Distributing Translating Embeddings for Modeling Multi-relational Data, using Shared Memory

Xhulja Shahini
Paluno - The Ruhr Institute for Software Technology
Email: xhulja_shahini@hotmail.com

***Abstract*—The interest that Knowledge graph embeddings (KGE) models have received, has significantly increased over the last years. Training KGE models is a tedious process that requires high CPU and memory capacity. Different industries are training their huge, often distributed datasets using KGE models. An urge to scale KGE models has evolved, to ensure that the researcher's hard work is being aligned with the needs of the industry of big data. To tackle the issue of resource availability or usage limitations, we have come up with an approach for distributing both training and testing of Translating embeddings for modeling multi-relational data [1], using a cluster with shared memory layer[1]. We show that distributing the model reduces it's run-time for training and testing while maintaining efficiency and achieving the results of the original paper. Although the focus of this work is to train and test TransE model in a distributed fashion, the underlying algorithm applies to any other KGE model.**

***Keywords*—Knowledge graphs embeddings, Translating embeddings, CPU and memory capacity, big data, distributed training and testing, shared memory.**

I. Introduction

A knowledge graph (KG) is a knowledge base that represents relational data in the structure of a graph. These graphs are composed of nodes that correspond to entities and edges that represent relations. After the launch of Google's knowledge graph in 2012 [2], KGs became very popular, nevertheless, the information contained in these graphs is often incomplete. Thus, it is necessary to perform tasks such as link prediction or error detection to fill in the missing information, as well as to remove false data. A commonly used approach for the efficient manipulation and analysis of knowledge graphs is to represent the entities and relations as vectors in a low-dimensional space, a.k.a knowledge graph embeddings[3]. This simple solution has proven to be very efficient and it has become widely used in different industries. Knowledge graphs embeddings models have proven to be successful even when applied on top of large-scale datasets.

With the evolving of big data, the size of information being stored in datasets is continuously becoming larger and larger. To manage to store massive amounts of information, the notion of distributed fashion datasets has been introduced. A distributed dataset is a storage model in which instead of using a single storage node/machine, the data is split and saved in a network of physical nodes. Sometimes the data is also replicated to prevent loss of information in cases of node's failure.

While it seems that we have a solution, at least temporarily, for storing and handling big data, processing and manipulation of it is quite of a challenging task in which we need to put a lot of effort since many industries are interested in efficient applications to extract information from their dataset.

Knowledge graph embeddings are well-known models to perform tasks such as knowledge extraction and completion. The issue of these models lies in the size of the knowledge graph. Training a KGE model with a big number of parameters over large datasets, significantly slows down the training process; moreover, the data will no more fit in the memory. Requirements for high CPU and memory availability during the training of these models leads to the need for parallelizing the training process in a distributed environment.

To address this issue, we developed a distributed version of TransE.
In this work we present our approach for efficiently distributing the training and the testing process of Translations Embeddings model in a cluster with shared memory layer. This work contributes to research in the following ways:

1) We describe our approach to develop a distributed version of TransE and we explain the logic behind our solution while comparing it with other proposed approaches for distributed KGE

[1]Link to our repository:

models.

2) We provide a detailed description of all the theoretical and technical steps taken to develop this model, to ensure the reproducibility of our work and evaluations.
3) We show that our model runs much faster than the original version of TransE and even though we had a limited testing environment we can show that the distribution speeds up the model runtime while preserving its accuracy.
4) We perform evaluations with a wider range of datasets and we show that for some cases we have achieved better results.

II. Foundations

A. Knowledge Graphs Embeddings

Knowledge graphs are representations of knowledge bases, usually containing millions of entities and many more facts. Consequently, performing tasks such as manipulating and analyzing the data, suffers from high computational and storage cost.

Researchers have proposed many different algorithms, intending to represent and manipulate KG efficiently. A well-known algorithm is to translate high dimensional knowledge graphs into low dimensional spaces using vectors representation, in such a way that the relations between the entities and the overall structure of the graph are kept intact. The vector representation of the graph's entities and relations is commonly known as embeddings. After learning the vector representations of the entities and relations, it is much easier to perform analysis and data operations. The typical algorithm of KGE models is[4]:

1) Translate the entities and relations in an embedding space R^d, $d \in$
2) Choose an appropriate score function
3) Train the entities and the relations

To predict the existence of a missing relation, knowledge graph models use a score function. This score determines the plausibility of a triple or fact, which we usually calculate via a distance-based or similarity-based score function.

In this work, we focus on the TransE model. Nevertheless, it is possible to extend the application for all other KGE models.

B. TransE

TransE stands for Translating Embeddings for Modeling Multi-relational Data. TransE is one of the models in the series of KGE Translational distance models[2]. Same to other KGE models, it translates knowledge graphs entities and relations in a low-dimensional embedding space. Multi-relational data are data organized as a directed graph of nodes that represent entities, and edges that represent relations. The whole graph consists of many types of relations. In TransE we translate all entities and relations in the same low dimensional space R^d (d is the number of dimensions), where we represent them as vectors. For every existing triple: (head, label, tail) in the KG, consider the respective triple embeddings in R^d to be: (hE, lE, tE). The sum vector of hE + lE should be the closest neighbor of the vector tE. TransE Algorithm 1 learns the embeddings of all entities and relations of the dataset by minimizing the overall loss which is calculated via the margin-based ranking criterion. TransE is a translational distance model, which makes use of distance-based score function. This means that to predict the score of a triple (hE, rE, tE), TransE measures the distance of that triple d(hE, rE, tE). It does so by using either L_1 or L_2 distance function.

Algorithm 1 LearningTransE

Input Training set $S = (h, l, t)$, entities and rel. sets E and L, margin Υ, embeddings dim. k
initialize $l \leftarrow uniform(\frac{-6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each l $\in L$
l$\leftarrow \frac{l}{\|l\|}$for each l $\in L$
e$\leftarrow uniform(\frac{-6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each entity e $\in E$
loop
e$\leftarrow \frac{e}{\|e\|}$for each e $\in E$
$S_{batch} \leftarrow sample(S, b)$ //sample a minibatch of size b
$T_{batch} \leftarrow \emptyset$ //initialize set of pairs of triplets
for (h,l,t)$\in S_{batch}$**do**
(h',l,t')$\leftarrow sample(S'_{(h',l,t')})$ sample a corrupted triple
$T_{batch} \leftarrow T_{batch} \cup (h, l, t), (h', l, t')$
Update embeddings w.r.t. $\sum_{(h,l,t),(h',l,t') \in T_{batch}} \nabla[\Upsilon + d(h + l, t) - d(h' + l, t')]_+$
end loop

C. Parallel and Distributed Training Methods

The training process of a model is a time-consuming task. Usually, the datasets we utilize for training are very big in size. Thus, training a model takes a long time to complete. While the accommodation of such large datasets doesn't appear to be very challenging, the situation is not the same for semantics derivation mechanisms from them. Therefore it is essential to focus more on developing efficient algorithms and applications for big data manipulation and knowledge extraction.

Big Data evolution has caused an increasing demand for diverse computing applications, therefore the development of highly efficient algorithms has become

[2]Translational distance models make use of a distance-based function to calculate the score of the triples. Some of these models are: TransH[5], TransR[6], TransD[7], TranSparse[8].

a requirement of high priority. Defining and achieving the efficiency of the algorithms for Big Data is a big challenge and work yet in progress.

Researchers have proposed parallel and distributed training to reduce the overall run-time of the model. According to the results of the previously done experiments, some best-practice suggestions regarding the use case and the available environment would be [9].

- *Parallel training in single node:* In cases where the runtime of the training process is the only concern that steers us to distribution mode, an appropriate solution for the problem is to use local parallel training. We can train models in parallel over multiple CPUs, GPUs, or both. This parallelization technique will drastically reduce the runtime of the model, but since we are using only a working node we should make sure that the memory availability is enough for the data and the selected model parameters.
  - We can employ CPU cores to parallelize tasks that can be computed simultaneously, such as batches of SGD.
  - GPU cores are useful for the fast processing of highly computational tasks.
- *Distributed Training in a cluster of nodes:* Another case where distribution methods would be necessary is when both the CPU and memory capacity of a single working node, is not enough for the training process of a model. In this case distribution in a cluster of nodes that communicate with each –other is needed, since parallel training in a single node can reduce the training speed, but it won't fit the data and the model parameters in the memory. Making use of the memory and CPU/GPU cores of multiple working nodes can solve the problem.
  Generally, algorithm parallelization strategies are composed of these main steps: [10]
  1) Analyse the problem
  2) Depict a parallel algorithm for the solution of the problem
  3) Choose an appropriate hardware architecture that matches the algorithm
  4) Implement the solution using a parallel programming language/framework

III. Methodology

A. Architecture

The first challenge of this work is to choose the appropriate cluster architecture. We based our solution in the gradient server architecture Figure 1 , with some modifications. In our case Figure 2, we replace the server group with a master node. The

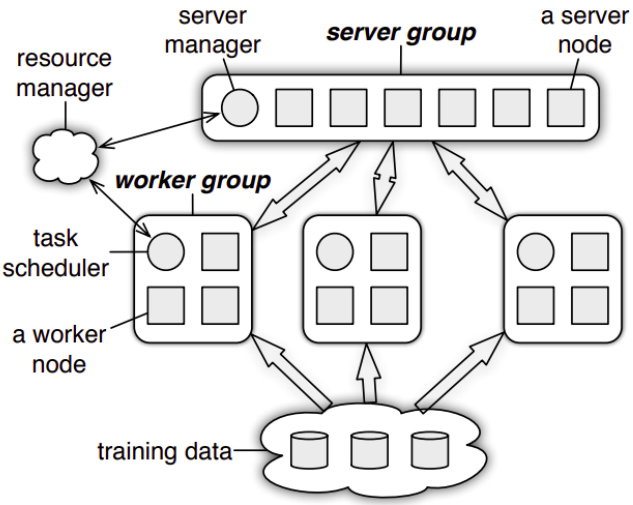

Figure 1: Illustrations of the parameter server architecture , adapted from [11] .

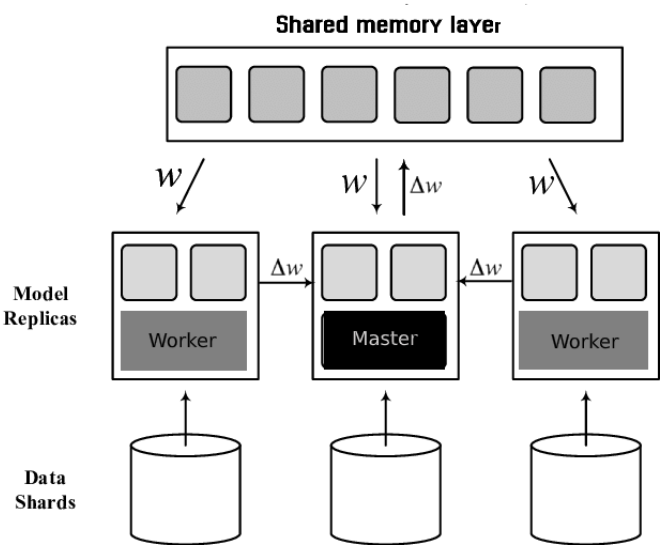

Figure 2: Illustrations of our cluster architecture.

main idea is to have a cluster of nodes composed of 1 master and many worker nodes. All nodes are computational nodes, including the master. The cluster has a shared memory layer, that we employ to store the information that needs to be accessed often, such as Triples and Embeddings. In this way, we reduce the communication between master and the working nodes for information retrieval.
Unlike the gradient server architecture, where the gradients are asynchronously accessed and updated by all the working nodes, to avoid the collision or overwrites during updates, we perform synchronous updates. The master node will be

the only node that performs the update of the gradients. The master node receives all the updates calculated by the slave nodes and performs the update. As mentioned previously, all nodes are computational nodes, which means that the master itself also performs batch computations for updating the gradient vectors, while waiting for the worker nodes to complete the same task.
While performing the evaluations, every node computes one batch of the test triples, and the master node aggregates the final result. We define the batch size as:

$$batchSize = \frac{testTripleNr}{clusterSize}$$

,where *testTripleNr* is the total number of triples used for testing and *clusterSize* is the number of nodes in the cluster.

B. Algorithm parallelization

In order to make the TransE model work efficiently in a distributed way, we made a slight adjustment to the algorithm. Since our goal is to minimize the run time of the model, it is necessary to minimize the communication back and forth between nodes. In our case, the master node is the only one that updates the gradients. This means that the worker nodes should send the calculated gradients back to the master every time they perform a computation. That would cause too much network communication, so we had to reduce the frequency of gradient updates. Every node computes $\frac{NrOfBatches}{clusterSize}$ batches. After each worker has computed a batch, we make an update of the embeddings. Our tests show that the efficiency of the model remains intact.

C. Shared layer memory

When transferring the model from a single working node environment to a cluster with multiple working nodes, a significant amount of time is added to the total run-time as a result of the communication that has to be carried between the working nodes of the cluster.
In a cluster of nodes that have their resources, there are 2 ways of arranging the communication between the nodes:

- Case 1: Every node contains a copy of the necessary data for the computations, and we perform updates at local and global levels. In this type of architecture, the most time-consuming step would be to update the data in every node (global update) every time an update takes place.
- Case 2: Only one node, the master node, contains all the necessary data, and the other nodes wait for instructions and data to perform computations. Here the most time-consuming step would be the movement of data from one node to another.

Given that in the TransE model it is necessary to access data very often, as well as do updates very often, none of these cases would be beneficial in terms of run-time. For this reason, we use a shared memory layer to reduce the data movement from one node to another. We implemented this with the help of the Data Grid of Apache Ignite and its distribution modes. Apache Ignite key-value cache distributes and stores the data among all nodes in the cluster. Via a hashing algorithm, clients can quickly localize where specific keys are stored. We use this functionality to store the training and test triples since they are frequently accessed data during the training and evaluation of the model. All the nodes can access this cache.
Regarding the gradient vectors, to avoid collisions during updates, we perform synchronous updates. This means that all the nodes send the calculated gradients back to the master node, and the master node does the update. We mentioned previously that this type of architecture could be unstable due to the single point of failure issue. We tackle that problem by deploying our cluster in Kubernetes, which automatically restarts a replacing node identical to the one who failed, ensuring a running cluster with the minimum number of nodes that the user has required.
Our model has 3 caches that are being accessed by all the nodes in the cluster:

- Training triples cache: Contains the triple ID-s that will be trained by the model:
  Key: Int - representing Triple's row Id
  Value: Tuple3[Int, Int, Int] - representing triples ids
- Existing triples cache: same as previous but with key-values the other way around to ensure that get() operation is performed in O(1) time:
  Key: Tuple3[Int, Int, Int] - representing triples ids
  Value: Int - representing Triple's row Id
- Test triples cache: cache containing all the triples used for the evaluation of the model
  Key: Int - representing Triple's row Id
  Value: Tuple3[Int, Int, Int] - representing test triples Ids

There are 3 different ways to configure a cache in

Apache Ignite. Based on how we want the cache to operate, we can select any of the following modes: Partitioned, Replicated, or Local.

- Partitioned mode is the most commonly used in distribution tasks. In this mode, Apache Ignite will share the dataset across the cluster giving every node a chunk of data.
- Replicated mode is similar to Partitioned mode, but instead of giving every node a chunk of data, it gives them a copy of the whole dataset.
- Local mode creates and stores the cache locally, no distribution of data happens over the cluster. We mostly use this mode for faster read-time and when the data needs to be updated frequently.

The Partitioned Cache mode, partitions the dataset among nodes, but it also keeps backup copies to prevent loss in case of node failures. The Replicated Cache mode also partitioned the dataset, making every node a primary storage uni for one chunk of the data. In this cache mode, the backup contains all the other chunks of the data. In this way, every node contains a copy of the whole dataset.

In our case, we found that only the first 2 cache operation modes are necessary. We use them as explained below:

- For the training triples, we use the Replicated mode cache, since there is no need to perform updates over this cache. This mode allows faster read time from the cache.
- For the filtering cache, we use Replicated mode cache for the same reason as mentioned above.
- For the test triples, we use the Partitioned mode cache accompanied by the affinity key functionality. This means that every node works only with the data that contains it. For example, we have a cluster of 10 nodes, each node contains $\frac{1}{10}^{th}$ of the test data and computes the test results just for that part of data. The results that each node computes, are aggregated at the master node.
- Initially we used Partitioned mode cache for the embeddings (gradients) vectors. But, given that all nodes need to have access to all the data due to randomly picking of the next Triple to be trained, we realized that this approach slows down the training of our model. Replicated mode cache doesn't work well either because after each update all the replicas of the embeddings cache would require a global update, which also results in a slow down of the model. Therefore, we implemented the embeddings using Scala's mutable Maps.

D. Files reading

Another step that we took to further reduce the run-time of our model, is distributing the task of reading knowledge from files. We have to read several files to create the dataset caches. For this reason, we read these files in parallel, in different nodes, instead of waiting for the master node to read all the files necessary. The initialization of the embeddings is also performed in a distributed way.

E. Reduced updates

At the beginning of this section, we claim that data transferal is a time-costly process and still we perform synchronous updates. This means that we aggregate the gradient embeddings in a single node before performing an update. The reason for that is that it is important to prevent the working nodes from overwriting each-others updates or cause errors while trying to write simultaneously. To reduce network communication at this step, we chose to reduce the frequency of updates. Instead of updating the embeddings after every batch, we update them once every $NumberOfNodes$-batches. We compute these batches in parallel. Basically, we have increased the batch size with a factor of $NumberOfNodes$. Our experiments showed that the accuracy results are preserved when running the model in a distributed fashion.

F. Zero gradients

There are 2 main causes for the latency of communication between nodes: the frequency of communication and the amount of data transferred. We already took a step to decrease the frequency of the embeddings update, but what can we do about the latter cause?

When we perform gradient calculations, some triples do not get updated. These are the pairwise triples, for which the loss of the false triple is already bigger than the loss of the true triple plus the margin.

Regarding these triples, we don't send their respective entities and relations embeddings back to the master node because it is useless and time-consuming to return gradients whose value is zero. Before sending the gradients back to the master, we delete the vectors with no gradient updates (zero gradients), minimizing in this way the amount of data to be transferred and updated.

G. Why didn't we scale our model with multiple GPUs ?

In the last years, a rapidly growing interest in the usage of GPU [12] for training machine learning models has been observed. Due to the capacity to scale horizontally and the fact that they have their memory, GPU-s are found very useful for speeding up the training process. Increasing the number of graphics cores will increase the speed of the processes that are being run on it. This technique is practical, but only for training models with a moderate number of parameters and dataset size. If the data and model parameters do not fit in the GPU, it will cause a bottleneck while doing the transfers from CPU to GPU. A solution for this case would be to reduce the size of the dataset or the number of parameters used in the model, but this approach does not work well for a very large dataset or a high number of embeddings dimensions used in KGE models. In the paper [13], Da Zheng et al. have developed DGL-KE, a package for KGE computations in different environments: multi-processing, multi-GPU, and distributed parallelism. According to their evaluation experiments, DGL-KE needs 100min to compute embeddings on an EC2 instance with 8 GPUs and less than 1/3rd of that amount of time to do the same task on an EC2 cluster with 4 machines with 48 cores/machine. Apart from the previously mentioned reasons, we aim to bring to a meeting point research approaches with applications in industry. Practically, many companies would choose CPU-s over GPU-s due to the price difference. In conclusion, we decided to develop a model that runs in a distributed fashion in a cluster of nodes.

IV. Experimental Design

A. Technology

We decided to implement our model using Scala since it's a programming language with strong concurrency support, which is a crucial property when it comes to parallelized processing. We use Apache Ignite [14] to obtain the shared memory layer. We used Google Kubernetes Engine (GKE) platform [15] as our testing environment. Due to the lack of funding we could only use the free resources that consist of a maximum of 8 simultaneously running vCPUs. Detailed information on how to set up the environment are given in our Git repository.

B. Baseline

Our work aim is to develop a distributed model of TransE. Therefore, we find it reasonable to refer to the results of the original TransE model as a baseline for our model evaluations, in terms of accuracy and runtime. To provide more accurate comparisons between these models, we ran in parallel identical experiments with our model in Scala as well as with the original TransE model implemented in C++, which can be found here.

C. Datasets

We trained and tested our distributed model using WordNet[16] and Freebase[17] datasets. We performed evaluations of our model using WN18 and WR18RR datasets. To compare our results with the baseline model evaluations we trained our model with FB15K database. Further, we performed evaluations also with FB15K-237 dataset.

D. Evaluation Metrics

Our evaluation protocol includes the same metrics used at our selected baseline model. The evaluation metrics include Hit@k with k= 10 and mean rank. We calculated both raw and filtered results with these metrics for head and tail corruption.

E. Testing environment

We created several GKE clusters to perform multiple tests, with different number of working nodes. Having a limited number of resources that we could use, we tested our approach in the following clusters:

- Cluster of 2 nodes, each node is of machine-type [18]: n1-standard-4. Thus, each node had 4 vCPU, 15GB Memory and $10(Gbps)$[3] Network egress bandwidth [3]
- Cluster of 3 nodes, each node is of machine-type: n1-standard-2. Thus, each node had 2 vCPU, 7.50GB Memory and $10(Gbps)$[3] Network egress bandwidth
- Cluster of 4 nodes, each node is of machine-type: n1-standard-2. Thus, each node had 2 vCPU, 7.50GB Memory and $10(Gbps)$[3] Network egress bandwidth

Given the fact that these nodes are very slow compared to a normal computer, for every evaluation that we performed in the cluster, we conducted an identical experiment in a single node to have fair runtime comparison analysis. Network egress bandwidth is up to the specified limit. Actual

[3]Egress is the traffic that exits a network [19]

performance depends on factors such as network congestion or protocol overhead [18].

F. Experimental results

The first experiment that we performed with our model was to test its performance in our local machine. This step was important to ensure that we implemented the TransE model correctly, and it produces the same results as the original code. We also ran our code in GKE cluster with a single node to test whether the model accuracy is preserved. The chart displayed in Figure 3, shows the accuracy comparisons between the original TransE code in C++ and our code in Scala running locally and our code running in GKE cluster with the same dataset and parameters:
Dataset : WN18, number of dimesions:20, learning rate: 0.01, margin : 1, distance: L_1, number of epochs : 1000.

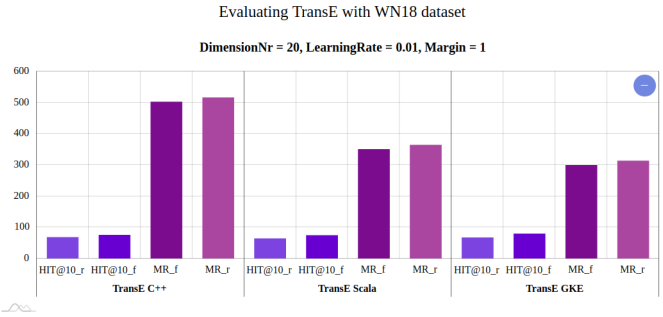

Figure 3: Accuracy comparisons.

1) ***Running on a 2-nodes cluster****:* We performed many experiments running our model with all the above-mentioned datasets in a GKE cluster with 2 worker nodes. After multiple hyper-parameters perturbations, the best results that we obtained while training the model with all the datasets are those shown in Figure 4. We performed evaluations for both head and tail corruption, but we present here only the one with the highest accuracy result.

Dimension	LearnRate	Margin	HIT@10 raw	HIT@10 filtered	MR raw	MR filtered
WN18						
20	0.01	1	67	79	313	299
50	0.02	1	77	91	471	457
50	0.005	2	77	90	445	431
WN18RR						
50	0.01	2	39	41	4488	4483
50	0.005	2	40	41	4697	4692
FB15k						
50	0.005	1	42	53	191	90
50	0.005	2	43	53	195	110
FB15k-237						
20	0.001	2	31	38	206	186
50	0.001	2	36	46	203	179
50	0.001	3	36	45	173	151

Figure 4: Accuracy results in 2-nodes cluster.

We can notice from Figure 4 that for certain combinations of hyper-parameters, the results of our model are better than those from the original paper Figure 5.

Dataset	WN				FB15k				FB1M	
Metric	Mean Rank		Hits@10 (%)		Mean Rank		Hits@10 (%)		Mean Rank	Hits@10 (%)
Eval. setting	*Raw*	*Filt.*	*Raw*	*Filt.*	*Raw*	*Filt.*	*Raw*	*Filt.*	*Raw*	*Raw*
Unstructured [2]	315	304	35.3	38.2	1,074	979	4.5	6.3	15,139	2.9
RESCAL [11]	1,180	1,163	37.2	52.8	828	683	28.4	44.1	-	-
SE [3]	1,011	985	68.5	80.5	273	162	28.8	39.8	22,044	17.5
SME(linear) [2]	545	533	65.1	74.1	274	154	30.7	40.8	-	-
SME(bilinear) [2]	526	509	54.7	61.3	284	158	31.3	41.3	-	-
LFM [6]	469	456	71.4	81.6	283	164	26.0	33.1	-	-
TransE	**263**	**251**	**75.4**	**89.2**	**243**	**125**	**34.9**	**47.1**	**14,615**	**34.0**

Figure 5: Accuracy results of original TransE paper [1]

For the dataset FB15k our model's optimal configurations were:

$$k = 50, \lambda = 0.005, \gamma = 1, d = L_1$$
$$k = 50, \lambda = 0.005, \gamma = 2, d = L_1$$

Evaluating our model with these hyper-parameters resulted in better accuracy measures than the original paper, for both HIT@10 and Mean Rank metrics.
Using the dataset WN18 our model's optimal configurations were:

$$k = 50, \lambda = 0.02, \gamma = 1, d = L_1$$
$$k = 50, \lambda = 0.005, \gamma = 2, d = L_1$$

With these configurations our model achieves higher HIT@10 percentage than the original paper.
As it appears in Figure 6, training the model in a distributed fashion preserves its efficiency. We noticed from several experiments that the achieved HIT@10 results remain almost the same, meanwhile the Mean Rank evaluations produce better accuracy in a distributed environment.
Regarding the runtime analysis, we can notice from Figure 7 that distributing the model in 2 nodes accelerates both training and testing the model. We performed the evaluations in the GKE cluster, running the model in a single working node and two working nodes to make fair comparisons since one node in GKE is much slower than a normal computer. We performed the evaluations with the FB15k dataset, with a different number of epochs. We can notice that for more epochs the difference in runtime for single workers versus 2 workers, increases.
We should keep in mind that the amount of data communicated back to the master node for every epoch is irrelevant to the number of nodes in the

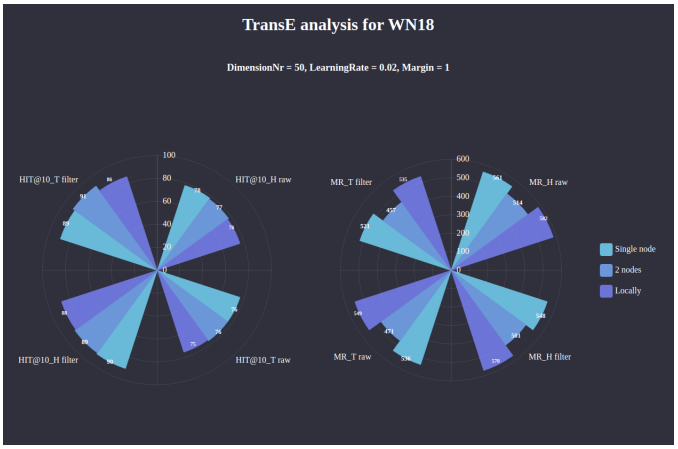

Figure 6: Some other accuracy measures comparisons of our model tested in: a single node in Kubernetes cluster, 2 nodes in Kubernetes cluster and in our local machine. We performed the experiments in the local machine to check if the accuracy of the models changes when we move from local to a distributed environment.

cluster. That means that having more nodes in the cluster would decrease the time needed to run every epoch, without additional network communication or communication time consumption.

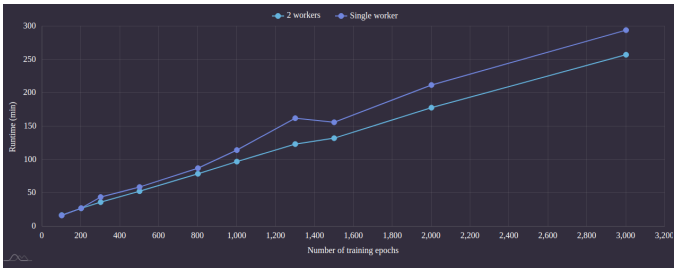

Figure 7: Runtime analysis of TransE in single node versus two nodes in Kubernetes cluster for various training experiments. The X-axis shows the number of training epochs for every experiment.

2) ***Running on a 3-nodes cluster****:* We created another cluster in the GKE platform to test the model with more working nodes. Our next cluster was composed of 3 nodes of machine type: n1-standard-2 with 2vCPU, 7.50GB memory, 10 $(Gbps)^3$ network egress bandwidth. To obtain equitable evaluations regarding the runtime metrics, we ran our model in this cluster using a single node, then we performed the exact experiments distributing the model in 2 and 3 nodes. Below we show in the accuracy and runtime results of the evaluations using FB15k dataset with these hyperparameters: $k = 50, \lambda = 0.01, \gamma = 1, d = L_1$

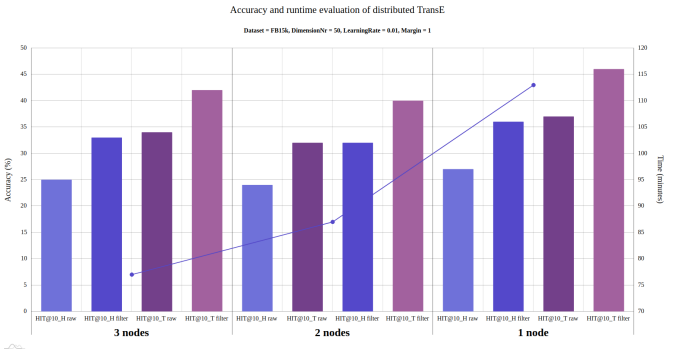

Figure 8: Accuracy and runtime results of our code in Scala with FB15k dataset, running in GKE cluster with 1, 2 and 3 nodes. The bars show the HIT@10 values and the line dots represent the runtime for each of the 3 different experiments.

As we can see from the Figure 8, the runtime of our model is inversely proportional with the number of nodes in the cluster. The runtime decreases with the increase of the number of nodes used to distribute it. Regarding the accuracy measures, they are close to the original results Figure 9.

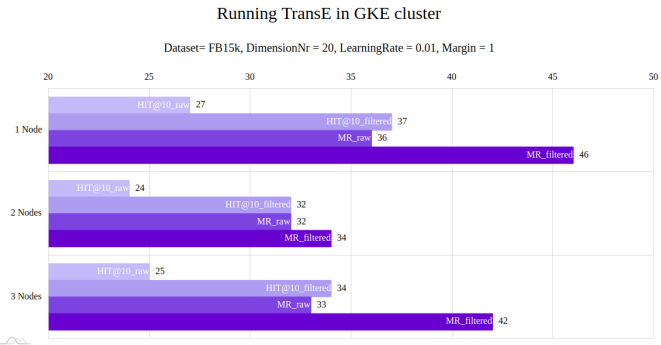

Figure 9: HIT@10 accuracy result of our code in Scala with FB15k dataset, running in GKE cluster with 1, 2 and 3 nodes.

3) ***Running on a 4-nodes cluster****:* To evaluate our model in a bigger environment, we created another cluster with 4 nodes and the same node capacity as the cluster above. From our experiments, we conclude that running the model in a cluster with 3 and 4 nodes results in the same accuracy measures. The number of nodes does not affect the accuracy, but increasing the number of training epochs has shown to improve the results. The reason for this is the fact that the distributed environment slows down the convergence. For this reason, we trained our model with a number of 1200 or 1500 epochs.

The results we present in Figure 10, show that

running the model in a cluster with 4 nodes requires less time than running it in a single working node. We can notice that increasing the number of epochs increases the runtime difference between running the model in a single node versus 4 nodes. This happens due to the fact that network bandwidth is very low. Thus, when we run short tasks, the network communication overhead takes over the speed-up effect of the distribution in such a small cluster.

Figure 10: Runtime results of our code in Scala with FB15k dataset, running in GKE cluster with 1 versus 4 nodes.The first column shows the number of training epochs performed in every single experiment.

V. Related Work

In the last decade, the amount of data stored as knowledge graphs is increasing rapidly and continuously. Many researchers have been focused on finding effective ways to scale machine learning models, as a response to the need of learning these KG using machine learning algorithms. Proposals for parallel algorithms/architectures design date back to 1995. The first methods on this subject proposed the distribution of machine learning algorithms by parallelizing the SGD method.

In the context of knowledge graphs, in 2015, the paper "Scalable Learning of Entity and Predicate Embeddings for Knowledge Graph Completion" presents an approach for faster training of KG. The authors have proposed Adaptive Learning Rates, a method for reducing the training time of KGE models.

In 2017 Xiao-Fan Niu et al.[20] proposed ParaGraphE. This is a framework for parallelizing Knowledge Graph embeddings models among CPU cores. ParaGraphE algorithm **??** works for all the KGE models that use of a margin ranking loss function, similar to the one used in TransE. Specifically, they have implemented the following models into their framework: TransE, TransH, TransR, TransD and SphereE. They evaluate their approach using a maximum number of 10 CPU cores and they show that the accuracy metrics of their models for WN and FB datasets are close to the reported values of the respective original models. The authors do not show the optimal parameters for achieving the reported results and the model has not been extended for the other KGE models, for example: similarity-based models.

In the same year, another framework for efficient parallel training of KGE models was proposed by Zhang et al. [21] in the paper "Efficient Parallel Translating Embedding For Knowledge Graphs", the authors propose a speedup algorithm to asynchronously train large and sparse KGs among CPU cores with shared memory. They make use of the law of collision emerging to tackle the issue of simultaneous updates in a single instance of data, in this case of the same embeddings vector. They convert the knowledge graph to a hypergraph to avoid update collisions. Due to the limitation of application just in sparse training data, the authors have developed and tested this framework only for TransE, TransH and TransE-AdaGrad models, which means that we can not use this approach for models with dense training data.

In 2019 Lerer et al.[22] presented in their paper "Pytorch-BigGraph a different approach for large-scale KGE models". This system uses the block decomposition method to partition the entities of the KG based on the type of the head and tail. This approach requires two main modifications of the traditional KGE model's algorithm:

1) The negative samples are not generated fully randomly, but rather from the set of different entities present in the current partition.
2) Training triples are not selected from a uniform I.I.D since they are dependent on the partition's groups of the head and tail entity.

These steps cause a slower convergence, which is by alternating the buckets frequently.

The efficiency of PBG has been a motivation for the development of other frameworks within the same

context of the application. One of the most recent ones is DGL-KE [13], which is implemented based on PBG combined with the Key-Value storage of C++. This framework implements various optimization methods to improve the training of KGE models, offering 3 levels of scaling, using: multiple CPU cores, multiple GPU cores, or a cluster of worker nodes. The results of the evaluations of these models, approximate the reported results of the respective original models, excluding the cases with large embeddings dimensions. Besides the broad application and its accuracy level, a drawback of this framework is that it does not work well with very large datasets. This is caused by low computational effectiveness, which makes the training time-consuming. the high data-transfer overhead plays an important role in this flaw of PBG.

VI. Conclusions and Future Work

The main work of this thesis is to tackle the issue of high resource requirements and long runtime while training KGE models. Our solution for resource availability is to distribute the training of these models in a cluster of working nodes. This way, we have a higher CPU and memory capacity available. We prove that using a distributed environment reduces the overall runtime of the model. To test our approach, we chose to work with TransE model since it is a simple and widely used as a baseline for the state-of-the-art proposals.

We developed a distributed version of TransE using a shared-memory layer. We set up the environment in the Kubernetes engine platform of Google Cloud to perform the evaluations of our model. We performed multiple evaluations with various hyper-parameters and different datasets, such as WN18, WN18RR, FB15k, and FB15k-237. The metrics that we used to evaluate our model accuracy are HIT@10 and Mean Rank.

Besides our cluster's size limitations and node's resources capacity, our model has shown to be able to reproduce the accuracy measures reported on the traditional TransE model. For some of the cases, we discovered hyper-parameters combinations that produce better results than the original TransE. We noticed that increasing the number of epochs while training our model, improves its accuracy. This happens due to the slower convergence caused by the distributed environment.

Apart from the accuracy measures, we also report the runtime analysis of our model. We provide comparisons between running the model in a single node versus running it in a cluster with 2, 3 and 4 nodes. We also provide comparisons between our model implemented in Scala and the original model implemented in C++, to denote that our model reduces the runtime while preserving the accuracy measures of the original TransE model. Even though the nodes used for the cluster are much slower than a normal computer, each node has 2 vCPU capacity, our model demonstrates an inversely proportional relationship between the cluster size and the runtime of the model. An increasing number of workers reduces the runtime of training and the evaluation of the model.

Our future work includes extending our model for other KGE models. We are now working on distributing TransH and DistMult model and our next step will be to implement it for the MDE model. We aim to have a platform of distributed KGE models for all the translational distance model, later on, we want to extend it further for the semantic matching models as well. We will also develop an interface for this platform, to ensure usability even for inexperienced users.

Extending our approach for other KGE models will require the implementation of more evaluation metrics to measure their accuracy.

An important step to do in the future is to set up a real distributed environment, to properly evaluate the runtime speed-up of our model compared to the runtime of the original models. We need to have a cluster with a minimum of 10 working nodes of machine-type: n1-highcpu-16. Having a real cluster will give us the possibility to efficiently test our model with larger datasets, for example, FB1M.

We plan to experiment implementing several accuracy optimizations methods for KGE models [23] [24], also different negative sampling methods [25].

Finally, we are planning to experiment with the run-time speed-up of our models, by incorporating the usage of multiple GPU cores.

References

[1] 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 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 2787–2795. [Online]. Available: http://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf

[2] C. Co, "Google knowledge graph," https://clix.co/things-not-strings-google-knowledge-graph/, 2012.

[3] Q. Wang, Z. Mao, B. Wang, and L. Guo, "Knowledge graph embedding: A survey of approaches and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 12, pp. 2724–2743, 2017.

[4] B. W. Quan Wang, Zhendong Mao and L. Guo, "Knowledge graph embedding: A survey of approaches and applications." [Online]. Available: https://persagen.com/files/misc/Wang2017Knowledge.pdf

[5] J. Feng, "Knowledge graph embedding by translating on hyperplanes," 06 2014.

[6] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," *Proceedings of AAAI*, pp. 2181–2187, 01 2015.

[7] G. Ji, S. He, L. Xu, K. Liu, and J. Zhao, "Knowledge graph embedding via dynamic mapping matrix," 01 2015, pp. 687–696.

[8] G. Ji, K. Liu, S. He, and J. Zhao, "Knowledge graph completion with adaptive sparse transfer matrix," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 985–991.

[9] S. U. Vishakh Hegde, "Parallel and distributed deep learning." [Online]. Available: https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf

[10] S. H. Roosta, "Parallel processing and parallel algorithms," 2000.

[11] M. Li, D. Andersen, J. Park, A. Smola, A. Ahmed, V. Josifovski, J. Long, E. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," *Proc. OSDI*, pp. 583–598, 01 2014.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng, "Large scale distributed deep networks," *Advances in neural information processing systems*, 10 2012.

[13] C. M. Z. T. Z. Y. J. D. H. X. Z. Z. G. K. Da Zheng, Xiang Song, "Dgl-ke: Training knowledge graph embeddings at scale." [Online]. Available: https://arxiv.org/abs/2004.08532

[14] A. I. I.-M. C. Platform, "Apache ignite„" https://ignite.apache.org/, 2015,.

[15] G. C. Platform, "Kubernetes engine," https://console.cloud.google.com/, 2008,.

[16] C. Fellbaum, "Wordnet: An electronic lexical database„" https://wordnet.princeton.edu/, 1998,.

[17] M. Technologies, "Freebase„" https://lod-cloud.net/dataset/freebase, 2007,.

[18] G. C. Platform, "Machine types„" https://cloud.google.com/compute/docs/machine-types, 2008,.

[19] I. Aviatrix Systems, "What do egress and ingress mean in the cloud?" https://a.aviatrix.com/learning/cloud-security-operations/egress-and-ingress/, 2020,.

[20] W.-J. L. Xiao-Fan Niu, "Paragraphe: A library for parallel knowledge graph embedding," 2017.

[21] D. Zhang, M. Li, Y. Jia, and Y. Wang, "Efficient parallel translating embedding for knowledge graphs," 03 2017.

[22] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large-scale graph embedding system," 03 2019.

[23] W. Z. Siheng Zhang, Zhengya Sun, "Improve the translational distance models for knowledge graph embedding." [Online]. Available: https://link.springer.com/article/10.1007/s10844-019-00592-7

[24] W. M. Robert Bamler, Farnood Salehi, "Augmenting and tuning knowledge graph embeddings." [Online]. Available: http://auai.org/uai2019/proceedings/papers/172.pdf

[25] V. N. Bhushan Kotnis, "Analysis of the impact of negative sampling on link prediction in knowledge graphs," 2017. [Online]. Available: https://arxiv.org/abs/1708.06816

[26] M. Nickel, V. Tresp, and H.-P. Kriegel, "A three-way model for collective learning on multi-relational data." 01 2011, pp. 809–816.

[27] B. Yang, W.-t. Yih, X. He, J. Gao, and l. Deng, "Embedding entities and relations for learning and inference in knowledge bases," 12 2014.

[28] M. Nickel, L. Rosasco, and T. Poggio, "Holographic embeddings of knowledge graphs," 10 2015.

[29] T. Trouillon, J. Welbl, S. Riedel, E. Gaussier, and G. Bouchard, "Complex embeddings for simple link prediction," 10 2016.

[30] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, "A review of relational machine learning for knowledge graphs," *Proceedings of the IEEE*, vol. 104, 12 2015.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2015.

[32] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," *Proceedings of AAAI*, pp. 2181–2187, 01 2015.

[33] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A survey on knowledge graphs: Representation, acquisition and applications," *ArXiv*, vol. abs/2002.00388, 2020.

[34] A. I. I.-M. C. Platform, "Apache ignite cache modes„" https://apacheignite.readme.io/v1.2/docs/cache-modes, 2015,.

[35] P. Minervini, N. Fanizzi, C. d'Amato, and F. Esposito, "Scalable learning of entity and predicate embeddings for knowledge graph completion," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 2015, pp. 162–167.