A TOPOLOGY-AWARE CODING FRAMEWORK FOR DISTRIBUTED GRAPH PROCESSING

Başak Güler, A. Salman Avestimehr and Antonio Ortega

University of Southern California
Ming Hsieh Department of Electrical and Computer Engineering
Los Angeles, CA

ABSTRACT

This paper proposes a coded distributed graph processing framework to alleviate the communication bottleneck in large-scale distributed graph processing. In particular, we propose a topology-aware coded computing (TACC) algorithm that has two salient features. First, we propose a topology-aware graph allocation strategy. Second, we propose a coded aggregation scheme that combines the intermediate computations for graph processes while constructing coded messages. The proposed setup builds on a trade-off between computation and communication, in that increasing the computation load at the distributed parties can in turn reduce the communication load. We demonstrate the effectiveness of the TACC algorithm by comparing the communication load with existing setups on a Google web graph for PageRank computations. In particular, we show that the proposed coding strategy can lead up to 82% improvement in reducing the communication load when compared to the state-of-the-art.

Index Terms— Distributed computing, large-scale graph processing, graph signal filtering

1. INTRODUCTION

Distributed graph processing provides a convenient way to scale-up data-intensive graph applications, by distributing the storage and computation load of a data processing application across multiple processors [1–5]. Such applications include graph filtering, where filter output at each node corresponds to a linear combination of signals from neighboring nodes scaled by filter coefficients [5–8], and PageRank evaluations for web search [3]. These frameworks operate on a *think like a vertex* principle, where each vertex is assumed to send and receive messages along the graph edges. A major challenge in distributed graph processing is inter-processor communication load, which can become a serious bottleneck by taking up to 50% of the overall execution time [9, 10].

Coding has proven to be an effective technique in tackling the communication bottleneck in distributed systems. It has been shown in [11] that coding can be utilized to achieve an inverse-linear trade-off between the communication and computation load in a MapReduce system [12]. The coded MapReduce framework from [11] has been extended to computing general graph processes in [13], providing a speedup of 47% over the baseline uncoded scheme. For the Erdős-Rényi graph, it has been shown in [13] that one can also achieve an inverse-linear trade-off between computation and communication. Recently, reference [14] has leveraged aggregation techniques to combine the intermediate results in the MapReduce framework of [11] to reduce the communication load.

This paper extends the topology-independent coded graph processing setup from [13] to networks with irregular graph topologies. Unlike classical random graph models such as the Erdős-Rényi graph, real-world graphs such as the WWW often have irregular degree topologies [15]. An important question for minimizing the inter-processor communication in such topologies is how to assign high degree nodes across processors. On the one hand, one can store a high degree node in multiple processors (each containing part of its neighbors), and completely eliminate the cost of communication for that node. Alternatively, one can place the node in a single processor and utilize multicasting to communicate its value with all its neighbors. By a careful placement of such nodes across the processors, this can also enable opportunities for network coding, e.g., multicasting coded messages that will be useful to multiple processors simultaneously.

To address the trade-offs involved in high-degree node assignment, we propose a topology-aware coded computing (TACC) framework that distributes the nodes across the processors to create multi-casting opportunities, but while doing so also ensures that high degree nodes are simultaneously stored at many processors. First, we propose a topology-aware graph allocation strategy to create redundancy in the computations performed by each processor, where different fractions of a processor's memory are allocated to vertices with different degree structures. Second, we propose a coded aggregation strategy that combines the intermediate computations prior to communication, and creates coded messages using the aggregated computations. We show that the proposed TACC algorithm can greatly reduce the communication load compared to the schemes that are oblivious to the graph structures.

We demonstrate the effectiveness of TACC on a real-world Google web graph [16]. By comparing the communication load of TACC to the state-of-the-art distributed coding algorithm which is oblivious to the graph topology [13], we show that TACC can lead up to 82% improvement in reducing the communication load.

Our main contributions over the topology-independent coded processing framework of [13] can be summarized as follows:

- We introduce a coding strategy for irregular graph topologies based on the degree structures of graphs.
- We propose a judicious aggregation strategy that combines the intermediate computations prior to coding.
- We show that TACC can reduce the communication load by more than 80% on real-world graphs over the state-of-the-art.

This work is also related to distributed matrix computations and distributed learning. For this setting, coding has been leveraged mainly for the straggler problem [17–21]. Our focus is not on the straggler problem, but instead in reducing the communication load of the distributed system. Moreover, unlike the general matrix-vector multiplication problem, graphs often exhibit special topological behavior. For instance, matrices corresponding to graphs (adjacency matrices) are often sparse, and node degrees can vary significantly, which we leverage in this paper.

Notation. In the remainder of the paper, x is a scalar variable, $\mathbf{x}$ is a vector, and $\mathcal{X}$ represents a set with cardinality $|\mathcal{X}|$. $\mathbf{A}$ is a matrix with A_{ij} denoting its $(i,j)^{th}$ element.

2. PROBLEM FORMULATION

We consider a graph $G = (\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}| = N$ nodes (vertices). If G is an undirected graph, $(i,j) = (j,i)$ represents an undirected edge between nodes i and j, whereas if G is directed, (i,j) indicates a directed edge from node i to node j. Vector $\mathbf{x} = (x_1, \ldots, x_N)$ represents vertex-labels, such that $x_i \in \mathbb{R}$ is the label of node i. Similarly, an edge label $e_{ji} \in \mathbb{R}$ is associated with edge $(i,j) \in \mathcal{E}$. We define the (open) neighborhood of node $i \in \mathcal{V}$ by $\mathcal{N}(i) = \{j : (i,j) \in \mathcal{E}\}$. We are interested in computing functions of the form

$$y_j = \sum_{i: j \in \mathcal{N}(i)} e_{ji} x_i \tag{1}$$

for $j = 1, \ldots, N$. By defining a matrix $\mathbf{M}$ such that

$$M_{ji} = \begin{cases} e_{ji} & \text{if} \quad j \in \mathcal{N}(i) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

one can represent (1) as $\mathbf{y} = \mathbf{M}\mathbf{x}$ with $\mathbf{y} = (y_1, \ldots, y_N)$. [1]

Equation (1) can also be represented as a MapReduce computation [12], by decomposing it into *map* and *reduce* functions. The map function represents the inner computation

$$g_{ji}(x_i) \triangleq e_{ji} x_i, \tag{3}$$

where $u_{ji} \triangleq g_{ji}(x_i)$ is called an *intermediate value*. The reduce function represents the summation

$$f_j(\{u_{ji} : j \in \mathcal{N}(i)\}) = \sum_{i: j \in \mathcal{N}(i)} u_{ji}, \tag{4}$$

to compute the output value y_j in (1). A large number of graph-based algorithms can be represented in the form of (1), including PageRank computation, graph signal filtering, or semi-supervised learning.

We consider a distributed scenario in which (1) is computed by K workers (processors), connected through a multicast network. Worker k stores a subgraph of G with $r\frac{N}{K}$ nodes, given by $\mathcal{M}_k \subseteq V$, where $\mathcal{V} = \bigcup_{k=1}^{K} \mathcal{M}_k$[2]. Worker k is also responsible for computing the outputs for $\frac{N}{K}$ nodes, denoted by $\mathcal{R}_k \subseteq \mathcal{V}$, where $\mathcal{R}_k \cap \mathcal{R}_j = \emptyset$ for $k \neq j$. Given a subgraph allocation $\mathcal{M} = (\mathcal{M}_1, \ldots, \mathcal{M}_K)$ and an output allocation $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_K)$, the distributed processing setup consists of three stages, map phase, communication (shuffle) phase, and reduce phase.

Map Phase. In this phase, workers utilize their allocated subgraphs to compute the corresponding intermediate values. For every $i \in \mathcal{M}_k$, worker k computes the intermediate values $\mathbf{g}_i = (\{u_{ji} : j \in \mathcal{N}(i)\})$. The computation load of the system is defined as follows.

Definition 1 (Computation Load). *Given a subgraph allocation $\mathcal{M} = (\mathcal{M}_1, \ldots, \mathcal{M}_K)$, the computation load of the distributed graph processing system is characterized as,*

$$r \triangleq \frac{\sum_{k=1}^{K} |\mathcal{M}_k|}{N}. \tag{5}$$

Shuffle Phase. After computing the intermediate values, worker k constructs a message $Z_k \in \mathbb{R}^{z_k}$ of length z_k, given by a function $Z_k = \phi_k(\{\mathbf{g}_i : i \in \mathcal{M}_k\})$, which consists of the computations required by other workers. In the shuffle phase, worker k multicasts Z_k to other parties in the system[3]. At the end of this phase, worker k has all the information it needs to compute (1) for the nodes in R_k. We formally define the communication load in this phase as follows.

Definition 2 (Communication Load). *The communication load is defined as the total number of messages communicated by K workers during the shuffle phase,*

$$L_G(\mathcal{M}, \mathcal{R}, r) \triangleq \sum_{k=1}^{K} z_k \tag{6}$$

for a given subgraph allocation $\mathcal{M} = (\mathcal{M}_1, \ldots, \mathcal{M}_K)$ and output allocation $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_K)$.

Reduce Phase. Worker k uses messages $Z_1, \ldots, Z_K$ communicated during shuffle phase and local computations $(\{\mathbf{g}_i : i \in \mathcal{M}_k\})$ from the map phase to compute (1) for the nodes in $\mathcal{R}_k$, via a decoding function $y_j = \psi_k^j(Z_1, \ldots, Z_K, \{\mathbf{g}_i : i \in \mathcal{M}_k\})$ for $j \in \mathcal{R}_k$.

We wish to identify the trade-off between computation and communication, as to how much reduction in communication is achieved by increasing the computation load[4]. To do so, we characterize the minimum communication load as a function of the computation load.

Definition 3. *Given a computation load r, we define the communication - computation function as,*

$$L^*(r) = \inf_{\mathcal{M}, \mathcal{R}} L_G(\mathcal{M}, \mathcal{R}, r) \tag{7}$$

$L^*(r)$ identifies the trade-off between the computation load and the communication load of the system.

Main Problem. *Given a computation load r, our goal is to find the optimal subgraph allocation $\mathcal{M}$, output allocation $\mathcal{R}$, and coding scheme that minimizes the communication load $L^*(r)$.*

3. TOPOLOGY-AWARE CODED COMPUTING (TACC)

In this section, we introduce our topology-aware coded computing (TACC) algorithm to perform computations of the form (1). An illustrative example of the algorithm is provided first.

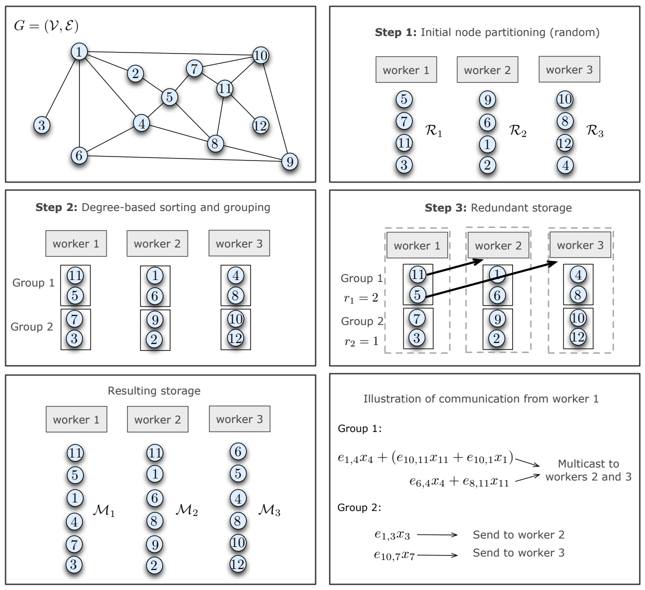
Consider the graph $G = (\mathcal{V}, \mathcal{E})$ depicted in Fig. 1 along with a distributed computing setup with $K = 3$ workers. In the first step, the set $\mathcal{V}$ is partitioned into K equal-sized parts and each part is associated with a distinct worker. Each worker is responsible from computing the output values in (1) corresponding to the nodes within the part it is assigned to. In the second step, the nodes in each part are sorted in descending order with respect to their degree, and divided into Q equal-sized chunks. In this example, we have $Q = 2$, therefore each part is divided into 2 groups with 2 nodes in each group. In the third step, we define a parameter r_q associated with part $q = 1, 2$. This parameter indicates that each node in group q is to be stored at r_q workers. For the example at hand, $r_1 = 2$, hence each node in the first group is replicated at 2 workers. On the other hand, for the second group we have $r_2 = 1$, hence the nodes in the second group is stored at a single worker. We call r_q the *computation load* for group $q = 1, 2$. The node allocation is done according

[1] In applications such as graph filtering, one may replace $\mathcal{N}(i)$ in (1) with the closed neighborhood $\{i\} \cup \mathcal{N}(i)$.

[2] Storing a subgraph corresponds to storing the labels x_i and e_{ji} for all nodes $i \in \mathcal{M}_k$ and $j \in \mathcal{N}(i)$.

[3] Existing practical systems that employ network-assisted multicast include IP multicast for streaming media [22]. We note that even if network multicast is disabled, one can use an application layer multicast, such as by using the MPI broadcast feature [23], that builds efficient multicast mechanisms at the application layer [24].

[4] We assume the graph is fixed and needs to be processed multiple times.



$G = (\mathcal{V}, \mathcal{E})$
Step 1: Initial node partitioning (random)
worker 1
worker 2
worker 3
$\mathcal{R}_1$
$\mathcal{R}_2$
$\mathcal{R}_3$
Step 2: Degree-based sorting and grouping
Group 1
Group 2
Step 3: Redundant storage
$r_1 = 2$
$r_2 = 1$
Resulting storage
$\mathcal{M}_1$
$\mathcal{M}_2$
$\mathcal{M}_3$
Illustration of communication from worker 1
Group 1:
$e_{1,4}x_4 + (e_{10,11}x_{11} + e_{10,1}x_1)$
$e_{6,4}x_4 + e_{8,11}x_{11}$
Multicast to workers 2 and 3
Group 2:
$e_{1,3}x_3$ Send to worker 2
$e_{10,7}x_7$ Send to worker 3

Fig. 1. Illustrative example for the proposed TACC algorithm.

to the following procedure. For each worker, the nodes associated with part q are divided into $\binom{K-1}{r_q-1}$ equal-sized segments, and each segment is replicated at a distinct subset of $r_q - 1$ other workers. As a result, each node in group q is stored at r_q workers in total. By selecting $r_1 > r_2$, one can ensure that each worker allocates a larger fraction of its memory to higher degree nodes. Finally, workers aggregate the intermediate computations that are targeted for the same destination node, and form coded messages to be communicated to the other workers. For instance, worker 1 multicasts the coded message $e_{1,4}x_4 + (e_{10,11}x_{11} + e_{10,1}x_1)$ to workers 2 and 3. The value $e_{10,11}x_{11} + e_{10,1}x_1$ is needed by worker 3 to evaluate y_{10}, whereas $e_{1,4}x_4$ is needed by worker 2 to compute y_1 from (1). To recover the missing values, workers 2 and 3 evaluate $e_{10,11}x_{11} + e_{10,1}x_1$ and $e_{1,4}x_4$, respectively, and remove it from the coded message. Workers communicate the missing values separately for the two groups.

The proposed topology-aware coded computing (TACC) algorithm is given in Algorithm 1. Initially, we partition $\mathcal{V}$ into K equal-sized non-overlapping parts $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_K)$. Then, we utilize the graph topology to design a subgraph allocation scheme $\mathcal{M} = (\mathcal{M}_1, \ldots, \mathcal{M}_K)$ in which higher degree nodes are stored at a larger fraction of workers. Our intuition is that higher degree nodes are needed by a larger fraction of workers. As such, we carefully design a subgraph allocation scheme so that a larger portion of each worker's memory is dedicated to store higher degree nodes. To do so, our scheme takes two system parameters. The first parameter, Q, is defined to group vertices with similar degrees. Specifically, nodes in each $\mathcal{R}_k, k = 1, \ldots, K$, are sorted according to their degrees and then divided into Q groups, such that each group has $\frac{|\mathcal{R}_k|}{Q} = \frac{N}{QK}$ nodes. The first (last) group consists of the $\frac{N}{QK}$ highest (lowest) degree nodes in $\mathcal{R}_k$. We represent the sorted and grouped version of $\mathcal{R}_k$ by $\widetilde{\mathcal{R}}_k = (\widetilde{\mathcal{R}}_{k1}, \ldots, \widetilde{\mathcal{R}}_{kQ})$, where $|\widetilde{\mathcal{R}}_{kq}| = \lceil \frac{N}{QK} \rceil$ for $q = 1, \ldots, Q$. The second parameter, defined as $\mathbf{r} = (r_1, \ldots, r_Q)$, where $r_q \in \{1, \ldots, K\}$, $q = 1, \ldots, Q$, specifies that each node in group q is stored at r_q workers. We assume $r_1 \geq \ldots \geq r_Q$ to ensure that high degree nodes are stored at a larger fraction of workers.

By controlling Q and $\mathbf{r}$, one can make sure that the memory constraints of the workers are satisfied. For instance, setting $Q = 1$ and choosing $r_Q = 1$ indicates that there is no overlap between the nodes stored at different workers, whereas choosing $r_Q = K$ indicates that every worker stores all of the nodes in $\mathcal{V}$.

Algorithm 1 Topology-Aware Coded Computing (TACC)

```
1: Initialize graph $G = (\mathcal{V}, \mathcal{E})$, number of workers $K$, parameters $Q$, and $\mathbf{r} = (r_1, \ldots, r_Q)$.
   Subgraph and Output Allocation Phase:
2: Partition $\mathcal{V}$ into $K$ equal-sized parts $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_K)$.
3: for $k = 1, \ldots, K$
4:     Let $\widetilde{\mathcal{R}}_k \leftarrow \{\mathcal{R}_k$ sorted using node degrees, in descending order$\}$
5:     Divide $\widetilde{\mathcal{R}}_k$ into $Q$ equal sized groups, $\widetilde{\mathcal{R}}_k = (\widetilde{\mathcal{R}}_{k1}, \ldots, \widetilde{\mathcal{R}}_{kQ})$, where $\widetilde{R}_{k1}$ holds the highest degree nodes from $\widetilde{\mathcal{R}}_k$.
6:     for $q = 1, \ldots, Q$
7:         Divide $\widetilde{\mathcal{R}}_{kq}$ into $\binom{K-1}{r_q-1}$ equal-sized, non-overlapping parts. Store each part at a distinct subset of $r_q$ workers (including worker k).
8: Denote the subgraph allocated to worker $k$ at the end of Step 3 by:
       $$\mathcal{M}_k = \bigcup_{q=1}^{Q} \mathcal{M}_k^q, \quad (8)$$
   where $\mathcal{M}_k^q$ is the subgraph stored at worker $k$ for group $q$.
   Map Phase:
9: for $k = 1, \ldots, K$
10:    Worker $k$ maps the nodes in $\mathcal{M}_k$ from (8) using the map function (3), and obtains the intermediate values,
       $$\{u_{ji} : i \in \mathcal{M}_k, j \in \mathcal{N}(i)\} \quad (9)$$
   Coded Aggregation and Shuffling Phase:
11: for $q = 1, \ldots, Q$
12:     for Each set of workers $\mathcal{S} \subseteq \{1, \ldots, K\}$ of size $r_q + 1$
13:         for $k \in \mathcal{S}$
14:             Workers in $\mathcal{S} \backslash \{k\}$ compute,
                $$u_j = \sum_{i \in \left(\bigcap_{k' \in \mathcal{S} \backslash \{k\}} \mathcal{M}_{k'}^q\right)} u_{ji}, \quad j \in \mathcal{R}_k, \quad (10)$$
                and form a vector of *aggregated intermediate values*
                $$\mathbf{u}_{\mathcal{S} \backslash \{k\}}^k = (\{u_j : j \in \mathcal{R}_k\}). \quad (11)$$
15:             Split $\mathbf{u}_{\mathcal{S} \backslash \{k\}}^k$ into $r_q$ equal-sized segments (chunks of equal vector size), given by $\mathbf{u}_{\mathcal{S} \backslash \{k\}, s}^k$ for $s \in \mathcal{S} \backslash \{k\}$. Associate each segment $s$ with a distinct worker $s \in \mathcal{S} \backslash \{k\}$.
16:         for $j \in \mathcal{S}$
17:             Worker $j$ computes a coded message by forming a linear combination of the segments associated with it
                $$Z_j^{\mathcal{S}} = \sum_{k \in \mathcal{S} \backslash \{j\}} \mathbf{u}_{\mathcal{S} \backslash \{k\}, j}^k \quad (12)$$
                by zero-padding if needed to ensure their sizes are equal, and multicasts to the workers in $\mathcal{S} \backslash \{j\}$.
   Reduce Phase:
18: for $q = 1, \ldots, Q$
19:     for Each set of workers $\mathcal{S} \subseteq \{1, \ldots, K\}$ of size $r_q + 1$
20:         for $k \in \mathcal{S}$
21:             After receiving $Z_j^{\mathcal{S}}$ from workers $j \in \mathcal{S} \backslash \{k\}$, worker $k$ removes the known segments to compute,
                $$\mathbf{u}_{\mathcal{S} \backslash \{k\}, j}^k = Z_j^{\mathcal{S}} - \sum_{k' \in \mathcal{S} \backslash \{j,k\}} \mathbf{u}_{\mathcal{S} \backslash \{k'\}, j}^{k'} \ \forall j \in \mathcal{S} \backslash \{k\}. \quad (13)$$
```

During the map phase, worker k maps the nodes in $\mathcal{M}_k$ from (8) using the map function (3), to obtain the intermediate values. The computation load of the system is

$$r = \frac{\sum_{k=1}^{K} |\mathcal{M}_k|}{N} = \frac{1}{Q} \sum_{j=1}^{Q} r_j. \quad (14)$$

We note that this phase follows the standard mapping stage of distributed algorithms, and unlike the previous stage, is not specific to the proposed TACC algorithm.

In coded aggregation and shuffling phase, the redundancy created during graph allocation is used to create coded multicasting opportunities. A key aspect of our strategy is to aggregate the intermediate values before forming coded messages to reduce the number of messages communicated by each worker and the overall communication load. Aggregation is achieved by taking linear combinations of the individual intermediate values targeted at a specific node.

In the reduce phase, each worker recovers the intermediate values needed to compute (1). For each group $q \in \{1, \dots, Q\}$, there are $\binom{K}{r_q+1}$ subsets $\mathcal{S}$ of size $r_q + 1$. Worker k is involved in $\binom{K-1}{r_q}$ of them, and within each subset, there are r_q distinct segments $\mathbf{u}^k_{\mathcal{S}\backslash\{k\},s}$ known by other workers $s \in \mathcal{S}\backslash\{k\}$ and needed by worker k. Hence, worker k needs to recover $\sum_{q=1}^{Q} \binom{K-1}{r_q} r_q$ distinct segments in total. Consider a group q and set $\mathcal{S} \subseteq \{1, \dots, K\}$ of size $r_q + 1$. For workers $j, k \in \mathcal{S}$ and $j \neq k$, there are $\binom{|\mathcal{S}|-2}{r_q-2} = r_q - 1$ subsets of $\mathcal{S}$ that has size r_q and contain both j and k. Among the r_q segments associated with worker j, $r_q - 1$ of them are known also by worker k, who needs the remaining segment. After receiving $Z_j^{\mathcal{S}}$ from workers $j \in \mathcal{S}\backslash\{k\}$, worker k can remove the known segments to compute (13). Hence, worker k recovers r_q distinct segments for the subset $\mathcal{S}$. Since there are $\binom{K-1}{r_q}$ such subsets per group q, worker k will recover all of the required $\sum_{q=1}^{Q} \binom{K-1}{r_q} r_q$ segments. A theoretical analysis of the proposed TACC algorithm is developed as part of a manuscript in preparation.

4. PERFORMANCE EVALUATION

We demonstrate the performance of our proposed TACC algorithm on a real-world scale-free network. An important real-world example of a scale-free network is a *web graph*, in which nodes represent webpages and edges denote the hyperlinks between webpages. Web graphs are utilized for various graph computations such as PageRank. We consider a Google web graph from [16], with 875,713 nodes, 5,105,039 edges, and an average degree of 5.57.

For the initial partitioning of the graph, we utilize the METIS graph partitioning tool [25, 26]. We consider a distributed system with $K = 5$ workers, and simulate the following three scenarios. The first one is coded computing without aggregation from [13]. The second one is topology-independent coded aggregation, obtained by setting $Q = 1$. The third scenario is topology-aware coded computing with aggregation from Algorithm 1, where we let $Q = 5$. We then compare the communication load for the three setups, by keeping the memory size of each worker the same, that is, each worker can store the same number of nodes in all three scenarios. To do so, we denote the computation load for the first two scenarios as $r \in \{1, \dots, K\}$, and select the computation load $\mathbf{r} = (r_1, \dots, r_Q)$ for the third scenario so that (14) is satisfied. In particular, we select,

$$\mathbf{r} = (r_1, \dots, r_5) = \begin{cases} (1,1,1,1,1) \text{ for } r = 1 \\ (5,2,1,1,1) \text{ for } r = 2 \\ (5,3,3,3,1) \text{ for } r = 3 \\ (5,4,4,4,3) \text{ for } r = 4 \\ (5,5,5,5,5) \text{ for } r = 5 \end{cases} \tag{15}$$

As a result, the average computation load is the same for all three scenarios. We remark that $\mathbf{r}$ is a parameter to be determined by the system designer, by taking into account the graph structure and the memory size of each worker.

We provide the comparison of the communication load for the three schemes in Fig. 2. As observed from Fig. 2a, aggregation can lead to an improvement of 57% over the coding strategy without aggregation, whereas topology-aware coding with aggregation can lead up to an 82% improvement. Hence, aggregation can be very useful in reducing the communication cost for sparse graphs with non-homogeneous degree structures. In Fig. 2b, we compare the communication load for topology-independent and topology-aware coding schemes, both with aggregation, and find that topology-aware subgraph allocation can lead up to 57% improvement over the topology-independent setup.

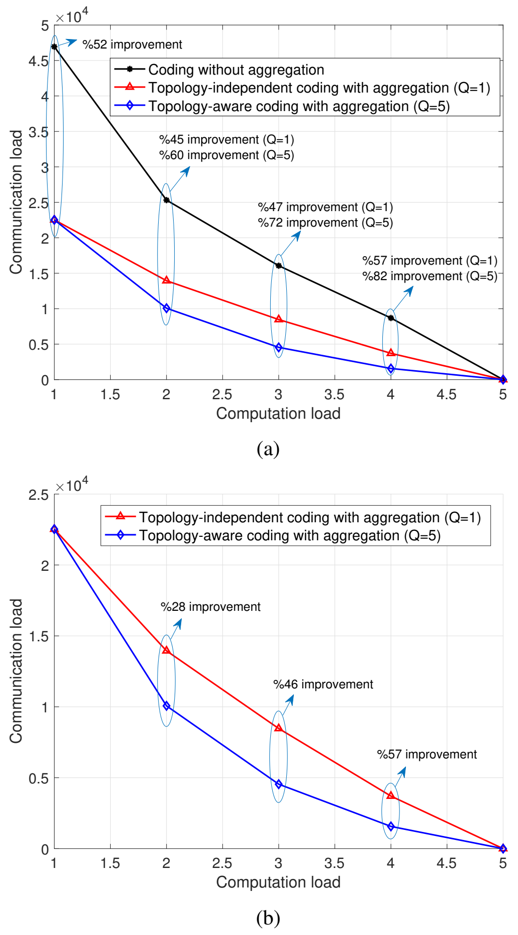


Fig. 2. Demonstrating the gain of the topology-aware coded computing (TACC) algorithm over (i) topology-independent coding with aggregation and (ii) coding without aggregation, in reducing the communication load for running PageRank on the Google web graph dataset consisting of 875,713 nodes and 5,105,039 edges.

5. CONCLUSION

We have considered the communication bottleneck in distributed graph processing. In order to reduce the communication load, we have proposed a coded graph computing algorithm (TACC) that leverages graph topology for subgraph allocation, in that nodes that are needed by a larger number of workers are stored at a larger fraction of workers. The proposed TACC algorithm carefully creates overlaps between the subgraphs allocated to different workers and aggregates the intermediate computations to enable coded multicasting opportunities. We then demonstrate the benefits of aggregation and topology-aware coding strategy over the existing schemes.

6. REFERENCES

[1] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of ACM International Conference on Management of Data*, 2010, pp. 135–146.

[2] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph 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.

[3] Sergey Brin and Lawrence Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[4] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, pp. 036106, 2007.

[5] Antonio Ortega, Pascal Frossard, Jelena Kovačević, José MF Moura, and Pierre Vandergheynst, "Graph signal processing: Overview, challenges, and applications," *Proceedings of the IEEE*, vol. 106, no. 5, pp. 808–828, 2018.

[6] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst, "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains," *IEEE Signal Processing Magazine*, vol. 30, no. 3, pp. 83–98, 2013.

[7] Aliaksei Sandryhaila and José MF Moura, "Discrete signal processing on graphs," *IEEE transactions on signal processing*, vol. 61, no. 7, pp. 1644–1656, 2013.

[8] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, 2016, pp. 3844–3852.

[9] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[10] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 215–226.

[11] Songze Li, Mohammad Ali Maddah-Ali, Qian Yu, and A Salman Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, 2018.

[12] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] Saurav Prakash, Amirhossein Reisizadeh, Ramtin Pedarsani, and Salman Avestimehr, "Coded computing for distributed graph analytics," in *IEEE International Symposium on Information Theory*, 2018.

[14] Songze Li, Mohammad Ali Maddah-Ali, and A Salman Avestimehr, "Compressed coded distributed computing," in *IEEE International Symposium on Information Theory*, 2018.

[15] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[16] Jure Leskovec and Andrej Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[17] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[18] Qian Yu, Mohammad Maddah-Ali, and Salman Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," in *Advances in Neural Information Processing Systems*, 2017, pp. 4406–4416.

[19] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Advances In Neural Information Processing Systems*, 2016, pp. 2100–2108.

[20] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *International Conference on Machine Learning*, 2017, pp. 3368–3376.

[21] Qian Yu, Netanel Raviv, Jinhyun So, and A Salman Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security and privacy," *e-print arXiv:1806.00939*, 2018.

[22] Steve Deering, "RFC 1112: Host extensions for IP multicasting," 1989.

[23] Lisandro Dalcín, Rodrigo Paz, and Mario Storti, "MPI for Python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[24] Songze Li, Sucha Supittayapornpong, Mohammad Ali Maddah-Ali, and Salman Avestimehr, "Coded Terasort," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 389–398.

[25] George Karypis and Vipin Kumar, "Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[26] George Karypis and Vipin Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.