Accelerating Neural Recommendation Training with Embedding Scheduling

Chaoliang Zeng*, Xudong Liao*, Xiaodian Cheng, Han Tian, Xinchen Wan, Hao Wang, Kai Chen
iSING Lab, Hong Kong University of Science and Technology

Abstract

Deep learning recommendation models (DLRM) are extensively adopted to support many online services. Typical DLRM training frameworks adopt the parameter server (PS) in CPU servers to maintain memory-intensive embedding tables, and leverage GPU workers with embedding cache to accelerate compute-intensive neural network computation and enable fast embedding lookups. However, such distributed systems suffer from significant communication overhead caused by the embedding transmissions between workers and PS. Prior work reduces the number of cache embedding transmissions by compromising model accuracy, including oversampling hot embeddings or applying staleness-tolerant updates.

This paper reveals that many of such transmissions can be avoided given the predictability and infrequency natures of in-cache embedding accesses in distributed training. Based on this observation, we explore a new direction to accelerate distributed DLRM training without compromising model accuracy, i.e., embedding scheduling—with the core idea of proactively determining *"where embeddings should be trained"* and *"which embeddings should be synchronized"* to increase the cache hit rate and decrease unnecessary updates, thus achieving a low communication overhead. To realize this idea, we design Herald, a real-time embedding scheduler consisting of two main components: an adaptive location-aware inputs allocator to determine where embeddings should be trained and an optimal communication plan generator to determine which embeddings should be synchronized. Our experiments with real-world workloads show that Herald reduces 48%-89% embedding transmissions, leading up to 2.11× and up to 1.61× better performance with TCP and RDMA, respectively, over 100 Gbps Ethernet for end-to-end DLRM training.

1 Introduction

Deep learning-based recommendation systems have been extensively applied to a wide range of online services [9, 53], consuming significant infrastructure capacity and compute cycles across production datacenters [2]. Training a deep learning recommendation model (DLRM) poses challenges in both memory and computation. A typical DLRM (§2.1) consists of (1) *embedding tables*, which are large lookup tables that store millions to billions of semantic embedding vectors (embeddings for short) and consume a tremendous memory footprint (a few KB per embedding [29] and up to tens of GBs to TBs in total [30, 47, 52–54]), and (2) *multi-layer perceptron (MLP)*, which makes up the dense model and contributes to most of the computation cycles. These hybrid requirements make it challenging to train a DLRM efficiently. Specifically, hardware accelerators notably GPUs, which are widely incorporated into deep learning training, can execute high-performance dense model computation, but fail to provide large enough memory capacity to fully support the embedding tables component.

To address this dilemma, a common practice [2, 21, 29, 38] separates dense model computations and embedding table operations (lookup and update) in DLRM training. It typically adopts the parameter server (PS) architecture [26] to maintain globally shared embeddings in memory-optimized and cost-effective CPU servers while leveraging GPUs to accelerate the dense model computation with data parallelism. GPU workers will further cache a few hot embeddings in their local memory to accelerate the embedding lookup operation. However, a single training sample may involve up to thousands of embeddings in production workloads [2]. A naive cache-enable system still suffers from significant communication overhead for embedding transmissions[1] in bulk synchronous parallel (BSP) [12] training (§2.2), due to stale cached-embedding updates and embedding gradient synchronizations.

Prior efforts [3, 31] mitigate this communication overhead by *reducing the number of embedding transmissions between workers and PS*. They either oversample training data containing only hot embeddings with fast inter-server collective communication [3] or apply a staleness-tolerant embedding update manner [31]. Both approaches deviate from the vanilla BSP training and do not provide any theoretical guarantee of model accuracy. However, a high and stable model accuracy is critical in production. For example, even an order of 0.1% accuracy loss is intolerable in Meta recommendations [2]. Therefore, DLRM is more favorable to BSP training without a bias on training embeddings [34, 38].

In this paper, we explore a new direction, i.e., *embedding scheduling* (§3), to accelerate distributed DLRM training without compromising model accuracy. The core idea is to determine (1) where embeddings should be trained by distributing

* Equal contribution.

[1] In this paper, we refer to both the transmission of embedding value and the transmission of embedding gradient as embedding transmission.

batch training samples to likely cache-hitting[2] workers and (2) which embeddings should be synchronized by identifying embeddings to be trained in upcoming iterations for a low communication overhead. These optimization opportunities come from two crucial characteristics of in-cache embedding accesses during the DLRM training.

- **Predictability:** since the embeddings required by training samples and the current cache snapshot of each worker are visible before the computation, the upcoming cache accesses and their results (hit or not) are predictable under a partition of batch samples.
- **Infrequence:** the physical accesses to most in-cache embeddings are infrequent enough that the popularity of an embedding may be less than the total number of training samples assigned to a worker in the entire training process.

These two characteristics imply that with proactive embedding scheduling during the input generation, there exists a potential to train each infrequently accessed embedding in a fixed worker, respectively, which can effectively increase the cache hit rate and avoid unnecessary synchronizations for consistency (as this embedding is not required by other workers). Furthermore, our theoretical analysis proves that accelerating DLRM training with embedding scheduling can preserve the model consistency and hence the exactly same model accuracy as vanilla synchronous training.

When considering the whole DLRM training data flow, it is important to make the embedding scheduling decision in real time for two main reasons. First, production recommendation systems require online training from the streaming data for high-velocity inference queries [21, 29, 38]. Second, even for offline training, preparing scheduling decisions for the entire training process in advance is difficult, if not impossible, given the huge scheduling space and the unpredictable number of training iterations (and thus the number of scheduling decisions), which depends on the real-time evaluation result of the model. As a result, for offline training, we can only prepare the scheduling result for the early iterations, and we still need real-time scheduling for the rest training iterations.

Nevertheless, designing a real-time embedding scheduler is non-trivial. The *scheduling budget*, i.e., the available scheduling time, to conduct a scheduling decision for an iteration is limited, as the scheduling overhead should not be larger than the training time of an iteration to prevent introducing additional overhead when overlapping training and scheduling. Moreover, the scheduling budget varies among different workloads and training settings, so the embedding scheduler must be adaptive to different training tasks.

To this end, we propose Herald[3] (§4), a real-time embedding scheduler that leverages information including the required embeddings of input samples and the locations of those embeddings, to reduce embedding transmissions and thus achieve efficient DLRM training. Herald addresses the above challenges with a decoupling idea. Specifically, it determines *"where embeddings should be trained"* via an adaptive location-aware inputs allocation (LAIA) algorithm. The adaptive LAIA algorithm leverages the diverse ratios of infrequently-accessed embeddings among embedding tables, and conducts partition decisions based on a subset of selective tables for the batch inputs partition. Therefore, it effectively meets various scheduling budgets with a high cache hit rate. Herald further identifies *"which embeddings should be synchronized"* for this iteration in conjunction with the batch partition result of the next iteration by enabling sample prefetching. Herald figures out a minimal list of embeddings that should be synchronized to minimize gradient transmissions via an optimal communication plan generator.

We have implemented Herald on top of HET [31] (§5) and evaluated it via extensive simulations and a small-scale testbed with 8 Nvidia 3090 GPU workers on typical real-world workloads (§6). Evaluation results show that compared with HET, Herald reduces 48%-89% embedding transmissions. As a result, Herald delivers up to 2.11× and up to 1.61× better performance with TCP and RDMA, respectively, over 100 Gbps Ethernet for end-to-end DLRM training.

This paper makes the following contributions:

- We explore embedding scheduling as a new direction to accelerate DLRM training by leveraging two key characteristics of in-cache embedding accesses, i.e., predictability and infrequency (§3).
- We design Herald, a real-time embedding scheduler to meet various scheduling budgets while preserving a high scheduling quality (§4).
- We verify the benefit of embedding scheduling via testbed experiments on our Herald implementation (§5 and §6).

This work does not raise any ethical issues. Herald is available at `https://github.com/HKUST-SING/herald`.

2 Background & Motivation

2.1 Deep Learning Recommendation

A deep learning recommendation system models the recommendation decision as a problem to predict the probability of a specific event, e.g., the likelihood of a web viewer watching the recommended content (video or article). To fully exploit high-dimensional (categorical) features, the technique of representation learning is widely applied to project a category ID to a dense feature vector, also called *embedding*. This embedding is used as a representation of the category ID to apply in numerical computation for recommendation processing.

DLRM architecture. Figure 1 illustrates the high-level architecture of a DLRM with the example of apps recommendation. There are two primary components: *embedding tables* and

[2] In this paper, we refer to hitting the latest version of an embedding in the cache as a cache hit for short.

[3] We presented a preliminary idea of Herald in an earlier workshop paper [49].

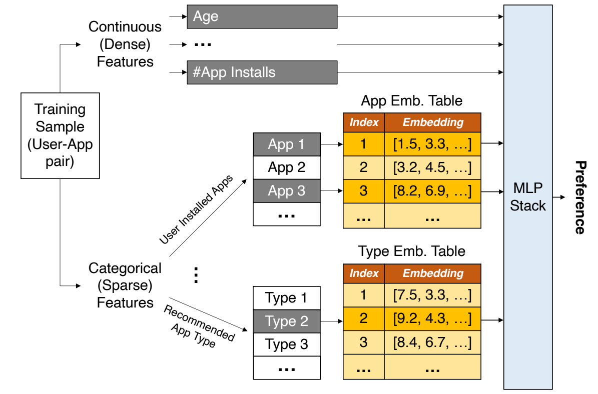


Figure 1: A high-level architecture of a typical DLRM with the example of apps recommendation.

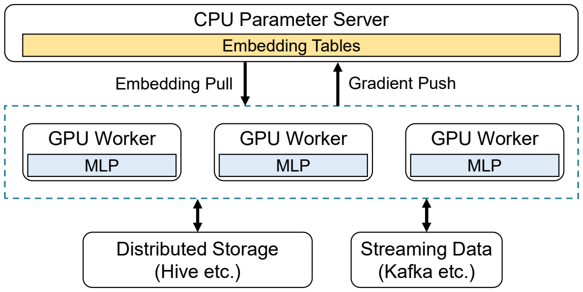


Figure 2: The data flow of DLRM training in production environments. We omit the data flow of MLP synchronization, which is not the focus of this paper.

multi-layer perceptron (MLP). The DLRM leverages embedding tables to project sparse features into dense representations, i.e., embeddings, by looking up every table with the corresponding category IDs as indexes. All these dense features, including embeddings and dense inputs, are processed by the MLP layer for the final prediction result. The value of embeddings, together with the parameters in MLP, will be iteratively updated during the training process.

Overview of distributed DLRM training systems. Figure 2 shows the data flow of the typical distributed DLRM training [2, 21, 29, 38]. DLRM training systems usually support different types of data I/O interfaces, e.g., Hive and Kafka, to provide good compatibility for various scenarios. Given the diverse requirements on computing and memory for MLP and embedding tables, respectively, DLRM training systems tend to separate the training of these two components. They train the MLP layer on compute-optimized GPU workers with data parallelism. To accommodate the scaling up of embedding table sizes (tens of GBs to TBs [30, 47, 52–54]), they maintain embeddings in memory-optimized CPU PS. This separation design introduces significant communication overhead when applying BSP for the non-degraded and reproducible model accuracy [5, 22, 34, 38]. In every training iteration, workers pull embeddings from PS on demand and push embedding gradients to PS at the end of this iteration.

	Model	Dataset
W1	Wide & Deep [7]	Criteo AD [10]
W2	Neural Collaborative Filtering [19]	MovieLens 25M [17]
W3	DeepFM [16]	Avazu [23]
W4	Deep & Cross [43]	Criteo Sponsored Search [40]

Table 1: Workloads in our case studies.

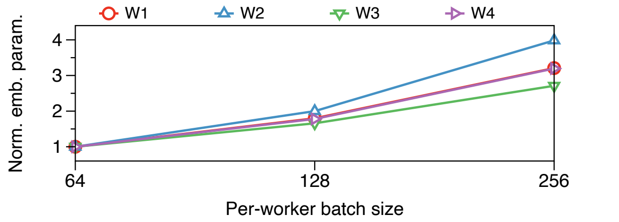


Figure 3: The number of embeddings used in an iteration increases with the per-worker batch size. The results are normalized to that number with a batch size of 64 for each workload.

Embedding cache. Given the skewed popularity distribution of embeddings, recent work [3, 31, 52] reduces embedding communications by exploiting an embedding cache in GPU workers. However, a naive cache-enable system provides limited performance improvement for BSP training, as it saves an embedding pull only when the queried embedding hits the cache (with the latest version) and does not save any embedding synchronizations. Therefore, they optimize the communication cost by compromising the BSP training procedure. For example, HET [31] tolerates a bounded staleness of each embedding by tracking the embedding version. Upon retrieving an embedding from the cache, it first compares the local version with the global version in PS and only pulls the embedding from PS if the difference between these two versions exceeds a threshold. A similar behavior happens for gradient synchronizations. HET still communicates with PS in every iteration for version checking, but these costs are much smaller than those of embedding transmissions.

2.2 Embedding Communication Matters

To quantify the contribution of embedding communication in the end-to-end DLRM training with BSP, we study typical DLRM workloads as listed in Table 1.

In the DLRM training, we find that a larger batch size (the number of training samples in an iteration) may lead to more embedding transmissions. The reason is that different samples may involve different sparse features and thus operate on different embeddings. Therefore, the number of operated distinct embeddings in an iteration grows with a larger group of samples (a larger batch size). Figure 3 demonstrates that the number of distinct embeddings used in an iteration can increase 2.7×-4.0× when the per-worker batch size increases from 64 to 256. However, the number of synchronized parameters in the MLP layer only depends on the MLP architecture, and thus does not change with the batch size.

We evaluate the distributed training efficiency of HET [31], the state-of-the-art cache-enabled embedding model training framework. The evaluation configuration follows the default setting as described in §6.1. We disable the embedding

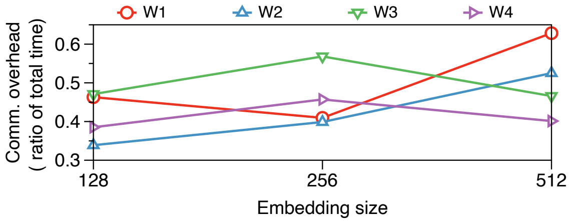


Figure 4: Embedding communication overhead can consume up to 63% of the end-to-end training time. The embedding size is the dimensions of an embedding or the number of columns of embedding tables.

prefetching[4] to precisely quantify the embedding communication overhead and its breakdown. The result is shown in Figure 4, where the embedding communication overhead for many of these workloads is high despite using state-of-the-art communication libraries like NCCL, and consumes up to 63% of end-to-end DLRM training time. It is worth noting that the ratio of the communication time to the total training time may not monotonically increase with the embedding size (the number of dimensions of an embedding or the number of columns of embedding tables), as the embedding size increases both computation time and communication time.

The embedding communication includes worker pull and worker push, which can be caused by either cache miss or cache update. When a cache miss occurs, the worker will pull the required embedding and push an evicted embedding if the cache is full. For cache updates, the worker will push the embedding gradients and pull the required embeddings with the latest version updated by other workers. Based on our evaluation, cache update is the major reason that causes embedding transmissions (71%-95%). In terms of the communication direction, pull and push contribute to a similar overhead (less than 9% coefficient of variance).

3 Embedding Scheduling

To reduce the number of worker pulls/pushes during the training, we propose a new direction to accelerate distributed DLRM training without compromising the model accuracy: *embedding scheduling*, which utilizes an embedding scheduler to determine where embeddings should be trained and which embeddings should be synchronized.

3.1 Rationales

In the forward propagation, a cache hitting on the required embedding can prevent worker pull and potential worker push caused by a cache eviction. Therefore, the embeddings to be trained can be scheduled to appropriate workers, where the training embeddings in each worker are most likely to hit the cache. Meanwhile, accessing an embedding in the forward propagation will incur a corresponding update in the backward propagation. However, synchronizing every embedding update is unnecessary, even in synchronous training. The reason is that an embedding is related to a sparse feature, and this feature may not be trained in the following iterations by other workers. It may happen in two scenarios: (1) this feature does not appear in the later training samples, or (2) this feature is only trained by the same worker. In other words, the updated embeddings only need to be synchronized to PS when they are required by other workers in the following training. Putting all this together, we derive two philosophies to reduce the embedding transmissions:

$\mathcal{P}1$. training in-cache embeddings as much as possible, and

$\mathcal{P}2$. performing on-demand synchronizations.

These philosophies can be followed by proactively partitioning the batch samples among workers and identifying the necessary embeddings for synchronizations.

Figure 5 describes a contrived and illustrative example to elaborate on how embedding scheduling reduces communication overhead. The vanilla training without embedding scheduling (Figure 5b) neither optimizes for cache hit rate nor avoids unnecessary embedding synchronizations due to the ignorance of workers' cache content and the following inputs. As a result, the cache miss or hit during the training is totally random, and every updated embedding should be synchronized. However, such embedding communication can be optimized with embedding scheduling (Figure 5c).

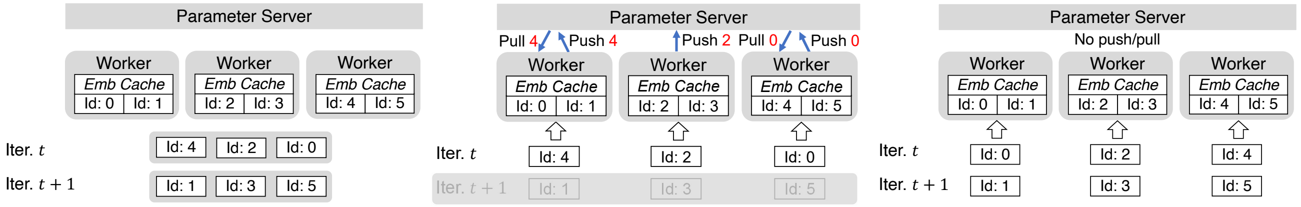
We can formulate the scheduling problem as a Markov Decision Process (MDP) as described in Appendix A. However, given the complexity of finding a globally optimal scheduling decision for the entire training process, we downgrade the scheduling problem to find a local quality decision for every iteration in this work.

3.2 Opportunities

Embedding scheduling is feasible given two characteristics of in-cache embedding accesses.

Predictability. There are two prerequisites for embedding scheduling: knowing current cache snapshots as well as predicting and determining future embedding accesses. Fortunately, both are achievable in DLRM training. As modern training frameworks decouple the training computation and the input preparation, we can foresee and decide proper embedding accesses a-priori when proceeding with input generation for each worker. The input generation is usually performed in a data loader, which partitions a batch of inputs into multiple micro-batches for every worker. Meanwhile, maintaining cache snapshots in the data loader is trivial. Therefore,

[4] HET supports prefetching the embeddings that will be used in the next iteration from the PS at the beginning of an iteration. We will enable this feature in end-to-end evaluation. However, prefetching brings limited improvement in vanilla BSP training, since most embedding pulls are for the updated embeddings as shown in the later paragraph, and these updated embeddings are valid after finishing the synchronization of the current iteration.



(a) Toy example (b) Training w/o embedding scheduling (c) Training w/ embedding scheduling

Figure 5: (a) A contrived example showing performance gain with embedding scheduling, assuming that all caches have empty slots for new embeddings. (b) The vanilla training incurs a total of 2 times worker pulls and 3 times worker pushes at iteration t. (c) The training with embedding scheduling does not incur any embedding transmissions at iteration t, since all trained embeddings hit caches and their updates are not required by the others in the later iteration.

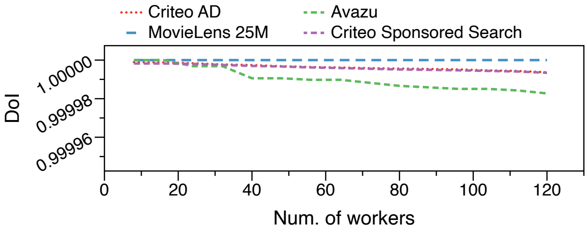


Figure 6: The degree of infrequence is consistently high across different scales of training clusters in studied datasets.

a tailored data loader with an embedding scheduler can allocate input samples to workers based on their in-cache embeddings and identify embedding dependencies among iterations for on-demand synchronizations.

Infrequency. Although a small set of "hot" embeddings could contribute to the majority of the total number of accesses [3,31], the access of individual in-cache embeddings is usually infrequent. We define that an in-cache embedding is infrequently accessed when the popularity of this embedding is less than the total number of samples trained by a worker during the entire training epoch. The more infrequently-accessed embeddings there are the more potential transmissions can be optimized. To evaluate the *degree of infrequence (DoI)*, i.e., the ratio of infrequently-accessed embeddings over the whole in-cache embeddings, we make profiling on real-world datasets as listed in Table 1, and consider each worker can cache 10% of the total embeddings. As shown in Figure 6, our profiled datasets exhibit consistently high DoI (> 99%) even in large training clusters. The high DoI indicates that most in-cache embeddings can each be trained in a consistent worker and their updates can avoid global synchronizations.

3.3 Model Consistency Analysis

Accelerating distributed DLRM training under BSP with embedding scheduling can preserve model consistency and thus the exactly same model accuracy. Compared with vanilla distributed training, embedding scheduling makes two changes: a tailored input partition algorithm (vs. a random partition algorithm) and on-demand synchronizations (vs. full-set synchronizations). In BSP, on-demand synchronizations ensure that all workers use the latest parameters for the incoming training iteration, which is the same behavior as the vanilla distributed training performs.

We prove that the training model will not be affected by the choice of input partition algorithm under BSP. Considering a parameter optimizer followed by the SGD algorithm [14], the gradient calculation for model weights w on a given batch of n training samples is expressed as follows:

$$\nabla w = \frac{1}{n}\Sigma_{i=1}^{n}\frac{\partial L(x_i,w)}{\partial w}, \qquad (1)$$

where x_i is the i-th training sample of the batch, and L is the loss function. Based on Equation 1, the gradient of the batch is the sum of the individual gradient for each training sample in the batch. Since the individual gradient depends only on input samples and current model weights, which have been synchronized before this iteration begins under BSP, partitioning the batch into m micro-batches takes no effect on the gradient result:

$$\frac{1}{n}\Sigma_{i=1}^{n}\frac{\partial L(x_i,w)}{\partial w} = \frac{1}{n}\Sigma_{i=1}^{m}\Sigma_{j=1}^{n/m}\frac{\partial L(x_{ij},w)}{\partial w}, \qquad (2)$$

where x_{ij} is the j-th training sample in the i-th micro-batch (for worker i). Therefore, any partition result generated by any partition algorithm preserves the same gradients as in BSP throughout training and hence converges to the same model.

4 Design

Based on the above ideas, we design an embedding scheduler, called Herald, to accelerate distributed DLRM training on top of the state-of-the-art cache-PS architecture [3,31]. Real-time scheduling is a crucial property when 1) training over streaming samples [21,29,38], as pre-processing the whole dataset is difficult, if not impossible, in this case, and 2) the number of training iterations (scheduling decisions) is unpredictable. In this work, we target to support real-time scheduling, where only a few batch inputs instead of the whole dataset are available at a time and the available batch inputs keep updating during the training procedure. We leave discussions on breaking this assumption in §7.

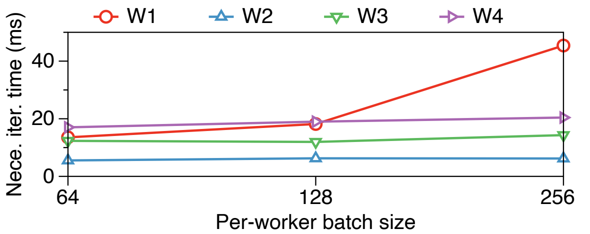


Figure 7: The necessary iteration time varies among different datasets and different per-worker batch sizes, where the embedding size is 256.

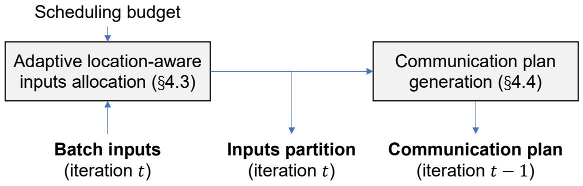

Scheduling budget
Adaptive location-aware inputs allocation (§4.3)
Communication plan generation (§4.4)
Batch inputs (iteration t)
Inputs partition (iteration t)
Communication plan (iteration $t-1$)

Figure 8: Herald overview.

4.1 Challenges

Integrating real-time embedding scheduling into training frameworks is challenging due to limited and varied *scheduling budgets*, i.e., the available scheduling time.

Limited scheduling budget for real-time scheduling. For a scheduling problem, there exists a tradeoff between scheduling quality and scheduling overhead. High-quality embedding scheduling can effectively reduce communication overhead, but at the cost of consuming a long scheduling time. Such long scheduling time, however, prevents the scheduling from being hidden by the iterative training, and thus introduces extra training overhead. The scheduling budget of real-time scheduling should be smaller than the iteration training time.

Varied scheduling budgets across different workloads and training settings. The scheduling budget is not static and is affected by many factors, including workload and training setting. We regard *necessary iteration time*, i.e., the training time excluding the embedding communication time in an iteration, as a valid scheduling budget, and show these results in Figure 7[5] with the same setting as Figure 4. It reveals an up to 78% coefficient of variance across different workloads within the same setting and an up to 67% coefficient of variance across different settings within the same workload. Varied scheduling budgets indicate that a static scheduling algorithm is not generic enough to support highly variant workloads and settings.

4.2 Overview

Figure 8 overviews Herald, which decouples the schedule targets to support real-time scheduling. Specifically, Herald determines *"where embeddings should be trained"* via an adaptive location-aware inputs allocation (LAIA) algorithm (§4.3), which meets varied scheduling budgets with a high cache hit rate. Herald determines *"which embeddings should be synchronized"* under a given inputs partition via an optimal communication plan generator (§4.4). The above scheduling process is executed on every worker instead of a centralized orchestrater, so that the scheduling can be computed locally to save the distribution time of scheduling results.

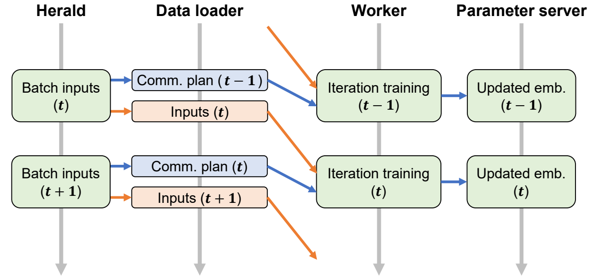


Figure 9: Herald training data flow. Herald leverages input prefetching to generate the communication plan in time.

As discussed before (§3.1 and Appendix A), the scheduling scope in the current design choice is within the single batch to achieve a low scheduling overhead. Given a batch of inputs of iteration t, Herald first generates a partition solution, which allocates each input to a proper worker for iteration t based on current embeddings' location information. However, there is a lag in the generation of the communication plan, which depends on the requirements of the next iteration. To guarantee that each worker can retrieve the latest embeddings under the generated partition solution, Herald figures out an optimal communication plan, which lists a minimal list of embeddings that each worker should synchronize with PS, for the previous iteration, i.e., iteration $t-1$. We discuss extending the scheduling scope to multiple batches in §7.

Figure 9 illustrates the data flow of the training with Herald. From a worker's point of view, in every iteration, the worker receives both training samples and a communication plan from the data loader before proceeding computation, and synchronizes the embeddings listed in the communication plan after backward propagation. Due to the time lag of communication plan generation, Herald needs to partition the next batch inputs before proceeding to that batch, which is possible as prefetching next batch samples is already commonly adopted in today's data loaders [1, 6, 36].

It is worth noting that Herald only controls the embedding update from workers to PS (i.e., worker push), but remains the update manner from PS to workers (i.e., worker pull) the same as existing cache systems operate. Herald relies on the internal cache consistency protocol in existing cache systems (e.g., version check in HET [31]) to pull the updated embeddings from PS and cache them in individual workers. Therefore, Herald preserves the same cache consistency as the underlying cache system. Moreover, as the cache miss results in only a minor fraction of embedding transmissions (§2.2), Herald does not optimize the policy of embedding

[5] Following the industry practice [2], we focus on *weak scaling*, which linearly increases the total batch size with the number of workers, instead of *strong scaling*, which keeps the per-iteration total batch size constant.

Algorithm 1: Static LAIA

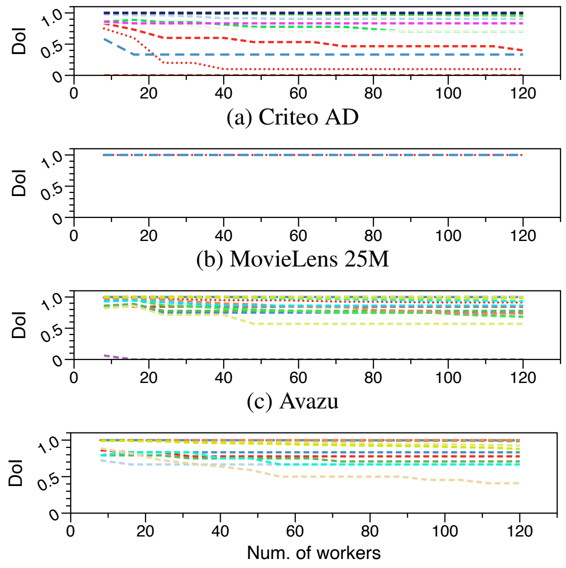
```
   input  : Batch samples ($Inputs$) and worker list ($Workers$)
   output : Inputs partition ($Alloc$)
1  Init $Alloc$;
2  Init all workers as available;
3  $capacity = size(Inputs)/size(Workers)$;
4  for $i$ in $Inputs$ do
5      for $w$ in $Workers$ do
6          $score_{(i,w)} = |cache(w) \cap embs(i)|$;
7      end
8      Find worker $w$ with the largest score among the available workers;
9      $Alloc_{(i,w)} = 1$;
10     if $\Sigma_i Alloc_w == capacity$ then
11         Mark $w$ as unavailable;
12     end
13 end
14 return $Alloc$;
```

cache evictions caused by the cache miss to avoid additional scheduling overhead.

Scheduling budget measurement. To preserve real-time scheduling, the scheduling budget should not be larger than the iteration time, which consists of both computation time and communication time. However, with embedding scheduling, the communication depends heavily on the scheduling results, which are hard to predict precisely before training[6]. Therefore, we utilize the necessary iteration time (as defined in §4.1), which can be profiled in advance, as a loose scheduling budget. For Herald scheduling, as only the input partition can adapt to the scheduling budget, and the upper bound of the communication plan generation time is predictable given the workload and the training setting, the real scheduling budget used in Herald should be the necessary iteration time minus the communication plan generation time. As our measurement is conservative (by considering a loose metric), it is not likely to overestimate the scheduling budget. Furthermore, our evaluation (§6.2.2) shows that Herald is resilient to the underestimation of the scheduling budget.

4.3 Location-aware Inputs Allocation

We first elaborate a static LAIA algorithm that heuristically partitions batch inputs into proper workers based on the current cache snapshots and a full set of table information. Then we observe that embedding tables are not equally important for scheduling, with the diverse DoIs among embedding tables. Based on this observation, we introduce an adaptive LAIA algorithm, which conducts the partition with selective embedding tables to meet the scheduling budget.



(a) Criteo AD

(b) MovieLens 25M

(c) Avazu

(d) Criteo Sponsored Search

Figure 10: The DoI of different embedding tables exhibits a large variance. Each line represents an embedding table.

Static LAIA. As it is impractical to conduct a brute-force search in real time for the optimal partition solution that maximizes the cache hit rate, we propose a heuristic partition algorithm, LAIA, as shown in Algorithm 1. LAIA calculates a score to quantify the relevance between every input sample and worker (Line 6). The score is defined as the number of embeddings that reside in the worker cache (with the latest version) and are required by the input sample simultaneously. After scoring, LAIA allocates each input to the worker with the highest score (Line 8-9) while ensuring evenly distributed inputs among workers (Line 10-12).

Furthermore, when there is a tie in finding the largest score (Line 8), LAIA will randomly select a worker from the candidates. This randomization tends to evenly cache embeddings across all workers. Therefore, in the later scheduling, LAIA can achieve a good balance of the total score (i.e., cache hit rate) among workers. We do not proactively treat the load balance as a scheduling goal in Algorithm 1 for a low scheduling overhead. With consecutive iterations, LAIA moves towards gathering embeddings that tend to be accessed together in many training samples on the same worker.

DoI diversity among tables. We observe that since every embedding table represents a unique physical feature, the distribution of embedding popularities in different tables may exhibit different skewness and hence diverse table DoIs. Figure 10 shows this DoI variance, where we measure the DoI of an individual table as the ratio of the number of infrequently accessed embeddings over the total number of in-cache embeddings of this table. Among the profiled datasets (except for MovieLens 25M which has only two tables[7]), there exist

[6] Even during training, the communication time after scheduling can vary among iterations.

[7] For those datasets that have a small number of tables, although the DoI diversity is not obvious, they have a low demand for this property, as their scheduling overhead is inherently small.

both tables with consistently high DoI ($> 99\%$) and tables with relatively low DoI (as low as $< 1\%$). Recalling that the potential performance gain achieved by embedding scheduling comes from the infrequent embedding updates (§3.2), this DoI diversity indicates that the embedding tables are not equally important for scheduling.

Being adaptive. Based on the above observation, we can accommodate Algorithm 1 to varied scheduling budgets. In Algorithm 1, scoring (Line 4-7) is one of the most time-consuming functions due to its three-layer loop, including inputs iteration, workers iteration, and tables iteration (implicitly shown in Line 6). Therefore, by only considering scheduling-worthy (i.e., high-DoI) embedding tables, the time overhead of scoring and thus Algorithm 1 can be reduced. Given the scheduling budget, we can determine the maximum number, k, of embedding tables that should be considered in scoring.

Table profiling. To identify the k embedding tables with the highest DoI in runtime, we incorporate a table profiler into Herald to continuously measure the DoI of each table. Specifically, the profiler counts the appearance of every embedding in every batch input and maintains the DoI of each table dynamically. The profiling runs parallel to the allocation function to avoid interference with the scheduling process. The profiler periodically updates the k embedding tables with the highest DoI to Laia for scheduling. The interval to update the profiling result can be decided by either the change of the selective embedding tables or the profiling time.

4.4 Communication Plan Generation

To figure out the necessarily synchronized embeddings for on-demand synchronizations, we introduce a concept of *embedding dependency*. An embedding dependency appears when an embedding with the latest version is cached on a worker and there are other workers assigned to train this embedding in this iteration. The worker caching the latest version should synchronize this embedding to PS before this iteration begins. Therefore, all embedding dependencies of this iteration become the communication plan of the previous iteration. As discussed in §4.2, Herald can timely generate the communication plan with the inputs prefetching.

Based on the above idea, Herald generates the communication plan for each worker given the inputs partition result of the next iteration, as described in Algorithm 2. The *embs_by_others* are all embeddings required by workers except for worker w itself (Line 2) in the next inputs, and the embedding dependencies (communication plan) of worker w are the intersection of *embs_by_others* and its cached embeddings (Line 3). The generated communication plan is optimal between two iterations, as it contains the minimal list of embeddings required by other workers in the next iteration. Since we focus on the PS architecture, the embedding dependencies of a worker do not need to be distinguished among target workers. We will discuss how to support on-demand synchronization with peer-to-peer communication in Appendix B.

Algorithm 2: Embedding dependency generation

```
   input  : Batch samples ($Inputs$), worker list
            ($Workers$), and inputs allocation ($Alloc$)
   output : Embedding dependency ($Dep$)
1  for $w$ in $Workers$ do
2      $embs\_by\_others = embs(Inputs - Alloc_w)$;
3      $Dep_w = embs\_by\_others \cap cache(w)$;
4  end
5  return $Dep$;
```

5 Implementation

We have implemented a system prototype of Herald with C++ and Python. Besides the design described in §4, Herald maintains cache snapshots of all workers inside the scheduler to provide information on embedding locations. These cache snapshots are used for the scoring function in inputs allocation, and updated at the end of the allocation based on the partition result, which avoids synchronizations among schedulers in different workers. Cache snapshots follow the same caching policy as workers' for consistency.

HET plugin. We have integrated Herald in HET[8]. We replace the data loader implementation in HET with communication plan support. Herald data loader returns training samples and synchronized embedding indexes as sparse inputs. To support prefetching batch inputs while preventing from interfering with the training process, Herald is launched by dedicated threads, and the scheduling results from Herald are transmitted to the data loader via shared memory.

Multi-threading. We utilize spare CPU resources to accelerate the scheduling by enabling multi-threading. We manage a thread pool for multi-threading processing. The parallelism can be easily applied to most Herald functions, including scoring (Line 4-7 in Algorithm 1), cache snapshots update, and communication plan generation (Algorithm 2). However, it is non-trivial to parallelize the function of worker allocation (Line 8-12 in Algorithm 1) due to its dependency between two allocation actions. Preserving the consistency of the allocation result inevitably introduces a lock and limits the parallelism of this function. We implement two versions of the worker allocation function: a single-threading version and a lock-free multi-threading version which may compromise the result consistency. In the multi-threading version, we evenly divide the worker capacity and training samples among threads. Therefore, each thread can independently allocate its samples to the workers without considering the allocation result of other threads. The allocation result of each thread is then merged into the final allocation.

[8] https://github.com/Hsword/Hetu

6 Evaluation

Our evaluation seeks to answer the following three questions:
How does Herald improve cache embedding communication overhead? Simulation experiments (§6.2.1) with representative datasets show that Herald reduces the average number of embedding transmissions by 48%-89%, where Laia algorithm contributes to the major improvement according to the performance breakdown.
How lightweight and effective is Herald for real-time and adaptive scheduling? Deep dive profiling (§6.2.2) on Herald shows that even adopting the single-threading worker allocation function, multi-threading benefits 92% of scheduling time. Furthermore, adaptive Herald preserves consistently high scheduling quality with less than 1.11× transmission increase compared with static Herald across different scales of training clusters.
How does Herald perform in end-to-end DLRM training? Testbed experiments (§6.3) with real-world workloads show that Herald provides 1.09×-2.11× and 1.02×-1.61× speedup over HET in end-to-end training with TCP and RDMA, respectively, over 100 Gbps Ethernet. Herald preserves the effectiveness in the multi-GPU and multi-node scenario.

6.1 Evaluation Settings

Our evaluations include both cache simulation for micro-benchmarks and testbed for end-to-end training. Unless mentioned otherwise, both the simulation and testbed contain 8 workers and 1 PS. Each worker contains an embedding cache following LRU policy. The embedding cache can house 10% of PS-side embedding tables, following the caching strategy studied in [25, 31, 48]. We adopt the single-threading version of the worker allocation function whenever possible for the best scheduling quality, and adopt the multi-threading version when evaluating the scalability of Laia algorithm (§6.2.2). The default thread pool size in Herald is 16.
Testbed. Both workers and PS are equipped with a 20-core Intel Xeon Gold 5218R CPU at 2.1 GHz and 64 GB (256 GB in PS) of RAM. Each worker is also equipped with an Nvidia 3090 GPU. These workers and PS are connected at 100 Gbps Ethernet with Mellanox ConnectX-5 NICs. The default underlying transport is TCP. We also incorporate RDMA into HET implementation. All machines run Ubuntu 18.04 (Linux 5.4.0), CUDA 11.3.1[9], cuDNN 8.2.0, and NCCL 2.9.9.
Baseline. We compare Herald with HET [31] and FAE [3]. FAE is designed in a static caching manner. For a fair comparison, we implement FAE on top of HET. As discussed in §1 and §8, both baselines reduce the embedding transmissions by compromising model accuracy and are orthogonal to Herald. Therefore, we adopt vanilla BSP training in HET and FAE. Other settings are configured as same as Herald. Baselines and Herald leverage a hybrid synchronization model [24], where MLP parameters are synchronized in ring-based all-reduce and embedding parameters are synchronized with PS. Baselines and Herald enable the embedding prefetching.

Optimization	Reduced embedding transmissions		
	Pull	Push	Overall
Vanilla (w/o embedding scheduling)	-	-	-
Laia (§4.3)	54%	17%	35%
Communication plan (§4.4)	0%	13%	7%
Herald	54%	63%	59%

Table 2: Breakdown of contribution by each optimization with the W1 dataset.

Workloads. We use four real-world models with representative datasets for end-to-end experiments, as listed in Table 1. We exclude the first 10 iterations for warm-up and report the performance for the subsequent iterations. We do not include training accuracy results because Herald can preserve a consistent model accuracy, as analyzed in §3.3. The default batch size is 128 in the simulation and 256 in the testbed, and the default embedding size is 512. Since Herald with multi-threading can meet the scheduling budget of the above workloads in the default training setting based on our evaluation (§6.2.2), we adopt static Laia (Algorithm 1) rather than its adaptive version in end-to-end experiments by default.

6.2 Micro-benchmarks

6.2.1 Cache Performance

We first study the cache behaviors by comparing Herald with vanilla training manner, i.e., a random inputs partition and full-set embedding synchronizations.
Cache behaviors. Figure 11 shows the number of cache embedding transmissions by normalizing to that of vanilla training across different datasets. Overall, applying Herald can reduce 48%-89% embedding transmissions. We further decompose the cache embedding transmissions based on the communication direction. It shows that Herald effectively reduces the number of worker pulls by 43%-88% and worker pushes by 51%-90%.
Improvement breakdown. To figure out the reason behind these improvements, we further break down the contribution made by each optimization with the Criteo AD dataset (W1), as shown in Table 2. We find that Laia contributes to the major improvements. For worker pull, Laia distributes the embeddings required by inputs to most likely hit worker caches (reducing the number of worker pulls by 54%). Meanwhile, the worker push performance is jointly optimized by Laia and communication plan, where the location-aware partition mechanism reduces embedding dependencies while on-demand synchronizations make such dependencies reduction benefit to embedding transmissions reduction. Therefore, the optimization of Laia and communication plan alone only reduces the number of worker pushes by 17% and 13%, respectively, but the combination of the two, i.e., Herald, can reduce the number of worker pushes by 63%.

[9] We upgrade HET from CUDA 10 to CUDA 11.

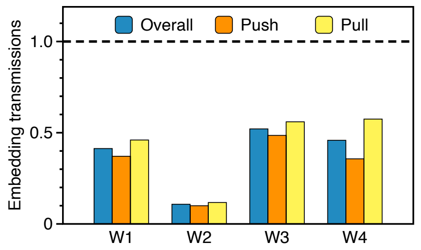


Figure 11: Herald can reduce 48%-89% embedding transmissions.

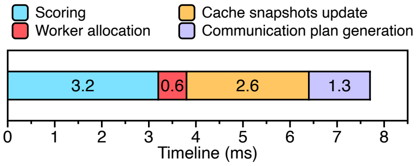


Figure 12: Breakdown of scheduling overhead with the W1 dataset.

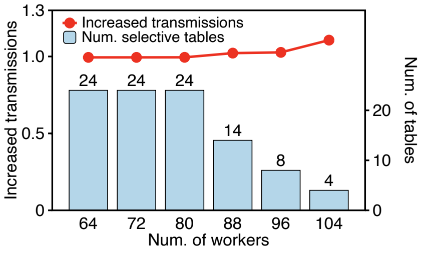


Figure 13: Adaptive LAIA is scalable with less than 1.11× transmissions increase.

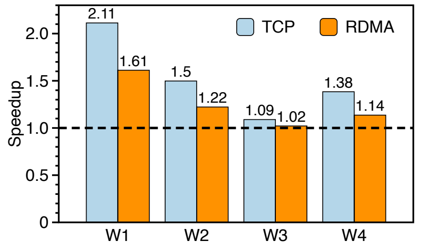


Figure 14: Herald improves end-to-end training by up to 2.11× speedup over TCP and up to 1.61× over RDMA.

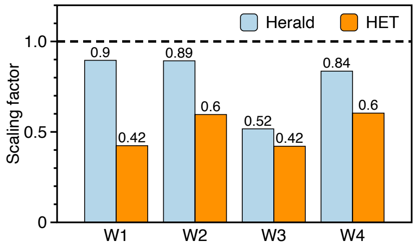


Figure 15: Herald utilizes distributed computing resources efficiently with high scaling factors.

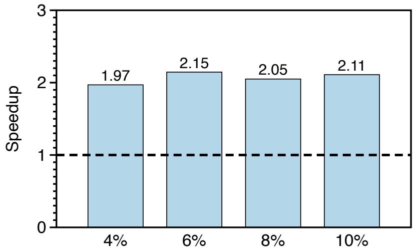


Figure 16: Herald preserves a consistent effectiveness under limited cache sizes.

6.2.2 Scheduler Performance

We profile the Herald scheduler overhead with the same machine as the testbed (§6.1) to evaluate its time consumption and scheduling quality. In this part, we focus on the Criteo AD dataset only.

Overhead breakdown. Through our measurement, the per-batch scheduling overhead is 7.7 ms (16 threads) on average. Figure 12 shows the breakdown of this scheduling overhead, where the major overhead is caused by the scoring function, consuming 3.2 ms (42%). 92% of the overhead, including scoring, cache snapshots update, and communication plan generation, are accelerated by multi-threading. Although the worker allocation function runs in a single thread, it contributes to only 0.6 ms (8%) overhead.

Scalability of adaptive LAIA. We further evaluate the adaptiveness of LAIA across different scales of training clusters. We assume that the scheduling budget[10] does not change when increasing the scale of training clusters. Thereby, we find the number of tables that LAIA can be considered during the scoring, so that the whole scheduling time is not larger than the scheduling budget, i.e., 18.2 ms in this case. In this experiment, we use a multi-threading worker allocation function for adaptive LAIA. We report the results of different cluster scales, where the number of selective tables is smaller than the total number of tables (i.e., 26) in Criteo AD dataset. The increased transmissions are the ratio of the embedding transmissions incurred by adaptive LAIA to that incurred by static LAIA. As shown in Figure 13, adaptive LAIA introduces less than 1.11× transmission increase by scoring with at least 4 tables. This result indicates that Herald is scalable and adaptive to different scales of training clusters. Moreover, it also reveals that Herald is resilient to the underestimation of the scheduling budget by preserving a high scheduling quality with a small set of selective tables.

Herald vs. brute-force search. The performance of Herald compared with the brute-force search is shown in Appendix C due to the space limitation.

6.3 End-to-end Training

We demonstrate that Herald accelerates end-to-end training, improves scalability, and preserves performance superiority even under limited cache sizes and large-scale clusters.

Training speedup. Figure 14 shows the training speedup of Herald over HET across different workloads. We observe that Herald can achieve 1.09×-2.11× speedup over TCP and 1.02×-1.61× over RDMA. Specifically, we find that the speedup of W1 is the highest among all, which accords with the results in Figure 4 that W1 has the highest embedding communication ratio. The above speedups are attributed to the communication reduction of Herald in the number of embedding transmissions, as explained in §6.2.1.

Scalability. The high training performance makes the distributed training system more scalable. We evaluate the scalability of Herald and HET with the metric of scaling factor defined as in [51]: $\frac{T_N}{NT}$ where T is the throughput of a single worker, N is the number of workers, and T_N is the overall throughput of a cluster with N workers. Note that T in Herald

[10] The scheduling budget increases with the scaling training clusters due to the larger dense parameters synchronization time. Therefore, we can score with a larger number of embedding tables in adaptive LAIA than we report.

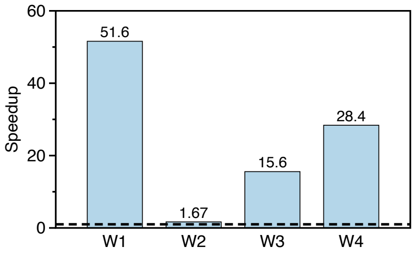


Figure 17: Herald achieves tremendous speedup compared with FAE-like static caching.

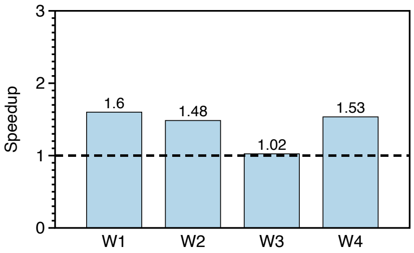


Figure 18: Herald preserves the performance superiority in a large-scale testbed.

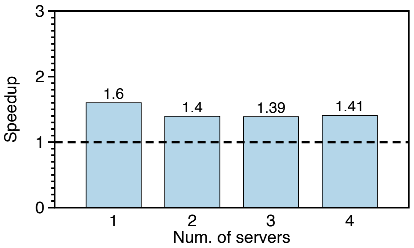


Figure 19: Herald preserves the performance superiority when increasing the number of servers for PS.

equals that in HET, as both frameworks behave the same in the single-worker training. We measure the throughput of a single worker with a single GPU server, where the PS runs on CPUs in the same server to eliminate network communication. Figure 15 shows the scaling factor of Herald and HET in every workload. Herald improves the scaling factor from 0.42-0.60 to 0.52-0.90, which indicates that incorporating Herald into distributed DLRM training can utilize resources more efficiently.

Training performance under limited cache sizes. Given the ever-expanding size of the embedding tables and the relatively slow evolution of accelerator device memory, it is likely that the ratio of in-cache embeddings becomes smaller. We study the impact of different embedding cache sizes on W1. The cache ratio, i.e., the ratio of the number of in-cache embeddings to the total number of embeddings, ranges from 4% to 10%. As shown in Figure 16, Herald preserves the performance superiority over HET and the speedup (1.97×-2.15×) does not vary a lot across different sizes of embedding cache. The reason is that, as discussed in §2.2, embedding transmissions are dominated by the embedding update, and this dominance holds across different embedding cache sizes. Therefore, the distributed DLRM training framework still benefits from reducing embedding pulls and pushes in embedding updates even under a small embedding cache size. This experiment verifies the consistent effectiveness of Herald under limited cache sizes.

Compared with FAE (static caching). FAE [3] is quite a different cache model from Herald and HET. First, FAE is a static cache, i.e., the cached embeddings are predetermined and fixed before training. Second, all workers in FAE cache exactly the same embeddings, and they synchronize all cached embeddings with all-reduce as dense parameters. We build FAE-like static caching on top of HET. Our FAE-HET implementation supports caching at most 1% embeddings in each worker and training workloads with a per-worker batch size of 128. For implementation efficiency, we change embedding interactions from concatenation to sum pooling in all workloads. As shown in Figure 17, Herald outperforms FAE with a 1.67×-51.59× speedup. The huge improvement mainly comes from two reasons. First, since Herald's workers can cache arbitrary embeddings, Herald's overall cache size is larger than FAE's, thus leading to a higher cache hit rate. Second, the embedding scheduling in Herald further improves the efficiency of the cache usage (by reducing the number of worker pulls and worker pushes) compared with FAE.

Multi-GPU and multi-node scenario. Lastly, we evaluate Herald in a multi-GPU and multi-node testbed, consisting of 10 × 8-GPU servers as workers. Each worker contains 80 CPU cores, and the thread pool size is also set to 80. The other settings remain the same as the default. At such a large scale, Herald adopts adaptive scheduling (§4.3) to bound the scheduling time. Figure 18 shows the results. By comparing to HET, Herald can achieve 1.02×-1.60× speedup among four workloads by leveraging 77%-100% of tables for scheduling. We further evaluate the Herald performance when increasing the number of servers for PS on the W1 workload. As shown in Figure 19, the improvement of Herald may have a slight decrease (achieving about 1.4× speedup) when the capability of PS increases. The reason is that increasing the number of servers for PS can relieve or even eliminate the network bottleneck in the PS architecture [41].

7 Discussions

Scheduling without batch prefetching. If the streaming dataset only provides a single batch of inputs at a time, i.e., no batch prefetching support, Herald will fail to generate communication plans for the on-demand synchronizations. In this case, Herald's LAIA algorithm still works to provide a cache-friendly inputs partition. Based on the optimization breakdown (Table 2), scheduling with only LAIA can preserve a considerable overall improvement.

Scheduling among multiple batches. If the scheduling budget is sufficient or even unlimited (offline scheduling in this case), Herald's scheduling scope can be extended to multiple batches to achieve a better scheduling quality.

Inputs partition. Instead of finding a locally high-scoring allocation of a single batch as in Algorithm 1, the partition algorithm for multiple batches should seek an allocation with a globally high score. However, searching for such globally high-scoring allocation is onerous, as allocation scores are dependent on batches. Different allocation results of the front batch may result in totally different cache snapshots, and thus affect the allocation score of the following batch. A potential heuristic method to reduce the exploration space is to leverage the power of two choices [32], which considers the top-2 highest scoring allocations of a batch, explores the next batch based on these two allocations in parallel, and finally returns the allocation with the highest overall scoring.

Beyond preserving the same batch order when allocating the multi-batch inputs, there exists work to reorder inputs among training iterations for embedding transmissions saving [4]. However, the non-random ordering of training samples may increase the gradient variance during the SGD training and lead to overfitting, as the model may learn some spurious patterns from these handcrafted sample orders [13].

Communication plan generation. In current practice, workers may execute different sizes of communication plans in an iteration. This kind of work imbalance will result in idle workers at the synchronous barrier of the iteration. This issue can be improved by considering multiple batches when generating the communication plan for both embedding pull and push, as discussed in Appendix A, with additional embedding dependency checking. For embedding pull, the worker can prefetch an embedding if it is not trained by other workers until the time this worker accesses it. For embedding push, the worker can synchronize more embeddings beyond the requirement in the next iteration. Based on this idea, we can re-schedule communication plans to balance the communication workload among workers, and thus reduce the overall communication time.

8 Related Work

Distributed recommendation systems. Several specialized systems have been proposed for scalable and efficient training upon DLRMs. Persia [28] advocates mixing the synchronous and asynchronous mechanisms to update MLP and embedding tables, respectively. However, the asynchronous scheme is not scalable and can compromise accuracy with an increasing number of workers [34]. XDL [21] proposes optimizations including hierarchical sample compression, workflow pipeline, and zero copy. It provides systematical optimizations on the DLRM training pipeline, and can benefit from the embedding scheduling to further optimize inter-worker/PS communication. We acknowledge that the embedding scheduling only works for data parallelism. For the system adopting a hybrid parallelism strategy, like Neo [33], embedding scheduling can provide a partial system acceleration.

Communication acceleration. There are many efforts exerted to accelerate communication for deep learning training. A line of work speeds up individual messages with efficient collective communication design. Besides those designed for deep neural networks [8, 35, 39, 41, 42, 45], many collective communication approaches [11, 24, 27, 37] are proposed to optimize the synchronization of sparse parameters. Another line of work exploits communication scheduling [18, 20, 50], which organizes the message transmission order of different layers to overlap communication with computation. All the above communication acceleration methods try to answer *"how to efficiently transmit messages"*. In contrast, Herald accelerates communication by answering *"which messages should be transmitted"*.

Serving large embedding tables. There are two common architectures to resolve the large memory requirement on embedding tables. The first one applies model parallelism directly across multiple GPU workers, where each GPU stores a shard of tables on its high-bandwidth memory (HBM) [27,44]. However, this manner is sub-optimal, as the GPU resource is usually scarce and expensive. The second architecture is the cache-PS architecture [52] as the focus of this paper. This architecture leverages the skewness feature of datasets to accelerate embedding accesses with high popularity, while Herald further identifies the infrequency feature among those cached embeddings. To reduce the communication between cache and PS, existing works apply accuracy-compromising optimizations including oversampling hot inputs (accessing only hot embeddings) [3], reordering training samples among batches [4], and updating embeddings with staleness-tolerance [31]. On the contrary, Herald optimizes the embedding communication with embedding scheduling, which can preserve the model (accuracy) consistency theoretically. Moreover, when the accuracy is not that sensitive, Herald can also benefit from the above philosophies to optimize the scheduling process. Another orthogonal line of optimization focuses on cache prefetching by scheduling the embedding IO and the computation inside a worker [5, 15, 25].

9 Conclusion

This paper presents Herald, a runtime embedding scheduler for efficient cache-enabled recommendation model training. By leveraging the predictability and infrequency of embedding cache accesses, Herald applies an adaptive location-aware inputs allocation mechanism and an on-demand synchronization strategy to reduce the embedding transmissions between workers and PS during training. Extensive experimental results show that Herald can significantly reduce the embedding communication overhead and thus boost the end-to-end recommendation model training.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Mohammad Alizadeh for their constructive comments. We thank Yilun Jin for insightful discussions, and appreciate HET maintainers for their timely responses to our issues. This work is supported in part by Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, Key-Area Research and Development Program of Guangdong Province (2021B0101400001), and the Turing AI Computing Cloud (TACC) [46]. Kai Chen is the corresponding author.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *HPCA*, 2021.

[3] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Accelerating recommendation system training by leveraging popular choices. In *VLDB*, 2021.

[4] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Heterogeneous acceleration pipeline for recommendation system training. In *arXiv*, 2022.

[5] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *arXiv*, 2022.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *arXiv*, 2015.

[7] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *DLRS*, 2016.

[8] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *SysML*, 2019.

[9] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *RecSys*, 2016.

[10] CriteoLabs. Criteo display ad challenge. https://www.kaggle.com/c/criteodisplay-ad-challenge.

[11] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *SIGCOMM*, 2021.

[12] Saeed Ghadimi, Guanghui Lan, and Hongchao Zhang. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. In *Math. Program.*, 2016.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. In *MIT press*, 2016.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. In *arXiv*, 2017.

[15] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. Scalefreectr: Mixcache-based distributed training system for ctr models with huge embedding table. In *SIGIR*, 2021.

[16] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: a factorization-machine based neural network for ctr prediction. In *arXiv*, 2017.

[17] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. In *TIIS*, 2015.

[18] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In *MLSys*, 2019.

[19] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *WWW*, 2017.

[20] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *MLSys*, 2019.

[21] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. Xdl: an industrial deep learning framework for high-dimensional sparse data. In *DLP-KDD*, 2019.

[22] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.

[23] Kaggle. Avazu mobile ads ctr. https://www.kaggle.com/c/avazu-ctr-prediction.

[24] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *EuroSys*, 2019.

[25] Youngeun Kwon and Minsoo Rhu. Training personalized recommendation systems from (gpu) scratch: look forward not backwards. In *ISCA*, 2022.

[26] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NeurIPS*, 2014.

[27] Shengwei Li, Zhiquan Lai, Dongsheng Li, Yiming Zhang, Xiangyu Ye, and Yabo Duan. Embrace: Accelerating sparse communication for distributed training of deep neural networks. In *ICPP*, 2022.

[28] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *SIGKDD*, 2022.

[29] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, et al. Monolith: real time recommendation system with collisionless embedding table. In *arXiv*, 2022.

[30] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding capacity-driven scale-out neural recommendation inference. In *ISPASS*, 2021.

[31] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. Het: Scaling out huge embedding model training via cache-enabled distributed framework. In *VLDB*, 2021.

[32] Michael Mitzenmacher. The power of two choices in randomized load balancing. In *TPDS*, 2001.

[33] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *ISCA*, 2022.

[34] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. In *arXiv*, 2020.

[35] NVIDIA. Collective communications library (nccl). https://developer.nvidia.com/nccl.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[37] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *SC*, 2019.

[38] Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, et al. Distributed equivalent substitution training for large-scale recommender systems. In *SIGIR*, 2020.

[39] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. In *arXiv*, 2018.

[40] Marcelo Tallis and Pranjul Yadav. Reacting to variations in product demand: An application for conversion rate (cr) prediction in sponsored search. In *arXiv*, 2018.

[41] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat - resilient allreduce tree for distributed machine learning. In *APNet*, 2020.

[42] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards domain-specific network transport for distributed dnn training. In *NSDI*, 2024.

[43] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *AD-KDD*, 2017.

[44] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G Abel, Xu Guo, Jianbing Dong, et al. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *RecSys*, 2022.

[45] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking transport layer design for distributed machine learning. In *APNet*, 2019.

[46] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[47] Xinyang Yi, Yi-Fan Chen, Sukriti Ramesh, Vinu Rajashekhar, Lichan Hong, Noah Fiedel, Nandini Seshadri, Lukasz Heldt, Xiang Wu, and Ed H Chi. Factorized deep retrieval and distributed tensorflow serving. In *MLSys*, 2018.

[48] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. In *MLSys*, 2021.

[49] Chaoliang Zeng, Xiaodian Cheng, Han Tian, Hao Wang, and Kai Chen. Herald: An embedding scheduler for distributed embedding model training. In *APNet*, 2022.

[50] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *ATC*, 2017.

[51] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *NetAI*, 2020.

[52] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. In *MLSys*, 2020.

[53] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *AAAI*, 2019.

[54] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *SIGKDD*, 2018.

A The Problem Definition of Embedding Scheduling

Prerequisites. Considering a distributed learning system with n workers and a per-worker batch size of m, the overall batch size is $n \times m$. The embedding table length, i.e., the total number of embeddings in the table, is d. The total number of iterations is T. The goal of embedding scheduling is to minimize the communication overhead, specifically the operation time of push and pull operations between workers and the PS.

For modeling simplicity, we delay the operation of the embedding push to the next iteration without violating the BSP training. In this model, an iteration proceeds with the following operations: retrieving the batch inputs, pushing the embeddings, pulling the embeddings, DLRM forward propagation, DLRM backward propagation, and dense parameter synchronization. There exists a synchronous barrier between two operations. Finally, we can model the problem as a Markov Decision Process (MDP) with the following components.

State space $\mathcal{S}$. An iteration can be described by a state including the training batch of the current iteration and the current state of embeddings. The training batch is related to the dataset and the current iteration number. The state of embeddings S is a matrix with d rows and $n+1$ columns, where the value of $s_{i,j}$ is 1 if the j-th worker ($j = n+1$ refers to the PS) has the latest version of the i-th embedding, a value between 0 to 1 if the j-th worker has a non-aggregated gradient (a partially aggregated gradient for PS) of the i-th embedding, and 0 for the rest.

Action space $\mathcal{A}$. The action space is defined by three action matrices: a sample assignment matrix P, an embedding push matrix Q, and an embedding pull matrix R. State actions are enforced based on these three matrices sequentially.

P is a matrix with $n \times m$ rows and n columns, where the value of $p_{i,j}$ is 1 if the i-th sample is assigned to the j-th worker, and 0 otherwise. Each sample can only be assigned to one worker, i.e., $\sum_j p_{i,j} = 1$. The sample assignment matrix P can transfer to the embedding assignment matrix U according to the accessing embeddings of each sample. The embedding assignment matrix U is a matrix with d rows and n columns, where the value of $u_{i,j}$ is 1 if the i-th embedding is assigned to the j-th worker, and 0 otherwise.

Q is a matrix with d rows and n columns, where the value of $q_{i,j}$ is 1 if the j-th worker pushes the i-th embedding to the PS, and 0 otherwise. Q should preserve the availability of the latest version of embeddings at PS if this embedding is not available by its assigning workers:

$$\begin{aligned} q_{i,j} = 1, \text{ if } & 0 < s_{i,j} < 1 \text{ and } \Sigma_{j' \in n} u_{i,j'} >= 1 \\ & \text{or } s_{i,j} = 1 \text{ and } s_{i,n+1} = 0 \text{ and } \Sigma_{j' \in n, j' \neq j} u_{i,j'} >= 1 \end{aligned}$$

R is a matrix with d rows and n columns, where the value of $r_{i,j}$ is 1 if the j-th worker pulls i-th embedding from the PS, and 0 otherwise. R is subject to two rules that ① the worker should pull the assigning embedding if it does not maintain the latest version of this embedding, and ② the PS has the latest version of the embedding when there is a pull request:

$$① \; r_{i,j} = 1, \text{ if } u_{i,j} = 1 \text{ and } s_{i,j} \neq 1$$
$$② \; r_{i,j} = 1 \Rightarrow s_{i,n+1} = 1$$

There is an implicit assumption that the PS has the latest version of the i-th embedding when performing rule ①. This assumption is maintained by Q.

State transition. The transition of the training batch is trivial. In terms of the state of embeddings, the transition is as follows: for $j \in [1,n]$ (workers),

$$s'_{i,j} = \begin{cases} \frac{1}{\Sigma_{j' \in n} u_{i,j'}}, & \text{if } u_{i,j} = 1 \\ 0, & \begin{array}{l}\text{if } u_{i,j} = 0 \text{ and } \exists j' \text{ such that } u_{i,j'} = 1 \\ \text{or } \Sigma u_{i,j} = 0 \text{ and } q_{i,j} = 1 \text{ and } s_{i,j} \neq 1\end{array} \\ s_{i,j}, & \text{otherwise} \end{cases}$$

for $j = n+1$ (PS),

$$s'_{i,n+1} = \begin{cases} 0, & \text{if } \Sigma_{j' \in n} u_{i,j'} >= 1 \\ min(1, s_{i,n+1} + \Sigma_{\forall j', q_{i,j'}=1} s_{i,j'}), & \text{otherwise} \end{cases}$$

Reward function $\mathcal{R}$. The reward is the negative of the communication latency. We assume the latency to push or pull an embedding remains the same. Since we preserve a strict order between the operations of push and pull, and the latency of these two operations depends on the worker with the largest workload, we define the reward function as follows:

$$\mathcal{R}(s,a) = -[\max_j \sum_{i=1}^{d} q_{i,j} + \max_j \sum_{i=1}^{d} r_{i,j}]$$

Objective. Based on the above-defined MDP, our objective is to find a policy $\pi(\mathcal{S}) \rightarrow \mathcal{A}$ that maps states to actions, which maximizes the expected cumulative reward $\mathbb{E}\left[\sum_{t=0}^{T} \mathcal{R}(\mathcal{S}_t, \mathcal{A}_t)\right]$.

A conventional solution to solve the above problem is dynamic programming. However, it does not work well in practice given the curse of dimensionality (the number of iterations in this problem), not to mention the fact that sometimes it is difficult to retrieve the entire dataset in advance from the streaming data and hard to predict the number of training iterations. Herald focuses on single batch scheduling, i.e., finding a quality action for the current state to get a high reward. Therefore, both the pull operation and the push operation perform on demand, and existing cache systems (like HET [31]) work with on-demand embedding pulls by design. Moreover, delaying the push operation after retrieving the batch inputs may be different from the practice, where the embedding push operation happens during the synchronization of the previous iteration. According to our analysis, the push action (Q) is mainly affected by the sample assignment. Therefore, in practice, we can decide on the push operation with an early sample assignment for the next iteration.

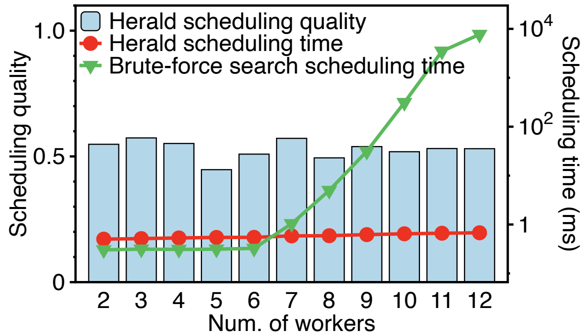


Figure 20: Herald vs. brute-force search (scheduling time is in log scale).

B Point-to-point Embedding Synchronization

In this paper, we follow the same distributed cache model as HET's [31], where each worker only communicates with PS. In this cache model, embedding synchronization requires at least two steps, one worker push and one worker pull. We can reduce the synchronization path to one step by direct P2P synchronization between two workers. Moreover, P2P embedding synchronization can eliminate the potential network bottleneck caused by the PS architecture [41]. Herald can support P2P embedding synchronization by distinguishing synchronization targets during the communication plan generations. These targets finally become a receiving communication plan to indicate which embeddings would be received in an iteration for each worker.

C Herald vs. Brute-force Scheduling

To evaluate the effectiveness of Herald scheduler, we also implement a scheduler that leverages brute-force search as the baseline. The brute-force search scheduler explores all possible allocations and returns an allocation that incurs minimal embedding transmissions. Since the search space expands exponentially with the number of workers and per-worker batch size, we restrict the per-worker batch size to 1. We define the scheduling quality of Herald as the ratio of the number of embedding transmissions incurred by the brute-force search scheduler to that incurred by Herald.

We measure the scheduling quality and the scheduling time across different numbers of workers and report their average values for 90 iterations after 10 warm-ups used to fill the cache. For easy parallelism, the thread pool size is identical to the number of workers for both schedulers. Figure 20 demonstrates that Herald can achieve a scheduling quality from 0.45 to 0.57, indicating that there is still room to exploit embedding scheduling in the future. In terms of scheduling time, Herald consistently preserves a low scheduling overhead (< 0.7 ms). Meanwhile, the scheduling time of the brute-force search scheduler increases rapidly, consuming up to a few seconds when there are more than 10 workers in this tiny search space. This result reveals that the brute-force search scheduler is impractical for large-scale DLRM training.