Distributed Training of Embeddings using Graph Analytics

Gurbinder Gill
Katana Graph Inc.
Austin, TX, USA
gill@katanagraph.com

Roshan Dathathri
Katana Graph Inc.
Austin, TX, USA
roshan@katanagraph.com

Saeed Maleki
Microsoft Research
Redmond, WA, USA
saemal@microsoft.com

Madan Musuvathi
Microsoft Research
Redmond, WA, USA
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research
Redmond, WA, USA
toddm@microsoft.com

Olli Saarikivi
Microsoft Research
Redmond, WA, USA
olsaarik@microsoft.com

***Abstract*—Many applications today, such as natural language processing, network and code analysis, rely on semantically embedding objects into low-dimensional fixed-length vectors. Such embeddings naturally provide a way to perform useful downstream tasks, such as identifying relations among objects and predicting objects for a given context. Unfortunately, training accurate embeddings is usually computationally intensive and requires processing large amounts of data.**

This paper presents a distributed training framework for a class of applications that use *Skip-gram-like* models to generate embeddings. We call this class Any2Vec and it includes Word2Vec (Gensim), and Vertex2Vec (DeepWalk and Node2Vec) among others. We first formulate Any2Vec training algorithm as a graph application. We then adapt the state-of-the-art distributed graph analytics framework, D-Galois, to support dynamic graph generation and re-partitioning, and incorporate novel communication optimizations. We show that on a cluster of 32 48-core hosts our framework GraphAny2Vec matches the accuracy of the state-of-the-art shared-memory implementations of Word2Vec and Vertex2Vec, and gives geo-mean speedups of $12\times$ and $5\times$ respectively. Furthermore, GraphAny2Vec is on average $2\times$ faster than DMTK, the state-of-the-art distributed Word2Vec implementation, on 32 hosts while yielding much better accuracy.

***Index Terms*—Machine Learning, Graph Analytics, Distributed Computing**

I. Introduction

Embedding is a popular technique that associates a fixed-length vector to objects in a set such that the mathematical closeness of any two vectors approximates the relationship of the associated objects. Applications in areas such as natural language processing (NLP) [1]–[3], network analysis [4], [5], code analysis [6], [7], and bioinformatics [8] rely on semantically embedding objects into these vectors. Embeddings provide a way to perform downstream tasks, such as identifying relations among words, by studying their associated vectors. Training these embeddings requires large amounts of data to be used and is computationally expensive, which makes distributed solutions especially important.

An important class of embedding applications use *Skip-gram-like* models, like the one used in Word2Vec [1], to generate embeddings. We call this class Any2Vec and it includes Word2Vec (Gensim [9]), Vertex2Vec (DeepWalk [4] and Node2Vec [5] are examples), Sequence2Vec [8], and Doc2Vec [3] among others. Applications in this class maintain a large embedding matrix, where each row corresponds to the embedding vector for each object. Sequences of objects in the training data (text segments for Word2Vec and graph paths in Vertex2Vec) identify their relations, which are used during (semi-supervised) training to update the associated embedding vectors. The relations between objects are irregular, which leads to sparse accesses to the embedding matrix. Furthermore, the relations change with the training data, making accesses dynamic during the training process. Due to these dynamic and sparse accesses, distributing the training of Any2Vec embeddings is challenging and existing distributed implementations [10]–[13] scale poorly due to the overheads of frequent communication and even then they do not match the accuracy of the shared-memory implementations.

We first formulate the Any2Vec class of machine learning algorithms that use *Skip-gram-like* models as a graph application. The objects are vertices and the relations between them are edges. An edge *operator* updates the embedding vectors of the edge's source and destination vertices. The algorithm applies these edge operators in a specific order or *schedule* (the schedule is critical for accuracy). However, these applications cannot be implemented in existing distributed graph analytics frameworks [14]–[17] because it requires: (1) edge operators, (2) general (vertex-cut) graph partitioning policies, and (3) dynamic graph re-partitioning. None of the existing frameworks support all these required features.

This paper introduces a distributed training framework for Any2Vec applications called GraphAny2Vec, which extends D-Galois [17], a state-of-the-art distributed graph analytics framework, to support dynamic graph generation and re-partitioning. GraphAny2Vec leverages the communication optimizations in D-Galois and adds novel application-specific communication optimizations. It also leverages the Adaptive Summation technique [18] for preserving accuracy during synchronization and exposes a runtime parameter to transparently

trade-off synchronization rounds and accuracy.

We evaluate two applications, Word2Vec and Vertex2Vec, in GraphAny2Vec on a cluster of up to 32 48-core hosts with 3 different datasets each. We show that compared to the state-of-the-art shared-memory implementations (the original C implementation [2] and Gensim [9] for Word2Vec and DeepWalk [4] for Vertex2Vec) on 1 host, GraphAny2Vec on 32 hosts is on average 12× and 5× faster than Word2Vec and Vertex2Vec respectively, while matching their accuracy. GraphAny2Vec can reduce the training time for Word2Vec from 21 hours to less than 2 hours on our largest dataset of Wikipedia articles while matching the accuracy of Gensim. On 32 hosts, GraphAny2Vec is on average 2× faster than the state-of-the-art distributed Word2Vec implementation in Microsoft's Distributed Machine Learning Toolkit (DMTK) [10].

The rest of this paper is organized as follows. Section II provides a background on Any2Vec and graph analytics. Section III describes our GraphAny2Vec framework and Section IV presents our evaluation of GraphAny2Vec. Related work and conclusions are presented in Sections V and VI.

II. Background

In this section, we first briefly describe how stochastic gradient descent is used to train machine learning models (Section II-A), followed by how Any2Vec models are trained with Word2Vec as an example (Section II-B). We then provide an overview of graph analytics (Section II-C).

A. Stochastic Gradient Descent

A machine learning model is usually expressed by a function $M : \mathbb{R}^n \times \mathbb{R}^k \rightarrow \mathbb{R}^r$ or $M(w, x) = y$ where $w \in \mathbb{R}^n$ is the weight of the model parameters, $x \in \mathbb{R}^k$ is the input example and $y \in \mathbb{R}^r$ is the prediction of the model. The training task of a machine learning model is defined by a set of training examples (x_i, y_i) where $x_i \in \mathbb{R}^k$ and $y_i \in \mathbb{R}^r$ and a loss function $L : \mathbb{R}^r \times \mathbb{R}^r \rightarrow \mathbb{R}^+$ ($\mathbb{R}^+$ is the set of non-negative reals) which assigns a penalty based on closeness of $M(w, x_i)$, the model prediction, and y_i, the true answer. In an ideal world, $M(w, x_i) = y_i$ and the loss function would be 0 for all training examples i. Since (x_i, y_i) are constant during the training process, we will use function $L_i : \mathbb{R}^n \rightarrow \mathbb{R}^+$ defined by $L_i(w) = L(M(w, x_i), y_i)$ which is the loss for training example i given model parameters w for convenience. Since perfect prediction is usually impossible, for a given set of training examples, a w with minimum $\sum_i L_i(w)$ is desired.

Stochastic Gradient Descent (SGD) [19] is a popular algorithm for machine learning training to find w with minimum total loss. Suppose that there are l training examples and $\{r_1, \ldots, r_l\}$ is a random shuffle of indices from 1 to l. The model is initially set to a random guess w_0 and at iteration i,

$$w_i := w_{i-1} - \alpha_i \cdot g_{r_i}(w_{i-1}) \tag{1}$$

where α_i is the *learning rate*, and $g_{r_i}(w_{i-1}) = \frac{\partial L_{r_i}}{\partial w}\big|_{w_{i-1}}$ is the *gradient* of L_{r_i} at w_{i-1} for training example i. The learning rate determines how much the model is moved in the gradient direction and is a sensitive hyper-parameter that needs to be carefully set and decayed as the training progresses. Training is complete when the model reaches a desired loss or evaluation accuracy. An *epoch* of training is the number of updates needed to go through the whole dataset once or equivalently, l iterations. For subsequent epochs, the dataset is shuffled again. For the rest of this paper, we will drop w_{i-1} from $g_{r_i}(w_{i-1})$ as the model is known from the context.

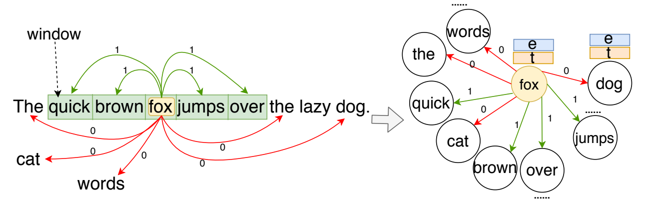

Fig. 1. Viewing Word2Vec in Skip-gram model as a graph. The unique words in the training data corpus form vertices of the graph and edges represent the positive (edge label: 1) and negative (edge label: 0) correlations among vertices. Labels on the vertices are: *embedding* (e) and *training* (t) vectors.

As it is evident from Equation 1, SGD's update rule for w_i depends on w_{i-1} which makes it an inherently sequential algorithm. Luckily, there are several parallelization approaches for SGD algorithm each of which changes the semantics of the algorithm. The most well-known technique is *mini-batch* SGD [20], where the gradients for multiple training examples are computed from the same point and then averaged. Mini-batch algorithm is semantically different from SGD and therefore, the convergence of the algorithm is negatively affected for the same number of gradients computed. Hogwild! [21] is another well-known SGD parallelization technique, wherein multiple threads compute gradients for different training examples in parallel in a lock-free way and update the model in a racy fashion. Surprisingly, this approach works well on a shared-memory system, especially with models where gradients are sparse. The staleness of the models following this parallelization technique hurts the accuracy with too many working threads. Adaptive Summation (AdaSum) [18] is another technique similar to mini-batch algorithm which computes the gradients for multiple gradients in parallel but instead of averaging, it uses the following formula to reduce two arbitrary gradients, g_a and g_b, into one:

$$AdaSum(g_a, g_b) = g_a + g_b - \frac{g_a^T \cdot g_b}{\|g_b\|^2} g_b \tag{2}$$

Using this reduction rule, AdaSum takes any number of gradients and produces one. Maleki et al. [18] show that AdaSum preserves the convergence of the sequential SGD algorithm when applied to training of dense deep neural networks (DNNs).

B. Training Any2Vec Embeddings

An embedding is a mapping from a dataset D to a vector space $\mathbb{R}^N$ such that elements of D that are related are close to each other in the vector space. The length N of the *embedding vectors* is typically much smaller than the size of D.

Many models have been proposed for learning embeddings. We will focus on the popular Skip-gram model [1] together with the negative sampling introduced in [2], and explain it here in a form suitable for a graph analytic understanding. Note that all the ideas proposed in this work are applicable for Continuous Bag-of-Words (CBOW) model as it shares architectural similarities with the Skip-gram model for computing vector representations of words.

Skip-gram uses a training task where it predicts if a target word w_O appears in the context of a center word w_I. Figure 1 (left) illustrates this for an example sentence, with "fox" as the center word. The context is then defined as the words inside a window of size c (a hyper-parameter) centered on "fox". In Figure 1 (left), these *positive* samples have green edges with label 1. For each positive sample, Skip-gram picks k (a hyper-parameter) random words as *negative* samples such as the red edges with label 0 in the figure. In case of Vertex2Vec, words correspond to the vertices of the network and context is defined by the paths generated by doing random walks on the network [4], [5].

The Skip-gram model consists of two vectors of size N for each word w in the vocabulary: an *embedding* vector e_w and a *training* vector t_w. For a pair of words $\langle w_I, w_O\rangle$, the model predicts the label with $\sigma(e_{w_I}^T \cdot t_{w_O})$. $e_{w_I}^T \cdot t_{w_O}$ is the inner product of the two vectors and $\sigma(x) = \frac{1}{1+exp(-x)}$ is the sigmoid function which is a strictly increasing function from 0 to 1. For a pair of related words, a close to 1 prediction is desired and a close to 0 otherwise. The loss function is the defined by $-\log(1 - |y - \sigma(e_{w_I}^T \cdot t_{w_O})|)$ where y is 1 (0) for related (unrelated) words. Following the standard gradient rules, e_{w_I} and t_{w_O} are update as follows:

$$\begin{aligned} e_{w_I}^i &= e_{w_I}^{i-1} - \alpha^i(y - \sigma(e_{w_I}^T \cdot t_{w_O}))t_{w_O}^{i-1} \\ t_{w_O}^i &= e_{w_I}^{i-1} - \alpha^i(y - \sigma(e_{w_I}^T \cdot t_{w_O}))e_{w_I}^{i-1} \end{aligned} \quad (3)$$

where the superscript identifies the iteration similar to the subscripts in Equation 1. Note that only one vector from each word is updated. Also, α^i is the learning rate for each iteration and our experiments suggested it is a sensitive hyper-parameter for Any2Vec and needs to set and decayed carefully.

We base the work in this paper on Google's Word2Vec tool[1], which uses the Hogwild! parallelization technique. Each thread is given a subset of the corpus and goes through the words in it sequentially (skipping some due to sub-sampling frequent words as described in [2]). On each thread and for each word w_I in the assigned text, Word2Vec finds the related words in the context window centered around w_I. For each pair of related words, Word2Vec randomly samples k unrelated words to update the vectors. Therefore, for each pair of related words, a total $k+1$ updates occur, 1 with label 1 and k with label 0. As discussed before, these updates occur in a racy manner.

For GraphAny2Vec in this paper, we use Hogwild! parallelization strategy on each machine in a cluster and Adaptive Summation (AdaSum) for synchronization of updates across machines. The combination of these approaches provides sufficient parallelism while preserving the accuracy in a distributed training. As we will discuss in Section IV-C, the original AdaSum reduction technique was proposed for DNNs but we adapted it to Any2Vec for performance gains.

C. Graph Analytics

In typical graph analytics applications, each vertex has one or more labels, which are updated during algorithm execution until a global quiescence condition is reached. The labels are updated by iteratively applying a computation rule, known as an *operator*, to the vertices or edges in the graph. The order in which the operator is applied to the vertices or edges is known as the *schedule*. A *vertex operator* takes a vertex n and updates labels on n or its neighbors, whereas an *edge operator* takes an edge e and updates labels on source and destination of e.

To execute graph applications in distributed-memory, the edges are first partitioned [22] among the hosts and for each edge on a host, proxies are created for its endpoints. As a consequence of this design, a vertex might have proxies (or replicas) on many hosts. One of these is chosen as the *master* proxy to hold the canonical value of the vertex. The others are known as *mirror* proxies. Several heuristics exist for partitioning edges and choosing *master* proxies [22].

Most distributed graph analytics systems [14]–[17] use bulk-synchronous parallel (BSP) execution. Execution is done in rounds of computation followed by bulk-synchronous communication. In the computation phase, every host applies the *operator* on vertices/edges in its own partition and updates the labels of the local proxies. Thus, different proxies of the same vertex might have different values for the same label. Every host then participates in a global communication phase to synchronize the labels of all proxies. Different proxies of the same vertex are reconciled by applying a *reduction* operator, which depends on the algorithm being executed.

III. DISTRIBUTED ANY2VEC

In this section, we first describe our formulation of Any2Vec as a graph application and provide an overview of our distributed GraphAny2Vec (Section III-A). We then describe dynamic graph generation and partitioning (Section III-B), model synchronization (Section III-C), and communication optimizations (Section III-D) in more detail.

We use Word2Vec as the running example in this section as other examples of Any2Vec follow the same logic.

A. Overview of Distributed GraphAny2Vec

We formulate Any2Vec as a graph problem and call it GraphAny2Vec. Each unique element in the dataset D (such as unique words in Word2Vec) corresponds to a vertex in a graph, and the *positive* and *negative* samples correspond to edges in the graph with label 1 and 0 respectively. Figure 1 (right) illustrates this for Word2Vec. Training the Skip-gram model is now a graph analytics application. Each vertex has two vectors — e and t — for embedding and training vectors, respectively, of size N (labels on the vertices as described in

[1]https://code.google.com/archive/p/word2vec/

Algorithm 1 Execution on each distributed host of GraphAny2Vec.

```
 1: procedure GRAPHANY2VEC(Corpus C, Num. of epochs R,
    Num. of sync rounds S, Learning rate α)
 2:     Let h be the host ID
 3:     Stream C from disk to build set of vertices V
 4:     Read partition h of C that forms a work-list W
 5:     for epoch r from 1 to R do
 6:         for sync round s from 1 to S do
 7:             Let W_s be partition s of W
 8:             Build graph G = (V, E) where E are samples in W_s
 9:             Compute(G, W_s, α)          ▷ Updates G locally
10:             Synchronize(G)              ▷ Updates G globally
11:         end for
12:     end for
13: end procedure
```

Section II-C). The model corresponds to these vectors for all vertices. These vectors are initialized randomly and updated during training by applying an *edge operator*, that takes the source src and destination dst of an edge, and updates $e_{\texttt{src}}$ and $t_{\texttt{dst}}$ using Equation 3 where src corresponds to w_I and dst corresponds to w_O. In each epoch, the operator is applied to all edges once in a specific *schedule*, which we describe in Section III-B. This schedule is critical for accuracy.

Algorithm 1 gives a brief overview of our distributed GraphAny2Vec execution. The first step is to construct the set of vertices V (unique words in case of Word2Vec) by making a pass over the training data corpus C on each host in parallel (line 3). As C may not fit in the memory of a single host, we stream it from disk to construct V. The corpus C is then partitioned (logically) into roughly equal contiguous chunks among hosts. All hosts read their own partition of C in parallel (line 4). The list of vertices in a host's partition of C constitutes the work-list[2] W that the host is responsible for computing Any2Vec on in each epoch (line 5). The graph edges (positive and negative samples as described in Section II-B) are dynamically generated as described in Section III-B. The same vertex may occur in W on different hosts, so its edges might be generated on different hosts. Such graph partitions are known as a *vertex-cut* [22] because edges of a vertex could be split across partitions. Since the number of edges generated and processed are determined by the window size (c) and number of negative samples (k) (both hyper-parameters), the equal-sized chunk assignment of the training data corpus aids in *load-balancing computation* among hosts.

We introduce a new (hyper-)parameter for controlling the number of synchronization rounds within an epoch. In each epoch on each host, W is partitioned into S roughly equal contiguous chunks (one for each round) as shown in Algorithm 1 (line 6). In each round s, W_s is the corresponding subset of W and it forms a smaller subgraph. The Any2Vec *operator* is then applied to all edges in the graph (line 9) in a specific *schedule*. The operator updates the vertex vectors directly using Equation 3. Then, all hosts participate in a bulk-synchronous communication to synchronize the vertex vectors (line 10). We explain this communication in more detail in Section III-C.

[2]The work-list W does not change across epochs, so we construct it once and reuse it for all epochs and synchronization rounds in our evaluation. However, if it does not fit in memory, partition s of W can be constructed from the corpus in each synchronization round.

GraphAny2Vec exploits both data and model parallelism. The edges on the graph represent the positive and negative samples and as discussed, the edges are partitioned among distributed hosts and are operated on in parallel. This corresponds to data parallelism. The vectors associated with vertices of the graph represent the model parameters. While the vertices are distributed among hosts, the same vertex might have proxies on multiple hosts. Updates for these vectors (the proxy copies or the master ones) may occur in the same round and therefore, the model parameters are updated in parallel which corresponds to model parallelism.

There are many distributed graph analytics systems [14]–[17] but GraphAny2Vec cannot be implemented in any of them without significant effort because none of them provide the following three requirements in the same framwork: (1) edge operators, (2) vertex-cut partitioning policies, and (3) *dynamic* graphs generation where the vertices and edges change during algorithm execution out of the box. We extend D-Galois [17], the state-of-the-art distributed graph analytics system, to support GraphAny2Vec. D-Galois consists of the Galois [23] multi-threaded library for computation and the Gluon [17] communication substrate for synchronization. Galois provides efficient, concurrent data structures like graphs, work-lists, dynamic bit-vectors, etc. Gluon incorporates communication optimizations that enable it to scale to a large number of hosts. D-Galois already supports edge operators and vertex-cuts. We adapted D-Galois to handle dynamic graph generation efficiently during computation and communication. as explained in Sections III-B and III-D respectively.

B. Graph Generation and Partitioning

As explained in Section II-B, the Skip-gram model generates positive and negative samples using randomization. Consequently, the samples or edges generated for the same element or vertex in the corpus in different epochs may be different. As the same edge may not be generated again, one way to abstract this is to consider that the edges are being streamed and each edge is processed only once, even across epochs. Due to this, the graph needs to be constructed dynamically in each synchronization round, as shown in Algorithm 1 (line 8).

The graph can be explicitly constructed in each round. However, this may add unnecessary overheads as each edge is processed only once before the graph is destroyed. More importantly, this does not distinguish between edges (samples) from different occurrences of the same vertex (element) in the corpus. Consequently, the relative ordering of the edges is not preserved. We observed that the accuracy of the model is highly sensitive to the order in which the edges are processed because the learning rate decays after each occurrence of the vertex is processed. Hence, the key to our graph formulation is

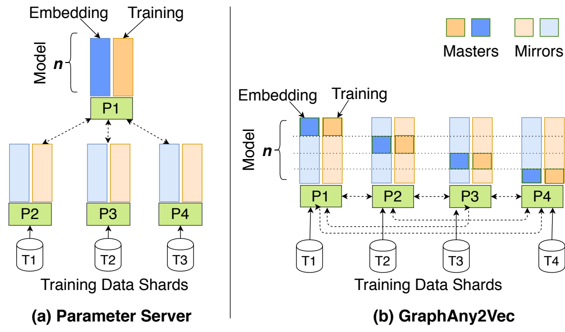

Fig. 2. Synchronization of the Skip-gram Model.

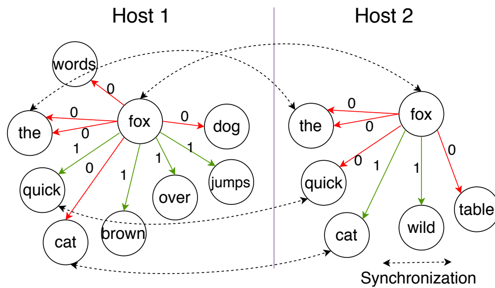

Fig. 3. Synchronization of proxies in graph partitions.

that on each host, *the schedule of applying operators on edges in GraphAny2Vec must match the order in which samples would be processed in Any2Vec*. Note that the work-list W preserves the ordering of vertex occurrences in the corpus. Thus, GraphAny2Vec generates or streams edges on-the-fly using partition s of W in round s, instead of constructing the graph.

Each host generates edges for its own partition of the data corpus in W_s in each synchronization round. In other words, the graph is re-partitioned in every round. Each host owns the edges generated in a given synchronization round. As mentioned in Section II-C, vertex proxies are created for the endpoints of edges on a host. The *master* proxy for each vertex can be chosen from among its proxies thus created, but this would incur overheads in every round. We instead logically partition the vertices (or unique words in case of Word2Vec) once into roughly equal contiguous chunks among the hosts and each host creates master proxies for the vertices in its partition; this is also referred to as the *model partitioning* in the literature. Master proxies do not change across rounds or epochs. Proxies for other vertices on the host would be *mirror* proxies. Each mirror knows the host that has its master using the static partitioning of vertices. Each master also needs to know the hosts with its dynamic mirrors as the edges change in each round. At the end of each round, the updates to mirrors are sent to the master proxy to be reduced. Section III-C and III-D discuss the update exchange schemes and reduction operators.

C. Model Synchronization

Figure 3 shows an example where proxies on two hosts need to be synchronized after computation. Consider the word "*fox*" that is present on both hosts. It may have different values for the embeddings on both hosts after computation. Prior works for distributed Word2Vec [10]–[13] such as Microsoft's Distributed Machine Learning Toolkit (DMTK) [10] (and other machine learning algorithms) use a parameter server to synchronize the model, as illustrated in Figure 2(a). One of the hosts (say P_1) is chosen as the parameter server which holds the master proxy of the vectors. Other hosts asynchronously receive the updated model from the server and update their model locally based on the assigned computation. Finally, each host independently sends its local model to the P_1 to update the global model.

GraphAny2Vec, however, uses a different synchronization model based on D-Galois [17], as illustrated in Figure 2(b). Abstractly, this can be viewed as a decentralized parameter server model where each host acts as a parameter server for a partition of the model and the updates occur in a bulk-synchronous fashion. In Figure 2(b), P_i has the master proxies for the ith partition of the vertices (model). Dividing the model equally among hosts and making them responsible for maintaining the canonical state of their partitions of the model aids in *load-balancing communication* among hosts. As described in Section III-B the local updates to mirrored proxies are sent to the host with the master proxy to be reduced. This is more efficient than parameter server in modern interconnects because one server host does not become the bottleneck in communication.

Different hosts may update the vectors for the same vertex in the same round and therefore, the host with the master proxy needs to reduce multiple updates into one. Prior works for distributed Word2Vec use averaging but GraphAny2Vec leverages the AdaSum [18] reduction operator as it allows us to match the accuracy of shared-memory implementations.

The AdaSum paper shows that the optimal way to perform such distributed reductions for all parameters in the model is by using vector-halving distance-doubling technique [24]. However, in GraphAny2Vec, not all vectors are updated in every round. Consequently, such communication would result in redundant communication. As described above, in GraphAny2Vec, the master is responsible for applying the updates from all mirrors. Suppose that in a synchronization round, $u_0 = g_0$ is the value of an embedding on the master proxy and $g_1, \ldots, g_n$ are the set of updates received from its n mirror proxies. Then the updates are reduced sequentially on that host in the following way: $\forall 1 \leq i \leq n, u_i = AdaSum(u_{i-1}, g_i)$.

D. Communication Schemes and Optimizations

We provide two major schemes to exchange updates between mirror and master proxies with communication optimization for each.

RepModel-Base: In this scheme, each host has proxies for all vertices, so the entire model is replicated on each host. Thus, each host statically knows that every other host has

mirror proxies for the masters on it. This means that at the beginning of a round, each host sends the master proxy of each vertex to every other host and at the end of the round, each host sends back its mirror proxies to the host with master proxy to update. Obviously, this approach has an overwhelming volume of communication as a host may not need the update for a mirror proxy in a round.

RepModel-Opt: This is the same scheme as RepModel-Base with a communication optimization. In this version, each host maintains a bit-vector for the mirror proxies updated in each round. Note that from Equation 3, for edge corresponding to src and dst, only e_{src} and t_{dst} are updated and e_{dst} and t_{src} are neither read nor updated. Therefore, we need two bits for each vertex, one for each vector. Synchronization in D-Galois uses this bit-vector to ensure that only the updated mirrors are sent to their masters and the masters broadcast their values to all other hosts only if it was updated on any host in that round.

PullModel-Base: In this scheme, each host makes an inspection pass over W_s (its work-list) before computation in each round to track the vertices that would be accessed during computation. Mirror proxies are then created for the vertices tracked and the master proxy communicates both vectors to the mirror proxy. Then each host updates its vectors and communicates back the vectors of tracked vertices.

PullModel-Opt: As discussed in RepModel-Opt, e_{src} are accessed only at the src of an edge and t_{dst} are accessed only at the destination. If a mirror proxy on a host has only outgoing (or incoming) edges, then it will not access t_{dst} (or e_{src}). This is not exploited in PullModel-Base because masters and mirrors are not label-specific in D-Galois (labels are 2 vectors e and t). We modified D-Galois to maintain masters and mirrors specific to each label. We also modified our inspection phase to track sources and destinations separately, and create mirrors for e_{src} and t_{dst} respectively. Due to this, masters will broadcast e and t only to those hosts that access it.

Generally, RepModel schemes requires the entire model to fit in the memory of a host and the entire model is updated on each vertex for every round which requires a huge volume of communication. However, PullModel allows larger models with limited required communication for each round at the expense of an inspection pass. We will compare these schemes and optimization in Section IV.

IV. Evaluation

We implement distributed Word2Vec and Vertex2Vec in our GraphAny2Vec framework, and we refer to these applications as GraphWord2Vec (GW2V) and GraphVertex2Vec (GV2V) respectively. Our evaluation methodology is described in detail in Section IV-A. First, we compare GW2V and GV2V with the state-of-the-art third-party implementations (Section IV-B). We then analyze the impact of AdaSum [18] (Section IV-C) and communication optimizations (Section IV-D).

A. Experimental Methodology

Hardware: All our experiments were conducted on the Stampede2 cluster at the Texas Advanced Computing Center using up to 32 Intel Xeon Platinum 8160 ("Skylake") hosts, each with 48 cores with clock rate 2.1Ghz, 192GB DDR4 RAM, and 32KB L1 data cache. Machines in the cluster are connected with a 100Gb/s Intel Omni-Path interconnect. Code is compiled with g++ 7.1 and MPI mvapich2/2.3.

The evaluation in this paper focuses on distributed CPUs because the existing (shared-memory and distributed-memory) Word2Vec and Vertex2Vec frameworks [2], [4], [9], [10] have been implemented and evaluated for CPU architectures. Nevertheless, the techniques in GraphAny2Vec described in Section III are not restricted to CPU architectures. Furthermore, the GraphAny2Vec framework is based on the state-of-the-art distributed and heterogeneous graph analytics framework, D-Galois [17], which supports both CPUs and GPUs. Thus, GraphAny2Vec can be extended to run on GPUs but that is beyond the focus of this work. Due to the limited memory of GPUs, larger GPU clusters may be required to process the bigger datasets considered in this study. However, we expect GraphAny2Vec's online graph generation, model partitioning, and communication optimizations to reduce memory usage and benefit distributed execution on GPUs as well.

TABLE I
DATASETS AND THEIR PROPERTIES.

Applications	Datasets	Vocabulary Words	Training Words	Size	Labels
Word2Vec (Text)	1-billion	399.0K	665.5M	3.7GB	N/A
	news	479.3K	714.1M	3.9GB	N/A
	wiki	2759.5K	3594.1M	21GB	N/A
Vertex2Vec (Graph)	BlogCatalog	10.3K	4.1M	0.02GB	39
	Flickr	80.5K	32.2M	0.18GB	195
	Youtube	1138.5K	455.4M	2.8GB	47

TABLE II
WORD2VEC TRAINING TIME (HOURS) ON A SINGLE HOST COMPARED WITH GW2V(AS) ON 32 HOSTS. SPEEDUP GAINED BY DISTRIBUTED GW2V (AS) ON 32 HOSTS OVER THE STATE-OF-THE-ART SHARED-MEMORY W2V.

	1 host				32 hosts	
Dataset	W2V	GEM	DMTK (AVG)	GW2V (AS)	GW2V (AS)	Speedup
1-billion	4.24	4.39	4.21	3.98	0.32	13x
news	4.45	4.66	4.28	4.51	0.37	12x
wiki	20.49	OOM	25.43	22.34	1.85	11x

Datasets: Table I lists the training datasets used for our evaluation: text datasets for Word2Vec and graph datasets for Vertex2Vec. Prior Word2Vec and Vertex2Vec publications used the same datasets. The wiki (21GB) and Youtube (2.8GB) datasets are the largest text and graph datasets respectively. We used DeepWalk [4][3] for generating training corpus for Vertex2Vec by performing 10 random walks each of length 40 from all vertices of the graph. We limit our Word2Vec and Vertex2Vec evaluation to 32 and 16 hosts of Stampedes respectively because these datasets do not scale beyond that. For all frameworks and datasets, the preprocessing steps like

[3] https://github.com/phanein/deepwalk

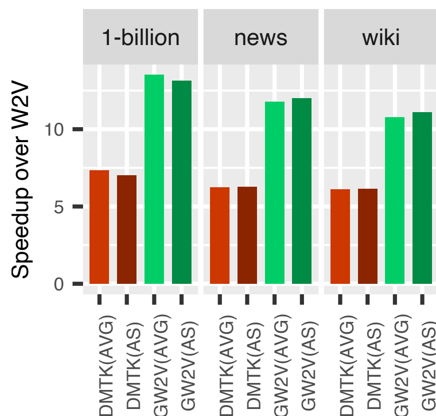

Fig. 4. Speedup of DMTK and GW2V on 32 hosts over W2V on 1 host.

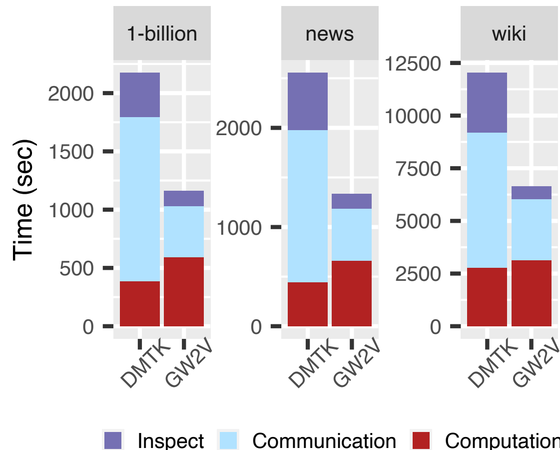

Fig. 5. Breakdown of time of DMTK(AS) and GW2V(AS) on 32 hosts.

TABLE III
WORD2VEC ACCURACY (SEMANTIC, SYNTACTIC, AND TOTAL) OF DIFFERENT FRAMEWORKS RELATIVE TO W2V. BLUE (FIRST ROW FOR EACH ACCURACY TYPE) INDICATES THE BASELINE ACCURACY ALONG WITH THE TOLERANCE BOUNDS. GREEN AND RED INDICATE ACCURACY CHANGE $\geq$ AND $<$ THE TOLERANCE LOWER BOUND RESPECTIVELY.

	Framework	1-billion	news	wiki
Semantic	W2V (1 Host)	75.86±0.07	70.79±0.54	79.10±0.31
	GEN (1 Host)	-0.22	-0.22	OOM
	DMTK (1 Host)	-13.79	-18.43	-7.46
	DMTK(AVG) (32 Hosts)	-57.36	-57.15	-34.39
	DMTK(AS) (32 Hosts)	-10.93	-17	-5.17
	GW2V (1 Host)	+0.07	-0.08	+0.26
	GW2V(AVG) (32 Hosts)	-7.00	-9.15	-4.03
	GW2V(AS) (32 Hosts)	+0.21	+0.07	-0.17
Syntactic	W2V (1 Host)	50.0±0.18	50.0±0.26	49.22±0.12
	GEN (1 Host)	-0.14	-0.12	OOM
	DMTK (1 Host)	-1.89	-0.67	-3.11
	DMTK(AVG) (32 Hosts)	-24.89	-25.11	-23.11
	DMTK(AS) (32 Hosts)	-3.56	-1.78	-1.44
	GW2V (1 Host)	-0.37	0.0	-0.12
	GW2V(AVG) (32 Hosts)	-4.89	-4.11	-7.55
	GW2V(AS) (32 Hostss)	+0.10	+0.11	+0.18
Total	W2V (1 Host)	72.36±0.21	69.21±0.42	74.10±0.42
	GEN (1 Host)	+0.0	-0.14	OOM
	DMTK (1 Host)	-11.65	-15.42	-3.03
	DMTK(AVG) (32 Hosts)	-51.29	-51.71	-32.03
	DMTK(AS) (32 Hosts)	-9.86	-14.78	-5.24
	GW2V (1 Host)	-0.14	-0.28	+0.1
	GW2V(AVG) (32 Hosts)	-6.79	-9.28	-5.17
	GW2V(AS) (32 Hostss)	+0.14	+0.29	-0.17

vocabulary building take $< 5\%$ of the execution time and can be amortized. Thus, we report the accuracy and the training (or execution) time for all frameworks on these datasets, excluding preprocessing time, as an average of three distinct runs.

Shared-memory third-party implementations: We evaluated the Skip-gram [2] (with negative sampling) training model for both Word2Vec and Vertex2Vec. We compared GraphWord2Vec (GW2V) with the state-of-the-art shared-memory Word2Vec implementations, the original C implementation (W2V) [2] as well as the more recent Gensim (GEN) [9] python implementation. We also compared our GraphVertex2Vec (GV2V) with the state-of-the-art shared-memory Vertex2Vec framework, DeepWalk [4] (both DeepWalk and Node2Vec [5] use Gensim's [9] Skip-gram model).

Distributed-memory third-party implementations: We compared GraphWord2Vec with the state-of-the-art distributed-memory Word2Vec from Microsoft's Distributed Machine Learning Toolkit (DMTK) [10], which is based on the parameter server model. The model is distributed among parameter server hosts. During execution, hosts acting as workers request the required model parameters from the servers and send model updates back to the servers. Each host in the cluster acts as both server and worker, and it is the only configuration possible. DMTK uses OpenMP for parallelization within a host, while GW2V uses Galois [23] for parallelization within a host. Both GW2V and DMTK use MPI for communication between hosts. We modified DMTK to include a runtime option of configuring the number of synchronization rounds. DMTK uses averaging as the reduction operation to combine the gradients. We refer to this as DMTK(AVG). We also implemented AdaSum [18] in DMTK and we call this DMTK(AS). There are no prior distributed implementations of Vertex2Vec. Unless otherwise specified, GW2V and GV2V use AS to combine gradients and use PullModel-Opt communication optimization.

Hyper-parameters: We used the hyper-parameters suggested by [2], unless otherwise specified: window size of 5, number of negative samples of 15, sentence length of 10K, threshold of 10^{-4} for Word2Vec and 0 for Vertex2Vec for down-sampling the frequent words, and vector dimensionality N of 200. All models were trained for 16 epochs. For distributed frameworks, GV2V, GW2V, and DMTK, we compared 2 gradient combining methods: Averaging (AVG) and AdaSum (AS) [18] method. Synchronization rounds are increased linearly (roughly) with the number of hosts in order

to maintain the desired accuracy (discussed in Section IV-C). Therefore, Unless otherwise specified, GV2V, GW2V, and DMTK use the same number of synchronization rounds: 1 for 1 host, 3 for 2 hosts, 6 for 4 hosts, 12 for 8 hosts, 24 for 16 hosts, and 48 for 32 hosts. Note that DMTK uses only 1 synchronization round by default, but this yields much lower accuracy, so we do not report these results.

Accuracy: In order to measure the accuracy of trained models of Word2Vec on different datasets, we used the analogical reasoning task outlined by Word2Vec [2]. We evaluated the accuracy using scripts and question-words.txt provided by the Word2Vec code base[4]. Question-words.txt consists of analogies such as "Athens" : "Greece" :: "Berlin" : ?, which are predicted by finding a vector x such that embedding vector(x) is closest to embedding vector("Athens") - vector("Greece") + vector("Berlin") according to the cosine distance. For this particular example the accepted value of x is "Germany". There are 14 categories of such questions, which are broadly divided into 2 main categories: (1) the syntactic analogies (such as "calm" : "calmly" :: "quick" : "quickly") and (2) the semantic analogies such as the country to capital city relationship. We report semantic, syntactic, and total accuracy averaged over all the 14 categories of questions. For Vertex2Vec, we measured *Micro-F1* and *Macro-F1* scores using scoring scripts provided by DeepWalk [4][5].

B. Comparing With The State-of-The-Art

Word2Vec: We compare GW2V with distributed-memory implementation DMTK and shared-memory implementations, W2V and GEN. Table II compares their training time on a single host. Figure 4 shows the speedup of both GW2V and DMTK on 32 hosts over W2V on 1 host. Note that AVG and AS methods are used to combine gradients during inter-host synchronization, so they have no impact on a single host.

Performance: We observe that for all datasets on a single host, the training time of GW2V is similar to that of W2V, GEN, and DMTK. GW2V scales up to 32 hosts and speeds up the training time by $\approx 13\times$ on average over 1 host. GW2V is $\approx 2\times$ faster on average than DMTK on 32 hosts. Figure 4 also shows that there is negligible performance difference between using AVG and using AS to combine gradients in both DMTK and GW2V. Training wiki using GW2V takes only **1.9 hours**, which saves **18.6 hours** and **1.5 hours** compared to training using W2V and DMTK respectively.

To understand the performance differences between DMTK and GW2V better, Figure 5 shows the breakdown of their training time into 3 phases: inspection, (maximum) computation, and (non-overlapped) communication. Firstly, GW2V's inspection phase as well as serialization and de-serialization during synchronization are parallel using D-Galois parallel constructs and concurrent data-structures [17], [23], whereas these phases are sequential in DMTK as it uses non-concurrent data-structures such as set and vector provided by the C++ standard template library. Moreover, in GW2V, hosts can update their masters in-place. This is not possible in DMTK as workers on each host have to fetch model parameters from servers on the same host to update, incurring overhead for additional copies. Secondly, DMTK communicates much higher volume ($\approx 3.5\times$) than GW2V. GW2V memoizes [17] the vertex IDs exchanged during inspection phase and exchanges only the required values. In contrast, DMTK sends the vertex IDs along with the updated values to the parameter servers before and after the communication. In addition, GW2V's inspection precisely identifies both the positive and negative samples required for the current round. DMTK, on the other hand, only identifies precise positive samples, and builds a pool for negative samples. During computation, negative samples are randomly picked from this pool. The entire pool is communicated from and to the parameter servers, although some of them may not be updated, leading to redundant communication.

Accuracy: Table III compares the accuracies (semantic, syntactic, and total) for all frameworks on 1 and 32 hosts relative to the accuracies achieved by W2V. On a single host, GW2V is able to achieve accuracies (semantic, syntactic and total) comparable to W2V. DMTK on a single host is less accurate due to implementation differences in the Skip-gram model training: DMTK only updates learning rate between mini-batches, whereas others continuously degrade learning rate, and DMTK uses a different strategy to choose negative samples as described earlier. On 32 hosts, DMTK(AVG) has a much lower accuracy than the baseline while GW2V(AVG) has a better accuracy than DMTK(AVG) but still subpar compared to the baseline. AS significantly improves the accuracies

TABLE IV
VERTEX2VEC TRAINING TIME (SEC) OF DEEPWALK ON 1 HOST VS. GRAPHVERTEX2VEC (GV2V) ON 16 HOSTS.

Dataset	DeepWalk	GV2V	Speedup
BlogCatalog	115.3	28.8	4.0x
Flickr	976.7	183.1	5.3x
Youtube	11589.2	2226.2	5.2x

TABLE V
VERTEX2VEC ACCURACY (MACRO F1 AND MICRO F1) OF GV2V ON 16 HOSTS RELATIVE TO DEEPWALK ON 1 HOST (NOTE: THE SCORING SCRIPTS PROVIDED BY DEEPWALK WERE NOT ABLE TO HANDLE THE YOUTUBE DATASET).

		% Labeled Vertices	30%	60%	90%
Micro-F1	BlogCatalog	Deepwalk	34.0	37.2	38.4
		GV2V	-0.1	+0.1	+0.7
	Flickr	Deepwalk	38.6	40.4	41.1
		GV2V	+0.1	-0.1	-0.2
Macro-F1	BlogCatalog	Deepwalk	34.1	37.2	38.4
		GV2V	-0.3	+0.1	+0.7
	Flickr	Deepwalk	26.5	28.7	29.5
		GV2V	+0.1	-0.1	+0.3

[4]https://github.com/tmikolov/word2vec

[5]https://github.com/phanein/deepwalk/blob/master/example_graphs/scoring.py

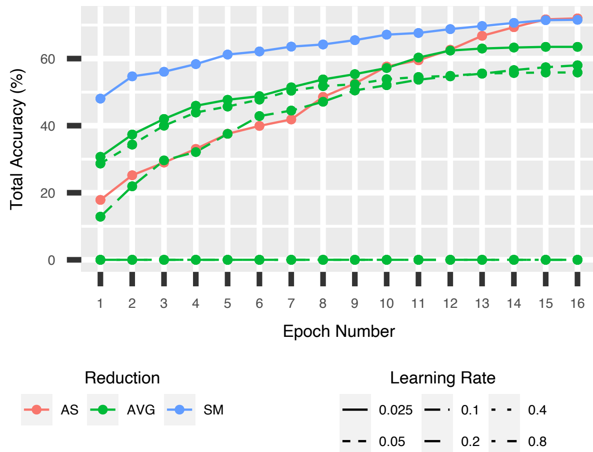

Fig. 6. Accuracy of GW2V after each epoch for 1-billion dataset on 1 host (SM) and on 32 hosts using AS and AVG.

over AVG for both DMTK and GW2V. DMTK(AS) matches its own single host accuracy and GW2V(AS) matches the accuracy of W2V.

Vertex2Vec: Table IV compares the training time of DeepWalk [4] on a single host with our GV2V on 16 hosts. We observe that for all datasets, GraphVertex2Vec can train the model $\approx 4.8\times$ faster on average. Similar to GraphWord2Vec, this speedup does not come at the cost of the accuracy, as shown in Table V, which shows the *Micro-F1* and *Macro-F1* score with 30%, 60%, and 90% labeled vertices.

Discussion: GraphAny2Vec significantly speeds up the training time for Word2Vec and Vertex2Vec applications by distributing the computation across the cluster without sacrificing the accuracy. Reduced training time also accelerates the process of improving the training algorithms as it allows application designers to make more end-to-end passes (with different hyper-parameters) in a short duration of time.

C. AdaSum Reduction Operator in GraphAny2Vec

If time were not an issue, all machine learning algorithms would run sequentially. A sequential SGD is simple to tune and converges fast. Unfortunately, it is slow. A point (x, y) on Figure 6 denotes the total accuracy (y) as a function of epoch (x). The blue line (SM) shows the accuracy of GW2V on a single shared-memory host. It clearly converges to a high accuracy quickly. In contrast, the green lines plot accuracy of distributed GW2V that uses averaging the gradients (AVG) with different learning rates on 32 hosts. The learning rate of 0.025 is the same as SM while the learning rate of 0.8 is 32 times larger. The former converges slowly while the latter does not converge at all (accuracy is 0) because the learning rate is too large. As it can also be observed, none of the other learning rates have an on-par accuracy with averaging. Finally, the red line plots accuracy of distributed GW2V that uses AS and 0.025 as the learning rate on 32 hosts. AS has no problem meeting the accuracy of the sequential algorithm. In addition to providing the same accuracy as SM, it is $\approx 12\times$ times faster on 32 hosts than SM (as shown in Table II). Not having to tune the learning rate and still getting accuracy at scale is made possible by AdaSum.

Synchronization Rounds: AdaSum (AS) improves the accuracies significantly but in order to get accuracies comparable to shared-memory implementations, the number of synchronization rounds in each epoch is an important knob to tune. We observe that accuracies improve as we increase the number of synchronization rounds within an epoch for both AS and AVG. Nonetheless, accuracies show more improvement for AS (for example, semantic: 3.07%, syntactic: 3.99% and total: 3.36% when synchronization rounds are increased from 12 to 48 on 32 hosts) as opposed to AVG, which shows very little change in accuracies with synchronization rounds. In general, we have observed that in order to maintain the desired accuracy, the synchronization rounds needs to be increased (roughly) linearly with the number of hosts. We have followed this rule of thumb in all our experiments.

D. Impact of Communication Optimizations

Figure 7 compares the strong scaling of GW2V with RepModel-Base (RMB), RepModel-Opt (RMO), PullModel-Base (PMB), and PullModel-Opt (PMO). Each of the optimized variants scale well for all datasets but the performance differences between them increase with more hosts. This is due to two reasons: (a) synchronization rounds doubles with the number of hosts, thus communicating more data, and (b) as training data or corpus gets divided among more hosts, sparsity in the updates increase. Figure 8 shows the breakdown of the training time into inspection, (maximum) computation, and (non-overlapped) communication time on 32 hosts.

RepModel-Opt, which avoids communicating untouched vectors reduces communication volume by $2.5\times$ and is 16% faster on average than RepModel-Base on 32 hosts. This shows that RepModel-Opt is able to exploit the sparsity in the communication. In contrast to RepModel-Opt, PullModel-Base only communicates the vectors that are needed in a round. This is done through an inspection pass which by itself has a compute and a communication overhead. On 32 hosts, PullModel-Base sends $1.3\times$ more communication volume on average than RepModel-Opt. PullModel-Opt enhances the inspection pass to only communicate half of the vectors as explained in Section III-D. It is 14%, 22%, and 34% faster on average than PullModel-Base, RepModel-Opt, and RepModel-Base respectively. It also reduces communication volume by $1.5\times$, $1.1\times$, and $2.7\times$, respectively.

Summary: PullModel-Opt in GraphAny2Vec always performs better than the other variants due to the enhanced inspection phase which reduces communication volume exchanged among hosts at the cost of slightly increased bookkeeping. These improvements are expected to grow as we scale to bigger datasets and number of hosts. Hence, PullModel-Opt not only allows one to train bigger models, but also gives the best performance.

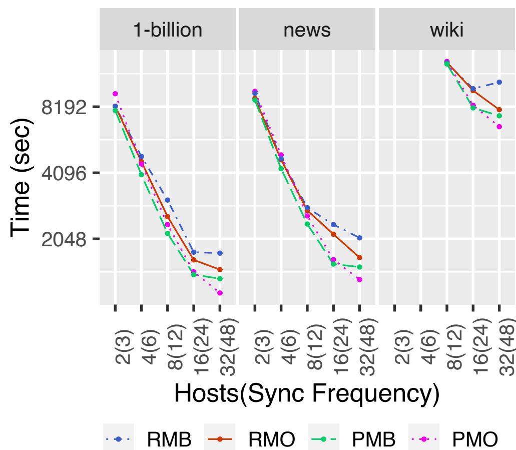

Fig. 7. Strong scaling of GW2V (RMB: RepModel-Base, RMO: RepModel-Opt, PMB: PullModel-Base, PMO: PullModel-Opt).

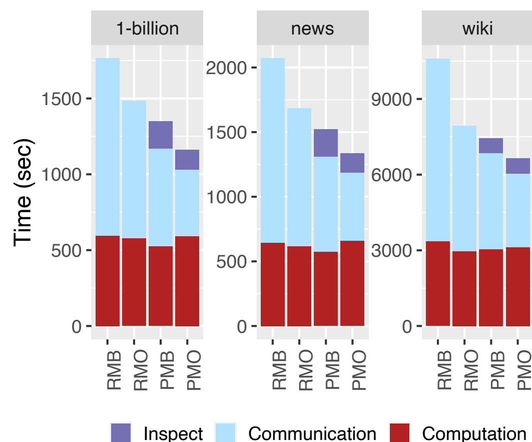

Fig. 8. Breakdown of training time of GW2V with different communication schemes on 32 hosts.

V. Related Work

Training models: Mikolov et al. [1] proposed two simple model architectures for computing continuous vector representations of words from very large unstructured data sets, known as Continuous Bag-of-Words (CBOW) and Skip-gram (SG). These models removed the non-linear hidden layer and hence avoid dense matrix multiplications, which was responsible for most of the complexity in the previous models such as LSA [25] and LDA. CBOW is similar to the feedforward Neural Net Language Model (NNLM) [26], where the non-linear hidden layer is removed and the projection layer is shared for all words. SG on the other hand unlike CBOW, instead of predicting the current word based on the context, tries to maximize classification of a word based on another word within a sentence. Later Mikolov et al. [2] further introduced several extensions, such as using hierarchical softmax instead of full softmax, negative sampling, subsampling of frequent words, etc., to SG model that improves both the accuracy and the training time. SG has also been adopted in other domains to represent vertices of a graph/network (DeepWalk [4] and Node2Vec [5]), genome sequences in bio-informatics (Sequence2Vec [8]), and documents (Doc2Vec [3]).

Shared-memory frameworks: Our work adapts the algorithm from Mikolov et al. [2] for distribution. This work, together with many current implementations [9] are designed to run on a single machine but utilizing multi-threaded parallelism. Our work is motivated by the fact that these popularly used implementations take days or even weeks to train on large training corpus. GraphAny2Vec is able to significantly speedup the training process while maintaining the accuracy comparable to the state-of-the-art shared-memory frameworks.

Distributed-memory frameworks: Prior works on distributing Word2Vec either use synchronous data-parallelism [11]–[13], [27] or parameter-server style asynchronous data parallelism [10], [28]. However, they perform communication after every mini-batch, which is prohibitively expensive in terms of network bandwidth. Our design was motivated by the need to use commodity machines and network available on public clouds. Our approach communicates infrequently and uses AdaSum [18] to overcome the resulting staleness.

Ordentlich et al. [27] can handle models that do not fit on a single machine in a cluster by vertically partitioning the embedding and training vectors and assigning parts of it to different machine. These partitions compute partial dot products locally but communicate to compute global dot products. However, in order to compute partial dot products, all hosts have to process the same training data at a given instance, whereas, in GraphAny2Vec, each host can independently process different chunks of the training data corpus in parallel. Therefore, our design not only allows for horizontal partitioning of large models that do not fit in memory, but also enables parallel processing of large corpses of data.

Big data analytics frameworks like Spark [29]) and machine learning frameworks like TensorFlow [30] and DMTK [10] are sufficiently general to support Any2Vec applications. Consequently, there are Word2Vec implementations in Spark [12], [27], TensorFlow [28], and DMTK [10]. These implementations cannot be adapted to support the graph-specific optimizations and Any2Vec-specific optimizations introduced in this paper. That is why GraphAny2Vec outperforms and scales even better than DMTK which is the state-of-the-art for such problems. Thus, the generality of these frameworks comes at the cost of performance. On the other hand, traditional HPC programming models like MPI [31] can also be used to implement Any2Vec like the Word2Vec implementation [13] that uses MPI. Optimizations introduced in this paper can also be implemented with MPI but they would require a significant effort to support a new Any2Vec application as MPI is too low-level.

Our formulation of Any2Vec as a graph problem enables Any2Vec applications to be implemented in existing graph frameworks. Any2Vec requires (1) edge operators, (2) vertex-cut partitioning policies, and (3) *dynamic* graph re-partitioning, but none of the existing distributed graph frameworks [14]–

[17] support all these features. Gemini [16] does not support edge operators or vertex-cut partitioning policies. PowerGraph [14] supports vertex-cuts but does not support edge operators. TuX^2 [15] is a graph engine that supports subset of distributed machine learning applications by extending the Gather-Apply-Scatter model [14]. D-Galois [17] is the state-of-the-art graph engine that optimizes communication based on invariants in the graph partitioning policies. Both TuX^2 and D-Galois support edge operators and vertex cuts, but they do not support dynamic graphs. GraphAny2Vec extends D-Galois to support dynamic graph generation and re-partitioning. It is the first distributed graph framework to support Any2Vec and it introduces communication optimizations specific to Any2Vec applications.

VI. Conclusions

Through GraphAny2Vec, we show that popular Any2Vec ML applications can be mapped to graph applications. GraphAny2Vec highlights and implements the extensions required in the current state-of-the graph analytics framework to support Any2Vec applications, namely dynamic graph generation, graph re-partitioning, and communication optimization.

GraphAny2Vec substantially speeds up the training time for Any2Vec applications by distributing the computation across the cluster without sacrificing the accuracy. Reduced training time also accelerates the process of improving the training algorithms as it allows application designers to make more end-to-end passes in a short duration of time. GraphAny2Vec thus enables more explorations of Any2Vec applications in areas such as natural language processing, network analysis, and code analysis.

References

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space."

[2] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," ser. NIPS'13, USA.

[3] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," ser. ICML'14. JMLR.org, 2014.

[4] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," ser. KDD '14. New York, NY, USA: ACM.

[5] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," ser. KDD '16. New York, NY, USA: ACM.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019.

[7] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2018.

[8] H. Dai, R. Umarov, H. Kuwahara, Y. Li, L. Song, and X. Gao, "Sequence2Vec: a novel embedding approach for modeling transcription factor binding affinity landscape," *Bioinformatics*, vol. 33, 2017.

[9] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *LREC 2010*.

[10] D. M. L. Toolkit, http://www.dmtk.io/.

[11] word2vec in Deeplearning4j, https://deeplearning4j.org/docs/latest/deeplearning4j-nlp-word2vec.

[12] word2vec in Spark, spark.apache.org/docs/latest/mllib-feature-extraction.html.

[13] S. Ji, N. Satish, S. Li, and P. K. Dubey, "Parallelizing word2vec in shared and distributed memory," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," ser. OSDI'12, CA, USA.

[15] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux2: Distributed graph computation for machine learning," ser. NSDI'17. Berkeley, CA, USA: USENIX Association, 2017.

[16] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A Computation-centric Distributed Graph Processing System," ser. OSDI'16, CA, USA.

[17] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," *SIGPLAN Not.*, 2018.

[18] S. Maleki, M. Musuvathi, T. Mytkowicz, O. Saarikivi, T. Xu, V. Eksarevskiy, J. Ekanayake, and E. Barsoum, "Scaling distributed training with adaptive summation," 2020.

[19] L. Bottou, *Stochastic Gradient Descent Tricks*, ser. Lecture Notes in Computer Science (LNCS). Springer, January 2012.

[20] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning," *arXiv e-prints*, Jun 2016.

[21] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NeurIPS 24*, 2011.

[22] L. Hoang, R. Dathathri, G. Gill, and K. Pingali, "CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics," in *IPDPS*, 2019.

[23] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," ser. SOSP '13. New York, NY, USA: ACM.

[24] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, 2005.

[25] T. Mikolov, S. W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *NAACL-HLT-2013*.

[26] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, Mar. 2003.

[27] E. Ordentlich, L. Yang, A. Feng, P. Cnudde, M. Grbovic, N. Djuric, V. Radosavljevic, and G. Owens, "Network-efficient distributed word2vec training system for large vocabularies," in *CIKM 2016*.

[28] W. in TensorFlow, https://www.tensorflow.org/tutorials/text/word2vec.

[29] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2934664

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[31] M. P. Forum, "Mpi: A message-passing interface standard," USA, Tech. Rep., 1994.