needed for storing vectors.

CSC can scale poorly if the number of zero columns grows significantly [7]. DCSC tackles this problem by converting $A$ to $\bar{A}$, which does not contain zero columns. Figure 3 shows how SpMV operations are executed on top of DCSC, wherein $\bar{A}$ is multiplied by an uncompressed input vector $x$ and the results are stored in an uncompressed output vector $y$. Note that Sparse Matrix - Sparse Vector (SpMSpV) operations can also be ran on top of DCSC, with $x$ being compressed (which can be represented by (index, value) pairs) and $y$ being uncompressed or dense [22]. Although compressed input $x$ can be indexed through an uncompressed output $y$ using $JC$, most implementations do not exploit such an option [17], [18] in order to use the output of one SpMSpV directly as an input to the next SpMSpV operation [22].

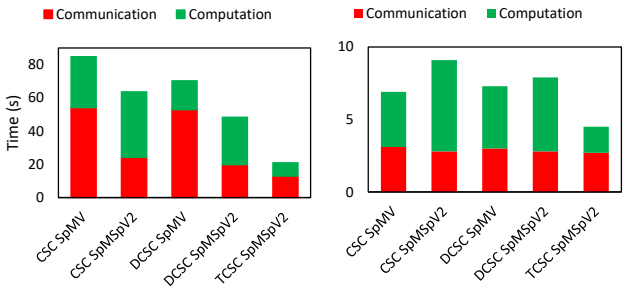
## III. Motivation

The standard CSC and DCSC runs SpMV kernels without any changes. CSC SpMV does not need any indirection to access the uncompressed input and output vectors, whereas DCSC SpMV requires one indirection because it compresses the $JA$. Luckily, in DCSC if there are enough zero columns to remove, the cost of this indirection would not hurt the runtime.

In a distributed setting where the elements of input and output vectors are transported over the network, vector sizes become highly important because they are acting as a proxy for communication. The communication volume can be reduced by compressing the input/output vectors through removing the zero columns/rows and then adding indirection to the CSC and DCSC formats to support SpMSpV$^2$ kernels on the compressed vectors. To index the compressed vectors, CSC SpMSpV$^2$ requires two indirections (both rows and columns) and DCSC SpMSpV$^2$ requires only one (given it has already supported compressed column, hence it only needs one indirection for indexing rows).

To demonstrate the tradeoff between communication reduction and runtime increase due to indirection, we profile the execution of 20 iterations of PageRank (PR) on two large graphs, Twitter and Rmat29 (see Table II for details), running on our GraphTap distributed platform using CSC and DCSC. As shown in Figure 4a, CSC/DCSC SpMV have roughly identical amounts of communication, whereas the computation time of DCSC SpMV is more than CSC SpMV. This is due to the DCSC SpMV's extra level of indirection. For a relatively less sparse graph like Twitter which only has a small number of empty columns, this indirection turns out to cause a computation penalty. Yet, this is not the case for a sparser graph like Rmat29 (Figure 4b), where DCSC SpMV's indirection contributes to a better runtime compared to CSC SpMV. Last, SpMV compressions are spending approximately half and three-quarter of their runtime for sending/receiving vectors, where a good portion of them are zeros.

Figure 4a shows that for Twitter graph, compressing vectors does not help CSC/DCSC SpMSpV$^2$ to achieve a better communication time because vectors are relatively dense. Whereas, for Rmat29 (Figure 4b) the communication time is cut in half compared to SpMV because there is a good number of zero columns/rows to remove. Finally, the computation time of SpMSpV$^2$ increases significantly in both Twitter and Rmat29 graphs because of the extra levels of indirections added to support SpMspV$^2$ kernels.




(a) Twitter (1.46 B edges) (b) Rmat29 (8.58 B edges)

Fig. 4: Comparison of SpMV with SpMSpV$^2$ using PR

Hence, there is a trade-off between SpMV and SpMSpV$^2$. As communication time goes down in SpMSpV$^2$ due to compressing the vectors, the computation time goes up due to adding the levels of indirections (note that this trade-off is beneficial when sparsity is large and detrimental when sparsity is small). Hence, it would be desirable to compress the vectors while adding no indirection to the SpMSpV$^2$ kernel, which is the rationale behind TCSC. This desirable feature is shown in the last columns of Figure 4 that shows using TCSC with the SpMSpV$^2$ kernel always decreases the computation time while either decreasing (in Rmat29) or not increasing (in Twitter) the communication time.

## IV. Triply Compressed Sparse Format

In this paper we propose a simple, yet highly efficient, co-compression technique called Triply Compressed Sparse Column (TCSC) (or Triply Compressed Sparse Row (TCSR) for row compressed data). By removing nonzero columns and rows of a sparse matrix, TCSC does not only store the sparse matrix in an efficient and cost-effective way, but further extends that to input and output sparse vectors. TCSC supports SpMSpV$^2$ operations on sparse matrix and vectors without requiring any indirection to access compressed vectors.

### A. Triply Compressed Sparse Column (TCSC)

DCSC compresses matrix $A$ by removing only its zero columns while retaining its zero and nonzero rows. TCSC capitalizes on DCSC's compression strategy via removing $A$'s zero rows as well. Like array $JC$ for indexing nonzero columns, TCSC introduces array $IR$, the row indices array for indexing nonzero rows, where $|IR| = nzr$. As illustrated in Figure 5, TCSC utilizes $IR$ to populate $IA$ with values within the range of nonzero rows. This eliminates the problem of row indexing upon executing SpMSpV$^2$ operations. Figure 5 shows how an SpMSpV$^2$ kernel can run on top of TCSC with fully compressed matrix $\bar{\bar{A}}$ and fully compressed input and output vectors $\bar{x}$ and $\bar{y}$, without requiring any additional support from a bitvector or a list of (index, value) pairs. More precisely, by using $JC$ and $IR$ together, TCSC provides direct accesses to $\bar{x}$ and $\bar{y}$. Lastly, the space requirement of TCSC is $2\ nzc + nzr + 2\ nnz + 1$.

TCSC consolidates the sparsity of matrix and vectors in a co-designed data structure to enable efficient executions of SpMSpV$^2$ operations. CSC and DCSC can also be used to run SpMSpV$^2$. However, to support SpMSpV$^2$, CSC requires two levels of indirections for indexing compressed input and output vectors, while DCSC requires only one indirection for indexing the compressed output vector.

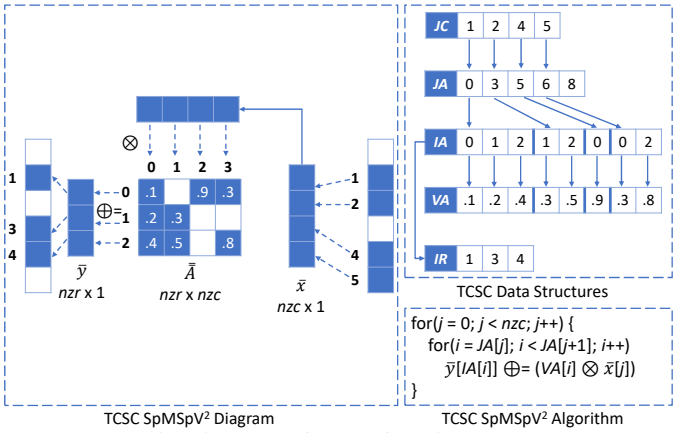



Fig. 5: TCSC format for Figure 1d.

### B. Comparison of Space Requirements

Table I shows a comparison between different compression techniques. For SpMSpV$^2$ operations, CSC requires using data structures like two lists of (index, value) pairs for input and output vectors. In addition, it needs to store metadata for column and row indirections. Therefore, its space requirement evaluates to $3\ n + 3\ nzc + 3\ nzr + 2\ nnz + 1$. DCSC requires maintaining information on input and output vectors and metadata for row indirection. Thus, its space requirement for SpMSpV$^2$ operations is $n + 3\ nzc + 3\ nzr + 2\ nnz + 1$. TCSC total space requirement is $3\ nzc + 2\ nzr + 2\ nnz + 1$.

In comparing space requirements for SpMSpV$^2$ operations, TCSC demands the least space due to uniquely addressing the sparsity of vectors in conjunction with the sparsity of the matrix. It can be proved that under certain conditions TCSC can save space when at least $40\%$ of rows/columns of the matrix are empty compared to CSC and DCSC with SpMV (see Figure 6; more on this shortly). Alongside space savings, TCSC provides faster SpMSpV$^2$ operations because: 1) it averts two levels of indirections compared to CSC and one level of indirection compared to DCSC, 2) it requires sending/receiving only values of compressed vectors (especially in distributed settings) without exchanging any metadata since it retains internally the nonzero indices, 3) it results in smaller vectors, which can potentially fit in cache, and 4) it exhibits sequential access patterns on the input vector (like DCSC), thus exploiting more cache locality (as compared to CSC).

Given the information reported in Table I, we can derive relaxed space formulas for all the compression schemes by ignoring the $IA$ and $VA$ arrays and the plus one in $JA$ array, which are equivalent across all the schemes. Thus, we can eliminate the term $2\ nnz + 1$. Furthermore, we assume $nzc \approx nzr \approx nz$ and thus $nz = n - z$, where $nz$ is the number of nonzero elements and $z$ is the number of zeros agnostic to rows and columns. Finally, by removing $2\ nnz + 1$ and, subsequently, substituting $nzc$ and $nzr$ with $n - z$, we obtain the following approximate space formulas:

TABLE I: Space required for storing matrix, vector, and column/row indirections of different compression schemes.

| | Arr | CSC SpMV | DCSC SpMV | CSC SpMSpV$^2$ | DCSC SpMSpV$^2$ | TCSC SpMSpV$^2$ |
|---|---|---|---|---|---|---|
| Mat | $JC$ | | $nzc$ | | $nzc$ | $nzc$ |
| | $JA$ | $n+1$ | $nzc+1$ | $n+1$ | $nzc+1$ | $nzc+1$ |
| | $IA$ | $nnz$ | $nnz$ | $nnz$ | $nnz$ | $nnz$ |
| | $VA$ | $nnz$ | $nnz$ | $nnz$ | $nnz$ | $nnz$ |
| | $IR$ | | | | | $nzr$ |
| Vec | $x/\bar{x}$ | $n$ | $n$ | $2\ nzc$ | $nzc$ | $nzc$ |
| | $y/\bar{y}$ | $n$ | $n$ | $2\ nzr$ | $2\ nzr$ | $nzr$ |
| Ind | $c$ | | $nzc \rightarrow n$ | $nzc \rightarrow n$ | | |
| | $r$ | | | $nzr \rightarrow n$ | $nzr \rightarrow n$ | |

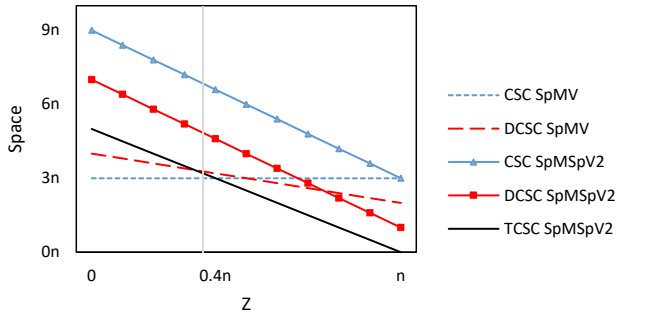



Fig. 6: Space of different compressions using (1).

$$\begin{aligned}
&\text{CSC SpMV} \rightarrow 3\ n \\
&\text{DCSC SpMV} \rightarrow 2\ n + 2\ (n - z) = 4\ n - 2\ z \\
&\text{CSC SpMSpV}^2 \rightarrow 3\ n + 3\ (n - z) + 3\ (n - z) = 9\ n - 6\ z \\
&\text{DCSC SpMSpV}^2 \rightarrow n + 3\ (n - z) + 3\ (n - z) = 7\ n - 6\ z \\
&\text{TCSC SpMSpV}^2 \rightarrow 3\ (n - z) + 2\ (n - z) = 5\ n - 5\ z
\end{aligned} \tag{1}$$

By varying the value of $z$ in equations (1) over the range $[0, n]$, the space of each compression can be computed in terms of $n$. As demonstrated in Figure 6, from $z = 0.4\ n$ onward (marked by the vertical gray line), TCSC will require less space as opposed to other schemes. In Section VI, we present experimental results that corroborate this observation.

### C. Translating Graph Algorithms onto SpMSpV$^2$ Operations

Leveraging the duality between graphs and matrices, many graph theory operations can be mapped onto certain linear algebra primitives on matrices [13]. As a simple primitive, SpMSpV$^2$ primitive, $\bar{y} = \bar{\bar{A}} \oplus.\otimes \bar{x}$ can be formalized as:

- $\bar{\bar{A}}$ is the $nzr \times nzc$ sparse matrix with $nnz$ entries (edges), where $nzr$ and $nzc$ are the number of nonzero rows and columns, respectively.
- $\bar{x}$ is the $nzc \times 1$ sparse input vector with $nzc$ entries (columns), which is multiplied in $\bar{\bar{A}}$ using the multiplication and addition operators.
- $\bar{y}$ is the $nzr \times 1$ sparse output vector with $nzr$ entries (rows), which stores the results of multiplying $\bar{\bar{A}}$ and $\bar{x}$.
- $\oplus.\otimes$ is a semiring equipped with $(+, \times)$ operators.

SpMSpV$^2$ requires a way of encoding the sparsity for both $\bar{x}$ and $\bar{y}$ vectors. Previous works have used bitvectors [17], [18] or lists of (index, value) pairs [19] to encode this information. In contrast, TCSC coalesces this information in the compressed sparse matrix format and assumes that sparse input and output vectors are of sizes $nzc$ and $nzr$, respectively.

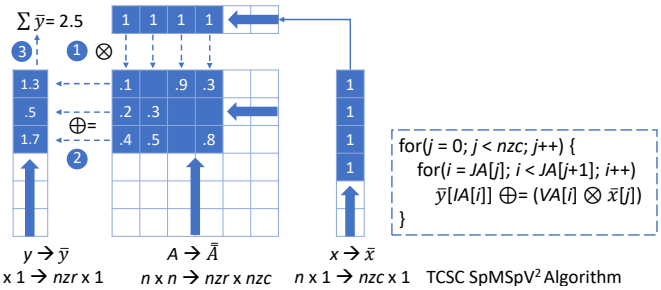



Fig. 7: Calculating weighted outgoing degree of Figure 1d.

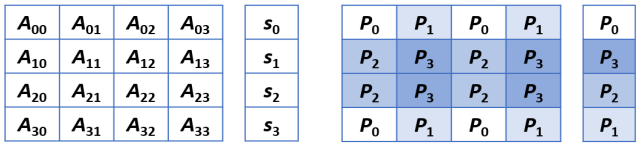



Fig. 8: (a) Matrix/vector partitioned to $p^2$ tiles / $p$ segments ($p = 4$); (b) Tiles/segments assigned to $p$ processes ($p = 4$).

To exemplify, we consider *weighted degree calculation* which calculates the outgoing degree of a graph ponderated by the weight of each edge. This problem can be solved via multiplying the outgoing edges of each vertex by one and summing up the results. Using SpMSpV$^2$ operations, first $\bar{x}$ is initialized with ones. Second, the weighted outgoing degree of each vertex is calculated by multiplying each entry of $\bar{x}$ to its corresponding column of $\bar{\bar{A}}$. Third, the result of each row is stored in the respective entry of $\bar{y}$, which will eventually hold the weighted outgoing degrees of all vertices.

## V. GraphTap: Distributed Graph Analytics using Triply Compressed Sparse Format

In this section, we introduce GraphTap, a new distributed system for scalable graph analytics that features a TCSC-based SpMSpV$^2$ system mated with a vertex-centric programming interface. As such, GraphTap can execute any user-defined vertex program on any input graph. This is done in two steps. First, GraphTap loads and partitions the input graph into TCSC tiles distributed across multiple processes. Next, it executes the user's vertex program in an iterative fashion via its distributed SpMSpV$^2$ core. The followings describe these steps in details.

### A. Matrix Partitioning

GraphTap can read graphs given in an edge-list format. It loads edges into an adjacency matrix representation that is partitioned in two dimensions and distributed for scalability [18], [19], [24], [25]. To elaborate, given $p$ processes, GraphTap partitions the matrix into $p^2$ *tiles* and any associated vector into $p$ *segments*, as exemplified in Figure 8a.

GraphTap assigns tiles and segments to processes while accounting for both load balancing and locality [19]. As Figure 8b shows, each process is assigned $p$ tiles and one of $p$ vector segments. In particular, the process owning the $i^{th}$ diagonal tile, $A_{ii}$, also owns the $i^{th}$ vector segment, $s_i$. We call this process the *leader* of the $i^{th}$ *row group* (i.e., the set of processes that own tiles in the $i^{th}$ row) and *column group* (i.e.,

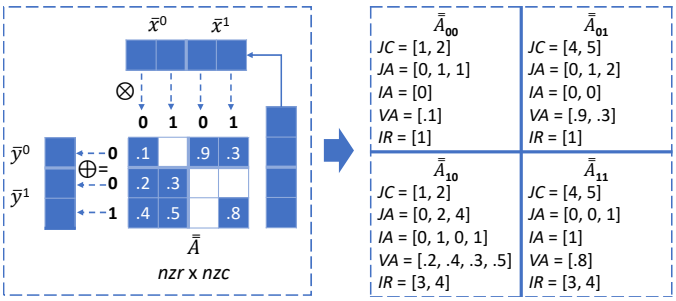



Fig. 9: Figure 1d matrix partitioned into four TCSC tiles.

the set of processes that own tiles in the $i^{th}$ column). During distributed SpMSpV$^2$ execution, each leader communicates with its row and column group *followers* (members) via MPI. For example, in Figure 8b, process $P_0$ owns tiles $A_{00}$, $A_{02}$, $A_{30}$, and $A_{32}$. Also, $P_0$ is the leader of the first row and column groups; thus, $P_1$ is $P_0$'s follower in the first row group and $P_2$ is $P_0$'s follower in the first column group.

GraphTap stores each tile using the TCSC format. The compressed height of any given tile, $\bar{\bar{A}}_{ij}$, is the number of nonzero rows across the entire $i^{th}$ row of tiles. Similarly, the compressed width of any given tile, $\bar{\bar{A}}_{ij}$, is the number of nonzero columns across the entire $j^{th}$ column of tiles. Moreover, in order to eliminate indirections during SpMSpV$^2$, the compressed sizes of the $i^{th}$ input (or output) vector segments are equal to the compressed width (or height) of the $i^{th}$ column (or row) of tiles. For example, in Figure 9, tiles $\bar{\bar{A}}_{00}$ and $\bar{\bar{A}}_{10}$ both have a compressed width of two, as does input segment $\bar{x}^0$. Similarly, tiles $\bar{\bar{A}}_{00}$ and $\bar{\bar{A}}_{01}$ both have a compressed height of one, as does output segment $\bar{y}^0$.

### B. Vertex Program Execution

Similar to other recent graph analytics systems [17]–[19], GraphTap translates a user-defined vertex-centric program into iteratively-executed SpMSpV$^2$ operations. Like these systems, GraphTap applies a variant of the *Gather*, *Apply*, *Scatter* (GAS) model [5]. To be more precise, GraphTap involves three method calls per SpMSpV$^2$ iteration: *Scatter-Gather*, *Combine*, and *Apply*, which we shall elaborate upon shortly.

In order to map a vertex-centric program to its SpMSpV$^2$ system, GraphTap maintains – in addition to the compressed input and output vectors, $\bar{x}$ and $\bar{y}$ – a *state* vector, $v$, which stores vertex states. The state vector is not compressed, and its size equals the number of vertices, because we assume that all vertices have states, even if some states may remain unchanged. The state vector is partitioned into $p$ segments, each assigned to its corresponding leader process. Thus, each process initializes the states of its own vertices. Thereafter, GraphTap launches its iterative SpMSpV$^2$ execution. Per iteration, each process calls the following methods.

*1) Scatter-Gather:* To begin an iteration, each $i^{th}$ leader, in parallel, prepares its new $\bar{x}^i$ and *scatters* it to its column group followers. $\bar{x}^i$ is essentially an interpolation of the old state, $v^i$ (i.e., resulting from the previous iteration).

Consequently, TCSC offers the following advantages during Scatter-Gather: 1) Since $|\bar{x}| = nzc < |x| = n$, less communication is required (per column group). 2) Given that TCSC already incorporates the sparsity information inside its data structures, there is no need to send the indices of the nonzero elements. Therefore, the communication volume is only limited to sending the values themselves, which is less compared to sending a list of (index, value) pairs in [18], [19]. 3) When calculating the new $\bar{x}$ from $v$, TCSC's $JC$ array efficiently enables direct indexing on both $\bar{x}$ and $v$ (i.e., without requiring any extra levels of indirection).

*2) Combine:* After the scattered $\bar{x}$ is gathered at all processes, each process starts processing the tiles that it owns in a row-wise fashion. For each tile, $T_{ij}$, in the $i^{th}$ row, the SpMSpV$^2$ operation is called on its TCSC value array, $VA$, and the $\bar{x}^j$ belonging to the $j^{th}$ column group. The result is *combined* (accumulated) locally in $\bar{y}^i$, which is indexed directly using the $IA$ array. After processing all its tiles belonging to the $i^{th}$ row, each follower sends its $\bar{y}^i$ to its row group leader, which combines it into its own $\bar{y}^i$. Given GraphTap uses asynchronous communication, leaders/followers post their receives/sends and move on to their next row of tiles.

Thus, TCSC offers the following advantages during Combine: 1) No indirections are needed while running SpMSpV$^2$ operations on $\bar{x}$, $VA$, or $\bar{y}$ (for storing the results). This is because, $\forall\ T_{ij}$, $|\bar{x}^j| = |JA|$ and $|\bar{y}^i| = |IA|$. 2) Since $|\bar{y}| = nzr < |y| = n$, less communication is required (per row group). 3) When followers send $\bar{y}^i$s to their leader, only the actual values are sent without their indices, further reducing communication volume. 4) Asynchronous communication allows GraphTap to overlap communication with computation.

*3) Apply:* To complete an iteration, each $i^{th}$ leader, in parallel, waits until its $\bar{y}^i$ is finalized, and then uses it to update its $v^i$ (to be used in the next iteration). Although $|\bar{y}| = nzr \neq |v| = n$, TCSC's $IR$ array circumvents an undesired indirection when computing $v$ from $\bar{y}$ since it contains the original row ids of the nonzero indices of $v$.

GraphTap continues iterating until $v$ converges or a specified maximum number of iterations is reached.

*4) Activity Filtering and Computation Filtering:* Graph applications may be classified as *stationary* or *non-stationary* [18], [19]. In a stationary application, all vertices remain active over all iterations. In a non-stationary application, only a subset of vertices is active during each iteration and this subset can change dynamically. GraphTap skips the communication and computation of inactive vertices in non-stationary applications. We implement *activity filtering* by communicating (index, value) pairs of active vertices only.

For directed graphs, it is possible to make the SpMV more efficient via *computation filtering* [19]. This firstly requires classifying vertices into regular vertices (have both incoming and outgoing edges), source vertices (have only outgoing edges), sink vertices (have only incoming edges), and isolated vertices (have no edges). Subsequently, processing only regular and source vertices in the first iteration, only regular vertices in the middle iterations, and only regular and sink vertices in the final iteration. We implemented computation filtering for stationary applications on directed graphs only.

## VI. Results

Experiments are conducted in two settings: single node processing and distributed processing, both written in C/C++. The single node implementation is a single thread PageRank application which basically compares CSC, DCSC, and TCSC SpMSpV[2]. The distributed implementation uses GraphTap[2], the proposed distributed graph analytics system which utilizes TCSC as its default compression technique and MPI for both inter and intra-node communication. GraphTap's experiments include both weak scaling comparison where graph size is scaled alongside the cluster size, and strong scaling where graph size is fixed, and the cluster size is varied.

### A. Experimental Setup

*1) Hardware and Software Configurations:* We ran experiments on a cluster of machines that uses Slurm workload manager for batch job queuing [26]. We used Intel MPI [27] to compile our program on the cluster. Moreover, for single node experiments, we used a machine with 12-core Xeon processor (@ 3.40 GHz speed) and 512 GB RAM. For the distributed experiments, we used a sub-cluster of 32 machines each with 28-core Broadwell Processor (@ 2.60 GHz speed), 192 GB RAM, and Intel Omni-path network (10 Gbps transfer speed). At our largest scale, we utilized all these 32 machines and launched 16 processes (cores) per machine without over subscription of cores (512 cores in total). Finally, any data point reported here is the average of multiple individual runs.

*2) Counterpart Systems:* GraphTap has been tested against GraphPad [18], a linear algebra-based system developed by Intel, and LA3 [19], a linear algebra based system with sophisticated communication and computation optimizations. After a careful assessment, we noticed that GraphPad works best when launched with two threads per MPI process and LA3 with one thread per MPI process (without multithreading). Furthermore, we allocate 16 cores per machine and thus GraphPad is launched with 8 processes and two threads (cores) per process, and LA3 and GraphTap are launched with 16 processes (cores) per machine.

*3) Graph Datasets:* Table II shows the collection of six real-world graphs and five synthesized graphs alongside their characteristics and the number of processes allocated to process them. This collection includes multiple web crawls and social network from LAW [28], and RMAT 26 - 30 graphs from the Graph 500 challenge [29].

*4) Graph Applications:* To evaluate TCSC, we implemented two types of graph applications: 1) stationary applications including Degree, and PageRank (PR) on unweighted directed graphs, and 2) non-stationary applications including Single Source Shortest Path (SSSP) on weighted directed graphs, and Breadth First Search (BFS) and Connected Component (CC) on unweighted undirected graphs. Note that similar to the setting used in [18], [19], we ran PR for 20 iterations and SSSP, BFS, and CC until convergence and report the average execution.

[2]GraphTap source code is online at https://github.com/hmofrad/GraphTap

TABLE II: Datasets used for experiments. $Z_c$ and $Z_r$ are the percentage of zero columns and rows. T is the type (including web crawl, social network and synthetic graphs). $N$ is the number of machines used to process the graph.

| Graph | $\|V\|$ | $\|E\|$ | $Z_c$ | $Z_r$ | T | $N$ |
|---|---|---|---|---|---|---|
| UK'05 (UK5) [28] | 39.4 M | 0.93 B | 0 | 0.12 | Web | 4 |
| IT'04 (IT4) [28] | 41.2 M | 1.15 B | 0 | 0.13 | Web | 4 |
| Twitter (TWT) [28] | 41.6 M | 1.46 B | 0.09 | 0.14 | Soc | 8 |
| GSH'15 (G15) [28] | 68.6 M | 1.80 B | 0 | 0.19 | Web | 8 |
| UK'06 (UK6) [28] | 80.6 M | 2.48 B | 0.01 | 0.14 | Web | 16 |
| UK Union (UKU) [28] | 133 M | 5.50 B | 0.05 | 0.09 | Web | 24 |
| Rmat26 (R26) [29] | 67.1 M | 1.07 B | 0.55 | 0.72 | Syn | 4 |
| Rmat27 (R27) [29] | 134 M | 2.14 B | 0.57 | 0.73 | Syn | 8 |
| Rmat28 (R28) [29] | 268 M | 4.29 B | 0.59 | 0.74 | Syn | 16 |
| Rmat29 (R29) [29] | 536 M | 8.58 B | 0.61 | 0.75 | Syn | 24 |
| Rmat30 (R30) [29] | 1.07 B | 17.1 B | 0.62 | 0.76 | Syn | 32 |

### B. Single Node Results

To experimentally measure the performance of TCSC, we implemented a single thread PageRank application and reported its space, number of L1 cache misses, and speedup in Figure 10. We choose PageRank as it is a compute-intensive application and our focus in this section is more on identifying the computational characteristics of TCSC.

*1) Space Utilization:* Figure 10a shows the space utilization measured for different compressions. Similar to the TCSC space analysis (Section IV-B), we only report the space required for vectors and indirections for this comparison as the amount of storage required for storing the graph edges is the same across all compressions (see Table I).

From Figure 10a, we note that CSC and DCSC have approximately similar space utilization and TCSC has the least space requirement in both real-world and synthetic graphs. Compared to CSC, on average TCSC requires 45% and 70% less space in real-world and synthetic graphs. Also, compared to DCSC, on average TCSC requires 15% and 25% less space in real-world and synthetic graphs. This space efficiency roots in the indexing algorithm of TCSC where it stores the sparsity of vectors while constructing the compressed matrix data structure by renumbering its column and row indices and removing zero (empty) columns and rows. This successfully allows TCSC to trivially expand or compress the input and output vectors and at the same time consumes the least space.

*2) Cache Analysis:* We used CPU performance counters to collect data on L1 cache misses. Figure 10b shows the number of cache misses of different compressions. Comparing CSC and DCSC with TCSC, on average TCSC has 20% to 40% less cache misses across all real-world and synthetic graphs. TCSC is a cache friendly compression inasmuch as it can access the compressed input and output vectors without requiring any level of indirection while avoid trashing the L1 cache. TCSC sequentially indexes the input vector. This avoids unnecessary cache invalidations of the input vector and provides more cache locality. Moreover, TCSC can access the output vector with no level of indirection compared to CSC and DCSC, providing faster access to output vector entries. Last, given the compressed input and output vectors are essentially smaller than the original SpMV vectors, they can possibly fit in L2 cache which further yields better cache utilization.

*3) Time Analysis:* Figure 10c compares the speedup for different compressions. From this figure, compared to CSC and DCSC, TCSC is up to $2.2\times$ and $11\times$ faster in real-world and synthetic graphs, respectively. We believe this performance gain is mainly due to the direct indexing algorithm of TCSC which offers a better cache locality. CSC and DCSC underperform compared to TCSC because they suffer from access indirections and poor cache locality.

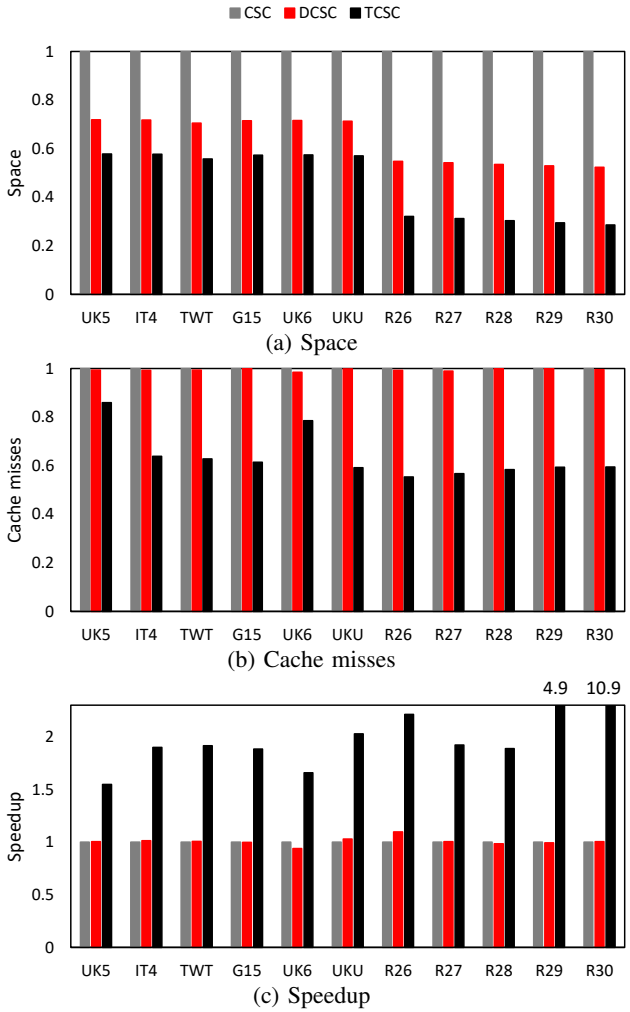
In Figure 10c, DCSC is slightly faster than CSC on average because it collapses the nonzero columns and skips the computation for nonzero columns. Furthermore, TCSC is faster than both CSC and DCSC because it additionally collapses the nonzero rows which further reduces the chances of L2 cache and memory thrashing. Moving to larger scales synthetic graphs such as RMAT30, the cache thrashing effect becomes more prominent and TCSC is $11\times$ faster than CSC and DCSC.

There are two levels of indirection while running the SpMSpV$^2$ kernel: 1) indirection used for the input vector while accessing column data using pairs of (index, value), and 2)indirection used for sparse output vector while writing the result of executing the operation. Although CSC and DCSC are adapted to work with sparse vectors, CSC requires both levels of indirections and DCSC requires the latter one. TCSC, on the other hand does not need these levels of indirections because for the former one, like DCSC the number of columns in the sparse matrix are aligned with the size of input vector. For the latter indirection, since TCSC's row indices array is populated using values derived from the number of nonzero rows, the row indices stored in the compressed matrix are essentially able to directly index the output vector.

### C. Distributed Processing Results

In this section we discuss experimental results of GraphTap. In the first and second experiments we compare different compression techniques implemented inside GraphTap and in the third experiment, GraphTap is compared with GraphPad [18] and LA3 [19], two state-of-the-art linear algebra-based graph analytics systems. The graphs and cluster sizes used for these experiments are reported in Table II.

*1) Speedup Comparison of CSC, DCSC, and TCSC in GraphTap:* We implemented CSC, DCSC, and TCSC SpMSpV$^2$ in GraphTap and benchmarked them using PR (a stationary application). As shown in Figure 11, on real-world and synthetic graphs, CSC and DCSC perform comparatively with DCSC performing slightly better. Also, TCSC performs the best compared to CSC and DCSC with up $3.5\times$ and $5.7\times$ speedup, respectively. From the results, CSC and DCSC are not scaling well compared to TCSC as while solving PR they become slower as dataset size increases (especially in synthetics). TCSC on the other hand is scalable because as dataset size increases, the runtime also improves in both real-world and synthetic graphs. This is because TCSC not only compresses vectors leading to less communication, but also has a better indexing algorithm, leading to more efficient computation.



(a) Space
(b) Cache misses
(c) Speedup

Fig. 10: Normalized space, speedup, and cache misses of compressions on a single node for PR with CSC as baseline.

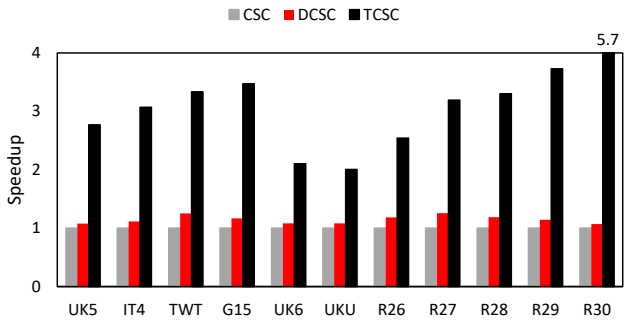
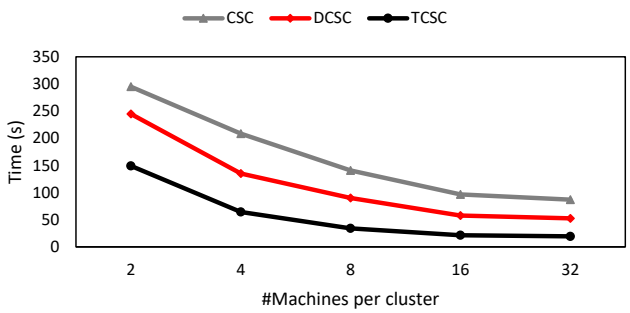
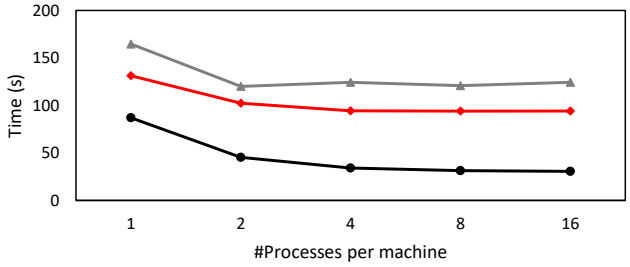



Fig. 11: Normalized speedup of compressions on GraphTap for PR with CSC as baseline

*2) Scalability Comparison of CSC, DCSC, and TCSC in GraphTap:* Figure 12a shows the results of *cluster scalability test*. In this experiment, we keep the number of processes per machine to 16 but change the number of machines from 2 to 32 and run PR on R29. Here, TCSC improves the runtime as we add more machines (or processes) to solve the problem because TCSC's communication volume is smaller compared to CSC and DCSC. Thus, increasing the communication volume by having more machines does not hurt its performance.




(a) Cluster scalability test using PR on R29 with 8.58 B edges




(b) Process scalability test using PR on R30 with 17.1 B edges

Fig. 12: Scalability tests for different compressions

Figure 12b shows the *process scalability test* of different compressions. In this experiment, we run PR on R30 using 32 machines while changing the number of processes per machine from 1 to 16. From this figure, TCSC is scalable because it can efficiently harvest the added processes to achieve a better runtime while maintaining a decent gap with other compressions. Moreover, CSC is not scalable because it achieves worse or comparable runtimes with more than 2 processes, whereas TCSC even with 16 processes per machine can still slightly improve the runtime.

### *D. Runtime Comparison of GraphPad, LA3, and GraphTap*

In this experiment, we compare GraphTap with GraphPad and LA3 using PR, SSSP, BFS, and CC applications on selected datasets from Table II. GraphPad [18] uses DCSC for compressing the sparse matrix and bitvectors for representing the sparse vectors. Similarly, LA3 [19] uses DCSC for sparse matrices, but uses lists of (index, value) pairs for representing sparse vectors. On the other hand, GraphTap uses TCSC that compress both matrix and vectors simultaneously and uses lists of (index, value) pairs for representing sparse vectors.

Figure 13a reports the results for PR. Based on this figure, GraphTap is up to $1.5\times$ faster than GraphPad and $7\times$ faster than LA3 in real-world datasets. Also, GraphTap is up to $2\times$ faster than GraphPad and $4\times$ faster than LA3 in synthetic datasets. LA3 uses aggressive communication optimizations that tailor the communication per tile while sending the input vectors. The overhead of this optimization becomes a bottleneck when running on a cluster with a fast communication infrastructure. Specifically, LA3 spends a significant amount of time on constructing these tailored input vectors. GraphTap, on the other hand, tailors input vectors for each column group of tiles so that it can skip the overhead of constructing individual input vectors per receiver process like LA3, while still efficiently utilizing the network bandwidth. Similarly, GraphPad performs better than LA3 because of its efficient communication.

Figure 13b and Figure 13c show the results for the non-stationary applications SSSP and BFS. From these figures, GraphTap is 2–3$\times$ faster than GraphPad and LA3. SSSP runs on weighted directed graphs, it starts from a source vertex and converges when it finds the shortest path to all vertices inside the connected component the source vertex is belonged to. Clearly, executing vertices which are not at the same component with source is unnecessary. Thus, activity filtering removes them from the main loop of computation. Moreover, vertices that have converged already are also factored out of the computation. For non-stationary applications, activity filtering significantly reduces the volume of communication compared to stationary applications like PR. Therefore, having less communication is the reason that LA3 performs better than GraphPad while running BFS on synthetic graphs. Also, GraphTap performs worse than GraphPad in SSSP and BFS on TWT; this is because TWT is among the relatively high-density real-world graphs where there is a small number of zero rows and columns to filter for TCSC.

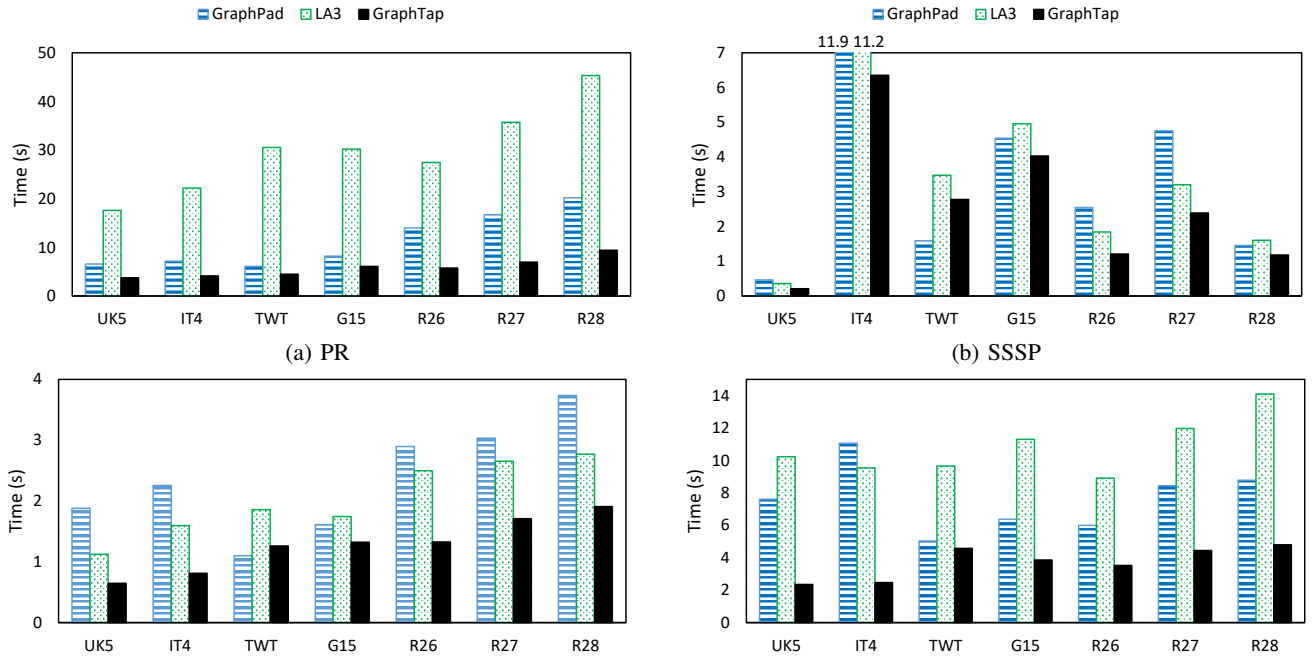
Figure 13d shows the result for CC. GraphTap is $1.2-4.5\times$ faster than GraphPad and $2-4\times$ faster than LA3 in real-world graphs. Also, GraphTap is $2\times$ faster than GraphPad and $3\times$ faster than LA3 in synthetic graphs. From Figure 13d, GraphPad performs better than LA3 because CC deals with significant amount of messaging to identify the connected components and the communication optimizations of LA3 are extremely expensive for such an application. Also, comparing GraphTap's TCSC with DCSC used in GraphPad, DCSC uses a bitvector to locate the nonzero entries of output vectors, whereas TCSC can directly index the output vectors.

Last, in Figure 13 on average GraphTap is $2-4\times$ faster than others on all scales which is due to the proposed TCSC format. Moreover, GraphTap scales better compared to GraphPad and LA3 because while adding more processes for larger graphs, it can efficiently utilize the additional processes with a negligible increase in runtime (this trend is more visible in Rmat synthesized graphs).

### *E. Discussion of Results*

TCSC introduced in this paper has significant space and indexing advantages over CSC and DCSC. Moreover, GraphTap which uses TCSC as its core compression format, outperforms GraphPad and LA3 distributed systems with DCSC compression scheme. The following are a summary of TCSC and GraphTap results:

1) TCSC is more cache friendly than CSC and DCSC. The input and output vectors are intrinsically smaller for TCSC and are accessed directly without indirection. The smaller vector sizes and the locality of access patterns cause fewer cache misses and less cache pollution in TCSC.




(a) PR (b) SSSP

(c) BFS (d) CC

Fig. 13: Runtime of GraphPad, LA3, and GraphTap

2) GraphTap communication volume is less than GraphPad and LA3 because the sizes of its vectors are equal to the number of nonzero columns/rows. There is no need for an auxiliary mechanism to index input and output vectors as they are aligned to the number of nonzero columns/rows. Therefore, input vectors are scattered without any change in their size and partial output vectors are aggregated without requiring any extra indexing metadata.
3) The proposed triple compression can be applied to row major compression resulting in a TCSR scheme. However, we picked TCSC for the same reason CSC and DCSC are preferred over CSR and DCSR. Specifically, in column major compressions, like CSC, DCSC or TCSC, access to the input vector is sequential and infrequent and access to the output vector is random and frequent providing better cache locality for input vectors. This flips for row major compressions like CSR, DCSR and TCSR. In non-stationary applications, given that input vector only carries information about active vertices, a column compression can immediately locate the active columns and runs the SpMV kernel, whereas in row compression, the algorithm first needs to scan all rows and locates the active vertices and then runs the SpMV which significantly requires more effort. Thus, column compressions are expected to, and have been shown to, perform better for graph applications.
4) TCSC is a scalable compression format. We have used it to process big graphs as large as 17.1 B edge on up 32 machines with 16 processes per machine (512 processes in total). From our experiments, by adding more machines per cluster or more processes per machine, TCSC can harvest additional processes efficiently because it compresses empty rows/columns and reduces the problem space.

## VII. Conclusion

This paper presents Triply Compressed Sparse Column (TCSC), a novel compression technique which leads to efficient Sparse Matrix - Sparse input and output Vectors (SpMSpV$^2$) operations. TCSC logically compresses both columns and rows of a sparse matrix and hence integrates the sparsity of input and output vectors within the sparse matrix. In our experiments, we analyzed the performance of TCSC on real-world and synthetic graphs with different sizes and demonstrated that TCSC has less space requirement while offering up to $11\times$ speedup compared to common CSC and DCSC. TCSC is implemented in GraphTap, a new linear algebra-based distributed graph analytics system introduced in this paper. We compared GraphTap with GraphPad and LA3, two state-of-the-art linear algebra-based distributed graph analytics systems on different graph sizes and numbers of machines and cores. We showed that GraphTap is up to $7\times$ faster than these distributed systems due to its efficient sparse matrix compression format, faster SpMSpV$^2$ algorithm, and smaller communication volume.

## VIII. Acknowledgments

This publication was made possible by NPRP grant #7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). This research was supported in part by the University of Pittsburgh Center for Research Computing through the resources provided. Finally, we thank the IEEE Cluster 2019 reviewers for their valuable suggestions and comments.

## REFERENCES


[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] M. Han and K. Daudjee, "Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[3] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, pp. 1–10, 2014.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12. Usenix, 2012, p. 2.

[6] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[7] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system." in *OSDI*. Usenix, 2016, pp. 301–316.

[8] 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*. ACM, 2017, pp. 527–543.

[9] H. Liu and H. H. Huang, "Graphene: Fine-grained {IO} management for graph computing," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, 2017, pp. 285–300.

[10] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *ACM SIGPLAN Notices*, vol. 53, no. 1. ACM, 2018, pp. 246–260.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[12] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[13] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[14] A. Lugowski, A. Buluç, J. R. Gilbert, and S. Reinhardt, "Scalable complex graph analysis with the knowledge discovery toolbox," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 5345–5348.

[15] D. Bader, A. Buluç, J. Gilbert, J. Gonzalez, J. Kepner, and T. Mattson, "The graph blas effort and its implications for exascale," in *SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14)*, 2014.

[16] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM, 2014, pp. 1–6.

[17] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[18] 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 *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 313–322.

[19] Y. Ahmad, O. Khattab, A. Malik, A. Musleh, M. Hammoud, M. Kutlu, M. Shehata, and T. Elsayed, "La3: a scalable link-and locality-aware linear algebra-based graph analytics system," *Proceedings of the VLDB Endowment*, vol. 11, no. 8, pp. 920–933, 2018.

[20] T. Davis, "Algorithm 9xx: Suitesparse: Graphblas: graph algorithms in the language of sparse linear algebra," *submitted to ACM Trans on Mathematical Software*, 2018.

[21] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–11.

[22] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 688–697.

[23] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "Petsc users manual," Technical Report ANL-95/11-Revision 2.1. 5, Argonne National Laboratory, Tech. Rep., 2004.

[24] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[25] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, "A study of partitioning policies for graph analytics on large-scale distributed platforms," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 321–334, 2018.

[26] Schedmd. (2019) Slurm workload manager. [Online]. Available: https://slurm.schedmd.com/

[27] Intel. (2019) Intel mpi library. [Online]. Available: https://software.intel.com/en-us/mpi-library

[28] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *13th ACM WWW*, 2004, pp. 595–601.

[29] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.