Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques

Adrian Kochsiek
University of Mannheim
Mannheim, Germany
adrian@informatik.uni-mannheim.de

Rainer Gemulla
University of Mannheim
Mannheim, Germany
rgemulla@uni-mannheim.de

ABSTRACT

Knowledge graph embedding (KGE) models represent the entities and relations of a knowledge graph (KG) using dense continuous representations called embeddings. KGE methods have recently gained traction for tasks such as knowledge graph completion and reasoning as well as to provide suitable entity representations for downstream learning tasks. While a large part of the available literature focuses on small KGs, a number of frameworks that are able to train KGE models for large-scale KGs by parallelization across multiple GPUs or machines have recently been proposed. So far, the benefits and drawbacks of the various parallelization techniques have not been studied comprehensively. In this paper, we report on an experimental study in which we presented, re-implemented in a common computational framework, investigated, and improved the available techniques. We found that the evaluation methodologies used in prior work are often not comparable and can be misleading, and that most of currently implemented training methods tend to have a negative impact on embedding quality. We propose a simple but effective variation of the stratification technique used by PyTorch BigGraph for mitigation. Moreover, basic random partitioning can be an effective or even the best-performing choice when combined with suitable sampling techniques. Ultimately, we found that efficient and effective parallel training of large-scale KGE models is indeed achievable but requires a careful choice of techniques.

PVLDB Reference Format:
Adrian Kochsiek and Rainer Gemulla. Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques. PVLDB, 15(3): 633 - 645, 2022.
doi:10.14778/3494124.3494144

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/uma-pi1/dist-kge.

1 INTRODUCTION

Knowledge graph embedding (KGE) models [4, 5, 15, 23, 27, 28, 30, 35] represent the entities and relations of a knowledge graph (KG) in a low-dimensional continuous space; the resulting representations are referred to as *embeddings*. KGE models are used to reason about the KG and provide suitable representations for downstream learning tasks. In particular, KGE models have been explored for knowledge graph completion tasks [22], visual relationship detection [3], drug discovery in biomedical KGs [20], and recommender systems [31]. A large part of the available literature focuses on small KGs. Recently, a number of frameworks that are able to train KGE models for large-scale KGs by parallelization across multiple GPUs or machines have been proposed, namely PyTorch BigGraph [19], GraphVite [37], and DGL-KE [36]. Each of these frameworks proposed a number of different parallelization techniques. In this paper, we report on an independent investigation of the efficiency and effectiveness of these techniques within a common framework.

A knowledge graph (KG) consists of *facts* about a set of entities, where each fact is a (subject, relation, object)-triple such as (Hitchcock, Directed, Psycho). A KG can be viewed as a graph in which vertices correspond to entities, edges to triples, and edge labels to relations. KGE models represent each entity and each relation of the KG by an embedding (e.g., a 200-dimensional real vector); the embeddings are subsequently used to reason about the plausibility of unseen facts. Large-scale knowledge graphs contain millions of entities, which makes training of KGE models challenging. The Freebase[1] KG, for example, contains more than 80M entities; even relatively low-dimensional KGE models may exceed the memory of GPUs. Moreover, Freebase contains 100s of millions of triples, which leads to long training times. While parallel training methods can handle KGs of such scale and provide reasonable training times, their use may also impact model quality negatively.

The key problem in parallelizing KGE training is that the entity and relation embeddings need to be synchronized across workers. Accesses to these embeddings are required both when processing a triple from the KG but also in a step called *negative sampling* that is essential for training. Each access potentially involves communication between workers, which is costly. To reduce this cost, parallel training methods [19, 36, 37] try to keep embedding accesses local to each worker; this is done by carefully partitioning the KG across workers and by employing customized negative sampling techniques. The available techniques include *relation partitioning* [36], *graph-cut partitioning* [36], *stratification* [19, 21, 37], *shared sampling* [19, 36], *local sampling* [19, 36, 37], and *batch sampling* [19, 36]. A comprehensive study of the benefits and drawbacks of these techniques has not been conducted so far.

In this paper, we report on an extensive experimental study in which we investigated the efficiency and effectiveness of the available parallelization methods. To ensure a fair comparison, we re-implemented all techniques on top of the LibKGE framework [6]. We found that the evaluation methodologies used in prior work are

[1] https://developers.google.com/freebase/, last accessed on Nov. 15th 2021

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494144

often not comparable and can be misleading in that degradations in model quality due to parallel training may remain undetected. Our results suggest that current (combinations of) training methods tend to have a negative impact on embedding quality and/or do not provide substantial speedups. We propose a simple but effective variation of the stratification technique used in PyTorch BigGraph that mitigates these problems. We also found that on some datasets (and when combined with suitable sampling techniques) the random-partitioning baseline outperformed more sophisticated methods. Ultimately, we found that efficient and effective parallel training of large-scale KGE models is indeed achievable but requires a careful choice of techniques; the best choice is dataset-dependent. For example, training a large-scale KGE model for Freebase on 8 GPUs is possible with 8x speedup and a model quality that is competitive to sequential methods and exceeds prior results.

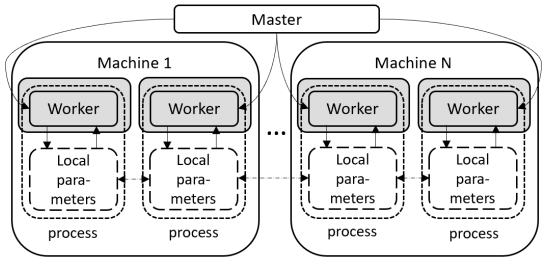


Figure 1: General architecture for parallel KGE training. A master process coordinates the distribution of data to workers. Parameters are distributed across workers using a parameter server. Training is performed on each worker in parallel using a GPU (marked in gray).

2 PRELIMINARIES

2.1 Background

Notation. We model a *knowledge graph* $\mathcal{K} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ as a collection of $N = |\mathcal{K}|$ subject-predicate-object (SPO) triples, where $\mathcal{E}$ denotes the set of entities and $\mathcal{R}$ the set of relations. KGE models associate an *embedding* with each entity and each relation; the embeddings are taken from a vector space specific to the respective KGE model. For example, in the well-known ComplEx model [30], each embedding constitutes a d-dimensional complex vector, where dimensionality d is a hyperparameter (e.g., 256). The plausibility of an SPO triple is modeled via a scoring function $f(\boldsymbol{s}, \boldsymbol{p}, \boldsymbol{o})$, which uses the embeddings of subject ($\boldsymbol{s}$), predicate ($\boldsymbol{p}$), and object ($\boldsymbol{o}$) of the triple as inputs. Generally, high scores indicate that the triple is likely to be correct according to the model, low scores indicate incorrect triples. For ComplEx, the scoring function has the form

$$f(\boldsymbol{s}, \boldsymbol{p}, \boldsymbol{o}) = \mathrm{Re}(\boldsymbol{s}^\top \operatorname{diag}(\boldsymbol{p})\bar{\boldsymbol{o}}), \tag{1}$$

where $\mathrm{Re}(x)$ refers to the real part of $x \in \mathbb{C}$ and $\bar{\boldsymbol{x}}$ to the element-wise complex conjugate of $\boldsymbol{x} \in \mathbb{C}^d$.

Models and frameworks. Many KGE models–e.g., RESCAL [23], TransE [5], DistMult [33] ComplEx [30], RotatE [28], SimplE [15], RGCN [27], QuatE [35], TuckER [4] and HittER [7]–have been proposed in the literature; a key difference between these models is the form of the scoring function. Frameworks such as OpenKE [13], Ampligraph [8], PyKeen [2] or LibKGE [6] provide implementations of various KGE models as well as training algorithms and evaluation methods. Different KGE models are compared in [1, 26].

Negative sampling. In general, KGE models aim to assign high scores to *positive* (i.e., correct) triples and low scores to *negative* (incorrect) triples. The knowledge graph $\mathcal{K}$ generally provides only positive triples but no negative triples. For this reason, KGE training methods make use of *pseudo-negative* triples, i.e., triples that are likely but not guaranteed to be actual negatives. A common and effective method to obtain informative pseudo-negative triples is to corrupt the subject or object (and sometimes also the relation) of known positive triples. For example, the positive triple (Hitchcock, Directed, Psycho) can be corrupted by replacing the object by some other entity obtaining, say, the pseudo-negative triple (Hitchcock, Directed, Avatar). Pseudo-negatives are commonly generated using *negative sampling*, in which the replacement entity is sampled randomly from the set of all entities. Each positive triple is used to generate multiple pseudo-negative samples in this way; the number of such samples is a hyperparameter. Note that an alternative approach is to use all entities from $\mathcal{E}$ for corruption. Although this "1vsAll" method [9] can be very effective on smaller KGs, we do not consider it further because it does not scale to large KGs with millions of entities.

Training. KGE models are trained using gradient-based optimization. The cost function is composed of the mean loss (e.g., binary cross entropy) taken over each positive triple $t \in \mathcal{K}$ and its associated pseudo-negative triples $S^-(t)$—i.e, $|\mathcal{K}|^{-1} \sum_{t \in \mathcal{K}} \mathrm{loss}(t, S^-(t))$—and regularization terms. We denote throughout by N_s^- (N_o^-) the number of pseudo-negative triples obtained by corrupting the subject (object) slot of a positive triple and set $N^- = N_s^- + N_o^-$. Processing is performed in mini-batches of batch size B (e.g., 1024); each batch thus consists of B positive and BN^- pseudo-negative triples. Batches are processed in parallel (on a GPU). This involves computing the batch loss (forward pass), its gradient (backward pass) and updating model parameters using an optimizer such as Adagrad [10] or Adam [16]. For large models, Lerer et al. [19] propose to use "row-wise" optimizers that significantly reduce the storage overhead of Adagrad or Adam by maintaining an optimizer state per embedding (instead of per component of each embedding). This reduces the space consumption of the optimizer from $O(d(|E| + |R|))$ with Adagrad to $O(|E| + |R|)$ with Row-Adagrad.

2.2 Parallel Training

The key insight that is used by existing parallel training techniques is that when processing a batch, *only the parameters relevant for the batch need to be accessed and are updated.* Here and subsequently, *parameters* refers to both model parameters (such as embeddings or other entity-/relation-specific parameters) as well as their associated optimizer states. In particular, the parameters of the entities and relations of the batch's triples (positives and pseudo-negatives) are relevant, whereas the parameters of all other entities and relations are not. This insight is exploited to reduce the overhead of parallel processing to the extent possible.

Algorithm 1 Framework for parallel KGE training

Input: A knowledge graph (set of positive triples)
Output: KGE model parameters (entity and relation embeddings)

```
1:  Master():
2:  Partition knowledge graph
3:  Initialize model parameters (parallel)
4:  for each epoch do
5:      Repartition data                                   ▷ optional
6:      Distribute partitions to workers
7:      Wait for all workers to complete
8:  end for
9:  return model parameters
10:
11: Worker():
12: while true do
13:     Retrieve partition from master                     ▷ latency
14:     Relocate parameters (async)                        ▷ optional
15:     for each batch do   ▷ constructed from current partition
16:         Sample negative examples                       ▷ work
17:         Pull batch parameters                          ▷ latency
18:         Process batch on GPU                           ▷ work, latency
19:         Push batch parameter updates                   ▷ latency
20:     end for
21:     Signal completion to master
22: end while
```

To facilitate our goal of investigating various techniques to scale KGE training, we make use of a simple architecture (Fig. 1) and computational framework (Alg. 1) that generalizes prior multi-GPU and multi-machine KGE training methods and allows to mix and match multiple techniques. We first summarize this general framework and subsequently use it to describe the different parallelization methods proposed in the literature.

General architecture. We assume a setup in which multiple workers are coordinated by a master process as in Fig. 1. The workers may be situated on a single machine (e.g., for *multi-GPU training*) or distributed across multiple machines. Each worker stores a subset of the *parameters* of the KGE model. A worker may communicate with other workers (either in shared memory or via network) to synchronize or exchange parameters. We denote the number of workers by W.

Master. The general computational framework is described in lines 1–9 of Alg. 1. We start by partitioning the knowledge graph $\mathcal{K}$ into multiple partitions $\mathcal{K}_1, \ldots, \mathcal{K}_P$ (line 2) and initialize the model parameters stored at each worker (line 3). The partitioning technique, the number P of partitions as well as the initial location of each parameter depends on the method being used. Training is then performed in multiple epochs until some convergence criterion is met. In each epoch, the master process distributes partitions to workers (after optionally repartitioning the data and potentially in a dynamic fashion or in multiple rounds) and waits until all workers have processed their partitions. Note that the cost of distributing partitions to workers is generally not a bottleneck: it is done at most once per epoch and communication cost can be further reduced when workers have access to the entire knowledge graph.

Parameter management. We make use of a co-located parameter server (PS) to exchange and synchronize parameters between workers. The parameters are partitioned across the individual workers and stored in their main memory; the architecture is similar to the key-value store of [36]. Each worker may *pull* (i.e., read) or *push* (i.e, update) any parameter. The pull operation first retrieves the current value of a parameter from its current location and stores them in GPU memory; likewise, the push operation copies parameter updates from the GPU back to their storage location. If a parameter is stored locally at a worker, this process is fast, otherwise latency and communication overheads occur (none if in that worker's GPU memory, some if in that worker's main memory, more if on some other worker). Workers may request to *relocate* a parameter from its current location at a remote worker to themselves in order to minimize these overheads.[2] Such relocation allows for efficient implementation of the parameter management schemes used in [36] and [19] within a common framework. Unless stated otherwise, parameters are relocated only into the worker's main memory. In some cases (which we will point out explicitly), relocated parameters are subsequently copied to GPU memory; in this case, the pull and push operations are omitted.[3]

Workers. Each worker repeatedly processes a partition as described in lines 11–22 of Alg. 1. After retrieving a partition from the master (line 13), each worker may initiate an (asynchronous) relocation of all parameters relevant for the partition to itself, i.e., the parameters of the entities or relations that actually occur in the partition. The partition is then divided into mini-batches of size B, which are processed in sequence. For each mini-batch, the worker first samples pseudo-negative triples using negative sampling (line 16), then obtains the values of all parameters relevant for the batch (line 17), processes the batch on a GPU (line 18), and (asynchronously) writes back the values of updated parameters (line 19). After the entire partition has been processed, the worker signals completion to the master.

Concurrent parameter accesses. During training, some parameters may be concurrently accessed by multiple workers, leading to conflicts. Due to the sparsity of KGs, the number of conflicts is low for most parameters [34], but they may arise more frequently for relation parameters and highly-connected entities. Some partitioning techniques (Sec. 3) specifically minimize these access conflicts, but they generally cannot be completely avoided. In such cases, access to stale parameter versions may occur. Our PS ensures sequential consistency, however. Moreover, lost updates do not occur because the push operation forwards deltas instead of actual values, and deltas are applied sequentially.

Discussion. In Alg. 1, we marked each step of the worker with whether (i) actual work is performed and (ii) latency due to data movement may arise. Note that only negative sampling (line 16) and batch processing (line 18) of our framework correspond to "training work"; the rest of the framework constitutes coordination and communication overhead. Key to effective parallel training is to reduce

[2] In our implementation, we use the Lapse parameter server [24], which supports transparent parameter relocation (see Sec. 5.2).

[3] Parameters are relocated to a GPU only if (i) there is sufficient memory on the GPU and (ii) it can be ensured that no other worker accesses these parameters. In this case, the updated parameters are copied back from the GPU to the worker's main memory once the entire partition has been processed (line 21 in Alg. 1).

Table 1: Summary of techniques for parallel KGE training. 1 refers to single-machine multi-GPU setup, *D* means multiple machines.

		PBG [19]		DGL-KE [36]		GraphVite [37]		This paper	
		1	D	1	D	1	D	1	D
Partitioning	Random						N/A	✓	✓
	Relation			✓			N/A	✓	✓
	Graph-Cut				✓		N/A	✓	✓
	Stratification	✓	✓			✓	N/A	✓	✓
Neg. samp.	Uniform						N/A	✓	✓
	Shared	✓	✓	✓	✓		N/A	✓	✓
	Local	✓	✓		✓	✓	N/A	✓	✓
	Batch	✓	✓	✓	✓		N/A	✓	✓

this overhead to the extent possible (most notably, in lines 17–19), e.g., by reducing the amount of data that is communicated between workers and between main memory and GPU memory as well as by minimizing the impact of communication latency.

Techniques. Pytorch BigGraph (PBG [19]), GraphVite [37], and DGL-KE [36] provide methods for scalable training of KGE models. The main goals of these methods are (i) low communication overhead, (ii) fast epoch times, and (iii) high embedding quality. Generally, careful partitioning and parameter relocation as well as customized negative sampling strategies are required to obtain effective methods, and the frameworks differ mainly w.r.t. how this is done. We discuss partitioning in Sec. 3 and negative sampling in Sec. 4. An overview of the various techniques is given in Tab. 1.

3 PARTITIONING

We are now ready to describe and compare various approaches to partition the knowledge graph across workers (lines 2 and 5 of Alg. 1). The most basic partitioning scheme, which serves as a baseline for our study, is random partitioning. We also describe relation partitioning, graph-cut partitioning, and finally stratification and discuss their influence on the performance of parallel training. An overview of which framework proposed or used which partitioning method is given in Tab. 1.

3.1 Desiderata for Partitioning Techniques

The choice of the partitioning method influences all three goals of parallel training mentioned above: low communication cost, fast epoch times, and high quality. We discuss each goal in turn; see Tab. 2 for a coarse overview of the influence of each partitioning scheme on these three criteria.

First, the choice of partitioning scheme influences communication cost because it determines which data is processed at which worker and consequently which parameters are accessed at which workers. If the partitioning of the parameters across workers is well-coordinated with the partitioning of the knowledge graph, latency can be largely avoided and overall communication cost reduced. Generally, this is done by allocating parameters to the workers that access them frequently (line 14 of Alg. 1).

Table 2: Influence of partition approaches on balancing of partition sizes, variety and communication cost.

	Load balancing	Variety	Comm. cost
Random	+	+	-
Relation	◦	+	◦
Graph-cut	-	-	+
Stratification (plain)	◦	-	+
Stratification (CARL)	◦	◦	+

Next, assuming that communication latency has been addressed and is not a bottleneck, the wall-clock time taken for an epoch is mainly driven by load balancing: if each worker has a similar amount of work, all workers can operate in parallel. Otherwise, overloaded workers may stall progress and become a bottleneck (e.g., line 7 in Alg. 1). To assess whether a partitioning scheme balances computation, we use partition sizes as a proxy: We generally prefer partitionings in which partition sizes are balanced over partitionings in which they are not. Another factor is the number P of partitions: many small partitions ($P \gg W$) may induce higher cost than few larger partitions (e.g., $P = W$), mainly due to overheads incurred when switching partitions at a worker.

Finally, and perhaps less obviously, the partitioning scheme also influences the quality of the resulting embeddings. On the one hand, this is because some partitioning schemes limit "variety" in that some triples (or entities) cannot co-occur in a partition or a batch. On the other hand, and perhaps more importantly, the partitioning schemes also influence the impact on quality of the parallel negative sampling techniques that will be discussed in Sec. 4.

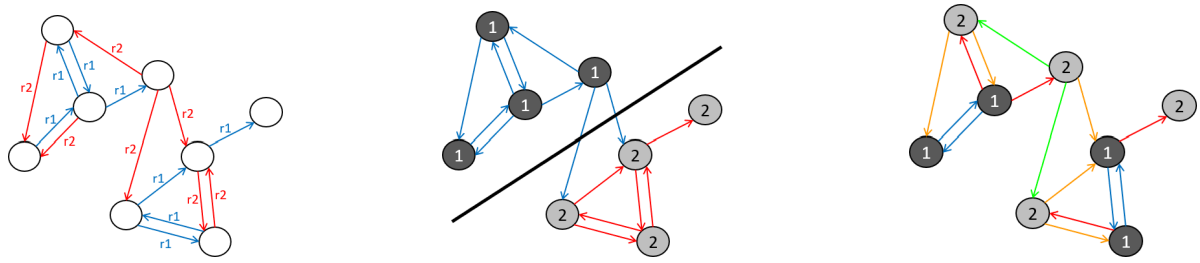
3.2 Random Partitioning

In random partitioning, the knowledge graph $\mathcal{K}$ is randomly divided into P equally-sized partitions $\mathcal{K}_1, \ldots, \mathcal{K}_P$, where typically P equals the the number of workers W. Likewise, all parameters are partitioned randomly across workers. To avoid confusion, we subsequently refer to the former as *triple partitions* and the latter as *entity or relation partitions*.

Discussion. Random partitioning ensures perfect load balancing and high variety of the triples within each triple partition. Its main problem is communication cost. Since the partitioning ignores the structure of the knowledge graph, most parameter accesses (pull and push in Alg. 1) will be non-local and incur communication and latency: each individual parameter access is local with a probability of only $1/W$. The resulting cost of remote parameter accesses may exceed potential parallelization benefits.

3.3 Relation Partitioning

The random partitioning scheme can be improved by making use of the following observation. In most large knowledge graphs, the number of entities (say, multiple millions) exceeds by far the number of relations (say, a few thousands). Since processing of each SPO triple in a batch requires access to its corresponding entity and relation parameters, a substantial fraction of the parameter accesses are to relation parameters. In addition to being responsible for a substantial amount of communication in random partitioning, there



(a) Relation partitioning: Triple partition membership is determined by each triples relation (here: r_1 and r_2).

(b) Graph-cut partitioning: Triple partition membership is determined by each triples subject entity. Entities are partitioned via a graph cut (indicated by vertex color).

(c) Stratification partitioning: Triple partition membership is determined by each triple's subject and object entity. Entities are partitioned randomly (indicated by vertex color).

Figure 2: Illustration of partitioning approaches for $W = 2$ workers. Colors indicate triple partition membership.

are generally more conflicts between these accesses in that multiple workers may access given relation parameters simultaneously (just because there are few relations).

Both problems can be avoided by relation partitioning [36]. Here the set $\mathcal{R}$ of relations is partitioned into P subsets $\mathcal{R}_1, \ldots, \mathcal{R}_P$, where as before $P = W$. The knowledge graph is then partitioned accordingly: triple partition $\mathcal{K}_p$ consists of the triples with a relation in $\mathcal{R}_p$, i.e., $\mathcal{K}_p = \mathcal{K} \cap \mathcal{E} \times \mathcal{R}_p \times \mathcal{E}$. An example of relation partitioning with $P = 2$, $\mathcal{R} = \{ r_1, r_2 \}$ and $\mathcal{R}_p = \{ r_p \}$ is shown in Fig. 2a.

Discussion. In general, relation partitioning aims to provide balanced triple partition sizes. Finding such a partition is a *multiway number partitioning* problem [12] (where the numbers correspond to relation sizes) and greedy approximation methods[4] empirically tend to work well as long as there are no overly large relations. However, such relations do arise in practice. In such cases, triple partitions either become unbalanced (impacting load balancing) or the relation partitioning is "softened" by splitting the triples of large relations across workers (impacting communication cost).

Since in (hard) relation partitioning each relation occurs only at a single worker, relation parameters do not need to be communicated across workers and can, in fact, be stored in each worker's GPU memory during the relocation step (line 14 in Alg. 1); no subsequent costs arise for pulling or pushing these parameters. If a soft relation partitioning is used, this benefit vanishes for frequent relations. There is also no benefit for accessing entity parameters so that substantial communication costs may still arise. These costs may be slightly reduced by relocating each entity to the worker that accesses it most often; this is especially beneficial if certain entities only arise in a single relation.

Finally, relation partitioning reduces variety when compared to random partitioning. Whether two triples may co-occur in a triple partition (and consequently a batch) depends on the relations of these triples.

3.4 Graph-Cut Partitioning

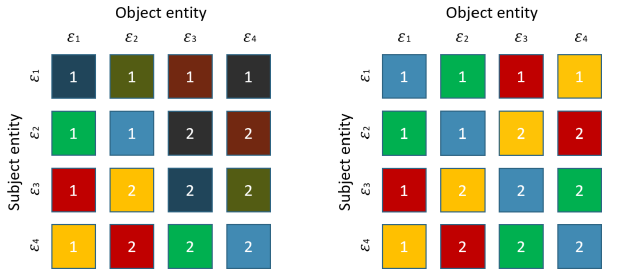
An alternative to partitioning the relations is to partition the set $\mathcal{E}$ of entities into P entity partitions $\mathcal{E}_1, \ldots, \mathcal{E}_P$, which are distributed across the workers. The knowledge graph $\mathcal{K}$ is partitioned accordingly in that each triple is assigned to a triple partition that corresponds to the entity partition of its subject entity: We have $\mathcal{K}_p = \mathcal{K} \cap \mathcal{E}_p \times \mathcal{R} \times \mathcal{E}$ for $1 \leq p \leq P$. Triple partition $\mathcal{K}_p$ is processed by the worker at which $\mathcal{E}_p$ is located. The idea of graph-cut partitioning [36] is to partition the entities in such a way that (i) partition sizes $|\mathcal{K}_p|$ are balanced and (ii) most triples are local to their partition. A triple $(s, r, o) \in \mathcal{K}_p$ is *local* if $s, o \in \mathcal{E}_p$: in this case, no communication is required to access the corresponding entity parameters during training. Zheng et al. [36] use METIS [14] to create such a balanced graph cut; see Fig. 2b for an example.

Discussion. Entity partitions influence the communication cost between workers and triple partitions the computational cost at each worker. There is generally a trade-off between balancing the sizes of entity and triple partitions and the fraction of local triples. The communication overhead for relation parameters is not reduced. Most knowledge graphs admit a good balanced cut in terms of entity partitions, but these cuts result in unevenly-sized triple partitions (c.f. Tab. 11). As the effectiveness in terms of training times reduction is heavily dependent on this balancing, graph-cut partitioning is beneficial for datasets for which there is a good graph cut with balanced entity partitions *and* balanced triple partitions. In general, many entities occur in multiple partitions so that data movement across workers as well as between main memory and GPU memory cannot be avoided. Finally, entity partitions tend to contain highly-interconnected entities (*communities*), which substantially reduces variety across partitions. This is especially problematic in conjunction with some of the sampling techniques discussed in Sec. 4.

3.5 Stratification Partitioning

Stratification partitioning was originally introduced in the context of matrix factorization [11] and subsequently adapted to training KGEs [19, 37]. The idea behind stratification is to create triple partitions (termed *buckets* in [19]) that access *pairwise disjoint sets of entities*. A set of S such triple partitions is called a *stratum* [11]. The key insight behind stratification is that we can process the triple partitions within a stratum in parallel across S workers without any need to synchronize entity parameters in between as they are

[4] We use the popular heuristic of sorting the relations in a descending order first, before applying greedy-number-partitioning.



(a) Plain stratification

(b) Stratification with combined mirror triple partitions

Figure 3: Possible schedule for $W = 2$ workers with stratification using $M = 4$ entity partitions. Triple partitions marked in the same color form a stratum and are processed in parallel; the number inside the triple partition indicates the worker number.

pairwise disjoint.[5] Strata can be obtained in the following way: first partition the set $\mathcal{E}$ of entities randomly into M entity partitions $\mathcal{E}_1$, …, $\mathcal{E}_M$ (where generally $M = \Theta(W)$) and subsequently partition the triples based on their subject *and object* entities. For example, a triple with a subject of entity partition p_1 and an object of entity partition p_2 is assigned to the triple partition $\mathcal{K}_{p_1p_2}$. We obtain $P = M^2$ triple partitions of form $\mathcal{K}_{p_sp_o} = \mathcal{K} \cap (\mathcal{E}_{p_s} \times \mathcal{R} \times \mathcal{E}_{p_o})$ for $1 \leq p_s, p_o \leq M$. Such a partitioning is visualized in Fig. 2c for $M = 2$. The resulting triple partitions are then scheduled across workers such that at any point of time, the set of currently processed triple partitions forms a stratum (see below). An example schedule for $M = 4$ and $W = 2$ is shown in Fig. 3a, where each color indicates a stratum. Before processing a triple partition, workers relocate their entity parameters to themselves (line 14 of Alg. 1) so that accesses to these entities are local.[6]

Partitioning and scheduling. The number M of entity partitions must be carefully chosen. Generally, any two triple partitions $\mathcal{K}_{p_sp_o}$ and $\mathcal{K}_{p'_sp'_o}$ within a stratum must be guaranteed to contain disjoint entities, i.e., we require $\{ p_s, p_o \} \cap \{ p'_s, p'_o \} = \emptyset$. A natural choice is to set $M = W$; however, such a choice does not ensure that all strata contain W triple partitions and thus saturate all workers. In Fig. 2c, where $M = 2$, it is only possible to build the non-trivial stratum $\{ \mathcal{K}_{11}, \mathcal{K}_{22} \}$; all other pairs of triple partitions do not form a stratum and thus cannot be processed in parallel. During training, this means that $\mathcal{K}_{12}$ and $\mathcal{K}_{21}$ are processed sequentially and one worker remains idle. We call such a schedule *blocking*.

To increase worker saturation, Lerer et al. [19] proposed to set $M > W$; in particular $M = 2W$ (less does not suffice). The resulting $P = (2W)^2$ partitions can be grouped into $4W$ strata, each of size exactly W. An example of such a non-blocking schedule is shown in Fig. 3a for $W = 2$ and consequently $M = 4$. Scheduling must be done carefully. A best-effort scheduler such as the one used in PBG [19] does not always produce a non-blocking schedule.

[5]Note that synchronization may be required to process negative samples. We come back to this point in Sec. 4.

[6]Relocation is to main memory in general to allow across-partition negative sampling. When local sampling is used (see Sec. 4.4), relocation to GPU is possible.

Processing order. Processing a triple partition only leads to updates in its corresponding entity partitions. For this reason, the embedding spaces of each entity partition may not be aligned in the first epochs when trained with a single worker. This problem can be mitigated by a carefully-chosen processing order [19]. In our setting, this issue vanishes as concurrent updates of relation parameters by multiple workers automatically lead to aligned embedding spaces. In fact, a random ordering of strata performed best in our experiments.

Discussion. Since entities are partitioned randomly, all KG partitions have the same size in expectation ($= N/P = N/M^2$). Partition sizes may still vary somewhat; the variance depends on the data distribution. A key problem with stratification is that the number of partitions scales quadratically with the number of workers (since $M = \Theta(W)$). This is problematic especially when the KG is small relative to the number of workers. The main advantage of stratification is that (i) overall communication cost is reduced and (ii) no access latency arises during batch processing for the embeddings of the partition's entities. Assuming a non-blocking schedule and ignoring negative sampling, each entity parameter is relocated at most P/W times (e.g., $4W$ times for the choice of $M = 2W$). In practice, roughly half of the relocations can be avoided by scheduling partitions to workers that already have its subject or object entities localized. Finally, stratification does reduce variety between partitions, but it does so in a less systematic way than graph-cut partitioning.

3.6 Improved Stratification Partitioning (CAR)

Stratification partitioning is a promising technique but, as discussed above, may suffer from (i) small partitions, (ii) many relocations, and (iii) reduced variety. We propose three simple techniques that mitigate these problems: (i) combining mirror partitions (C), (ii) relocating only active entities (A), and (iii) repartitioning between epochs (R). We refer to the resulting variant of stratification as *CAR stratification* and study its impact in the experimental section.

Combining mirror partitions (C) is a technique that halves the number of partitions required to create a non-blocking schedule. It consequently reduces communication cost and leads to more variety within partitions. The approach works as follows: First compute partitions as in plain stratification and subsequently merge each pair of *mirror partitions* into a single partition. The mirror partition of $\mathcal{K}_{p_sp_o}$ is $\mathcal{K}_{p_op_s}$ when $p_s \neq p_o$. When $p_s = p_o = p$, the mirror partition is $\mathcal{K}_{(p+1)(p+1)}$ for odd p and $\mathcal{K}_{(p-1)(p-1)}$ for even p. An example is shown in Fig. 3b; here, mirror partitions are indicated by the same color *and* the same worker number. One can show that when $M = 2W$, mirror partitions also admit a non-blocking schedule such as the one shown in Fig. 3b. Roughly speaking, when some worker w processes $\mathcal{K}_{p_sp_o}$, plain stratification ensures that no other worker processes mirror partition $\mathcal{K}_{p_op_s}$ concurrently. We can thus merge each partition with its mirror.

Active entities (A). In plain stratification, the entity parameters of the combined entity partitions $\mathcal{E}_{p_s} \cup \mathcal{E}_{p_o}$ are relocated to the worker before processing triple partition $\mathcal{K}_{p_sp_o}$. However, only a fraction of these entities may actually occur either as subject or object of a triple in $\mathcal{K}_{p_sp_o}$. We denote those entities as *active* and propose a simple variant that relocates only the parameters

Table 3: Number of unique entities per batch (upper bound) by sampling technique. Corresponding entity parameters need to be communicated and copied to GPU memory.

Technique	Unique entities per batch
Uniform sampling	$2B + BN^-$
Shared sampling	$2B + N^-$
Batch sampling	$2B$

of active entities instead. This approach reduces communication cost significantly when partitions are sparse (as is often the case in large KGs). Furthermore, this affects the pool for the local sampling technique discussed in Sec. 4.4.

Repartitioning (R). The entity partitioning and hence the triple partitioning in plain stratification is static, i.e., it does not change between epochs. To increase variety, one may repartition both entities (randomly) and triples between epochs. This repartitioning can be performed in background and only incurs minor communication costs when each worker has access to the complete knowledge graph. Note that repartitioning is also possible for random partitioning, but generally not for graph-cut and relation partitioning.

Discussion. All of the above modifications are simple. Our experimental study suggests that they are effective or even instrumental for the efficient use of stratification.

4 NEGATIVE SAMPLING

The partitioning techniques discussed in the previous section aim to make access to (the parameters of) positive triples from $\mathcal{K}$ more efficient. We now turn attention to pseudo-negative triples. Recall that these triples are constructed by negative sampling, i.e., for each positive triple in a batch, we create N^- pseudo-negative triples by corrupting either its subject or its object with a randomly sampled entity. All parameters of the so-sampled entities needed to be pulled to the worker as well, which induces communication overhead. If N^- is large, this overhead is substantial and may by far outweigh the communication overhead required to process positive triples.

The communication overhead is mainly driven by (i) the number of unique entities sampled for a batch[7] and (ii) the location of the corresponding entity parameters. Both factors can be influenced by biasing the sampling distribution used for corruption. The key techniques are *shared sampling* [19, 36], *batch sampling* [19, 36], and *local sampling* [19, 36, 37]. An overview of the techniques used in various frameworks is given in Tab. 1. In the following, we describe the techniques and analyze them in terms of the number of unique entities being used (Tab. 3).

4.1 Uniform Sampling

Uniform sampling is the basic technique used in sequential methods for training KGE embeddings; we use it as a baseline. Each negative sample is obtained by sampling the corrupted entity uniformly and independently from the set $\mathcal{E}$ of all entities. This leads to a large number of unique entities per batch: with a batch size of B, up to $2B$ unique entities occur in the positive triples (subject and object entity) and up to BN^- additional unique entities in the pseudo-negative triples (corrupted entity). Consequently, even for small choices of N^-, most entities relevant for a batch arise from negative sampling. In expectation, only a fraction of $1/W$ of these entities is local to the worker so that communication costs are high.

4.2 Shared Sampling

The idea of *shared sampling* [19, 36] is to use the same corrupted entities for triples in a batch. In more detail, N^- triples are sampled uniformly and independently from the set of all entities and then shared across positive triples.[8] This substantially reduces the number of unique corrupted entities from BN^- to N^- and consequently leads to a significant reduction of communication costs and epoch time. For many KGE models, computational cost can also be reduced because it facilitates the scoring of all negative triples in a batch on the GPU via matrix multiplication. This approach is used in PBG and DGL-KE. A key drawback of shared sampling is that variety is reduced, which can have a negative impact on embedding quality. In fact, our experimental study (Sec. 5.5) suggests that (i) this issue can be mitigated by increasing the number of negatives drastically, and (ii) even with the higher number of negative samples, shared sampling is far more efficient than uniform sampling.

4.3 Batch Sampling

To avoid transferring additional parameters for the entities of negative triples (both to worker and to GPU memory), *batch sampling* (termed degree-based sampling in [36]) does not sample from the set $\mathcal{E}$ of all entities but from the set of entities that already occur in a positive triple of the batch. The number of unique entities per batch reduces to $2B$ and in particular, does not depend on the number N^- of negative samples anymore. The size of the sampling pool for the negative samples is heavily reduced and their distribution is data-dependent (more frequent entities are more likely to occur in a batch and thus to occur as a negative). Zheng et al. [36] combine batch sampling with uniform sampling to increase variance. Our experimental study suggests that the benefits of batch sampling depend on both dataset and partitioning technique. On most datasets, batch sampling heavily deteriorated embedding quality, but it was highly beneficial in terms of both embedding quality and communication overhead on the Freebase dataset. Here, uniform negatives may lead to "easy" negative samples [17], whereas batch samples may contain harder negatives.

4.4 Local Sampling

Another approach to bias the sampling distribution is *local sampling*, where corrupted entities are drawn from the set of entities whose parameters are located at the worker (in the parameter server). This ensures that pull operations for negatives do not incur communication with other workers. Local sampling is used by DGL-KE, PBG and GraphVite and can be combined with shared sampling.

Since local sampling biases the sampling distribution, it has an impact on embedding quality. This impact is mainly determined by the allocation of entity parameters across workers, which in

[7] If an entity occurs multiple times in a batch, its parameters need to be pulled/pushed only once.

[8] Shared sampling can also be applied to a subset of the batch as done in [36]. However, we did not see any benefits of this approach.

turn is driven by the partitioning scheme being used. For example, if graph-cut partitioning is used, the allocation is static: for all triples of partition $\mathcal{K}_p$, negative samples are taken from $\mathcal{E}_p$. Likewise, when processing partition $\mathcal{K}_{p_s p_o}$ in stratification partitioning, negative samples are taken from the set $\mathcal{E}_{p_s} \cup \mathcal{E}_{p_o}$ of entities (or the subset of active entities in CAR stratification). Consequently, local sampling may degrade quality in all these cases; we investigate this degradation empirically in our experimental study.

One counter-measure to avoid degradation effects is to repartition the data between epochs as in CAR stratification. Such an approach is not possible for graph-cut partitioning (since partitions are deterministic), however, and it is very restricted in relation partitioning. An alternative approach that ensures a dynamic local-sampling pool is to reshuffle the entity partitions (but not the triple partitions) randomly between epochs. This approach—denoted (r)—can be used with random and relation partitioning since these methods use a random entity allocation.

Note that when local sampling is combined with stratification partitioning, it is guaranteed that (i) all entity parameter accesses (positive and negative) of a worker are local and (ii) no other worker accesses these parameters concurrently. For this reason, the parameters can be stored directly in GPU memory and the pull/push operations (lines 17 and 19 of Alg. 1) can be omitted.

5 EXPERIMENTAL STUDY

We conducted an experimental study in which we investigated the various partitioning and negative sampling techniques in a common framework. Our main goal was to provide insight into (i) the efficiency of the techniques in terms of runtime as well as, (ii) their effectiveness in terms of embedding quality, and ultimately (iii) whether and to what extent parallelization is beneficial to train KGE models. We consider both single-machine multi-GPU scenarios as well as multi-machine multi-GPU training.

5.1 Key Findings

Before describing the experimental setup and results of our study in detail, we briefly summarize our key findings.

General findings.

(1) The right choice of partitioning and negative sampling technique was crucial. With such a choice, both multi-GPU and multi-machine parallelization were effective and efficient, especially for larger knowledge graphs (Sec. 5.3).
(2) The right choice is dataset-dependent.
(3) The sampled MRR metric used in prior work to evaluate KGE model quality on large KGEs is misleading and should not be used (Sec 5.4).
(4) Row-based optimizers such as Row-Adagrad reduced communication overhead while preserving a high embedding quality (online repository).

Partitioning techniques (Sec. 5.3).

(5) CARL stratification was most efficient and effective on most datasets. It also outperformed plain stratification.
(6) Random partitioning consistently led to high-quality embeddings. On Freebase, it was the overall method of choice.
(7) Graph-cut partitioning was least effective due to unbalanced triple partition sizes (Sec. 5.7).

Table 4: Dataset statistics

Dataset	Entities	Relations	Triples (total)
FB15k	14,951	1,345	592,213
Yago3-10	123,182	37	1,089,040
Wikidata5m	4,594,485	822	20,624,575
Freebase	86,054,151	14,824	338,586,276

(8) Relation partitioning led to time improvements but had a negative influence on embedding quality on some datasets.

Sampling techniques (Sec. 5.5, 5.6).

(9) Shared sampling was instrumental for efficiency and should always be used.
(10) Local sampling with a dynamic sampling pool improved efficiency and should be used on large graphs.
(11) A static local sampling pool often had a negative influence on embedding quality.
(12) Batch sampling significantly deteriorated resulting quality on most graphs. On Freebase, however, it significantly improved quality.

5.2 Experimental Setup

We provide code, search configurations, and resulting hyperparameters at https://github.com/uma-pi1/dist-kge.

Datasets. We used knowledge graphs of varying sizes; see Tab. 4. *FB15k* [5], a small subset to compare to prior work on KGE, *Yago3-10* [9], *Wikidata5m* [32] and the *Freebase* KG as used in [19, 36].[9] Note that a KGE model for Freebase does not fit on a single GPU (e.g., a ComplEx model with $d = 400$ takes 128GB plus optimizer state). For all datasets except Freebase, we use the validation and test sets that accompany the datasets. For Freebase, we sample 10,000 triples from the original validation and test sets (which contain about 17M triples each) to keep evaluation costs feasible. These sets are still sufficiently large to estimate model quality accurately.

Hardware. We ran all experiments on the same hardware. We used two machines with 40 CPUs (Intel(R) Xeon(R) CPU E5-2640 v4, 2.40GHz) and 4 GPUs (GeForce RTX 2080 Ti). The bandwidth between the machines was 1 GB/s. For each experiment, we write *G@C* to indicate that *G* GPUs were used on each of *C* machines. For example, a 1@2 experiment uses two machines with one GPU each. On each GPU, we run two workers so that the 1@2 setup trains with four workers.

Implementation. We implemented parallel training on top of the PyTorch-based LibKGE library [6, 26], which provides a number of KGE models, training methods, and evaluation techniques. KGE models can be grouped into semantic matching models and translational distance models [31]. We consider ComplEx [30] and RotatE [28], which are are among the currently best-performing models [1, 18, 26, 28] for the former and latter group, respectively. For single-machine multi-GPU training, we used a shared-memory PyTorch tensor to store parameters in main memory. For multi-machine training, we used the parameter server Lapse [24].

[9] dump: https://developers.google.com/freebase/, preprocessed: http://web.informatik.uni-mannheim.de/pi1/kge-datasets/freebase.tar.gz.

Metrics. We compared parallel to sequential training methods. We used two different sequential settings: *sequential (GPU memory)* and *sequential (main memory)*, which differ in the location of the KGE model parameters. For the main memory version, parameters are copied to and retrieved from the GPU memory when processing each batch; this allows to handle very large KGE models. We evaluated efficiency in terms of runtime and communication cost (GBs transferred) per epoch. To evaluate quality, we follow standard practice and compute the filtered mean reciprocal rank (MRR) for the link prediction task. In particular, for each triple (s, p, o) in the test set, we rank all triples of form $(s, p, ?)$ (and subsequently $(?, p, o)$) by their predicted scores, filter out all triples that occur in the training, validation or test data, and finally determine the reciprocal rank of the test triple. The so-obtained filtered reciprocal ranks are averaged; higher values are better. In the case of ties, we use the mean rank to avoid misleading results [25, 26, 29]. Finally, to evaluate effectiveness, we combined efficiency and result quality, which may be affected by parallelization. In particular, for each setting, we report the time required to reach a MRR that exceeds 95% of the best MRR achieved in the sequential setting.

Hyperparameter optimization. A solid choice of hyperparameters is key to obtain high-quality KGE models. In general, we followed [26] and performed the model selection using a quasi-random search with 30 trials in a sequential setup. Each trial was run for 20 epochs using Adagrad; the best resulting model was then run for up to 300 epochs (Wikidata5m) or 400 epochs (FB15k, Yago3-10). We used 128-dimensional embeddings for all datasets. For our parallel experiments, we generally used the best-performing sequential hyperparameters. An exception is Freebase (which would be infeasible to run sequentially); here we used the best performing setting on FB15k with Row-Adagrad and trained up to 10 epochs.

Techniques. As shared negative sampling was instrumental for efficient parallel training, we consistently use it unless noted otherwise. We explored local (L) and batch (B) sampling, repartitioning of triple partitions and relocation of corresponding entity partitions (R), shuffling of entity partitions (r), combined mirror triple partitions (C) and active entities (A). We use batch sampling combined with uniform sampling (50/50) unless stated otherwise. For stratification we set $M = 2W$ for all datasets but Freebase. For Freebase, we use $M = 32$ throughout so that the entity partitions fit in GPU memory in all settings.

5.3 Partitioning Techniques (Tab. 5–6)

Tab. 5a (ComplEx) and Tab. 5b (RotatE) summarize the most effective parallelization techniques for various numbers of GPUs and machines. For all settings but 4@2, we ran all partitioning techniques and report the one with lowest time to 0.95 MRR. For 4@2, we used the best performing partitioning technique found for 1@2 (2@2 for Freebase). Parallelization was effective on all datasets, but the speedups on Yago3-10 were small; here parallelization overheads and quality degradation dominated. For the larger Wikidata5m and Freebase datasets, speedups were significant: up to 8x without any quality degradation. The two models behaved similar but overall RotaE train times were higher compared to ComplEx. Furthermore, graph-cut partitioning had a smaller negative influence on quality with RotatE than with ComplEx.

Table 5: Partitioning techniques (best-performing variant in terms of time to 0.95 MRR). Calculating time to 0.95 MRR was too expensive for Freebase.

(a) ComplEx model (n.r. means not reached).

	Set up	Partitioning technique	Epoch time	Time to 0.95 MRR	MRR	Hits @10
FB15k	1@1	Seq. (GPU memory)	5.9s	3.9min	**0.779**	0.862
	1@1	Seq. (main memory)	7.7s	5.1min	**0.779**	0.862
	2@1	Random (R)	2.6s	2.0min	0.775	0.859
	1@2	Random (R)	2.9s	2.2min	0.775	0.859
	4@2	Random (R)	**1.3s**	**1.3min**	0.766	0.858
Yago3-10	1@1	Seq. (GPU memory)	24.3s	38.5min	**0.542**	0.675
	1@1	Seq. (main memory)	42.6s	67.5min	**0.542**	0.675
	2@1	Relation	19.0s	**33.2min**	0.538	0.669
	1@2	Random (RL)	19.5s	35.8min	0.547	0.679
	4@2	Random (RL)	**5.6s**	n.r.	0.503	0.653
Wikidata5m	1@1	Seq. (GPU memory)	438.4s	219.0min	0.297	0.385
	1@1	Seq. (main memory)	774.3s	387.0min	0.297	0.385
	2@1	Stratification (CARL)	232.8s	77.6min	**0.308**	0.398
	1@2	Stratification (CARL)	228.0s	**76.0min**	**0.308**	0.398
	4@2	Stratification (CARL)	**97.1s**	105.2min	0.294	0.377
Freebase	1@1	Seq. (main memory)	3929.0s	-	0.364	0.487
	4@1	Random (RLB)	704.6s	-	**0.426**	0.529
	2@2	Random (RLB)	966.7s	-	**0.426**	0.529
	4@2	Random (RLB)	**591.6s**	-	0.421	0.523

(b) RotatE model.

	Set up	Partitioning technique	Epoch time	Time to 0.95 MRR	MRR	Hits @10
FB15k	1@1	Seq. (GPU memory)	9.5s	11.9min	0.705	0.834
	1@1	Seq. (main memory)	11.4s	14.3min	0.705	0.834
	2@1	Stratification (CARL)	**4.6s**	**5.8min**	**0.725**	**0.835**
	1@2	Stratification (CARL)	5.9s	7.4min	**0.725**	**0.835**
Yago3-10	1@1	Seq. (GPU memory)	74.1s	259.3min	**0.451**	0.637
	1@1	Seq. (main memory)	88.0s	307.8min	**0.451**	0.637
	2@1	Stratification (CARL)	**40.8s**	**166.6min**	0.438	0.607
	1@2	Stratification (CARL)	43.3s	175.8min	0.438	0.607
Wikidata5m	1@1	Seq. (GPU memory)	798.4s	199.6min	0.258	0.348
	1@1	Seq. (main memory)	985.7s	246.4min	0.258	0.348
	2@1	Stratification (ARL)	**466.7s**	**77.8min**	**0.264**	0.344
	1@2	Stratification (ARL)	477.7s	79.6min	**0.264**	0.344
Freebase	1@1	Seq. (main memory)	6495.7s	-	**0.566**	**0.627**
	4@1	Random (RLB)	1290.8s	-	0.567	0.630
	2@2	Random (RLB)	1541.4s	-	0.567	0.630
	4@2	Random (RLB)	**938.3s**	-	0.562	0.621

Overall, the best performing techniques were dataset-dependent. In most cases, a variant of stratification or random partitioning outperformed other techniques or was close to the best result. We analyzed Freebase separately (Tab. 6). Here the random partitioning baseline in combination with local (L) and batch (B) sampling was

Table 6: Comparison of partitioning techniques (ComplEx, Freebase). Note that the sampled MRR (sMRR) metrics used in some prior studies are misleading. Lower part shows results with 10× # negatives.

	Set-up	Partition technique	Epoch time	Data sent per epoch	sMRR /1,000	MRR	Hits @10
	1@1	Seq. (mm)	3929s	-	0.811	0.364	0.487
	1@1	Seq. (B) (mm)	3925s	-	0.815	0.426	0.528
	2@2	Random (RLB)	966s	232.8GB	**0.816**	**0.426**	**0.529**
	2@2	Relation (rLB)	823s	205.9GB	0.801	0.397	0.507
	2@2	Strat. (CARLB)	**803s**	123.2GB	0.793	0.325	0.424
	2@2	Graph-cut (LB)	1170s	**42.5GB**	0.789	0.407	0.512
	4@2	Random (RLB)	591s	251.9GB	0.819	0.421	0.523
10× neg.	2@2	Random (RLB)	1481s	232.8GB	**0.841**	**0.478**	**0.588**
10× neg.	2@2	Relation (rLB)	1341s	205.9GB	0.808	0.454	0.569
10× neg.	2@2	Strat. (CARLB)	**1127s**	122.1GB	0.798	0.451	0.556
10× neg.	2@2	Graph-cut (LB)	1810s	**44.4GB**	0.786	0.467	0.567

Table 7: Avg. processing and wait time in seconds per worker and epoch with std. dev. (ComplEx, 1@2, Wikidata5m).

	Rand. (R)	Rand. (RL)	Rel.	GC (L)	Strat. (CARL)
Total time	338±6	275±0	315±1	232±54	227±0
Proc. time	228±4	218±0	219±4	193±44	217±2
Wait time	110±1	63±0	96±3	39.3±10	11±2

Table 8: Performance comparison to original implementations (ComplEx, Freebase).

Setup	Frame-work	Partition technique	Dim.	Epoch Time	MRR	Hits@10
2@2	PBG	Strat. (LB)	100	2856s	0.157	0.250
4@1	DGL-KE	Relation (B)	400	~600s[10]	0.614	0.665
2@2	Ours	Strat. (LB)	100	1983s	0.291	0.384
4@1	Ours	Relation (B)	400	860s	0.612	0.662

most effective; stratification led to the overall worst results. One reason for the weaker performance of stratification was the reduced effectiveness of batch sampling with this partitioning technique, see Sec. 5.6. Increasing the number of negative samples by 10× mildened this effect and led to an overall increased embedding quality (see also Tab. 6). Graph-cut partitioning further reduced communication cost by 5-6×, but did not provide faster epoch times due to unbalanced triple partition sizes.

Tab. 7 shows processing and wait time per worker and epoch for each partitioning technique. Stratification (CARL) and graph-cut heavily reduced wait times and therefore led to the shortest overall processing times. However, graph-cut showed a high variance between workers; see Sec. 5.7 for additional analysis on graph-cut.

5.4 Comparison to Original Work (Tab. 8)

The above results were all obtained using an independent implementation of parallel training. It is challenging to put these results in perspective with the results originally reported by the DGL-KE and PBG frameworks. The reason is that these prior studies (i) did not report MRR but *sampled MRR* (sMRR) and (ii) used different variants of sMRR. In general, sMRR/X is obtained by ranking test triples against a set of X random triples instead of all triples. Note that sMRR/X decreases with increasing X until it reaches the MRR. The use of sMRR is computationally cheaper, but we found that it produced misleading results. As can be seen in Tab. 6, sMRR highly overestimated MRR and, perhaps more importantly, different models suggested similar quality in terms of sMRR even when their MRR differed substantially. One reason for this misleading approximation is that "hard negative" entities (which appear before the target entity in the full ranking) are unlikely to be sampled. Addressing this issue [19] and [36] proposed a batch sampling approach. We did not consider these methods because they did not solve the problematic approximation in preliminary experiments and resulted in an evaluation metric that is batch size dependent. For this reason we use MRR, even though the computation is expensive.

To get some intuition into the framework performance, we trained a ComplEx model with DGL-KE and PBG with their provided hyperparameter settings. We then trained a corresponding model in our framework (using the same dimensionality). The results in Tab. 8 suggest that our implementation is competitive.

5.5 Sampling Techniques (Tab. 9)

Shared Sampling. To investigate the effect of shared sampling, we ran a hyperparameter search for the model ComplEx with uniform sampling (1@1) to obtain a suitable choice of the numbers N_s^- and N_o^- of negative samples. We fixed this configuration but (i) switched to shared sampling to measure impact on quality and (ii) used a 1@2 setup with random partitioning to measure impact on communication costs. Our results are summarized in Tab. 9a.

First observe that shared sampling is very efficient: its use led to a 40x reduction in network footprint and a 7x reduction in epoch time. However, the model quality suffered in that the resulting MRR decreased. This is a consequence of the reduced variety of negative samples introduced by shared sampling. We found that the quality degradation could be countered by using a large number of negative samples. We reran the hyperparameter search with shared sampling and a 10x larger limit in the upper bound on the number of samples; the results are also reported in Tab. 9a. As can be seen, the increased number of samples led to models of similar or better quality than obtained with uniform sampling but is substantially more efficient in terms of epoch time and network footprint. The lower part of Tab. 6 shows that these quality improvements also hold for the largest dataset Freebase.

Local Sampling. To study the impact of local sampling, we first investigated the performance obtained by the various partitioning techniques with shared sampling with and without local sampling; see Tab. 9b. We found that none of the partitioning techniques provided substantial benefits w.r.t. random partitioning without the

[10]This epoch time is approximated since DGL-KE does not have a concept of epochs but only reports in steps.

Table 9: Sampling techniques (ComplEx, 1@2)

(a) Shared sampling reduced communication overhead and epoch time (Random (R)).

	Setting	N_s^-	N_o^-	Epoch time	Data sent per epoch	MRR	Hits @10
Yago3-10	Uniform	892	894	114.9s	66.5GB	0.518	0.659
	Shared	892	894	9.3s	1.9GB	0.508	0.649
	Shared	8919	8942	21.1s	7.2GB	0.538	0.672
Wiki-data5m	Uniform	66	236	5112.3s	3262.0GB	0.218	0.325
	Shared	66	236	153.6s	24.6GB	0.204	0.310
	Shared	2176	7851	347.2s	114.2GB	0.296	0.395

(b) Local sampling reduced epoch time. The static sampling pool of relation/graph-cut can have a negative influence on quality.

	Sample from:	All Entities			Local Entities		
		Time	Comm.	MRR	Time	Comm.	MRR
FB15k	Random (R)	2.9s	0.7GB	**0.775**	2.9s	0.5GB	**0.775**
	Relation	**2.8s**	0.6GB	0.771	**2.8s**	0.4GB	0.729
	Strat. (CAR)	3.5s	0.3GB	0.771	2.9s	**0.1GB**	0.765
	Graph-cut	3.3s	0.3GB	0.766	3.3s	**0.1GB**	0.506
Yago3-10	Random (R)	21.1s	7.2GB	0.538	19.5s	1.3GB	**0.547**
	Relation	23.6s	7.1GB	0.538	22.6s	1.3GB	0.497
	Strat. (R)	29.8s	12.1GB	0.534	13.8s	1.2GB	0.531
	Graph-cut	28.3s	11.2GB	0.535	18.2s	**0.2GB**	0.211
Wikidata5m	Random (R)	347.2s	114.2GB	0.296	290.6s	27.2GB	0.297
	Relation	320.5s	111.4GB	**0.296**	275.6s	24.1GB	0.290
	Strat. (CAR)	393.1s	143.7GB	0.291	**228.0s**	15.0GB	**0.308**
	Graph-cut	417.4s	137.7GB	0.294	317.2s	**6.1GB**	0.192

(c) Batch sampling is dataset-dependent (Random (R)).

Dataset	Batch Neg.	Time	Comm.	MRR	Hits@10
FB15k	0%	2.9s	0.7GB	0.775	0.859
	50%	2.9s	0.6GB	0.766	0.852
	100%	2.7s	0.6GB	0.745	0.845
Yago3-10	0%	21.1s	7.2GB	0.538	0.672
	50%	14.4s	5.6GB	0.422	0.589
	100%	9.1s	1.0GB	0.375	0.555
Wiki-data5m	0%	347.2s	114.2GB	0.296	0.395
	50%	230.8s	72.3GB	0.224	0.321
	100%	142.6s	19.4GB	0.186	0.274
Free-base	0%	983.8s	295.0GB	0.364	0.483
	50%	944.7s	263.4GB	0.420	0.525
	100%	948.2s	232.0GB	0.405	0.512

use of local sampling. When local sampling was used, epoch times and communication cost decreased for all partitioning techniques on all datasets; stratification benefited the most.

While local sampling may be efficient, it is not always effective as it may decrease quality. To gain insight into why this is the case, Fig. 4 shows the progress of validation MRR during training for various techniques. Note that there was a substantial drop in

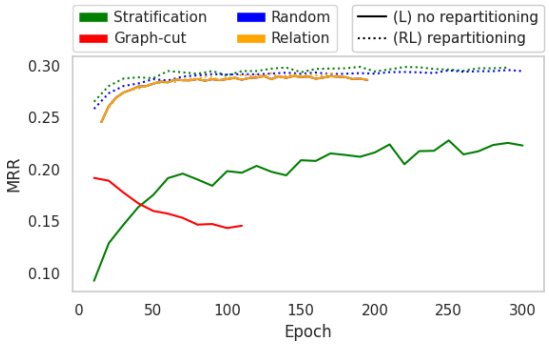


Figure 4: Local sampling without repartitioning led to substantial drops in embedding quality for stratification and graph-cut. (ComplEx, Wikidata5m)

validation MRR for stratification and graph-cut partitioning once local sampling was used. Stratification with repartitioning (as also done in CARL stratification) remained unaffected. Since repartitioning changes the sampling pool from epoch to epoch, the quality degradation was avoided.

We conclude that local sampling was generally efficient, but required data repartitioning between epochs to avoid large drops in embedding quality.

Batch Sampling. We analyzed the effect of batch sampling next. We trained ComplEx in a 2@1 setup. For each setting, we trained with 0%, 50% and 100% of entities sampled from within the batch (and the rest shared). The results are shown in Tab. 9c. We found that the substantial reduction of the sampling pool performed by batch sampling consistently deteriorated embedding quality on all datasets but Freebase, where batch sampling substantially improved embedding quality. Results are thus highly dataset-dependent. Over all datasets a combination of uniform and batch sampling was preferable. To see whether the quality degradation can be avoided, we performed a separate hyperparameter search with batch sampling enabled, but were not able to reach a similar quality as achieved without batch sampling.

5.6 Combination of Sampling and Partitioning Techniques (Tab. 10)

We investigated the interplay of partitioning techniques with local sampling as well as batch sampling on Freebase, the only dataset for which we saw a positive influence of batch sampling. Our results for random and stratification partitioning are shown in Tab. 10. Batch sampling was generally beneficial, whereas a combination of local and batch sampling deteriorated quality with stratification but not random partitioning.

5.7 Graph-Cut Partitioning (Tab. 11)

In our experiments, we often found that graph-cut partitioning led to lower speedups than CARL stratification (in addition to often leading to deteriorating embedding quality). The reason was that triple partition sizes were often quite unbalanced. Tab. 11 shows that this holds for all datasets, and that the time required to process a partition varied substantially between smallest and largest partition.

Table 10: Comparison of sampling technique combinations (ComplEx, Freebase).

Set-up	Partition technique	Epoch time	Data sent per epoch	MRR	Hits @10
1@1	Seq. (mm)	3929s	-	0.364	0.487
1@1	Seq. (B) (mm)	3925s	-	0.426	0.528
2@2	Random (R)	983s	295.0GB	0.364	0.483
2@2	Random (RL)	953s	232.0GB	0.365	0.485
2@2	Random (RB)	944s	263.4GB	0.420	0.525
2@2	Random (RLB)	966s	232.8GB	**0.426**	**0.529**
2@2	Strat. (CARL)	819s	124.3GB	0.327	0.439
2@2	Strat. (CARB)	1128s	182.4GB	0.391	0.491
2@2	Strat. (CARLB)	**803s**	**123.2GB**	0.325	0.424

Table 11: Graph-cut leads to unbalanced triple partitions. CV is the coefficient of variation of the triple partition sizes.

Dataset	#Partitions	#Triples			Epoch time	
		CV	Min	Max	Min	Max
FB15k	8	24.9%	6.3%	17.4%	0.8s	2.0s
Yago3-10	8	37.1%	7.3%	22.0%	5.6s	14.8s
Wikidata5m	8	28.9%	7.3%	17.3%	92.4s	194.7s
Freebase	8	15.9%	9.4%	15.1%	587.9s	1170.6s

Table 12: Stratification - average fraction of active entities per partition ($M = 8$).

	FB15k	Yago3-10	Wikidata5m	Freebase
Normal	78.6%	37.3%	27.3%	25.9%
Combined	82.6%	62.0%	43.2%	31.3%

5.8 Plain Stratification (Tab. 13)

We used the improved CARL stratification throughout the experimental study because it was consistently more efficient and effective than plain stratification. Here we report on the influence of combining mirror partitions (C), relocating only active entities (A), and repartitioning (R). We trained various variants of stratification in a 2@1 setting and measured training time, embedding quality, and communication costs.

Tab. 12 lists the average fraction of active entities for stratification with $M = 8$ with and without combining mirror partitions. Note that for the larger datasets, most entities are inactive, i.e., they do not occur in the respective partition. As shown in Tab. 13, the restriction of parameter relocation to active entities substantially reduced the network footprint (up to 70%) and also improved epoch times (by up to almost 20%). Combining partitions further reduced the network footprint and communication cost as parameters needed to be synchronized less often.

Repartitioning (R) mainly affected quality; see Fig. 5. In fact, stratification without repartitioning led to a substantial drop in quality due to local sampling, as discussed in Sec. 5.5. Stratification with repartitioning as well as CARL stratification did not lead to reduced quality. In contrast, we observed a small improvement with CARL stratification. This may be due to the change of sampling bias when using active entities: it is closer to the actual distribution of entities in the KG.

Overall, CARL stratification was instrumental for the effectiveness of stratification; plain stratification was not competitive.

Table 13: With Stratification sampling only from active entities (A) and combining mirror partitions (C) decreased network footprint and epoch time (ComplEx 1@2).

	Dataset	All entities		Active entities	
		Epoch time	Data sent	Epoch time	Data sent
Normal	FB15k	3.4s	0.2GB	3.6s	0.2GB
	Yago3-10	13.8s	1.2GB	10.7s	0.4GB
	Wikidata5m	296.7s	60.3GB	245.6s	26.5GB
Combined	FB15k	3.0s	0.1GB	2.9s	0.1GB
	Yago3-10	11.3s	0.6GB	10.9s	0.4GB
	Wikidata5m	252.3s	31.0GB	228.0s	15.0GB

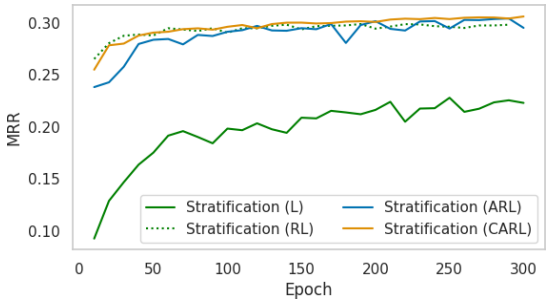


Figure 5: Sampling only from active entities (A) has a positive influence on quality (ComplEx, Wikidata5m).

6 CONCLUSION

We described and evaluated state-of-the-art techniques for parallel training of KGE models of large-scale knowledge graphs. We found that it is possible to achieve high speedup (up to 8× with 8 GPUs) with high embedding quality, both in a single-machine multi-GPU and in a multi-machine multi-GPU setup. However, the parallelization techniques currently implemented in large-scale KGE training frameworks did not realize these improvements and often led to a quality degradation compared to sequential training. This was mainly caused by the combination of a static partitioning and local sampling used by these implementations. Our experiments suggest that the overall choice of partitioning and sampling technique is highly dataset-dependent. For example, on Freebase, the random partitioning baseline in combination with improved sampling methods led to the overall best results. On all other datasets, CARL stratification—a variant of stratification as used in PyTorch BigGraph—along with shared local sampling often performed competitive or best.

REFERENCES

[1] Mehdi Ali, Max Berrendorf, Charles Tapley Hoyt, Laurent Vermue, Mikhail Galkin, Sahand Sharifzadeh, Asja Fischer, Volker Tresp, and Jens Lehmann. 2020. Bringing Light Into the Dark: A Large-scale Evaluation of Knowledge Graph Embedding Models Under a Unified Framework. *arXiv preprint arXiv:2006.13365* (2020).

[2] Mehdi Ali, Max Berrendorf, Charles Tapley Hoyt, Laurent Vermue, Sahand Sharifzadeh, Volker Tresp, and Jens Lehmann. 2021. PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings. *Journal of Machine Learning Research* 22, 82 (2021), 1–6. http://jmlr.org/papers/v22/20-825.html

[3] Stephan Baier, Yunpu Ma, and Volker Tresp. 2017. Improving visual relationship detection using semantic modeling of scene descriptions. In *International Semantic Web Conference*. Springer, 53–68.

[4] Ivana Balažević, Carl Allen, and Timothy M Hospedales. 2019. Tucker: Tensor factorization for knowledge graph completion. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[5] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems* 26 (2013), 2787–2795.

[6] Samuel Broscheit, Daniel Ruffinelli, Adrian Kochsiek, Patrick Betz, and Rainer Gemulla. 2020. LibKGE A knowledge graph embedding library for reproducible research. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[7] Sanxing Chen, Xiaodong Liu, Jianfeng Gao, Jian Jiao, Ruofei Zhang, and Yangfeng Ji. 2020. HittER: Hierarchical Transformers for Knowledge Graph Embeddings. *arXiv preprint arXiv:2008.12813* (2020).

[8] Luca Costabello, Sumit Pai, Chan Le Van, Rory McGrath, Nicholas McCarthy, and Pedro Tabacof. 2019. AmpliGraph: a Library for Representation Learning on Knowledge Graphs. https://doi.org/10.5281/zenodo.2595043

[9] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. 2018. Convolutional 2D Knowledge Graph Embeddings. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 1811–1818. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17366

[10] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* 12, 7 (2011).

[11] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 69–77.

[12] Ronald L Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell system technical journal* 45, 9 (1966), 1563–1581.

[13] Xu Han, Shulin Cao, Lv Xin, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. 2018. OpenKE: An Open Toolkit for Knowledge Embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[14] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing* 48, 1 (1998), 96–129.

[15] Seyed Mehran Kazemi and David Poole. 2018. SimplE Embedding for Link Prediction in Knowledge Graphs. In *NeurIPS*.

[16] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

[17] Bhushan Kotnis and Vivi Nastase. 2017. Analysis of the impact of negative sampling on link prediction in knowledge graphs. *arXiv preprint arXiv:1708.06816* (2017).

[18] Timothée Lacroix, Nicolas Usunier, and Guillaume Obozinski. 2018. Canonical tensor decomposition for knowledge base completion. In *Proceedings of 35th International Conference on Machine Learning*. PMLR, 2863–2872.

[19] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large-scale graph embedding system. *Proceedings of the 2nd SysML Conference* (2019).

[20] Sameh K Mohamed, Aayah Nounu, and Vít Nováček. 2019. Drug target discovery using knowledge graph embeddings. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 11–18.

[21] Jason Mohoney, Roger Waleffe, Yiheng Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Learning Massive Graph Embeddings on a Single Machine. *arXiv preprint arXiv:2101.08358* (2021).

[22] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs. *Proc. IEEE* (2015).

[23] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. 2011. A three-way model for collective learning on multi-relational data.. In *Proceedings of the 28th International Conference on Machine Learning*, Vol. 11. 809–816.

[24] Alexander Renz-Wieland, Rainer Gemulla, Steffen Zeuch, and V. Markl. 2020. Dynamic parameter allocation in parameter servers. *Proceedings of the VLDB Endowment* 13 (2020), 1877 – 1890.

[25] Andrea Rossi, Denilson Barbosa, Donatella Firmani, Antonio Matinata, and Paolo Merialdo. 2021. Knowledge graph embedding for link prediction: A comparative analysis. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 2 (2021), 1–49.

[26] Daniel Ruffinelli, Samuel Broscheit, and Rainer Gemulla. 2020. You CAN Teach an Old Dog New Tricks! On Training Knowledge Graph Embeddings. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BkxSmlBFvr

[27] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*. Springer, 593–607.

[28] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019. RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkgEQnRqYQ

[29] Zhiqing Sun, Shikhar Vashishth, Soumya Sanyal, Partha Talukdar, and Yiming Yang. 2019. A re-evaluation of knowledge graph completion methods. *arXiv preprint arXiv:1911.03903* (2019).

[30] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex Embeddings for Simple Link Prediction. In *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2071–2080. http://proceedings.mlr.press/v48/trouillon16.html

[31] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2724–2743.

[32] Xiaozhi Wang, Tianyu Gao, Zhaocheng Zhu, Zhiyuan Liu, Juanzi Li, and Jian Tang. 2021. KEPLER: A unified model for knowledge embedding and pre-trained language representation. *Transactions of the Association for Computational Linguistics* (2021).

[33] Bishan Yang, Scott Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[34] Denghui Zhang, Manling Li, Yantao Jia, Yuanzhuo Wang, and Xueqi Cheng. 2017. Efficient parallel translating embedding for knowledge graphs. In *Proceedings of the International Conference on Web Intelligence*. 460–468.

[35] Shuai Zhang, Yi Tay, Lina Yao, and Qi Liu. 2019. Quaternion Knowledge Graph Embeddings. In *NeurIPS*.

[36] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zi-Hao Ye, J. Dong, Hao Xiong, Zheng Zhang, and G. Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (2020).

[37] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *The World Wide Web Conference*. 2494–2504.