# Parallel and Scalable Implementations of Trans-E and Trans-H in Apache Spark using SPLASH

Krishna Manglani and Srinivas.K
Department of Computational and Data Science
Indian Institute of Science, Bangalore, India
srinivas.jdi@gmail.com krish2100@gmail.com

***Abstract*—Embedding a multi-relational graph in vector spaces while preserving certain properties of the graph is an important emerging trend in the area of Representation Learning. Once an embedding is learnt, it can be used for other tasks like link prediction and knowledge graph completion. Three of the most promising embeddings are those given by the algorithms - TransE, TransH and TransR. While these algorithms have been proven to work on small knowledge graphs, they need to be parallelized to run on really large knowledge graphs. In this project, we have explored the challenges involved in parallelization, identified the scalability bottlenecks and provided solutions to overcome these issues.**

## I. Introduction

Multi-Relational Graphs are an interesting way to store information of different kinds. Once information has been encoded as a graph, we can use it for many other tasks like link prediction and for efficiently collecting all the information available about any particular entity. Google has been successfully using its own multi-relational graph called as the Knowledge Graph to increase the quality of its search engine.

However, as the size of the graph increases, the complexities involved in inferring from them also increases drastically. One of the most important tasks that a multi-relational graph is used for is that of link prediction which is used to predict if there exists any relationship between two entities in the graph. Another important task is that of knowledge-graph completion which is used to discover the kind of relationship that exists between two entities in the graph. In order to perform these tasks, we need to first represent them in an appropriate manner. Regular graph representation techniques like that of an adjacency matrix or list don't work in this situation as we have a multi-relational graph and also such a representation is not of much use in inferring new knowledge from them.

Any good representation of these graphs must preserve the structural information available in the graph. An emerging approach to achieve such a representation is to embed the graph in a low-dimensional vector space. There have been many efforts in the past few years in this direction like Structured Embedding,Unstructured Model,Latent Factor Model,Semantic Matching Energy,Neural Tensor Model, TransE [1], TransH [2] and TransR [3] . The most promising ones among these are the TransE, TransH and the TransR approaches.

All the three approaches described above are very similar to each other. The fundamental idea in these is to represent every entity and relation in the graph with a point in the high dimensional vector space. Multi-relation graph are generally given in the form of a triplet *(entity1, relation, entity2) or ( e1, rel, e2)*. The presence of such a triplet in a multi-relational graph represents the fact that *entity1* is linked to *entity2* and the kind of link is given by the *relation*. After embedding the graph in the vector space, we get a vector corresponding to each entity and relation. Let the vectors corresponding to *entity1* and *relation* be **e1**,**e2** and **rel** respectively. Then in all these three models, we want them to satisfy the following equation **e1 + r = e2** and as far as possible, for those triplets which are not a part of the graph, this equation should not satisfied.

While all three algorithms work on this same principle, there are subtle differences among them. TransE embeds all the entities and relations in a single vector space, TransH embeds the entities linked by a relation as projections of the vectors onto a hyperplane corresponding to the relation. TransR embeds the entities and relations in two different vector spaces. Corresponding to every relation there is a projection matrix $M_r$ which defines a mapping from from the entity space to the relation space. The entities related by **rel** are first projected onto the relation space using the projection matrix and then the difference $M_r$**h + r -** $M_r$**t** is calculated.

These algorithms have so far been implemented only on small multi-relational graphs. In order to scale them up for larger graphs, existing implementations cannot work as it would take a lot of time to train such large models. In this project, we have tried to address the challenges involved in parallelizing and scaling up the existing implementations. We have analyzed the existing parallelization strategies, identified the scalability bottlenecks and provided solutions to address these issues.

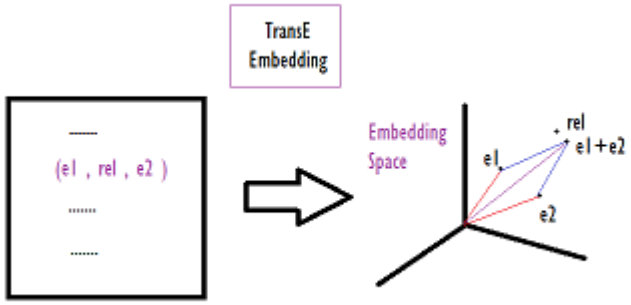


Fig. 1. The basic idea behind TransE

## II. Related Work

### A. Parallel Algorithms

The core of all the three algorithms is the stochastic gradient descent updates during the training phase. Though SGD is inherently serial in nature, a lot of effort has gone into coming up with parallelization of SGD updates for different scenarios. Most of the approaches can be classified as asynchronous and synchronous. In the asynchronous side, the most notable one is that of Hogwild! [4]. In the synchronous side, where the weight vectors are synchronized among the partitions, algorithms differ in the way in which conflicting updates are combined. Some famous approaches are those of bounded delay updates [5] and re-weighted averaged updates [6].

### B. Implementation Frameworks

Depending on the parallel algorithm we choose to use, the implementation framework needs to be chosen. Since the algorithm is iterative in nature and we want to scale it up to large graphs, we have chosen to implement the algorithms on a Spark cluster. But we now need to add a communication layer on top of existing the existing infrastructure to synchronize the weight vectors efficiently. This is generally done using an implementation of a parameter server. While there are many implementations of parameter servers [7] [8] available for different purposes, there are no open source implementations that have been integrated to work on big-data frameworks like Hadoop or Spark.

## III. Algorithms

### A. TransE

*1) The score function and SGD update:* As the figure shows we need to learn an embedding such that $||h + r - t||$ is minimized for samples present in the graph and not satisfied for samples not present in the graph. Here **h** represents the embedding of entity1, **t** represents the embedding of entity2 and **r** represents the embedding of relation. The score function in this algorithm is given as $f_r(h,t) = ||h + r - t||_2^{\ 2}$.

**Algorithm 1** Testing

```
Load Relation and Entity Vectors in RDD
Load Test Data in Test RDD
MeanRank and Hits@10 Computation:
  Map:
    Input: ⟨e1, e2, rel⟩
      w_e1 = Entity(e1)
      w_e2 = Entity(e2)
      w_rel = Relation(rel)
      w_left = w_e1 + w_rel
      w_right = w_rel - w_e2

      //compute left rank and left hits@10
      for every entity e do
        w_e = Entity(e)
        left_energy(e) = ||w_left-w_e||^2
      Sort(left_energy)
      left_rank = get_rank(left_energy,e2)
      if(left_rank ≤ 10)
        left_Hits@10 = 1
      else
        left_Hits@10 = 0

      //compute right rank and right hits@10
      for every entity e do
        w_e = Entity(e)
        right_energy(e) = ||w_right+w_e||^2
      Sort(right_energy)
      right_rank = get_rank(right_energy,e1)
      if(right_rank ≤ 10)
        right_Hits@10 = 1
      else
        right_Hits@10 = 0

    emit: (left_rank,right_rank,left_Hits@10, right_Hits@10)

  left_MeanRank = left_rank.mean()
  left_MeanHits@10 = left_Hits@10.mean()

  right_MeanRank = right_rank.mean()
  right_MeanHits@10 = right_Hits@10.mean()
```

*2) Initialization:* The embedding vectors for the entities and relations are first initialized randomly for this algorithm.The embeddings of relations are normalized.

*3) Training:* The loss function is defined as follows :From the score function we define our loss function to be
$L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} max(0, f_r(h,r,t) + f_r(h',r,t'))$

The training is done on the loss function by stochastic gradient descent.

*4) Testing:* We use the metrics of mean rank and hits@10 to measure performance.The algorithms for calculating mean

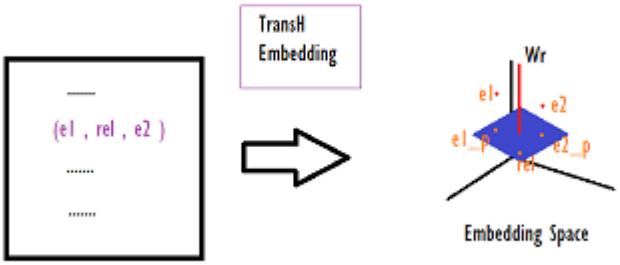


Fig. 2. The basic idea behind TransH

rank and hits@10 are given in the table.

### B. TransH

*1) The score function and SGD update:* As the figure shows we need to learn an embedding such that $||h+r-t||$ is minimized for samples present in the graph and not satisfied for samples not present in the graph. Given a relation **r**, we define a vector $W_r$ which defines a hyperplane in the embedding space. Then the mapped vectors are projected onto this plane and let $h_p$ and $t_p$ represent the projected vectors. The score function in this algorithm is given as $f_r(h,r,t) = ||h_p + r - t_p||_2^2$.

*2) Initialization:* The embedding vectors for the entities and relations are first initialized randomly for this algorithm.The embeddings of relations are normalized.

*3) Training:* From the score function we define our loss function to be
$L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} max(0, f_r(h,r,t) + f_r(h',r,t'))$

The training is done on the loss function by stochastic gradient descent.

*4) Testing:* As before, we use the metrics of mean rank and hits@10 to measure performance.

### C. TransR

*1) The score function and SGD update:* As the figure shows we need to learn an embedding such that $||h+r-t||$ is minimized for samples present in the graph and not satisfied for samples not present in the graph. Given a relation **r**, we define a matrix $M_r$ which defines a mapping from the entity embedding space to the relation embedding space. Then the mapped vectors are then used to calculate the score function.The score function in this algorithm is given as $f_r(h,r,t) = ||M_r(h) + r - M_r(t)||_2^2$.

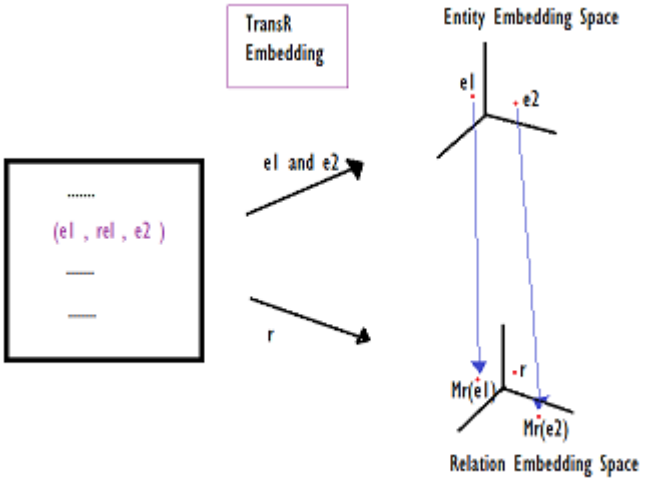


Fig. 3. The basic idea behind TransR

*2) Initialization:* The embedding vectors for the entities and relations are first initialized as the output of TransE algorithm of run on the correct space.The embeddings of relations are normalized.

*3) Training:* From the score function we define our loss function to be
$L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} max(0, f_r(h,r,t) + f_r(h',r,t'))$.

The training is done on the loss function by stochastic gradient descent.

*4) Testing:* As before, we use the metrics of mean rank and hits@10 to measure performance.

## IV. Strategies for Parallelization

The algorithms for parallelization have to be designed keeping in mind the architecture using which they will be implemented and vice-versa. For example, the bounded delayed update scheme provided in [5] was designed keeping in mind that the communication cost need to be reduced. While Hogwild! [4] has been proven to converge faster, implementing it in a distributed setting would incur heavy network traffic due to the high communication cost in each iteration. The standard solution is to use an implementation of a parameter server. While parameter servers have been very useful for such purposes, implementing them and integrating them with big-data frameworks can be quite challenging.

While Spark is yet to come up with an integrated parameter server, there are many other interfaces and frameworks that have been created to support development of applications on top of Spark. One such interface is that of **SPLASH** which stands for **S**ystem for **P**arallelizing **L**earning **A**lgorithms with

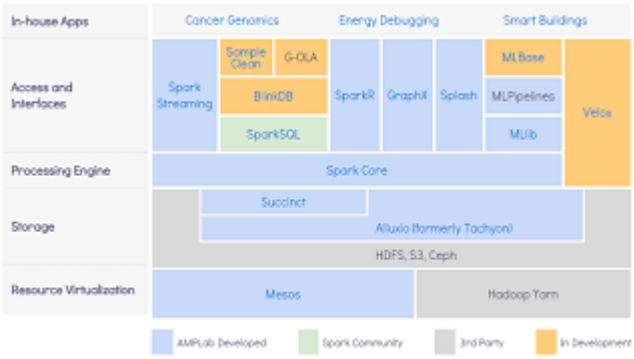


Fig. 4. Berkeley Data Analytics Stack

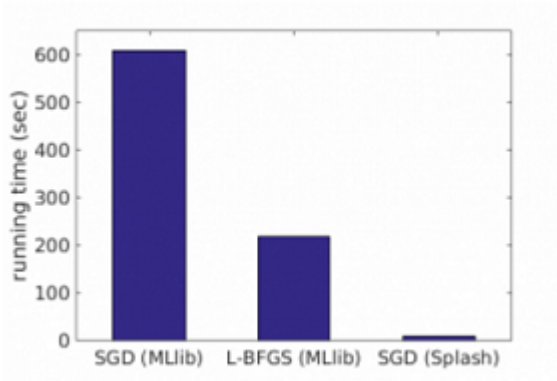


Fig. 5. Comparison of running time for SPLASH over other implementations for a 10 class logistic regression model using mnist8m dataset [6]

**S**tochastic **M**ethods.

SPLASH offers a suite of features that suit our needs for parallelization. The first one is that it acts like a (single master - multiple client) parameter server through a feature called shared variables. While creating the spark RDD , it allows the user to declare some of the variables as shared variables and others as local variables. While local variables in a partition are inaccessible from other partitions,shared variables are replicated across all the partitions and synchronized after every iteration. Thus, SPLASH acts implicitly like a parameter server and even performs the execution of the communication/synchronization step using a tree-reduce algorithm. The user needs to only declare the variables that need to be synchronized as shared variables.

While combining the updates from different machines, SPLASH does a re-weighting of the samples to improve convergence. The need for such a scheme and other details of this re-weighting scheme can be found in [6]. Also, the speed-ups, convergence results of this re-weighting strategy have been explained in detail in [6]. So, we have chosen to implement the three algorithms using the SPLASH interface.

## V. Scalability Bottlenecks

When we tried to scale up the implementation for really large graphs, we identified two potential scalability bottlenecks - sampling and synchronization.

### A. Synchronization of large number of parameters

The running time is dominated by the synchronization of shared variables while running on large datasets. This doesn't pose a problem for us due to the lazy evaluation scheme inherent in spark and SPLASH.

### B. Sampling

This proves to be the most challenging scalability issue. As discussed earlier, the SGD update require a negative sample for every triplet in the graph. This could be obtained easily for small datasets using a hashtable. The entire graph is stored in a hash table. Given a triplet present in the graph, another triplet is created which has the same relation but with one of the entities is different. Then, if this alternate triplet is not available in the hash-table, it is added to the triplet as a negative sample. While serial implementations perform this random sampling during the algorithm, it is not scalable. We have decided to create a dataset with both positive and the corresponding negative samples as a pre-processing step to this algorithm.

**Algorithm 2** getReplica(avg,max,min,density)

**if** (avg $\leq$ density) return 1
**else if** $\left( \frac{avg + min}{2} \leq \text{density} \right)$ return 2
**else if** $\left( \frac{avg + 3min}{4} \leq \text{density} \right)$ return 4
**else** return 10

**Algorithm 3** corrupt_right_elem( $\langle (e1, rel), \{e2\} \rangle$)

m = replication_factor(e1)
Declare Corrupted List = {1 2 .. entity_num }
Remove all e2 from corrupted list
**for** each e2 do
  **repeat** m times do
    take random t_corrupt from corrupted list
    emit $\langle (e1, e2, rel, e1, t_{corrupt}, rel) \rangle$

**Algorithm 4** corrupt_left_elem( $\langle (e2, rel), \{e1\} \rangle$)

m = replication_factor(e2)
Declare Corrupted List = {1 2 .. entity_num }
Remove all e1 from corrupted list
**for** each e1 do
  **repeat** m times do
    take random h_corrupt from corrupted list
    emit $\langle (e1, e2, rel, h_{corrupt}, e2, rel) \rangle$

**Algorithm 5** Pre-processing Phase

**Initialize:** `entity2id` and `relation2id` maps
Map each entity and relation in test and training datasets to ids

**Density computation:**
**Map:**
Input: $\langle e1, e2, rel\rangle$
emit: $\langle e1, 1\rangle$ , $\langle e2, 1\rangle$
**Reduce:**
**Input:** $\langle e, \{v\}\rangle$
**Output:** $\left\langle e, 50\ \frac{\Sigma v}{Train.size}\right\rangle$
max = max(Density.item2)
min = min(Density.item2)
avg = mean(Density.item2)

**Replication Factor Computation**
**Map:**
**Input:** $\langle e, \text{density}\rangle$
**Output:** $\langle e, \text{getReplica(avg,max,min,density)}\rangle$

**Right Corruption:**
Input: $\langle e1, e2, rel\rangle$
Map: $\langle e1, e2, rel\rangle \Rightarrow \langle (e1, rel), e2\rangle$
GroupByKey: $\langle (e1, rel), e2\rangle \Rightarrow \langle (e1, rel), \{e2\}\rangle$
Map: $\langle (e1, rel), \{e2\}\rangle \Rightarrow \{corrupt\_Right\_elem((e1, rel), \{e2\})\}$

**Left Corruption:**
Input: $\langle e1, e2, rel\rangle$
Map: $\langle e1, e2, rel\rangle \Rightarrow \langle (e2, rel), e1\rangle$
GroupByKey: $\langle (e2, rel), e1\rangle \Rightarrow \langle (e2, rel), \{e1\}\rangle$
Map: $\langle (e2, rel), \{e1\}\rangle \Rightarrow \{corrupt\_left\_elem((e2, rel), \{e1\})\}$

## VI. Experiments and Results

While the code is ready for all the three algorithms (TransE, TransH and TransR) and the pre-processing stage, due to lack of time we have been able to test only the TransE algorithm on one of the datasets used in [3]. The results of our implementation are shown below.

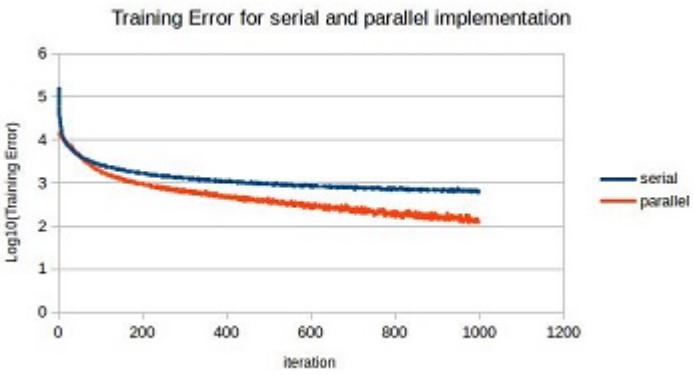


Fig. 6. Comparison of Training errors in FB15k

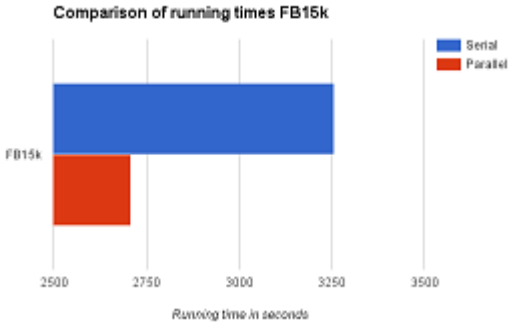


Fig. 7. Running Time for FB15k

## 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*, 2013, pp. 2787–2795.

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

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

[4] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[5] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 873–881.

[6] Y. Zhang and M. I. Jordan, "Splash: User-friendly programming interface for parallelizing stochastic algorithms," *arXiv preprint arXiv:1506.07552*, 2015.

[7] 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.

[8] W. Dai, J. Wei, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing, "Petuum: A framework for iterative-convergent distributed ml," 2013.