Fast and Accurate Learning of Knowledge Graph Embeddings at Scale

Udit Gupta
Department of Computational and Data Sciences
Indian Institute of Science
Bangalore, India
uditgupta@iisc.ac.in

Sathish Vadhiyar
Department of Computational and Data Sciences
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India
vss@iisc.ac.in

***Abstract*—Knowledge Graph Embedding (KGE) is used to represent the entities and relations of a KG in a low dimensional vector space. KGE can then be used in a downstream task such as entity classification, link prediction and knowledge base completion. Training on large KG datasets takes a considerable amount of time. This work proposes three strategies which lead to faster training in distributed setting. The first strategy is a *reduced communication* approach which decreases the All-Gather size by sparsifying the Sparse Gradient Matrix (SGM). The second strategy is a *variable margin* approach that takes advantage of reduced communication for lower margins but retains the accuracy as obtained by the best fixed margin. The third strategy is called *DistAdam* which is a distributed version of the popular Adam optimization algorithm. Combining the three strategies results in reduction of training time for the FB250K dataset from twenty-seven hours on one processing node to under one hour on thirty-two nodes with each node consisting of twenty-four cores.**

***Index Terms*—Knowledge graph embeddings, distributed learning, Horovod.**

I. Introduction

A. KG and KGEs

A Knowledge Graph (KG) is a multi-relational graph composed of entities as nodes and relations as edges. Each instance in the KG is represented as a triple also called a fact. As the name suggests, the triple contains three items out of which two are entities and one is relation. The most commonly used format is *head_entity, relation, tail_entity*; although the format *subject, predicate, object* is also used. An example of such a triple is *Bangalore, cityIn, Karnataka* where *cityIn* is a relation connecting the two entities *Bangalore* and *Karnataka*. KGEs are a way to represent the entities and relations present in the graph as a low dimensional vector embedding. The purpose of this representation is that the embeddings can be used in downstream tasks such as entity classification, link prediction and knowledge base completion.

B. Distributed Training (Machine Learning)

As the datasets become larger and the computational costs of the algorithms increase, there is an increasing need for distributed training of the model for improved scalability and to explore large problems. The parameter server approach for distributed training was proposed in [1]. In this approach the data and computation is distributed among the workers and the parameter server maintains the shared parameters. Training data is divided into shards and each shard is handled by a worker. Each worker calculates gradients with respect to the shard allotted to it and sends them to the parameter server. The parameter server aggregates gradients from all the workers and updates the shared parameters accordingly.

The parameter server approach though widely used and popular has the disadvantage of network bottleneck since many workers communicate to few servers. Due to this disadvantage, a new architecture called *Ring-AllReduce* was proposed by [2]. The working of *Ring-AllReduce* is as follows, the data to be aggregated is divided into P chunks therefore the size of each chunk is $\frac{N}{P}$ where P is the number of processes and N is the size of data within a process. Each process P sends its P^{th} chunk to the $(P+1)^{th}$ process and receives $(P-1)^{th}$ chunk from $(P-1)^{th}$ process, then it performs reduction on $(P-1)^{th}$ chunk. In the next iteration, it sends the reduced chunk to the next process and so on. After repeating the steps of *Receive-Reduce-Send* P-1 times each process holds one of the reduced chunks. The whole process has to be repeated (P-1) more times (without reduction operation) such that each process obtains all the reduced chunks.

In the case of machine learning, gradients calculated by different workers need to be aggregated before updating the parameters. The success of ring-allreduce algorithm in reducing communication bottleneck prompted its use in distributed machine learning and served as motivation for Horovod [3] which is currently the state-of-the-art framework for distributed training. In our work we use Horovod with Tensorflow to parallelize the KGE learning process. In this work, we propose three efficient strategies for distributed learning of KGEs using Horovod. The first strategy is *Reduced communication approach* which decreases the All-Gather size by sparsifying the Sparse Gradient Matrix (SGM). The second strategy is *Variable margin approach* that takes advantage of reduced communication for lower margins but retains the accuracy as obtained by the best fixed margin. The third strategy is *DistAdam* which is a distributed version of the popular Adam optimization algorithm [4]. This strategy suggests changes that need to be made to hyperparameters including learning rate for fast and accurate optimization in distributed mode. *DistAdam* is proposed as a general distributed optimization algorithm

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00030

which is applicable for all machine learning problems.

Combining the three strategies results in reduction of training time from twenty-seven hours on one processing node to under one hour on thirty-two nodes for FB250K dataset. The speedup was obtained without compromising the model accuracy.

Section II describes related work in distributed training of machine learning models. Section III provides background on KGE models and distributed learning using the Horovod framework. This section also provides profiling results highlighting the performance bottlenecks with the baseline Horovod strategy. In Section IV, all the three strategies for performance improvement of distributed training of KGE models are explained in detail. Section V presents details on experimental setups, and gives results and observations with the proposed strategies. Section VI gives conclusions and future work.

II. Related Work

Zhang et al. [5] proposed a parallel framework to make KGE training faster. The objective of their model was to avoid collisions when the same embedding vector is updated by two different processors. The embedding vectors are updated without synchronizations. This work is applicable only for shared memory systems. ParaGraphE [6] proposes a multi-threaded implementation which reduces the training time, but this approach is also restricted to the shared memory paradigm.

A recent work by Goyal et al. [7] trained a deep learning model (Resnet-50) on Imagenet dataset using 256 workers in one hour, achieving scaling efficiency of 90% while maintaining the accuracy. They used *distributed synchronous SGD* algorithm for multi-worker training, and MPI_Allreduce [8] (an MPI collective operation) for gradient aggregation. Ben-Nun et al. [9] found out that Map-Reduce posed a hindrance to deep neural network specific optimizations and MPI was better suited to implement fine grained parallelism features. Motivated by good performance achieved by MPI based methods of distributed training on deep learning tasks, this work explores the approaches to make KGE training faster in distributed memory systems using Horovod [3].

Various models have been proposed to learn KGEs. One of the common models is the *TransE* [10]. TransE achieved major improvements in terms of accuracy as compared to its predecessors, also it is a less complex model and requires relatively fewer parameters. TransE's success served as motivation for current state-of-the-art models for learning KGEs. *TransH* [11] was proposed to better capture the mapping properties of relations. It is similar to TransE except that the relation is represented by two embeddings instead of one. Each relation is modelled as a hyper-plane along with the translation operation.

Due to the large size of KGs, learning of KGEs takes a considerable amount of time. Our aim is to reduce the training time using distributed training.

III. Background

A. KGE Models

TransE and similar models use a margin based ranking loss function which encourages lower error for the correct triple present in the KG and higher error for the corrupted triple. The corrupted triple is constructed from the correct triple by replacing either the head entity or tail entity with some random entity present in the KG. The parameters, i.e. the low dimensional representation of entities and relations, are initialized randomly and an optimization algorithm such as SGD is used for updating the parameters such that the loss function is minimized.
The basic structure of the loss function is:

$$\mathcal{L} = \sum_{(h,r,t)\in\mathcal{S}} \sum_{(h',r',t')\in\mathcal{S'}_{(h,r,t)}} max[0, f_r(h,t) + \gamma - f_{r'}(h',t')] \tag{1}$$

Here, $\mathcal{S}$ is a set of correct training triples and $\mathcal{S'}_{(h,r,t)}$ is the set of corresponding corrupted triples. $f_r(h,t)$ is the model specific score function and γ is the margin. The correct triple is expected to have a lower score and the corrupted triple is expected to have a higher score. The major differentiating factor among the KGE models is the formulation of score function. In the TransE model, each entity and each relation is represented by a low dimensional vector embedding (of size say 50 or 100). The score function of TransE is given by:

$$f_r(h,t) = d(h + r, t) \tag{2}$$

where $d(.)$ is a distance metric and is generally taken to be the L1 norm.
TransH uses the following score function:

$$f_r(h,t) = d(h_\perp + d_r,\ t_\perp) \tag{3}$$

where $h_\perp = h - w_r^T h w_r$ and $t_\perp = t - w_r^T t w_r$
$h_\perp$ and $t_\perp$ are the projections of embeddings of h and t on the relation specific hyper-plane w_r; d_r is the translation vector for a relation.

B. Accuracy Metrics

We follow standard evaluation protocol for link prediction as was followed by TransE [10]. We present results for three metrics i.e. Mean Rank (avg), Hits@10 (avg) and Triple Classification Accuracy (TCA). Mean rank (tail), for a test triple, is calculated by replacing the tail by every other entity in the KG and calculating the score function; the scores are then ranked in ascending order and the rank of the correct triple is noted. This procedure is repeated for every instance in the test set and the ranks are averaged to get the Mean Rank (tail). Similarly, Mean Rank (head) metric is obtained by replacing the head. Mean Rank (avg) is calculated by averaging the Mean Rank (tail) and Mean Rank (head). For Hits@10 (avg), similar approach of averaging is followed where Hits@10 (tail) and Hits@10 (head) are averaged. The Hits@10 metric is computed by considering the correct entities ranked in the top ten after sorting them according to the score function's output.

The setting described above is termed as *raw*. We also used the *filtered* setting proposed by Bordes et al. [10] in which the corrupted triples occurring in the dataset were not considered for ranking. Another metric used is TCA, which is a binary classification metric which predicts whether the given triple is correct or incorrect (based on the learned KGEs).

C. Distributed Learning of KGE using Horovod: Baseline Strategy and Performance Bottlenecks

Horovod [3] is a distributed training framework which has been shown to be better than parameter server approach for deep learning tasks as it provides a network optimal way of aggregating gradients. Horovod's success in distributed training of deep learning tasks prompted its use in our work as KGE models are essentially single layer learning models.

In Horovod, global averaging of gradients across multiple workers is done using *All-Reduce* or *All-Gather* algorithm. In *All-Reduce*, the gradients corresponding to the whole embedding matrix are averaged and hence it corresponds to dense updates. In *All-Gather*, only active[1] indices and corresponding rows of gradients are collected (gathered) and hence it corresponds to sparse updates. For KGE models, dense updates are not efficient because a batch of input will involve only a certain number of entities and relations (much less than total number of entities and relations). So, applying *All-Reduce* on the whole embedding matrix will lead to unnecessary communication of zeros. In this work, we aggregate gradients using the sparse mode.

In all our experiments, the batch size per worker is fixed at 10,000. When k workers are involved in the training process, then the effective batch size becomes $10,000 \times k$. The gradients are calculated locally by each worker w.r.t to the current batch of training instances and communicated across multiple workers using MPI. A synchronous distributed strategy is followed where all workers maintain the same set of parameters after an update has been made.

We conducted experiments with the above baseline strategy of Horovod to identify potential performance and scalability bottlenecks. The experiments were performed on the FB250K dataset[2]. The learning rate was selected from $\{0.01, 0.001, 0.0001\}$ and margin was selected from $\{1, 2, 3, 4, 5\}$. Optimal values were, learning rate = 0.0001 and margin = 3. For establishing a baseline in distributed setting, *linear scaling rule* was used [12] as this is the widely adopted default strategy when training using multiple workers.

1) Profiling Results: Results in this section and subsequent sections on profiling have been obtained using Tuning and Analysis Utilities (TAU) [13] v2.27.

At any given time, a worker could be in one of the following phases:

1) Local computation phase : Each worker calculates its local gradients corresponding to the batch it is currently handling. This phase also includes the time required for the *All-Gathered* gradients to be applied to their respective parameters.
2) Negotiation phase: Computation time of workers will be different. Therefore, some workers may need to wait for other workers to finish their computations.
3) All-Gather phase: After every worker has completed their computations, each worker communicates its local gradients to every other worker for aggregation.

[1]Here active refers to the entities present in the current batch

[2]The dataset is described in section V-A1.

No. of Nodes	AG Time (hrs)	Other Time (hrs)	Total Time (hrs)	EFC	TCA (%)	MR		Hits@10	
						raw	filter	raw	filter
1	0	27.02	27.02	800	89.06	4883	359	0.245	0.592
2	11.64	14.16	25.80	798	89.45	4909	345	0.250	0.600
4	11.29	9.07	20.36	727	89.42	4911	346	0.248	0.601
8	9.98	7.11	17.09	778	89.58	4920	331	0.252	0.606
16	9.14	5.93	15.07	793	89.73	4951	336	0.253	0.610
32	8.95	4.88	13.83	780	89.75	4953	334	0.254	0.611
64	9.62	4.18	13.80	796	89.73	4965	336	0.252	0.611

TABLE I: Results for Baseline Model

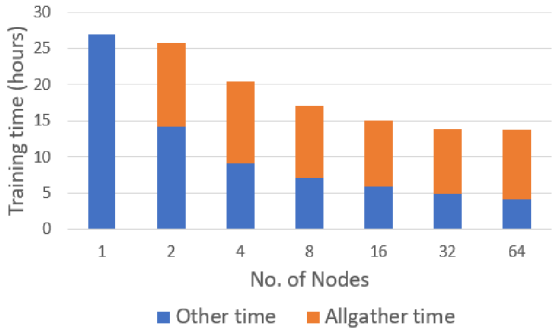


Fig. 1: Breakdown of Total time for multiple nodes (Baseline)

The *Profiling* results on FB250K dataset are presented in Table I. The times have been split up into *All-Gather time* and *Other time*. *Other time* includes the Local computation phase and Negotiation phase as mentioned above. Model is trained till convergence[3]. In Table I *EFC* stands for Epochs for Convergence, *TCA* stands for Triple Classification Accuracy and *AG Time* is the All-Gather Time.

From Figure 1, we observe that though the *Other time* is decreasing, *All-Gather time* remains almost constant which leads to saturation. From Table I we observe that the accuracy of model trained in distributed mode is at par with the accuracy obtained using single node training. The drawback of *baseline* is that the model takes too many epochs to converge which leads to longer training time.

Thus, the following improvements need to be made to eliminate the performance and scalability bottlenecks in the baseline multi node strategy. All-Gather time needs to be reduced as it forms a significant percentage of overall time for higher number of nodes. In order to decrease the total training time we formulate an accurate distributed optimization algorithm which takes fewer epochs for convergence.

[3]convergence criteria mentioned in section V-A3

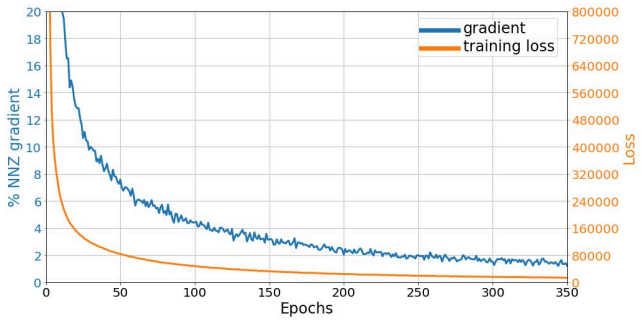


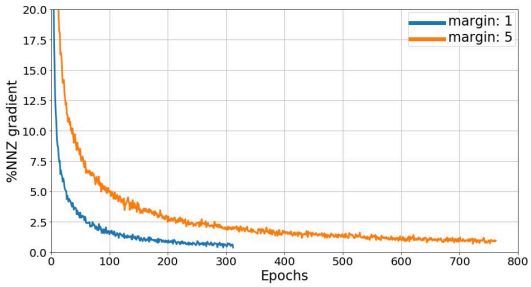
Fig. 2: Similarity between %NNZ rows in SGM and Loss value

IV. Strategies for Scalable KGE Learning

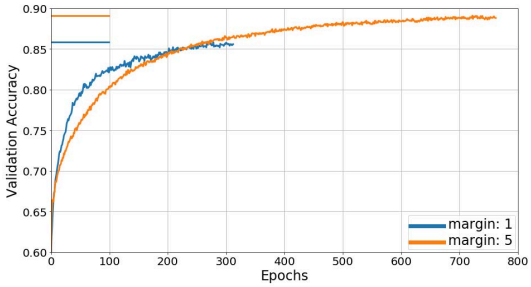
In this section, we discuss three different strategies for improved performance and scalability in the distributed learning of KGEs. These strategies are generic and can be applied to different models. The first two techniques, namely, *reduced communications* and *variable margin* can be applied to any model which uses a margin based ranking loss function. The third technique, the *DistAdam* Optimizer is a general optimization algorithm for distributed setting. It is not limited to the training of KGE models but can be applied to other domains including distributed deep learning.

A. Avoiding Zero-row Communications

It was observed that as the training progresses, many rows (corresponding to the parameters in *indices* array) of SGM become zero rows. This is because of the formulation of *margin based ranking loss function* (refer Equation 1). Such behavior is observed because as the training progresses and embeddings are learnt, the score function (refer Equation 2) for many correct triples decreases (desired behavior). The loss value for such triples becomes zero resulting in the gradients of the corresponding parameters also to be zero. Another explanation for this observation is the following. Gradient represents the amount of update to be made to a parameter. Initially, the parameters are randomly initialized and hence more number of parameters need to be updated. Hence, the Number of Non-Zero (NNZ) rows in the SGM is more (i.e. less sparse). In later stages of training much of the learning has already happened, hence less number of parameters need to be updated. Thus, the Number of Non-Zero (NNZ) rows in the SGM is less (i.e. more sparse). To confirm the above theory, experiments were performed on FB15K dataset on a single node. %NNZ rows in SGM was calculated for the first batch in each epoch and plotted against the loss for that epoch. Figure 2 shows that the %NNZ value decreases in a similar way as the training loss decreases. This implies that the full SGM need not be communicated in the multi-node setting. We can only communicate the NNZ rows without impacting accuracy of the model. This method of selectively sending the NNZ rows greatly reduces the All-Gather size in sparse mode, and hence reduces the communication time.



(a) Comparison of %NNZ gradients for two margins



(b) Comparison of Validation Accuracy for two margins

Fig. 3: %NNZ gradients and Validation Accuracy for two margins

B. Variable Margin Approach

The loss function mentioned in Equation 1 has a γ term which is the *margin* hyperparameter. The tuning of γ is done along with other hyperparameters such as learning rate and batch size. In the distributed setting, the effective batch size is $k \times n$, i.e the batch size keeps changing as we change the number of workers. This also means that the γ value that gives best validation accuracy for $k = 1$ may not be the best option for (say) $k = 32$. Continuous tuning of γ as the number of workers change, is a tedious and time consuming process. Instead, γ can be made self-adaptive. This is our first motivation for *Variable Margin approach*.

The value of γ directly affects the loss function value (refer Equation 1). The higher the γ, the higher is the loss value. According to the training objective we want lesser *score function* value for correct triple and higher score value for incorrect triple. A triple in the training data will either incur zero loss or a positive loss depending on the difference between the scores and γ value. It was shown in Figure 2 that the %NNZ gradients decrease as the loss value decreases. This means, for lesser loss value we obtain lesser %NNZ gradients which in turn reduces the All-Gather size. Reducing communication volume for multi-node setting is our second motivation for *Variable Margin approach*.

Plots supporting this theory are shown in Figures 3a and 3b. For these plots, experiments were performed on FB15K dataset and models trained till convergence. On this dataset, best validation accuracy is achieved with η = 0.001 and γ = 5. γ = 1 has been used for comparison.

It can be seen from Figure 3a that the %NNZ for $\gamma = 1$ is much less than the %NNZ for $\gamma = 5$. Although $\gamma = 1$ has the advantage of lesser All-Gather size, it performs poorly on the validation data as is shown in Figure 3b.

Ideally, we would like to maintain the validation accuracy as achieved by $\gamma = 5$ and also have lesser %NNZ for reduced All-Gather size as achieved by $\gamma = 1$. In this section we propose an approach to gradually change the margin such that both the desired properties can be achieved.

A very high value of margin allows the scores of correct and incorrect triples present in training data to be further apart. This generates embeddings which are suited for training data but performance on validation data is poor. In general, less value of margin may lead to under-fitting and high value of margin may lead to over-fitting of training data. Figure 3b shows that the validation accuracy for $\gamma = 1$ is better than that of $\gamma = 5$ for the first 150 epochs, but after that learning saturates for $\gamma = 1$. In our approach, we start with a lesser value of margin, such as $\gamma = 1$ and increase it dynamically until it reaches $\gamma = 5$. This has two advantages, first is the reduction in All-Gather size and second is higher validation accuracy during initial epochs. To dynamically vary the margin we link it to validation accuracy computed after each epoch. If the validation accuracy does not improve for five or ten consecutive epochs we increase the margin by five percent. After increasing the margin if the validation accuracy still does not improve for another ten epochs we increase the margin steeply i.e. by ten percent.

C. DistAdam: Adam Optimizer for Large Scale Training

In this section, we propose Adam optimizer for distributed (synchronous) training called *DistAdam*. Until now, for multi-node case we have used the *linear scaling rule* [7] which is formulated for SGD optimizer. Though Adam optimizer [4] is popular and works better than its contemporaries, there has not been an attempt to adapt it for distributed scenario. For the baseline model in Section III it was shown in Table I that the number of epochs for convergence is too high for multi-node setting. Formulation of an accurate optimization strategy will lead to fewer number of epochs for convergence thereby reducing the training time.

The Adam optimizer algorithm proposed by Kingma and Ba. [4] does not mention the batch size, so some changes are made in the notations to incorporate the batch size.

$f(\theta)$: Objective function with parameters θ

$g_{t,i}$: gradient of f w.r.t θ evaluated at timestep t for training instance i

m_t : 1^{st} order moving average at time t

v_t : 2^{nd} order moving average at time t

η : Base learning rate

$\alpha_t = \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$

Rule for updating the parameters is mentioned below in Equation 4

$$\theta_{t+1} \leftarrow \theta_t - \eta \cdot \alpha_{t+1} \cdot \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} \tag{4}$$

Equations written below are for m_{t+1}; for v_{t+1}, replace β_1 by β_2 and $g_{t,m}$ by $g_{t,m}^2$.

$$m_{t+1} = \beta_1 \cdot m_t + (1-\beta_1) \cdot \frac{g_{t+1,1} + g_{t+1,2} + \cdots + g_{t+1,n}}{n}$$

$$m_{t+k} = \beta_1 \cdot m_{t+k-1} + (1-\beta_1) \cdot \frac{g_{t+k,(k-1)n+1} + \cdots + g_{t+k,kn}}{n}$$

In general,

$$m_{t+p} = \beta_1 \cdot m_{t+p-1} + (1-\beta_1) \cdot \frac{\sum_{j=1}^{n} g_{t+p,(p-1)n+j}}{n}$$

Define $g(t,p,n) = \frac{\sum_{j=1}^{n} g_{t+p,(p-1)n+j}}{n}$
Therefore,

$$m_{t+p} = \beta_1 \cdot m_{t+p-1} + (1-\beta_1) \cdot g(t,p,n) \tag{5}$$

Similarly,

$$v_{t+p} = \beta_2 \cdot v_{t+p-1} + (1-\beta_2) \cdot g^2(t,p,n)$$

1) Correctly weighting moving averages: After k small batches of size n, the first order moving average m becomes:

$$\begin{aligned} m_{t+k} &= \beta_1 \cdot m_{t+k-1} + (1-\beta_1) \cdot g(t,k,n) \\ &= \beta_1^2 \cdot m_{t+k-2} + (1-\beta_1) \cdot [\beta^1 \cdot g(t,k-1,n) + g(t,k,n) \\ &\vdots \\ &= \beta_1^k \cdot m_t + (1-\beta_1) \cdot [\beta_1^{k-1} \cdot g(t,1,n) + \\ &\qquad \beta_1^{k-2} \cdot g(t,2,n) + \beta_1^0 \cdot g(t,k,n)] \end{aligned} \tag{6}$$

Taking one step in the case of large batch size kn, the first order moving average $\hat{m}$ becomes (hat has been used to differentiate large batch with small batch):

$$\begin{aligned} \hat{m}_{t+1} &= \beta_1 \cdot m_t + (1-\beta_1) \cdot \frac{g_{t+1,1} + g_{t+1,2} + \cdots + g_{t+1,kn}}{kn} \\ &= \beta_1 \cdot m_t + (1-\beta_1) \cdot g(t,1,kn) \end{aligned} \tag{7}$$

Note that m_t in first part of Equations 6 and 7 is the same since it is considered the starting point for both small and large batch. Or in other words $m_t = \hat{m}_t$. Ideally, we want value of m after taking 1 large step should be equal to value of m after taking k small steps. Currently, there is vast difference between m_{t+k} and $\hat{m}_{t+1}$ because in Equation 6, m is weighted by β_1^k whereas in equation 7, m is weighted by β_1^0. We make a few assumptions to ensure $\hat{m}_{t+1} \approx m_{t+k}$. For SGD, the assumption made by Goyal et al. in [7] (in terms of our notation) is mentioned in equation 8.

$$\sum_{j=1}^{k} g(t,j,n) = k \cdot g(t,1,kn) \tag{8}$$

In Equation 6, $g(t,1,n)$ is weighted by β_1^{k-1}, $g(t,k,n)$ is weighted by β_1^0 and so on. In the distributed case, $g(t,1,n), g(t,2,n), \cdots, g(t,k,n)$ are computed by workers individually and independently, so selective weighting of gradients is not appropriate. Equation 6 contains gradients w.r.t

batch size n, whereas Equation 7 contains gradients w.r.t batch size kn. To compare Equations 6 and 7, a stronger assumption has been made, the assumption is:

$$g(t,1,n) = g(t,2,n) = \cdots = g(t,k,n) \tag{9}$$

From equations 8 and 9, following is inferred:

$$g(t,1,n) = g(t,2,n) = \cdots = g(t,k,n) = g(t,1,kn) \tag{10}$$

Since $g(.)$ is computed for a batch size of 10,000 in our case, it can be argued that the assumption stated in Equation 10 is true by *Central Limit Theorem*. Based on stated assumptions, Equation 6 can be simplified as:

$$\begin{aligned} m_{t+k} &\approx \beta_1^k \cdot m_t + (1-\beta_1) \cdot g(t,1,kn) \cdot [\beta_1^{k-1} + \\ &\qquad \beta_1^{k-2} + \cdots + \beta_1^0] \\ &\approx \beta_1^k \cdot m_t + (1-\beta_1) \cdot g(t,1,kn) \cdot \frac{1 \cdot (1-\beta_1^k)}{1-\beta_1} \\ &\approx \beta_1^k \cdot m_t + (1-\beta_1^k) \cdot g(t,1,kn) \\ &\approx \hat{m'}_{t+1} \end{aligned} \tag{11}$$

The difference between Equations 7 and 11 is that β_1 in Equation 7 has been replaced by β_1^k in Equation 11. Finally, for the large batch case, using $\hat{m'}_{t+1}$ is more appropriate than using $\hat{m}_{t+1}$. Calculations can be worked out similarly for $\hat{v'}_{t+1}$ as well, and we get:

$$\begin{aligned} \hat{m'}_{t+1} &= \beta_1^k \cdot m_t + (1-\beta_1^k) \cdot g(t,1,kn) \\ \hat{v'}_{t+1} &= \beta_2^k \cdot v_t + (1-\beta_2^k) \cdot g^2(t,1,kn) \end{aligned} \tag{12}$$

For distributed setting with k workers
use $\boldsymbol{\beta_1^k}$ instead of $\boldsymbol{\beta_1}$ in case of m and **use $\boldsymbol{\beta_2^k}$ instead of $\boldsymbol{\beta_2}$** in case of v

Note: β_1 and β_2 used to calculate α_t are not changed.

2) Scaling factor: Scaling factor refers to the scaling of *learning rate* in case of large batch. Owing to less number of updates in large batch, its *learning rate* needs to be multiplied by a scaling factor such that its update value after one step can match update value of small batch after k steps. From Equation 4, after taking k small steps, the update will be:

$$\begin{aligned} \theta_{t+k} &= \theta_{t+k-1} - \eta \cdot \alpha_{t+k} \frac{m_{t+k}}{\sqrt{v_{t+k}} + \epsilon} \\ &= \theta_t - \eta \cdot [\alpha_{t+1} \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} + \alpha_{t+2} \frac{m_{t+2}}{\sqrt{v_{t+2}} + \epsilon} + \\ &\qquad \cdots + \alpha_{t+k} \frac{m_{t+k}}{\sqrt{v_{t+k}} + \epsilon}] \end{aligned} \tag{13}$$

After taking one large step, the update equation will be:

$$\hat{\theta}_{t+1} = \theta_t - \hat{\eta} \cdot [\alpha_{t+1} \frac{\hat{m'}_{t+1}}{\sqrt{\hat{v'}_{t+1}} + \epsilon}] \tag{14}$$

Note that $\hat{\eta}$ in Equation 14 is different from η in Equation 13 and in this section we determine the appropriate value of $\hat{\eta}$. (Note that according to *linear scaling rule* $\hat{\eta} = k\eta$)
According to Equation 11, $m_{t+k} \approx \hat{m'}_{t+1}$ and $v_{t+k} \approx \hat{v'}_{t+1}$. The assumption that is made here is:

$$\frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} \approx \frac{m_{t+2}}{\sqrt{v_{t+2}} + \epsilon} \approx \cdots \approx \frac{m_{t+k}}{\sqrt{v_{t+k}} + \epsilon} \approx \frac{\hat{m'}_{t+1}}{\sqrt{\hat{v'}_{t+1}} + \epsilon}$$

Incorporating this assumption in Equation 13 and comparing with Equation 14, we get:

$$\begin{aligned} \hat{\eta} \cdot \alpha_{t+1} &= \eta \cdot \sum_{j=1}^{k} \alpha_{kt+j} \\ \hat{\eta} &= \eta \cdot \frac{\sum_{j=1}^{k} \alpha_{kt+j}}{\alpha_{t+1}} \end{aligned} \tag{15}$$

Scaling factor to use for k workers
$\frac{\sum_{j=1}^{k} \alpha_{kt+j}}{\alpha_{t+1}}$ instead of k

The complete *DistAdam* algorithm is stated in Algorithm 1

Algorithm 1: DistAdam

η : base learning rate (say 0.0001)
$\beta_1, \beta_2 \in [0, 1)$
$f(\theta)$: Stochastic objective function with parameters θ
θ_0: Initial parameter vector
$\hat{m'}_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$\hat{v'}_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t *not converged* **do**
  $g(t,1,n) \leftarrow \nabla_\theta f_{t+1}(\theta_t)$
  All-Gather $g(t,1,n)$ from k workers and average to generate $g(t,1,kn)$
  $\hat{m'}_{t+1} \leftarrow \boldsymbol{\beta_1^k} \cdot \hat{m'}_t + (1-\boldsymbol{\beta_1^k}) \cdot g(t,1,kn)$
  $\hat{v'}_{t+1} \leftarrow \boldsymbol{\beta_2^k} \cdot \hat{v'}_t + (1-\boldsymbol{\beta_2^k}) \cdot g^2(t,1,kn)$
  $\alpha_{t+1} \leftarrow \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$
  scale = $\frac{\sum_{j=1}^{k} \alpha_{kt+j}}{\alpha_{t+1}}$
  $\theta_{t+1} \leftarrow \theta_t - \eta \cdot$ **scale** $\cdot \alpha_{t+1} \cdot \frac{\hat{m'}_{t+1}}{\sqrt{\hat{v'}_{t+1}} + \epsilon}$
  $t \leftarrow t+1$
end

V. Experiments and Results

A. Experimental Setup

1) Datasets: In this work, we used three datasets, namely *FB15K*, *FB250K* and *Wiktionary English* dataset. The FB15K dataset has been used to gain insights and confirm the working of proposed techniques. FB15K is an open-source dataset created by Bordes et al. [10] by skimming the original Freebase dataset [14], which is a very large dataset consisting of around 2 billion triples (facts).

FB15K dataset has 14951 unique entities, 1345 unique relations and close to 600k triples. To demonstrate the scalability of the results, we created a dataset called *FB250K*. This dataset is created by extracting most frequently occurring 250k entities from the Freebase dataset and then selecting only those triples which contained these entities. The FB250K dataset has about 16 million facts, 240k entities and 9280 relations.

The *Wiktionary English* [15] dataset was created by extracting the most frequently occurring one million entities and selecting triples containing only these entities. This resulted in six million training instances.

The datasets were divided into *train*, *valid 1*, *valid 2* and *test*. FB15K dataset already has three splits i.e. train, validation and test. The validation set was split in two halves to generate *valid 1* and *valid 2*. For FB250K dataset, instances were randomly chosen to be in either of the 4 splits with different probabilities. FB250K dataset has ∼15.1 million train instances, ∼375k instances each in *valid 1* and *valid 2* and ∼50k test instances. For the multi-worker setting involving k workers, the dataset is partitioned into k equal portions and each worker samples batches randomly from the partition of dataset available to it. Therefore, at each training step, we sample non-overlapping portions of the dataset which eliminates redundancy. The batch size per worker is fixed at 10,000.

2) Metrics: We compare in terms of both performance and accuracy. For evaluation of accuracy, we use the accuracy metrics described in Section III-B. For Triple Classification Accuracy (TCA), δ_r is the threshold learned individually for each relation and *valid 1* split is used to learn the optimum value of δ_r. For a given triple in *valid 2*, the score function is calculated using the learnt embeddings and if this value is less than the threshold δ_r, then the triple is classified as correct or incorrect otherwise. After each epoch, TCA is calculated on *valid 2* split (after optimizing δ_r on *valid 1*). This value of TCA is called validation accuracy in this work and is used to ascertain the convergence behavior of the model. Finally, after convergence, TCA is calculated on the *test* set.

3) Model: TransE [10] has been used to compare results across multiple workers and Horovod [3] has been used as the distributed training framework. The dimensions of the vector embedding for both, entities and relations, is set as 100. Construction of incorrect triples is done by generating one negative sample per training instance in which either the head or the tail entity is corrupted randomly. Margin hyperparameter is chosen among values$\{1, 2, 3, 5, 10\}$ based on validation accuracy. Adam optimizer [4] has been used for all KGE models unless otherwise specified. The preliminary experiments have been performed on our lab server with *Intel(R) Xeon(R) CPU E5-2670* processor which has 32 cores. The experiments with FB15K dataset have been done on this machine and *OpenMPI* has been used for spawning multiple workers. Each experiment is run until convergence and the best model is saved based on validation accuracy. The model is said to have converged if the validation accuracy does not improve for 40 consecutive epochs. Training is stopped at 800 epochs if convergence is not achieved by then. Selection of hyperparameters such as learning rate and margin was done on the basis of best validation accuracy.

4) Optimizer Selection: We performed experiments[4] with both SGD (Stochastic Gradient Descent) and Adam optimizer and found that better validation accuracy was obtained using Adam optimizer.

[4]Experiments performed using single worker on FB15K dataset

No. of Nodes	AG Time (hrs)	Other Time (hrs)	Total Time (hrs)	EFC	TCA (%)	MR		Hits@10	
		———				raw	filter	raw	filter
1	0	11.32	11.32	323	88.24	4826	380	0.246	0.582
2	0.73	6.32	7.05	425	89.00	4812	357	0.251	0.592
4	0.78	2.61	3.39	359	88.77	4869	352	0.251	0.597
8	1.06	1.57	2.63	403	89.11	4878	340	0.255	0.601
16	0.58	0.90	1.48	350	88.68	4956	346	0.254	0.605
32	0.40	0.56	0.96	387	88.89	4925	347	0.256	0.607
64	0.62	0.32	0.94	293	88.93	5086	361	0.254	0.608

TABLE II: Results for Combined Approach

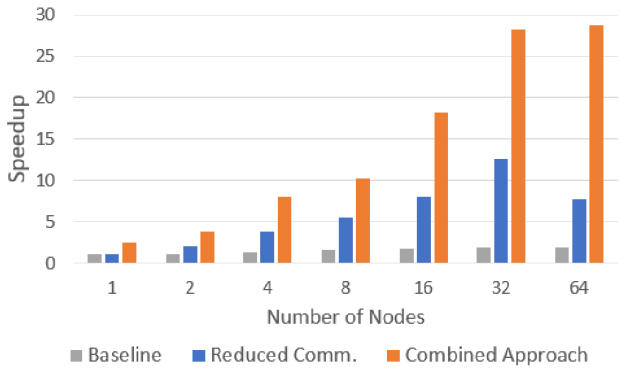


Fig. 4: Speedup comparison of proposed models

5) Parallel System: Our primary experiments were performed on a CrayXC40 supercomputing system in Supercomputer Education and Research Centre (SERC) in our Institute. The cluster has nodes with Intel Haswell processors running at 2.5 GHz and connected using Cray Aries interconnect. We used up to 64 nodes for our experiments where each node has 2 CPU sockets with 12 cores each and 128GB RAM.

B. Results

We show results for each of the three individual strategies described in Section IV and also with the combination of the strategies.

1) Combined Approach: In this section, results are presented for combined approach that is a combination of all the three strategies presented in Section IV. We present results of *Combined approach* on the FB250K dataset in Table II.

We observe that the evaluation metrics shown in Table II are similar to evaluation metrics shown in Table I which is the desired behavior. The major difference between the two tables is in terms of total training time. There is significant decrease in total time using the *combined approach* as compared to baseline.

We present speedup results in Figure 4. The speedups are calculated w.r.t to single node baseline time i.e. 27 hours. Training time of combined approach on 32 nodes is just 58 minutes whereas that of single node baseline is 27 hours and 32-node baseline is 13.8 hours. Therefore, with 32 nodes, the combined approach gives a speedup of 28 when compared to

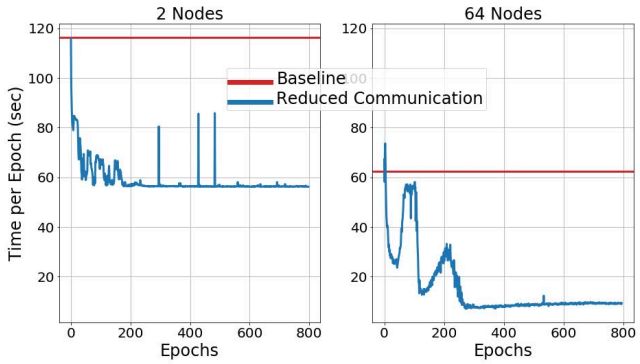


Fig. 5: Illustration of decrease in training time per epoch as training progresses

No. of Nodes	AG Time (hrs)	Other Time (hrs)	Total Time (hrs)	EFC	TCA (%)	MR		Hits@10	
						raw	filter	raw	filter
1	0	27.02	27.02	800	89.06	4883	359	0.245	0.592
2	1.22	11.78	13.00	798	89.45	4909	345	0.250	0.600
4	1.31	5.63	6.94	727	89.42	4911	346	0.248	0.601
8	1.57	3.37	4.94	778	89.58	4920	331	0.252	0.606
16	1.33	2.05	3.38	793	89.73	4951	336	0.253	0.610
32	0.97	1.18	2.15	780	89.75	4953	334	0.254	0.611
64	2.39	1.09	3.48	796	89.73	4965	336	0.252	0.611

TABLE III: Results for Reduced Communication Approach

single-node baseline and a speedup of 14.27 when compared to 32-node baseline.

2) Reduced Communication: We show results with the first strategy of avoiding communication of zero-rows described in Section IV-A. We ran experiments on the large dataset FB250K and found that there was significant reduction in training time. As was shown in Figure 2, the NNZ gradient rows decrease as the training progresses. Hence, we obtain decrease in training times with the number of epochs. In the plots shown in Figure 5, time per epoch decreases (in general) as the training progresses. This method results in lesser communication volume i.e. lesser All-Gather size without affecting the accuracy.

Table III shows the results for FB250K dataset using the reduced communication approach. For 32 nodes the reduced communication strategy is 6.5X faster than the baseline multi-node strategy. Note that the accuracy evaluation metrics in Table I and Table III are the same. This is because reduced communication approach reduces All-Gather size without impacting accuracy).

Figure 6 compares the multi-node baseline and reduced communication strategy (denoted by *B* and *R* respectively). We observe that the total time has decreased significantly as compared to baseline. This is largely attributed to decrease in *All-Gather time*.

3) Variable Margin Approach: From Figure 7 we observe that the validation accuracy for the *Variable Margin* approach is better than the best validation accuracy achieved for fixed margin. Also, we observe that the *Variable Margin* curve overlaps the $\gamma = 1$ curve for initial epochs but does not saturate early as that of $\gamma = 1$.

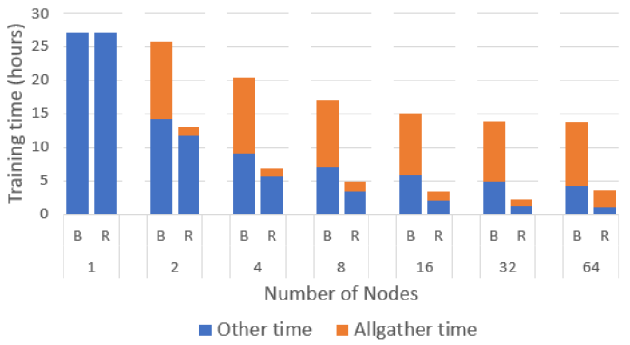


Fig. 6: Comparison of total training time for Baseline vs Reduced Communication model

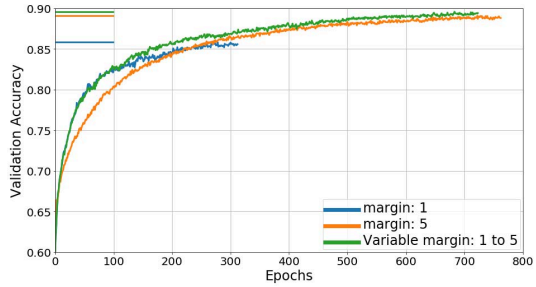


Fig. 7: Comparison of Validation Accuracy for Variable Margin Approach

Figure 8 shows the variation of %NNZ gradients with epochs. It shows that %NNZ for *Variable Margin* is lesser than best fixed margin of 5 (except for few later epochs), hence allowing the All-Gather size to be less as well.

4) DistAdam: Simulations in support of DistAdam[5]*:*

First, we show simulations for *correct weighting of moving averages*. We show comparisons between:

1) m_{t+k} i.e. m generated sequentially using small batches, considered as ground truth. (refer Equation 6)

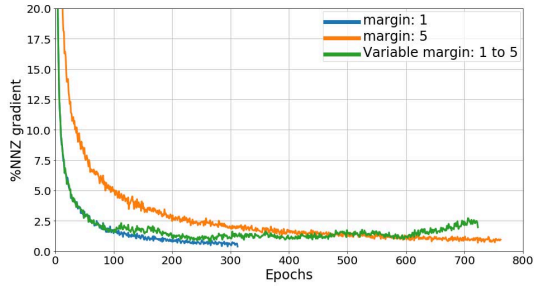
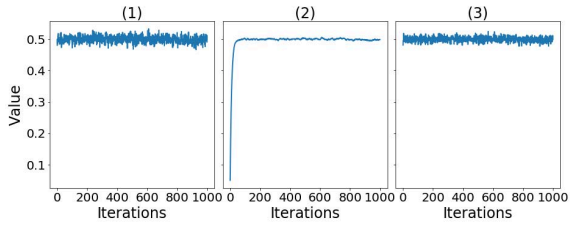
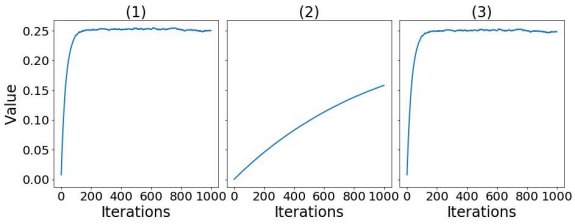


Fig. 8: Comparison of %NNZ gradients for Variable Margin Approach

[5]Code available at https://github.com/UditGupta10/DistAdam



(a) Comparison of m by simulation



(b) Comparison of v by simulation

Fig. 9: Simulations for m and v values

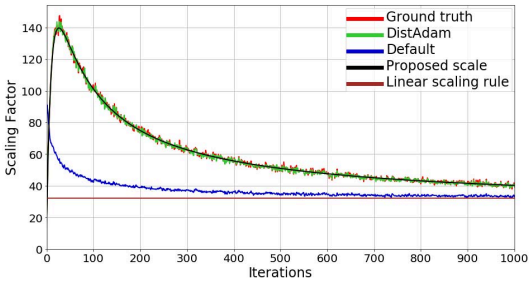


Fig. 10: Comparison of Proposed Scaling factor with Simulated Scaling factor

2) $\hat{m}_{t+1}$ i.e. m generated by default using large batches. (refer Equation 7)
3) $\hat{m'}_{t+1}$ i.e. m generated by *Distadam* using large batches. (refer Equation 12)

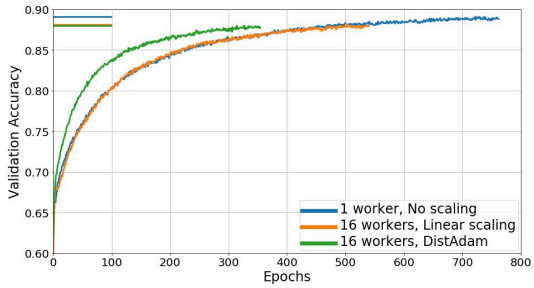
In Figures 9a and 9b, (1) sub-figure represents values generated sequentially using small batch (but shown at k intervals) to mimic large batch behavior; this is the ground truth. (2) sub-figure represents values generated by default setting using large batch. (3) sub-figure represents values generated by *DistAdam* using large batch.

Now, we show simulations for *scaling factor*. The aim is that the sum of k updates using small batch should be equal to one update using large batch.

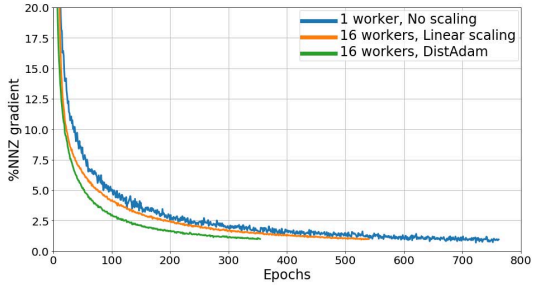
Figure 10 shows that the proposed scaling factor mimics the *ground truth* scaling factor. Also, the scaling factor corresponding to *linear scaling rule* is not an ideal choice during initial iterations although the proposed scaling factor approaches the *linear scaling rule* during later iterations.

Experiments with FB15K dataset

The best validation accuracy is achieved using (η) = 0.001 and (γ) = 5. Validation curve obtained using *DistAdam* for



(a) Comparison of Validation accuracy for Linear Scaling Rule and DistAdam



(b) Comparison of %NNZ for Linear Scaling Rule and DistAdam

Fig. 11: Comparison of Validation accuracy and %NNZ for Linear Scaling Rule and DistAdam

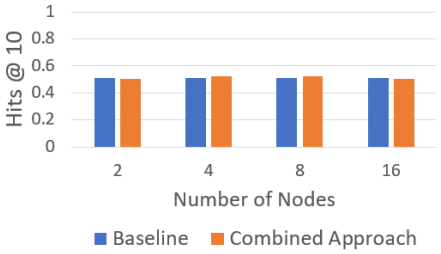
16 workers is shown in Figure 11a and is compared with the *linear scaling rule* for 16 workers. Validation curve for one worker serves as the baseline.

Though the maximum validation accuracy achieved by *DistAdam* and *linear scaling rule* are almost same, but *DistAdam* converges two hundred epochs earlier than the latter which saves training time. Figure 11b shows that %NNZ for *DistAdam* is always lesser than that of other two cases, thereby resulting in lesser *All-Gather* size.

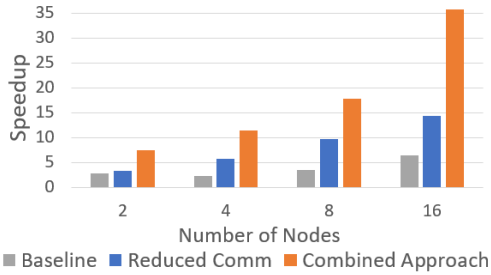
5) Results with Wiktionary English dataset: In this section we present results using *Wiktionary English* dataset. The best validation accuracy is achieved using $\eta = 0.001$ and $\gamma = 7$. For *variable margin* strategy the margin was varied from 1 to 7 dynamically.

Figures 12a and 12b show the results for accuracy and performance, respectively (speedup calculated w.r.t single node baseline). We find that the combined approach gives similar accuracy as baseline. As shown in the performance results, the combined approach with 16 nodes gives 35X speedup when compared to a single-node baseline, 6X speedup when compared to 16-node baseline and more than 2X speedup when compared to reduced communication approach.

6) Results with TransH Model: The results presented till now for all sections were performed using TransE model. But the approaches proposed in Section IV are not specific to TransE only. These techniques can be applied to other KGE models as well. Here we show the results of experiments performed on TransH [11] model using the *Combined approach*. Figure 13a shows the Hits@10 comparison for baseline vs

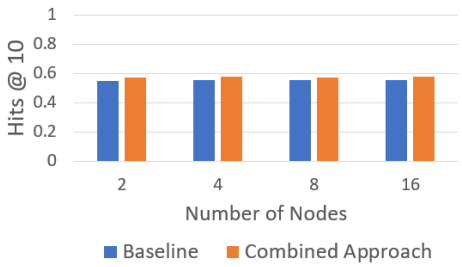


(a) TransE with WK1M - Accuracy comparison of Baseline and Combined Approach

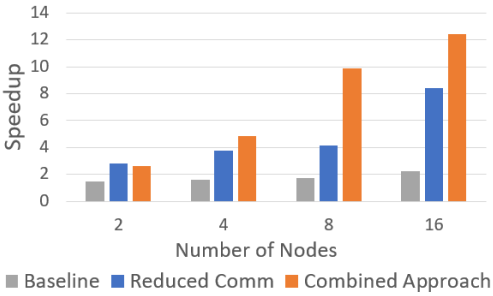


(b) TransE with WK1M - Speedup comparison of proposed models

Fig. 12: TransE with WK1M - Accuracy and Speedup



(a) TransH - Accuracy comparison of Baseline and Combined Approach



(b) TransH - Speedup comparison of proposed models

Fig. 13: TransH - Accuracy and Speedup

combined approach. We observe that the combined approach gives better accuracy than the baseline. Figure 13b shows the speedup comparison for reduced communication approach vs combined approach. We observe that combined approach gives better speedup than the reduced communication approach.

VI. Conclusions and Future Work

In this work we proposed three strategies to decrease the training time of KGE models. The fundamental strategy is the *reduced communication* strategy which significantly reduces the *All-Gather* size (and hence *All-Gather* time) as training progresses. *Variable margin* strategy increases the margin dynamically to decrease the All-Gather size. *DistAdam* algorithm proposes changes to be made to Adam optimizer such that it can be used accurately on large scale with multiple workers. Experiments show that it leads to faster convergence as compared to the default *linear scaling rule* for multiple-workers. Finally, we combine these three techniques and obtain a speedup of 28 on 32 nodes for TransE and 12.4 on 16 nodes for TransH on FB250K dataset. As future work, we plan to decrease the size of All-Gather even further, explore more robust techniques for variable margin and compare different methods of scaling the learning rate in DistAdam algorithm.

References

[1] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[2] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[3] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[5] D. Zhang, M. Li, Y. Jia, Y. Wang, and X. Cheng, "Efficient parallel translating embedding for knowledge graphs," in *Proceedings of the International Conference on Web Intelligence*. ACM, 2017, pp. 460–468.

[6] X.-F. Niu and W.-J. Li, "Paragraphe: a library for parallel knowledge graph embedding," *arXiv preprint arXiv:1703.05614*, 2017.

[7] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[8] W. Gropp, W. D. Gropp, A. D. F. E. E. Lusk, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[9] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, p. 65, 2019.

[10] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Advances in neural information processing systems*, 2013, pp. 2787–2795.

[11] Z. Wang, J. Zhang, J. Feng, and Z. Chen, "Knowledge graph embedding by translating on hyperplanes." in *AAAI*, vol. 14, 2014, pp. 1112–1119.

[12] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *Siam Review*, vol. 60, no. 2, pp. 223–311, 2018.

[13] B. Mohr, A. D. Malony, J. E. Cuny, and B. M. A. D. Malony, "Tau-tuning and analysis utilities for portable parallel programming," 1995.

[14] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. AcM, 2008, pp. 1247–1250.

[15] Wikimedia, "Wiktionary RDF extraction," https://wiki.dbpedia.org/wiktionary-rdf-extraction, 2014, [Online; accessed 20-Sep-2019].