HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework

[Scalable Data Science]

Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, Bin Cui
Department of Computer Science & Key Lab of High Confidence Software Technologies (MOE), Peking University
[7]Institute of Computational Social Science, Peking University (Qingdao), [1,6]Tencent Inc.
{xupeng.miao, z.hl, shiyining, xiaonan.nie, yangzhi, bin.cui}@pku.edu.cn, brucetao@tencent.com

ABSTRACT

Embedding models have been an effective learning paradigm for high-dimensional data. However, one open issue of embedding models is that their representations (latent factors) often result in large parameter space. We observe that existing distributed training frameworks face a scalability issue of embedding models since updating and retrieving the shared embedding parameters from servers usually dominates the training cycle. In this paper, we propose HET, a new system framework that significantly improves the scalability of huge embedding model training. We embrace skewed popularity distributions of embeddings as a performance opportunity and leverage it to address the communication bottleneck with an *embedding cache*. To ensure consistency across the caches, we incorporate a new consistency model into HET design, which provides fine-grained consistency guarantees on a per-embedding basis. Compared to previous work that only allows staleness for read operations, HET also utilizes staleness for write operations. Evaluations on six representative tasks show that HET achieves up to 88% embedding communication reductions and up to 20.68× performance speedup over the state-of-the-art baselines.

PVLDB Reference Format:
Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, Bin Cui. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework . PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/PKU-DAIR/Hetu/.

1 INTRODUCTION

To train a model on high-dimensional data, such as words in a corpus of text [8, 33, 36] or the user-item interaction data [39, 49], it is common to use an embedding model, which projects a sparse high-dimensional feature space, into a continuous low-dimensional *embedding* space. For example, in a language model, a training example might be a sparse vector with non-zero entries corresponding to the IDs of words in a vocabulary, and the distributed representation for each word will be a lower-dimensional vector. "Wide and deep learning" [9] creates distributed representations from cross-product transformations on categorical features. Embedding model is common at modern web companies (e.g., Facebook [34], Google [11] and Tencent [44]), which have been recognized as an effective learning paradigm to extract useful information for downstream tasks such as recommendation.

As each feature needs to be represented by a set of embeddings (i.e., latent vectors), many embedding models are at a *giant* scale and are too large to copy to a worker on every use, or even to store in RAM on a single host. For instance, the parameters of a real-world document embedding model in Google [6, 12] occupies several terabytes, and the industrial click-through rate prediction model in Baidu [48] has 10^{11} input sparse features and also requires 10 Tb parameters. For this reason, it is challenging to scale embedding models up to large-scale use cases, in which millions or even billions of parameters need to be learned.

Modern distributed ML systems (e.g., TensorFlow [6]) typically adopt the parameter server [26] framework to scale out models. The server usually maintains the globally shared parameters by aggregating updates from the workers and updating the global parameters. Workers communicate only with the server nodes, updating and retrieving the shared parameters. Existing ML systems usually support data parallelism where a worker usually contains a replica of the ML model and is assigned an equal-sized partition of the entire training data. Bulk Synchronous Parallel (BSP) [14] or Asynchronous Parallel (ASP) [30] are usually adopted for updating the model parameters during distributed training.

However, this setup faces a scalability issue for large embedding models [40, 48]. We observe that the greatest inefficiency comes from *updating and retrieving the shared feature embedding parameters* through a limited bandwidth link. For example, using TensorFlow with ASP, up to 86% of training time is spent on embedding fetching and updating, which dominates the training cycle. The major reason is that an embedding model often uses deep neural networks with low computational complexity, comparing with the giant embedding data. Accordingly, the computation takes a much shorter time than the reads and writes of remote embedding data. Moreover, with the increasing gap between emerging powerful accelerators and the slow growth of network bandwidth, the embedding communication bottleneck would become even more severe. To our knowledge, there is little prior work addressing the scalability issue of embedding models in a distributed environment.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

In this paper, we propose **HET**, a novel distributed system framework to scale **H**uge **E**mbedding model **T**raining. Our key idea is to exploit an efficient **embedding-cache-enabled** architecture, which is mainly inspired by the critical characteristics for embedding models: *popularity skewness* and *staleness tolerant*. Specifically, the popularity distribution of embeddings is often highly skewed, typically following power-law distributions [45], implying a performance opportunity: *a small cache of hot embedding at each worker can effectively save the network bandwidth while scaling training throughput with the number of workers.*

Replicating shared embedding data in multiple caches raises the problem of consistency in the presence of writes. Fortunately, embedding models are iterative convergent algorithms where some staleness errors during training are acceptable and will not prevent convergence. In other words, embedding models are robust to a bounded amount of inconsistency (e.g., reading out-of-date shared state). By relaxing the consistency guarantees properly, we can exploit the opportunity of caches to gain significant system improvements. However, conventional relaxed consistency models such as Stale Synchronous Parallel (SSP) [18] are not aware of the presence of skew access and require that every single worker should be able to hold an entire set of parameters. Moreover, they mainly target straggler problems rather than communication overhead. For example, SSP maintains an up-to-date global model through sent out write updates to servers each clock. Considering the large scale and communication cost of embedding models, the above issues become a critical limitation for scaling out the training.

To address the above issues, we incorporate a new consistency model into HET. Our consistency model differs from the traditional ones in two aspects. First, we enable the **fine-grained** caching and consistency that provides guarantees on a per-embedding basis. Specifically, for each cached embedding, we leverage an embedding-specific Lamport clock to manage its fine-grained consistency actions (e.g., validation, synchronization) and provide the concept of “per-embedding-clock-bounded” consistency — a worker can see all updates of an embedding older than certain embedding-specific clocks. We provide a fine-grained consistency model and theoretically prove its convergence guarantees. Second, compared to previous works that only allow staleness for read operations, we further utilizes staleness for writes, allowing **stale-writes** based on the timestamp deviation between the global and local clocks of each embedding. This means that writing to an embedding residing in the cache does not update the underlying global model until the embedding is invalided or evicted from the cache. This feature is critical in reducing communication overheads.

We summarize our contributions as follows: First, we reveal the performance bottleneck and the opportunity for scaling huge embedding models, and introduce a novel **system abstraction** with embedding cache. Second, we employ a new cache **consistency model** that provides (1) clock-bounded consistency at the fine-grained of each embedding, and (2) allows staleness for both caches read and write operations for minimizing communication overhead. Finally, we build **HET system**, a new framework that implements the proposed system abstraction and the consistency model, and supports 10^{12} parameters scale embedding model training, achieving $6.37-20.68\times$ speedup and up to 88% embedding communication reduction over the state-of-the-art baseline systems.

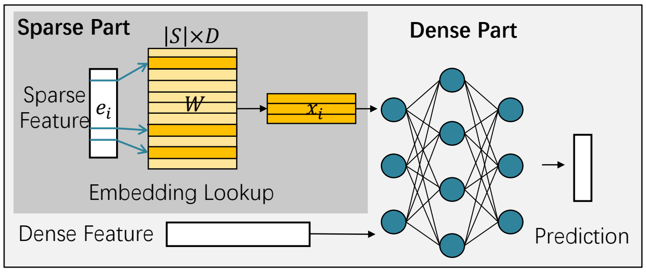


Figure 1: Illustrate of embedding model architecture

2 PRELIMINARY

2.1 Distributed Training

Distributed machine learning. The target of machine learning is to find a model $\mathbf{x} \in \mathbb{R}^d$ (d is the total number of parameters in the model) that minimizes the empirical risk:

$$\min_{\mathbf{x}} \left[F(\mathbf{x}) := \frac{1}{|\xi|} \sum_{i} f(\mathbf{x}; \xi_i) \right], \tag{1}$$

where $f(\cdot)$ is the loss function, ξ is the training dataset and ξ_i represents the i-th data sample. Distributed ML systems have been extensively studied in recent years to scale up ML for big data and large models. Parameter Server (PS) is a trendy data parallelism architecture for many existing systems (e.g., TensorFlow [6], PS2 [46]). Another choice is All-Reduce and several recent systems (e.g., PyTorch [27], Horovod [37]) show superior performance over PS with the help of NCCL [3], especially for dense models.

Parallel training paradigms. There are three paradigms for updating the model parameters during distributed deep learning with data parallelism. Most of these studies manage to keep consistent model performance as the standalone mini-batch SGD. BSP assumes that all N workers are fully synchronized and performing the following update rule:

$$\mathbf{x}(t+1) = \mathbf{x}(t) - \eta \left[\frac{1}{N} \sum_{i=1}^{N} G^i(\mathbf{x}(t); \xi^i) \right], \tag{2}$$

where η is the learning rate, ξ^i are randomly sampled from the training set, and $G^i(\cdot)$ denotes the gradient from the i-th worker. However, the frequent synchronization and straggler problem bring significant communication costs. ASP has been proposed to avoid such overheads by allowing the workers to proceed without waiting for each other. But the model degradation happens because of the stale model gradients. To balance the trade-off between training efficiency and model performance, SSP and several variants [21] have been proposed. It has been proved that stale synchronous methods could share the same convergence rate with BSP when the staleness is upper bounded [29]. Unfortunately, SSP requires to store the replication of the entire model inside every single worker, which is impractical for giant models.

2.2 Embedding Models

Many types of embedding models (e.g., Wide & Deep [9], Deep & Cross [38], DeepFM [15], xDeepFM [28] and Deep Interest Network [49]) have been developed for high-dimension data, and have achieved widespread success in recommender systems–a critical

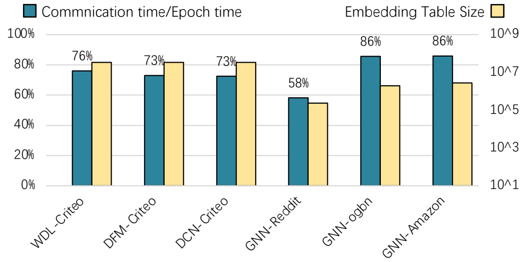


Figure 2: Large embedding model workloads on TensorFlow

service for internet companies. Figure 1 illustrates a common embedding model architecture. In order to handle categorical data, embedding tables map categorical features to dense representations in an abstract space. In particular, each embedding lookup may be interpreted as using a one-hot vector e_i (with the i-th position being 1 while others are 0, where index i corresponds to i-th category) to obtain the corresponding row vector of the embedding table $\mathbf{W} \in \mathbb{R}^{|S| \times D}$ as: $\mathbf{x}_i = \mathbf{W}^\top \mathbf{e}_i$.

2.3 Problems and Opportunities

Communication cost. Large-scale embedding models suffer communication bottlenecks from both dense parameters and sparse parameters. PS is often suitable for the sparse communication based on the embedding lookup operations. AllReduce is highly optimized for the dense communication across GPUs (e.g., NCCL). But it has to degenerate to the inefficient AllGather primitive for sparse communication. Considering the difference in the sparsity of model parameters, Parallax [24] proposes a hybrid communication architecture that combines PS and AllReduce to transfer sparse and dense parameters respectively. Kraken [40] follows the hybrid architecture and optimizes the embedding memory usage. HugeCTR [4] is NVIDIA's high-efficiency GPU framework designed for recommendation systems on multiple GPUs, but it is memory restricted since all embedding parameters must be maintained within GPUs.

In general, sparse embeddings dominate the communication bottleneck for large-scale embedding model training. We evaluate the distributed training efficiency of TensorFlow on six popular high-dimensional (almost up to 10^7) embedding workloads on a single worker, including both click-through rate prediction and graph representation learning. Only the embedding table is deployed on the remote parameter server. We use a small embedding size $D = 32$ and the network bandwidth is 1 Gbps. Figure 2 summarizes the time occupation of data transfer and the number of parameters over different embedding models and datasets. Clearly, across all the models and datasets, communication takes much longer time than computation. This situation will become more common at modern web companies as the embedding model is still growing in industrial applications (e.g., $|S| = 10^9$ to 10^{11} in [40, 48] and $D = 10000$ in [12]).

Meanwhile, we also find the following characteristics of the embedding models that provide us opportunities to further improve the performance of embedding model training.

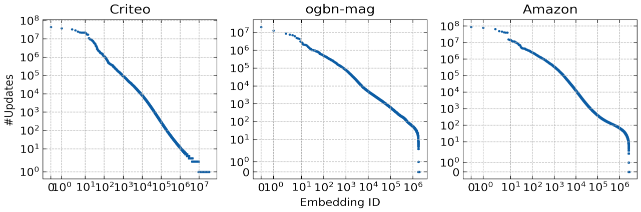


Figure 3: Embedding popularity skewness

Skewness. Figure 3 illustrates the skew distribution of embedding update frequency on some popular workloads, including click-through rate prediction (i.e., Criteo), citation network (i.e., ogbn-mag), and product co-purchasing network (i.e., Amazon). The top 10% popular embeddings on Criteo could account for 90% total number of updates. The observation motivates us to reduce the communication by caching frequently updated embeddings in limited local memory. Existing research provides evidence that parameter updates from various embedding models exhibit a universal skewed distribution [32], such as recommendation models [7, 22, 47], LDA topic models [23, 41, 43] and graph learning models [31].

Robustness. Introducing cache raises the problem of ensuring consistency in the presence of writes. Existing embedding models fall into the category of iterative convergent algorithms which start from a randomly chosen initial point and converge to optima by repeating iteratively a set of procedures. Such iterative convergent process has been shown robust to bounded amount of inconsistency and still converge correctly [30]. This property allows frameworks to improve system performance by relaxing cache consistency models and reading from local (out-of-date) caches.

3 HET DESIGN

This section describes a novel system solution to scale embedding models by exploiting cache. Figure 4 provides an overview of our system, which distributes the training data into multiple workers. Each worker holds a replication of dense model parameters and uses AllReduce for gradients synchronization during training. HET organizes shared embedding parameters as tables. The whole embedding parameters are stored in the global embedding table on the HET server. The client is responsible for the management of the local cache through communicating with servers to control the inconsistency between the local and the global embedding table. Below, we first introduce the design of cache management and read/write protocols. Then we analyze the cache consistency and model convergence guarantees enforced by our system.

3.1 Cache Management

The embeddings are organized in a collection of rows in the **embedding table**. Each embedding represents a sparse feature ID denoted by a unique key k. In the global embedding table, each global embedding $\mathbf{x}_k$ records a global Lamport clock [25] $\mathbf{x}_k.c_g$ indicating the total number of updates on this embedding. The servers storing the global embedding table act as the cache coordinator. Each worker can cache a small subset of the embedding table locally. Instead of recording the "clock time" of each client, HET supports fine-grained timing to coordinate cache synchronization at per-embedding basis. In the cache embedding table, each local embedding $\mathbf{x}_k^i$ on the i-th worker records two clocks. (1) The *start clock* $\mathbf{x}_k^i.c_s$ denotes the observed global clock when the last time the embedding $\mathbf{x}_k$ was fetched from server to worker i. (2) To guarantee "read-my-updates" for a worker i, we also increases the *local*

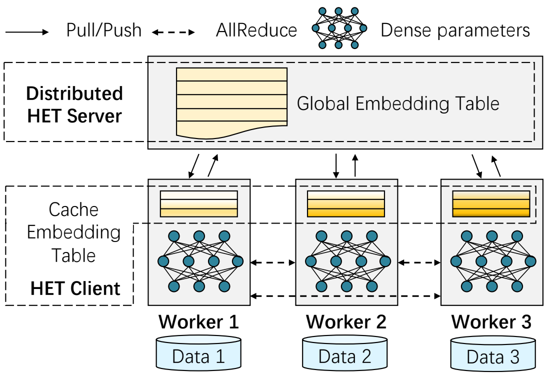


Figure 4: System architecture of HET

clock time $\mathbf{x}_k^i.c_c$ of the embedding $\mathbf{x}_k$ by one once it is updated in a iteration on that worker.

Our system provides operations on the embedding table to manage the cache. Listed below are the core methods of the HET client library. We illustrate some of these interfaces in the execution flow for one iteration in Figure 5.

`Het.Cache.Fetch(key)`: This operation directly reads the embedding indexed by `key` from the global embedding table on the server. For the fetched embedding $\mathbf{x}_k^i$, the start clock $\mathbf{x}_k^i.c_s$ and local clock $\mathbf{x}_k^i.c_c$ both are set to be equal to its global clock $\mathbf{x}_k.c_g$.

`Het.Cache.Evict(key)`: If input parameter `key` is provided, this operation finds and evicts the corresponding embedding and push the accumulated gradients and local clock $\mathbf{x}_k^i.c_c$ to the server. The server receives the accumulated gradients, applies them on the corresponding entry of global embedding table, and synchronizes the global clock $\mathbf{x}_k^i.c_g = \max(\mathbf{x}_k^i.c_g, \mathbf{x}_k^i.c_c)$. If `key` is not provided, this operation tries to evict the overflowed embeddings selected by certain cache policies (e.g., LRU, LFU) to prevent the cache table from exceeding the size limitation. We further discuss the selection of cache policies in Section 4.3.

`Het.Cache.CheckValid(key)`: This operation finds the local embedding x_k^i from the cache according to the `key` and returns true if its current clock $x_k^i.c_c$ satisfies the following two time-bound conditions: (1) the current clock should not be too far ahead of the start clock, i.e., $x_k^i.c_c \leq x_k^i.c_s + s$; (2) the current clock should not be too far behind its global clock, i.e., $x_k.c_g \leq x_k^i.c_c + s$. Here s is a user-defined staleness threshold to determine the cache validity. Since the global clock is recorded on the server, to validate condition (2), we have to send $x_k^i.c_c$ from the worker to the server when the cache hit occurs. Note that, the communication costs in this step are not significant because we only send the clocks, rather than the embedding vectors.

3.2 Read/write Protocols

We briefly introduce the workflow of HET client in Algorithm 1. After each worker initializes the model parameters, it repeats the iterations based on the mini-batch SGD algorithm and trains the embedding model. The client extracts the unique keys (i.e., feature ID) from the mini-batch of data and Reads the corresponding embeddings. The DL executor performs forward and backward computation using these model parameters and input data. After that, we could update the dense model parameters and sparse embeddings, respectively.

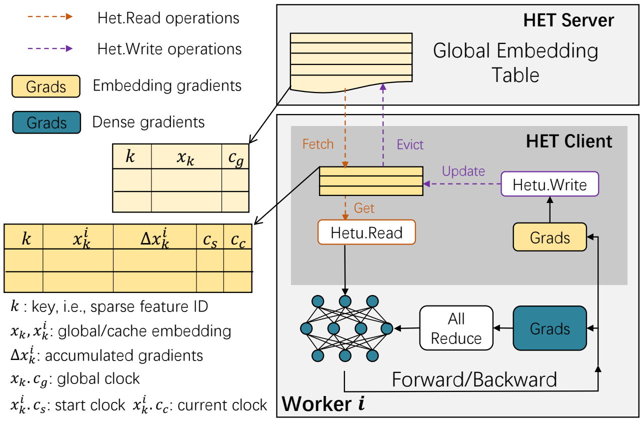


Figure 5: Workflow of HET

Algorithm 1: Het Client

```
Input: input dataset $\xi$, max iterations $T$, model parameters
       $\mathbf{x}_0$ and $\mathbf{x}_e$, DL runtime DL
Output: Trained model $\mathbf{x}_0$ and $\mathbf{x}_e$.
Initialize model parameter $\mathbf{x}_0$;
Het.Intialize($\mathbf{x}_e$);
for $i \in range(T)$ do
    $\xi_i$ ←sample a mini-batch of data from $\xi$;
    $K_i$ ← the unique embedding key set of $\xi_i$;
    $E_i$ ← Het.Read($K_i$) ;          /* Read embeddings */
    $G_i$ ← DL.Forward/Backward($\xi_i, E_i, \mathbf{x}_0$);
    DL.Update($G_i(\mathbf{x}_0)$) ;   /* Locally update dense */
    Het.Write($K_i, G_i(\mathbf{x}_e)$) ;   /* Write embedding */
end
```

`Het.Read(keys)`: Read a set of embedding vectors based on the requested `keys` as shown in Algorithm 2. For each key k, the client first checks whether k exists in the cache embedding table (i.e., `Het.Cache.Find(key)` in line 3). If not, the client fetches the latest version embedding from the server and adds it into the local cache embedding table temporarily (line 8); If so, the client further checks whether the caching embedding is valid (line 4) and manages to cache up with the server by synchronizations (line 5). For those embeddings within the staleness threshold, the client directly reads from the local embedding table (i.e., `Het.Cache.Get(key)`).

`Het.Write(keys, gradients)`: Writing back the embedding gradients as shown in Algorithm 3. Our cache embedding table allows **stale-writes** to reduce the communication cost between client and server. Since all the embeddings with keys have been loaded in the cache embedding table before the forward and backward computation, we could directly write the gradients locally by accumulating them on the corresponding rows of the cache embedding table (i.e., `Het.Cache.Update(key, grad)` in line 2). This enforces the read-my-updates property, which ensures that the data read by a client contains all its own updates. Meanwhile,

Algorithm 2: Het.Read

```
   Input: input key set $K$
   Output: Embedding set $E$
1  $E \leftarrow \{\}$;
2  for $k \in K$ do
3      if Het.Cache.Find(k) then
4          if not Het.Cache.CheckValid(k) then
5              Het.Cache.Evict(k) ;   /* Synchronize */
6              Het.Cache.Fetch(k) ;    /* embeddings */
7          end
8      else
9          Het.Cache.Fetch(k);
10     end
11     $E \leftarrow E \cup$ Het.Cache.Get(k);
12 end
```

Algorithm 3: Het.Write

```
   Input: input key set $K$, the embedding gradients $G$
1  for $k \in K$ do
2      Het.Cache.Update(k, $G_k$);
3      Het.Cache.Clock($k$) ;    /* Increase $c_c$ by 1 */
4  end
5  Het.Cache.Evict();
```

these cache embeddings should increase their current clocks by 1 (line 3). These accumulated updates could only be written back to server later through the cache eviction operation, which become "stale" relative to the global embedding table.

3.3 Cache Consistency Guarantee

From the per-embedding perspective, each embedding might exist in multiple cache embedding tables during training. Therefore, the cache consistency guarantee is crucial for the final model quality. Before interpreting the cache consistency model, we first clarify the following lemma on the clock consistency between any two embedding replications in different workers:

Lemma 1. *For any $\mathbf{x}_k$, let $\mathbf{x}_k^i, \mathbf{x}_k^j$ are its two replicas cached on worker i, j, respectively, HET guarantees that:*

$$\forall k, \max_{\forall 0 \le i,j \le N} \{|\mathbf{x}_k^i.c_c - \mathbf{x}_k^j.c_c|\} \le 2s. \tag{3}$$

Proof. For any embedding $\mathbf{x}_k$ at iteration t, the replication on the i-th worker is denoted by $\mathbf{x}_k^i$. Based on the conditions in CheckValid(key), we have: $\mathbf{x}_k.c_g - s \le \mathbf{x}_k^i.c_c$ and $\mathbf{x}_k^i.c_c \le \mathbf{x}_k^i.c_s + s \le \mathbf{x}_k.c_g + s$. Therefore, for any two different workers i and j, the difference between their local current clocks is upper bounded by $2s$. □

Lemma 1 formally describes the clock-bounded guarantee at per embedding basis. This guarantee enforces the following consistency model across embedding replications in multiple caches:

Definition 1 (**Per-embedding clock bounded consistency**). *For any embedding $\mathbf{x}_k$, the consistency model guarantees that a worker i sees the updates of any other worker j on embedding $\mathbf{x}_k$ in the range of $[0, \mathbf{x}_k^j.c_c - 2s]$.*

It is worth pointing out that Lemma 1 and the per-embedding clock bounded consistency only describe the observable embeddings. If a worker fetches a key, makes an update, and never sees that key again, while other workers continue to see that key. The embedding in that worker is going to be evicted when the cache capacity is full, and the update would be written back to the server. In the corner case, it might remain in the cache until the model completes the training process, we could simply ignore that update due to the robustness of iterative convergent algorithms.

3.4 Convergence Analysis

Recall that we differ from traditional SSP in the following aspects to improve performance: (1) SSP is not aware of the presence of skew access and provides bounded staleness at the coarse granularity measured by worker clocks, whereas we provide bounded staleness at the fine granularity of individual embedding clocks considering access heterogeneity. (2) SSP assumes a write-through cache so that the server is up-to-date, whereas we adopt a write-back-with-stale server to improve write performance. Given these key differences, we present a new theoretical analysis of the proposed consistency model, instead of reusing that of SSP. We summarize our proof results showing that our algorithm is guaranteed to converge under our consistency model. The details of the assumptions and proofs are in Appendix A. We decompose the d model parameters into two categories, including the dense parameters x_0 and the sparse parameters of $\mathbf{x}_e$ containing m embedding vectors $(x_1, x_2, \ldots, x_m)$. We denote x_i and $\nabla_i f(\mathbf{x})$ as the i-th component of $\mathbf{x}$ and $\nabla f(\mathbf{x})$, respectively. Clearly, $\mathbf{x} = (x_0, x_1, \ldots, x_m)$ and $\nabla f = (\nabla_0 f, \nabla_1 f, \cdots, \nabla_m f)$. We have the following theorem:

Theorem 1 (Global Convergence Rate). *Consider an arbitrary objective function f, under the per-embedding-clock-bounded consistency model and and certain assumptions, Given the success parameter $\epsilon > 0$, a constant learning rate value*

$$\eta \le \min\left(\frac{\sqrt{\epsilon}}{4\sqrt{3}sLMB}, \frac{\sqrt{\epsilon}}{4\sqrt{L}sMB}, \frac{\epsilon}{12M^2B^2L}\right)$$

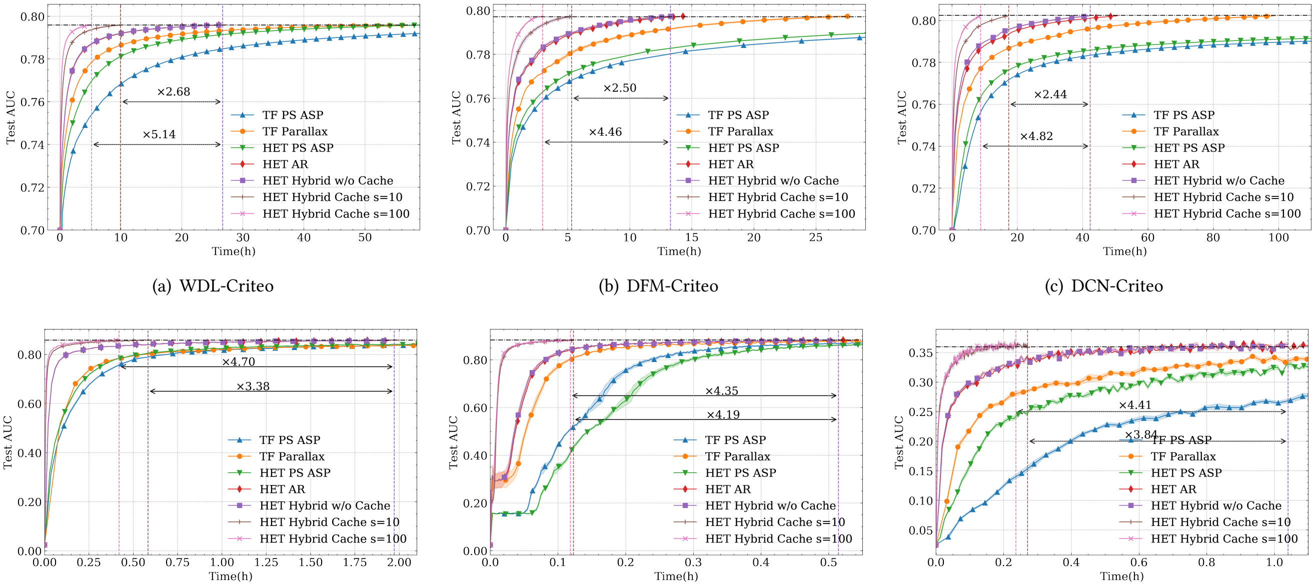
and $T = \Theta\left(\frac{f(x(0)) - f_{inf}}{\epsilon\eta}\right)$ iterations, for worker j, we are guaranteed to reach some iterate $\mathbf{x}^j(t^\star)$ with $1 \le t \le T$ such that

$$\mathbb{E}\|\nabla f(\mathbf{x}^j(t^\star))\|^2 \le \epsilon.$$

4 HET IMPLEMENTATION

HET's implementation consists of 14.5K LOC in C/C++/CUDA with a Python dataflow front-end (20.7K LOC)[1]. It is easy to extend our cache embedding mechanism to other DL systems by replacing the DL runtime (e.g., TensorFlow, PyTorch, MXNet). Taking TensorFlow (TF) as an example, we could first replace the native TF parameter server with our HET server to store the global embedding table. Then we could implement an embedding variable inheriting from TF, and encapsulate the lookup/update operations with HET client interfaces. We leave the extension as our future work.

[1] https://github.com/PKU-DAIR/Hetu/



(a) WDL-Criteo

(b) DFM-Criteo

(c) DCN-Criteo

(d) GNN-Amazon

(e) GNN-Reddit

(f) GNN-obgn-mag

Figure 6: Convergence performance comparison.

In our hybrid communication architecture, AllReduce is implemented by MPI [13] and NCCL [3]; the key components – HET client and server, are developed based on PS-Lite[5], a lightweight implementation of PS interface. Currently, we implement the embedding table in C++ and store the cache embedding table in the limited DRAM (e.g., 12 GB) of each worker in our experiments. We manage to improve the overall performance by leveraging the following implementation optimizations.

4.1 Asynchronous Communication Invocation

Similar to TensorFlow[6], we use a static computation graph abstraction to organize all the operations in HET. All operators implemented by GPU kernels are scheduled into the GPU stream. These operators will be launched and executed asynchronously to avoid blocking the CPU execution. In HET, the communication operations (e.g., AllReduce, Fetch and Evict) are also treated similarly to overlap with computations. To ensure dependencies between computation and communication, we borrow the idea of CUDA event from GPU. We asynchronously launch communication requests and record corresponding events to synchronize when the updated parameter should be used in the next iteration.

4.2 Message Fusion

Combining `Pull` (model parameters) and `Push` (model gradients) operations in parameter server architecture is a common technique [35] for performance improvement. It is easy to implement for traditional dense parameters. As for sparse parameters, especially for embedding tables, it is non-trivial since the sparse access property of embedding models. To combine the cache eviction and fetching, we need to pre-fetch the next mini-batch of data in advance to inform the embedding indices. Besides, we also remove the duplicate keys in the request of each mini-batch to reduce redundant communication costs.

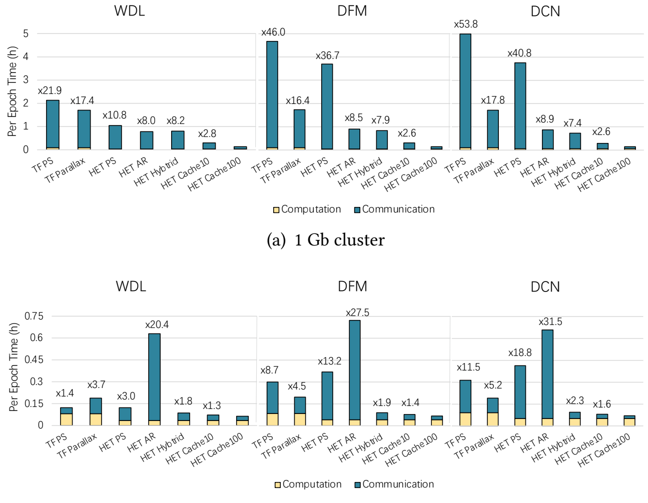
4.3 Cache Strategies

The goal of our cache strategy is to maximize the embedding lookup hit rate within a restricted amount of memory. The effectiveness of the cache is decided by two major factors: the query frequency from local workers and the length of its expiration period. The latter one is affected by other workers' workload and is hard to predict, so we only focus on optimizing the first objective, which is to cache the most frequently used embeddings by the local workers (e.g., LFU and LRU policy). Due to the high maintenance cost incurred by LFU, we provide a light-weighted version of LFU. When the frequency of an embedding is high enough, it will be assigned a direct access index, bypassing the cost of frequency maintenance. Under the same workload, it could have a similar miss rate as the original LFU while retaining a significantly small run-time cost.

5 EXPERIMENTS

Baselines. In this section, we compare our prototype system with two state-of-the-art systems: TensorFlow (TF) [6] and Parallax [24]. To alleviate the concerns on the difference from the system backbones and implementations, we implement three auxiliary baselines over our system named by HET PS (ASP), HET AR (AllReduce) and HET Hybrid (w/o Cache). HET PS follows the ASP algorithm in TensorFlow and each worker pushes its updates to the server without waiting for the others. HET Hybrid keeps the hybrid communication architecture in HET but removes the cache embedding table. All of these three baselines are sharing the same computation kernels and communication optimizations (i.e., overlapping, pre-fetching) as HET (denoted by HET Hybrid Cache or HET Cache in the following).

Datasets and models. We select two categories of representative embedding model workloads, including the deep learning recommendation model (DLRM) and the graph neural network



(a) 1 Gb cluster

(b) 10 Gb cluster

Figure 7: Per epoch time and the speedup of communication time on DLRM tasks.

(GNN). We use three industrial DLRM models consisting of Wide & Deep (WDL) [9], DeepFM (DFM) [15] and Deep & Cross (DCN) [38]. They are evaluated on a popular recommendation dataset, Criteo [1], which is also the largest standard benchmark in MLPerf [2]. There could be more than *one trillion model parameters* (i.e., 10^{12} floats) from the embedding table when we set $D = 4096$ on Criteo.

The second category is GNN model and we select the most popular GraphSAGE [17] algorithm on large graphs. We evaluate on node classification tasks and adopt several graph datasets with different scale: Reddit[16], Amazon[10] and ogbn-mag[19]. More details about the datasets and models are in the Appendix B.

Experimental setting. We implement all of these models in TensorFlow 1.15 and select SGD optimizer with the batch size of 128; the learning rate is selected from [0.001, 0.01, 0.1] by grid search. We have two GPU clusters for our evaluation. In cluster A, each node is equipped with an Nvidia RTX TITAN 24 GB card supporting PCIe 3.0 and 12 GB DRAM for cache (we temporarily improve the DRAM size to 48 GB in the last model scalability experiment). While the servers are in a CPU cluster and each node has two Intel Xeon Gold 5120 CPUs and 376 GB DRAM. The servers and workers clusters are connected by a 1 Gbit Ethernet. Cluster B has similar configurations, but the GPU is replaced by Nvidia V100 and the network is improved to 10 Gbit Ethernet. The testing AUC thresholds of convergence are set to be around 80% for the Criteo dataset, as reported in [9]. For the GNN datasets, due to the customized feature engineering step (e.g., introducing sparse features), we manually set the termination point by predefined values. All experiments are executed five times, and the averaged results are reported.

5.1 End-to-end Comparison

In this section, we first provide end-to-end comparison experiments with the baselines. These experiments are evaluated on 8 workers and 1 remote server. The size of the cache embedding table is set to be 10% size of the global cache embedding table (listed in Figure 2). We set $D = 128$ in the following experiments and tune it in Sec. 5.3.

Table 1: End-to-end convergence efficiency comparison.

Convergence time (h)	TF Parallax	HET Hybrid	HET Cache $s = 10$	HET Cache $s = 100$
WDL-Criteo	56.402 (×10.86)	26.668 (×5.14)	9.938 (×1.91)	**5.193**
DFM-Criteo	27.529 (×9.24)	13.273 (×4.46)	5.314 (×1.78)	**2.978**
DCN-Criteo	99.023 (×11.29)	42.296 (×4.82)	17.341 (×1.98)	**8.770**
GNN-Amazon	8.667 (×20.68)	1.972 (×4.71)	0.583 (×1.39)	**0.419**
GNN-Reddit	0.752 (×6.37)	0.514 (×4.36)	0.123 (×1.04)	**0.118**
GNN-ogbn-mag	3.869 (×16.39)	1.040 (×4.41)	0.271 (×1.15)	**0.236**

Convergence efficiency. Figure 6 shows the convergence curves on six different workloads on cluster A. TF PS and HET PS follow the ASP algorithm and cannot converge to the target thresholds in these workloads. We provide HET with different staleness thresholds $s = 10$ and 100. As we can see, our system always outperforms the other baselines on all tasks. When $s = 100$, benefited from our cache embedding table mechanism, we could achieve around 4.36-5.14× speed up compared to HET Hybrid. Compared to $s = 10$, it is natural that using a larger s could alleviate more communication costs. The variances regions are quite negligible, which verifies the convergence stability of our methods. Table 1 illustrates the end-to-end convergence time and our HET achieves 6.37-20.68× speedup compared to TF Parallax. PS-based ASP methods are not listed because they cannot achieve the convergence thresholds. Note that, in our system implementation, we make a unique operation for the keys before the embedding communication operations to avoid redundant embedding transferring costs. However, the GNN-Reddit workload is quite special. It only has node-id embeddings and all embeddings in a mini-batch of data samples are always unique naturally. Therefore, in this case, the costs of unique operation outweigh the benefits, making HET PS ASP slower than TF PS ASP.

Communication speedup. We also compare the per epoch time on DLRM tasks and provide the following findings. First, through comparative analysis on the per epoch time in Figure 7(a) and the learning curves in Figure 6, we find that HET PS and TF PS follow the same statistical efficiency [21] in Figure 6 (HET Hybrid and TF Parallax are also the same). Their different convergence speeds come from the backbone optimizations, which verify the correctness of our implementation. Second, PS-based methods often show poor performance compared to hybrid-based methods due to the dense communication. The phenomenon in WDL is not significant because it has fewer dense model parameters than DFM and DCN. Third, these results in Figures 6 and 7(a) also imply that existing PS and hybrid communication techniques cannot fundamentally solve the communication bottleneck in large-scale embedding model training. Fortunately, due to the fine-grained caching and consistency, our proposed HET achieves significant performance improvement and up to 88% ($\approx 1 - 1/8.2$) embedding communication reduction. Figure 7(b) shows the per epoch time and communication time speedup comparison on three DLRM tasks under 10 Gbit Ethernet cluster. As shown in Figure 7(b), although the speedups are smaller than those in the 1 Gbit Ethernet cluster due to the higher network bandwidth, the communication costs are still the bottleneck of the model training process. We see that our system still outperforms these baselines and achieves up to 2.3× and 5.2× speedup compared to HET Hybrid and TF Parallax respectively. Another interesting finding is that HET AR on

Table 2: Final test AUC (%) with different s on Criteo

Models	textbfs = 0	s = 100	s = 10k	s = ∞	Cache miss rate (WDL)	s = 0	s = 100
WDL	79.64	79.62	78.84	74.61			
DFM	79.74	79.73	78.96	70.91	0%	80.17%	80.15%
DCN	80.25	80.24	79.76	75.29	>0%	78.17%	78.12%

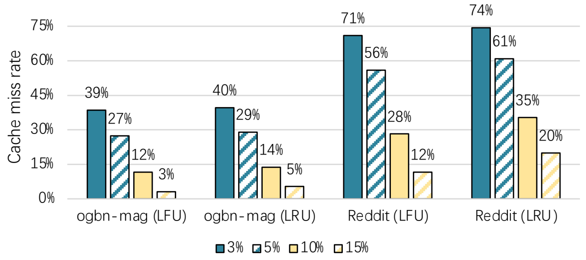


Figure 8: Cache miss rate under different cache space and strategy settings on GNN tasks.

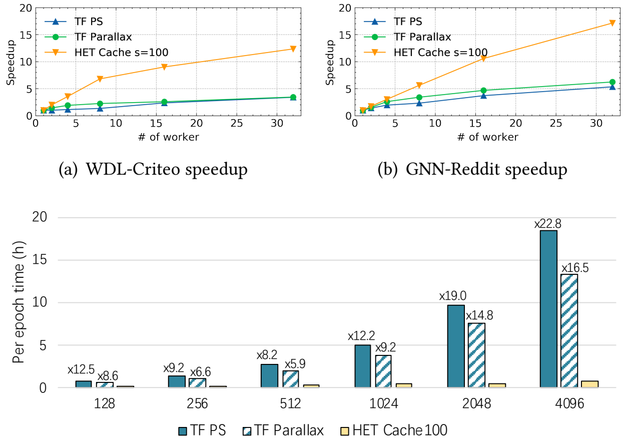
1 Gbit Ethernet cluster performs better than HET PS due to the utilization of the PCIe bandwidth cross GPUs. Although AllReduce degenerates to the inefficient AllGather primitive for sparse communication, it still achieves similar performance as HET Hybrid because these baselines involving the PS are suffering from the limited network bandwidth between servers and workers. When we use a 10 Gbit Ethernet, the high network bandwidth significantly improves the speed of PS-based methods. But HET AR maintains similar performance and becomes the slowest among all methods.

Convergence quality. We first investigate how much negative impact could the staleness have on the accuracy. We illustrate the convergent model performance (i.e., test AUC) with different staleness thresholds in Table 2 (left part). Due to the inherent robustness of iterative convergent algorithms, we find that our method reaches the target model quality even under moderate levels of staleness (s = 100), although the model degradation becomes apparent under high staleness levels.

We next investigate pathological cases (e.g. is it not possible that prediction with the embedding parameters that are less frequently synchronized may result in less accurate results (i.e., bias)?) We make a further study on the test dataset of Criteo on WDL. As a cache hit (miss) implies the prediction uses stale (up-to-date) embedding parameters, we use cache miss rate to measure the frequency of the prediction using the stale (less frequently synchronized) embedding parameters. Therefore, we split the test set into two sets based on the cache miss rate. As shown in Table 2 (right part), the predictions distribution from two models (s = 0 and s = 100) are very close, which demonstrates that the stale embeddings will not incur significant predication bias.

5.2 System Configuration Sensitivity

We study the impact of different cache embedding table sizes and cache strategy settings. We measure the cache miss rate on GNN task with ogbn-mag and Reddit datasets on cluster A. As shown in Figure 8, LFU often performs a lower cache miss rate than LRU. This is because LFU could reflect the long-term embedding access popularity better. We also evaluate different cache embedding table sizes, including 3%, 5%, 10% and 15%. As the cache table size growing, the cache miss rate significantly decreases. For ogbn-mag with LFU, given a piece of cache space whose size equals 15% of the global embedding table size, almost 97% embedding accesses are performed on the cache embedding table. This experiment strongly verifies the effectiveness of HET and explains how does our cache embedding table help to reduce the communication costs.



(a) WDL-Criteo speedup

(b) GNN-Reddit speedup

(c) WDL-Criteo with different embedding hidden size D

Figure 9: Scalability study

5.3 Scalability

We conduct a scalability study in terms of run time speedup on cluster A over 4 servers with 1, 2, 4, 8, 16, and 32 workers respectively. As shown in Figure 9(a) and Figure 9(b), both TF PS and TF Parallax have limited scalability suffering from large amounts of embedding communication costs. By contrast, our HET achieves improved scalability by caching the hot embeddings. We also note that all methods show better scalability on GNN-Reddit than WDL-Criteo. Because the latter has a larger embedding table and becomes more communication-intensive and more difficult to scale up. We also study the model scalability of HET on WDL-Criteo by increasing the embedding size D up to 4096 (around *one trillion model parameters*) over 32 workers. Figure 9(c) shows that HET significantly outperforms TF and Parallax since their PS architecture faces a more serious communication bottleneck with such a large scale embedding table.

6 CONCLUSION

Embedding model trained on high-dimensional data is common at modern web companies and poses an extra challenge to standard frameworks: the high communication overhead causes the embedding workloads to have low execution efficiency and scalability. To address this performance bottleneck, we presented HET, a system framework leveraging the embedding cache architecture combined with fine-grained consistency and stale-write protocols. Experimental results have shown that HET could reduce up to 88% embedding communication and achieve up to 20.68× performance improvements, compared to the state-of-the-art baselines. We hope that this work and the open-source release of HET helps motivate the release of larger high-dimension datasets from modern web companies and the increase of research on larger embedding models.

REFERENCES

[1] 2014. Criteo Kaggle Ad. https://www.kaggle.com/c/criteo-display-ad-challenge.

[2] 2020. MLPerf Benchmark. https://mlperf.org.

[3] 2021. NVIDIA collective communications library (NCCL). https://developer.nvidia.com/nccl.

[4] 2021. NVIDIA HugeCTR. https://github.com/NVIDIA/HugeCTR.

[5] 2021. PS-Lite, a lightweight parameter server interface. https://github.com/dmlc/ps-lite.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.

[7] Gediminas Adomavicius and Jingjing Zhang. 2012. Impact of data characteristics on recommender systems performance. *ACM Trans. Manag. Inf. Syst.* 3, 1 (2012), 3:1–3:17. https://doi.org/10.1145/2151163.2151166

[8] Avishek Anand, Megha Khosla, Jaspreet Singh, Jan-Hendrik Zab, and Zijian Zhang. 2019. Asynchronous Training of Word Embeddings for Large Text Corpora. In *WSDM*. 168–176.

[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *DLRS@RecSys*. 7–10.

[10] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *SIGKDD*.

[11] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys*. 191–198.

[12] Andrew M. Dai, Christopher Olah, and Quoc V. Le. 2015. Document Embedding with Paragraph Vectors. *CoRR* abs/1507.07998 (2015).

[13] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA.

[14] Saeed Ghadimi, Guanghui Lan, and Hongchao Zhang. 2016. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. *Math. Program.* 155, 1-2 (2016), 267–305.

[15] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *IJCAI*. 1725–1731.

[16] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216* (2017).

[17] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*. 1024–1034.

[18] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NeurIPS*. 1223–1231.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *NeurIPS*.

[20] Yaochen Hu, Di Niu, Jianming Yang, and Shengping Zhou. 2019. FDML: A Collaborative Machine Learning Framework for Distributed Features. In *SIGKDD*. 2232–2240.

[21] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *SIGMOD*. 463–478.

[22] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H. Chi. 2020. Learning Multi-granular Quantized Embeddings for Large-Vocab Categorical Features in Recommender Systems. In *Companion of The 2020 Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Amal El Fallah Seghrouchni, Gita Sukthankar, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 562–566. https://doi.org/10.1145/3366424.3383416

[23] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues (Eds.). ACM, 5:1–5:16. https://doi.org/10.1145/2901318.2901331

[24] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *EuroSys*. 43:1–43:15.

[25] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[26] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*. 583–598.

[27] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *PVLDB* 13, 12 (2020), 3005–3018.

[28] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *SIGKDD*. 1754–1763.

[29] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *NeurIPS*. 2737–2745.

[30] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *ICML*, Vol. 80. 3049–3058.

[31] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727.

[32] X. Miao, L. Ma, Z. Yang, Y. Shao, B. Cui, L. Yu, and J. Jiang. 2020. CuWide: Towards Efficient Flow-based Training for Sparse Wide Models on GPUs. *TKDE* (2020), 1–1. https://doi.org/10.1109/TKDE.2020.3038109

[33] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.

[34] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019).

[35] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *SOSP*. 16–29.

[36] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *SIGKDD*. 701–710.

[37] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018).

[38] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *ADKDD*. 12:1–12:7.

[39] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. SIGIR. 165–174.

[40] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *SC*. 21.

[41] Eric P Xing, Qirong Ho, Pengtao Xie, and Dai Wei. 2016. Strategies and principles of distributed machine learning on big data. *Engineering* (2016), 179–195.

[42] Hao Yu, Sen Yang, and Shenghuo Zhu. 2019. Parallel Restarted SGD with Faster Convergence and Less Communication: Demystifying Why Model Averaging Works for Deep Learning. In *AAAI*. 5693–5700.

[43] Lele Yu, Bin Cui, Ce Zhang, and Yingxia Shao. 2017. LDA*: A Robust and Large-scale Topic Modeling System. *Proc. VLDB Endow.* 10, 11 (2017), 1406–1417. https://doi.org/10.14778/3137628.3137649

[44] Junqi Zhang, Bing Bai, Ye Lin, Jian Liang, Kun Bai, and Fei Wang. 2020. General-Purpose User Embeddings based on Mobile App Usage. In *SIGKDD*. 2831–2840.

[45] Jia-Dong Zhang and Chi-Yin Chow. 2015. GeoSoCa: Exploiting Geographical, Social and Categorical Correlations for Point-of-Interest Recommendations. In *SIGIR*. 443–452.

[46] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, and Xupeng Miao. 2019. PS2: Parameter Server on Spark. In *SIGMOD*. 376–388.

[47] Qian Zhao, Jilin Chen, Minmin Chen, Sagar Jain, Alex Beutel, Francois Belletti, and Ed H. Chi. 2018. Categorical-attributes-based item classification for recommender systems. In *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*, Sole Pera, Michael D. Ekstrand, Xavier Amatriain, and John O'Donovan (Eds.). ACM, 320–328. https://doi.org/10.1145/3240323.3240367

[48] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *MLSys*.

[49] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *SIGKDD*. 1059–1068.

A PROOFS

A.1 Assumptions

Before our analysis, we make the following commonly used assumptions:

Assumption 1. *We claim the following standard properties:*

- ***Lower Bound**:* $\forall \mathbf{x} \in \mathbb{R}^d, f(\mathbf{x}) \ge f_{inf}$.
- ***Lipschitz continuity**:* $\forall$ $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$,
$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \le L\|\mathbf{x}-\mathbf{y}\|.$$

Assumption 2. *For the the j-th worker, $\forall \mathbf{x}^j \in \mathbb{R}^d$, the gradient of i-th model component is $G_i^j(\mathbf{x}^j)$, which satisfies:*

- ***Unbiased estimation**:* $\mathbb{E}_\xi[G_i^j(\mathbf{x}^j)] = \nabla_i f(\mathbf{x}^j)$,
- ***Bounded variance**:* $\mathbb{E}_\xi[\|G_i^j(\mathbf{x}^j)\|^2] \le B^2$.

We remark that both the above assumptions have been used in many related studies [20, 29, 30, 42] as well. Mathematically, the convergence property is formulated with Theorem 1.

A.2 Explanations on Per-embedding Clock Bounded Consistency

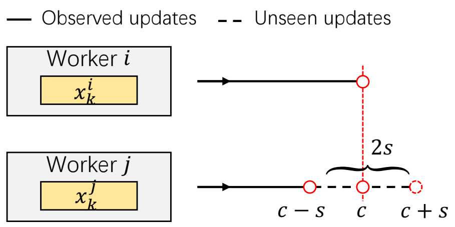


Figure 10: Illustration of per-embedding clock bounded consistency. Red circles represent cache synchronization.

Figure 10 illustrates a running example to explain the per-embedding clock bounded consistency guarantees. Considering for an embedding $\mathbf{x}_k$ at a certain iteration, $\mathbf{x}_k^i$ just finished the cache synchronization with server. We suppose the worst case that there exists $\mathbf{x}_k^j$ on worker j at clock c, which also synchronizes its updates on the embedding immediately after i's synchronization. Since the stale-write mechanism, worker i will not see the accumulated updates from j at clock c. But the updates before $c-s$ could be observed because they have already been written to the global embedding table and could be fetched by worker i during the synchronization. After clock c, due to the cache valid conditions, worker j could continue go forward at most s clocks (i.e., s updates) before i's next synchronization. Therefore, worker i could observe all updates from worker j older than $2s$ clocks, i.e., those in the range of $[0, \mathbf{x}_k^j.c_c - 2s]$.

Then we revisit the relationship between the embedding clocks and the iteration number t. Considering the case of a single worker, in our design, the embedding clock $\mathbf{x}_i.c_c$ represents the number of updates on $\mathbf{x}_i$. For dense models, it equals to the iteration number t. However, different embeddings in a model could have different popularity. Therefore, to bridge the gap between the clocks and t, we need to formally characterize the embedding access pattern. Suppose the input data are distributed to each worker in a unbiased manner, we define a *per-embedding existing probability p_i* to represent the probability of embedding x_i existing in each mini-batch of input data. Therefore, we can infer the following lemma to motivate our following proofs:

Lemma 2. *Given embedding per-embedding existing probability p_i for x_i, it needs around s/p_i iterations to perform s updates.*

Note that, our following results could also be proved similarly even on biased data distribution based methods by characterize the embedding existing probabilities on different workers.

A.3 Proof of Theorem 1

Our key tool for the convergence analysis is to define a reference model sequence $\mathbf{x}(t)$ without staleness, for each embedding $\mathbf{x}_i(t)$ in $\mathbf{x}(t)$, suppose the existing probability of $\mathbf{x}_i$ is p_i, we have:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) - \frac{\eta}{N}\sum_{n=0}^{N-1} G_i^n(\mathbf{x}^n(t))p_i. \tag{4}$$

Intuitively, $\mathbf{x}(t)$ would like to follow the "clean" SGD iteration, by including all the gradients generated by the end of round t. Without loss of generality, $\mathbf{x}(0) = \mathbf{x}^j(0)$ for any worker $0 \le j \le N$. Then we will prove that the "true" sequence is not too far from the models on the workers:

Lemma 3. *For any $t \ge 0$ and worker j, we have:*
$$\mathbb{E}[\|\mathbf{x}(t) - \mathbf{x}^j(t)\|^2] \le 4s^2\eta^2M^2B^2.$$

Proof. Based on the Cauchy-Schwarz inequality, we have:

$$\|\mathbf{x}(t) - \mathbf{x}^j(t)\|^2 \le M\sum_{i=1}^{M}\|\mathbf{x}_i(t) - \mathbf{x}_i^j(t)\|^2. \tag{5}$$

Then for each embedding $\mathbf{x}_i$, due to Lemma 2 we have:

$$\begin{aligned}
\|\mathbf{x}_i(t) - \mathbf{x}_i^j(t)\|^2 &\le \|\sum_{k=t-\frac{2s}{p_i}}^{t}\eta\sum_{n=1}^{N} G_i^n(\mathbf{x}^n(k))p_i/N\|^2 && (6)\\
&\le (\eta p_i/N)^2\|\sum_{k=t-\frac{2s}{p_i}}^{t}\sum_{n=1}^{N} G_i^n(\mathbf{x}^n(k))\|^2 && (7)\\
&\le (\eta p_i/N)^2\frac{2s}{p_i}N\sum_{k=t-\frac{2s}{p_i}}^{t}\sum_{n=1}^{N}\|G_i^n(\mathbf{x}^n(k))\|^2 && (8)\\
&\le 4s^2\eta^2B^2, && (9)
\end{aligned}$$

where we have used the properties stated in Lemma 1 (in particular the Staleness Bound), and the triangle inequality. Next, we notice that the expected squared norm of each of the missing gradients is bounded by B^2 (by the second moment bound). Combing Eq. (9) with Eq. (5) finally implies the claimed inequality.

□

Proofs on Theorem 1.

Proof. We begin from the definition of $\mathbf{x}(t)$ in Eq.(4). We will first prove the above statement for the iterate $\mathbf{x}(t)$, and then will extend the proof for $\mathbf{x}^j(t)$. For simplicity, let us denote $G(t) = \sum_{n=0}^{N-1} G^n(\mathbf{x}^n(t))$. We can use the Taylor expansion of $f(\mathbf{x}(t+1))$ around $\mathbf{x}(t)$ and the smoothness condition to obtain the following inequality that $f(\mathbf{x}(t+1))$

$$\le f(\mathbf{x}(t)) + (\mathbf{x}(t+1) - \mathbf{x}(t))^T \nabla f(\mathbf{x}(t)) + \frac{L}{2}\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2$$

$$= f(\mathbf{x}(t)) - \eta \nabla f(\mathbf{x}(t))^T \nabla f(\mathbf{x}(t)) + \frac{\eta^2 L}{2N^2}\|G(t)\|^2 +$$

$$+ \eta(\nabla f(\mathbf{x}(t)) - G(t)/N)^T \nabla f(\mathbf{x}(t)).$$

We can therefore apply the expectation with respect to the random sampling at step t, the second moment bound assumption:

$$\begin{aligned}\mathbb{E}\left[f(\mathbf{x}(t+1))|\mathbf{x}(t)\right] \le & f(\mathbf{x}(t)) - \eta\|\nabla f(\mathbf{x}(t))\|^2 + \frac{\eta^2 L}{2} M^2 B^2 \\ & + \eta(\nabla f(\mathbf{x}(t)) - \nabla f(\mathbf{x}^j(t)))^T \nabla f(\mathbf{x}(t)).\end{aligned}$$

To bound the last term, we can now apply the Cauchy-Schwarz inequality and the fact that the gradients are L-Lipschitz:

$$\begin{aligned}\mathbb{E}\left[f(\mathbf{x}(t+1))|\mathbf{x}(t)\right] \le & f(\mathbf{x}(t)) - \eta\|\nabla f(\mathbf{x}(t))\|^2 + \frac{\eta^2 L}{2} M^2 B^2 \\ & + \eta L\|\mathbf{x}(t) - \mathbf{x}^j(t)\|\|\nabla f(\mathbf{x}(t))\|.\end{aligned}$$

To further bound the last term, we can apply the classic inequality $a^2 + b^2 \ge 2ab$ together with Lemma 3 to obtain:

$$\begin{aligned}\mathbb{E}\left[f(\mathbf{x}(t+1))|\mathbf{x}(t)\right] \le & f(\mathbf{x}(t)) - \eta\|\nabla f(\mathbf{x}(t))\|^2 + \frac{\eta^2 L}{2} M^2 B^2 \\ & + \eta\|\nabla f(\mathbf{x}(t))\|^2/2 + 2s^2\eta^3 L^2 M^2 B^2.\end{aligned}$$

Rearranging terms and taking total expectation:

$$\begin{aligned}\mathbb{E}\left[\|\nabla f(\mathbf{x}(t))\|^2\right] \le & \frac{2\mathbb{E}\left[f(\mathbf{x}(t)) - f(\mathbf{x}(t+1))\right]}{\eta} + \eta M^2 B^2 L \\ & + 4s^2\eta^2 L^2 M^2 B^2.\end{aligned}$$

Summing across all t and dividing by T, we get:

$$\begin{aligned}\min_{1\le t\le T} \mathbb{E}\left[\|\nabla f(\mathbf{x}(t))\|^2\right] & \le \frac{1}{T}\sum_t \mathbb{E}\left[\|\nabla f(\mathbf{x}(t))\|^2\right] \\ \le & \frac{2\left(f(\mathbf{x}(0)) - f_{\text{inf}}\right)}{\eta T} + \eta M^2 B^2 L + 4s^2\eta^2 L^2 M^2 B^2. \quad (10)\end{aligned}$$

We now study the set of conditions for each of the three RHS terms to be less than $\epsilon/12$. We have that it is sufficient for the following three conditions to hold:

(1) $T \ge \frac{24(f(x(0) - f_{\text{inf}})}{\eta\epsilon}$;

(2) $\eta \le \frac{\epsilon}{12M^2B^2L}$;

(3) $\eta \le \frac{\sqrt{\epsilon}}{4\sqrt{3}sLMB}$.

All these conditions hold by assumption from the theorem statement. We have therefore obtained that there exists $t^\star$ such that $\|\nabla f(\mathbf{x}(t^\star))\|^2 \le \epsilon/4$. However, by smoothness and Lemma 3 we know that

$$\mathbb{E}\|\nabla f(\mathbf{x}(t^\star)) - \nabla f(\mathbf{x}^j(t^\star))\|^2 \le 4Ls^2\eta^2 M^2 B^2.$$

To prove our final target, we need to prove

$$\mathbb{E}\|\nabla f(\mathbf{x}(t^\star)) - \nabla f(\mathbf{x}^j(t^\star))\|^2 \le 4Ls^2\eta^2 M^2 B^2 \le \epsilon/4.$$

Based on this inequality, we have the second bound of the learning rate:

$$\eta \le \frac{\sqrt{\epsilon}}{4\sqrt{L}sMB},$$

which has been clarified as an assumption in the theorem statement on the upper bound on η.

Finally, we can apply the classic inequality $\|a+b\|^2 \le 2(\|a\|^2 + \|b\|^2)$ to obtain that

$$\mathbb{E}\|\nabla f(\mathbf{x}^j(t^\star))\|^2 \le \epsilon.$$

□

Corollary 1. *Under Theorem 1, with a proper learning rate $\eta = \frac{1}{L\sqrt{T}}$, we have:*

$$\frac{1}{T}\sum_t \mathbb{E}\left[\|\nabla f(\mathbf{x}(t))\|^2\right] \le \frac{2L\left(f(\mathbf{x}(0)) - f_{inf}\right) + M^2B^2}{\sqrt{T}} + \frac{4s^2M^2B^2}{T}. \quad (11)$$

We make the following observations regarding the bound. First, we prove that, for a given sequence of learning rates, the algorithm will converge to a point of negligible gradient. Corollary 1 implies that when the learning rate is $\eta = 1/(L\sqrt{T})$ and T is sufficiently large, we derive that the convergence rate is around $O(1/\sqrt{T})$, which is consistent with that of BSP. Second, we note that first RHS term in Eq. (11) is the standard SGD error and the third term comes from the staleness. This suggests to trade off the additional performance cost of synchronization with the slower convergence due to delayed gradient information in practice.

Table 3: Model Architectures

Model	Layer#	Hidden	Embedding Size	Dense Parameter
WDL	3	256	128	137.98K
DFM	3	256	128	917.77K
DCN	3	256	128	1.01M
GraphSage	2	128	128	2.40k-19.48k

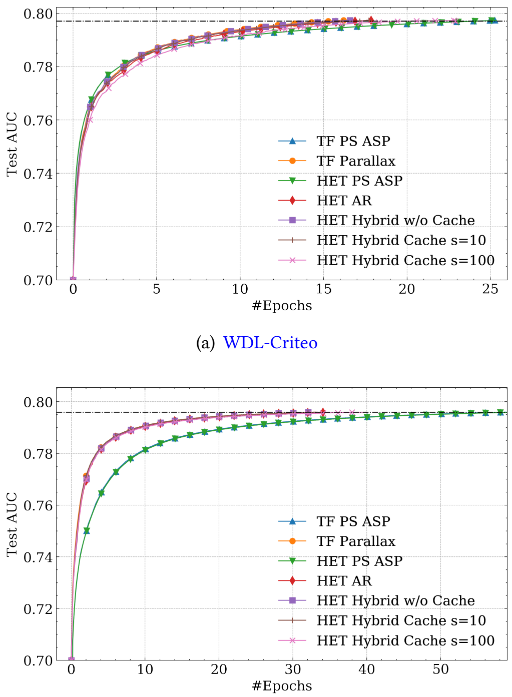
Table 4: Statistics of Datasets

Datasets	Embedding#	Sample#	Feature
Criteo	33.76M	45.84M	39
Amazon	2.79M	2.45M	17
Reddit	233.96K	232.96K	1
ogbn-mag	1.93M	736.39K	21

B MODELS AND DATASETS

We introduce more details about the model configurations in Table 3, and also some statistics of datasets in Table 4. In order to elucidate the effectiveness of our embedding learning system, we replace dense features in Amazon, Reddit and ogbn-mag with a trainable node-id embedding. In addition, we include each product's word of bag feature as sparse input in Amazon. And in ogbn-mag, we include each paper's author and field of interest as additional sparse features. We use the train/eval split provided in the original papers of these datasets. Note that, the embedding size is set to be 128 for almost of our experiments except for the last model scalability in Figure 9(c). The largest model in our experiments contains more than one trillion embedding parameters (i.e., 33.76M×4096).

C EXPERIMENT RESULTS



(a) WDL-Criteo

(b) DFM-Criteo

Figure 11: Convergence performance comparison.

We illustrate the convergence performance on WDL-Criteo and DFM-Criteo with the number of epochs as the x-axis. As shown in Figure 11, TF PS and HET PS have the same statistical efficiency and perform worse than the other methods due to asynchronous updates. But our proposed methods perform very close to the fully synchronous baselines including TF Parallax and HET Hybrid w/o Cache.