ABSTRACT
Parameter servers (PSs) facilitate the implementation of distributed training for large machine learning tasks. In this paper, we argue that existing PSs are inefficient for tasks that exhibit non-uniform parameter access; their performance may even fall behind that of single node baselines. We identify two major sources of such non-uniform access: skew and sampling. Existing PSs are ill-suited for managing skew because they uniformly apply the same parameter management technique to all parameters. They are inefficient for sampling because the PS is oblivious to the associated randomized accesses and cannot exploit locality. To overcome these performance limitations, we introduce NuPS, a novel PS architecture that (i) integrates multiple management techniques and employs a suitable technique for each parameter and (ii) supports sampling directly via suitable sampling primitives and sampling schemes that allow for a controlled quality–efficiency trade-off. In our experimental study, NuPS outperformed existing PSs by up to one order of magnitude and provided up to linear scalability across multiple machine learning tasks.

CCS CONCEPTS
• Information systems → Parallel and distributed DBMSs; • Computer systems organization → Distributed architectures.

KEYWORDS
parameter servers, distributed machine learning, large-scale machine learning, skew, sampling

1 INTRODUCTION
To keep up with increasing dataset sizes and model complexity, distributed training has become a necessity for large machine learning (ML) tasks. Distributed training enables (i) scaling to models and datasets that exceed the memory of a single machine by distributing them to the nodes of a compute cluster and (ii) faster training by performing distributed compute. Usually, each node accesses only its local part of the training data, but requires global read and write access to all model parameters. Parameter management is thus a key concern in distributed ML. Parameter servers (PS) ease distributed parameter management by providing primitives for reading and writing parameters, while transparently handling partitioning and synchronization across nodes [2, 15, 24, 38, 60]. Many ML system stacks employ a PS as a core component, e.g., TensorFlow [1], MXNet [8], PyTorch BigGraph [37], STRADS [33], STRADS-AP [32], or Project Adam [10], and there are many standalone PSs, e.g., Petuum [24], PS-Lite [38], Angel [30], FlexPS [26], Glint [27], PS2 [72], Lapse [57], and BytePS [31].

As cluster nodes access parameters over the network, distributed training induces communication overhead. For some ML tasks, this overhead causes the performance of distributed implementations to even fall behind that of single node baselines [57]; Figure 1 depicts this exemplarily for a large knowledge graph embeddings task. We observe that a key cause for such poor performance can be non-uniform parameter access and focus on ML tasks where this is the case. We identify two main sources of non-uniformity: skew and sampling. First, in a workload that exhibits skew, a (typically small) subset of parameters is accessed frequently (e.g., up to 100,000 times per second), whereas a large part of the parameters is accessed rarely (e.g., only once every minute) [9, 12, 20, 21, 44, 47]. The main reason for skew is that real-world datasets often have skewed

<table>
<thead>
<tr>
<th>Run time (hours)</th>
<th>Relocation PS (Lapse)</th>
<th>Replication PS (Petuum)</th>
<th>Classic PS (as PS-Lite)</th>
<th>Non-uniform PS (NuPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>7h</td>
<td>13h</td>
<td>38h</td>
<td>NuPS</td>
</tr>
</tbody>
</table>

Figure 1: Parameter server (PS) performance for training large knowledge graph embeddings (ComplEx [65], dimension 500 on Wikidata5m data) on an 8-node cluster (8 worker threads per node). The performance of state-of-the-art PSs falls behind that of a single node (8 worker threads) due to communication overhead. NuPS improves performance by up to one order of magnitude. Details in Section 5.1.
frequency distributions (e.g., graphs [9, 20, 21], texts [47], and others [12, 44]), and many ML models associate specific parameters with specific data items (e.g., with the tokens in a text document or with the vertices of a graph) [35, 45, 49]. The second source of non-uniformity is sampling: for a subset of parameter accesses, random sampling (rather than training data) determines which parameters are read and written [3, 6, 40, 45, 55, 58, 59]. One common reason for this access pattern is negative sampling [3, 22, 45, 58], which, for example, is used to reduce the cost of many-class classification tasks or to mitigate an absence of negative training data (e.g., in recommender systems with only positive feedback or in knowledge graphs that contain only positive edges).

In this paper, we explore how to extend the scope of PSs to ML tasks that exhibit such non-uniform parameter access. To this end, we present NuPS, a novel non-uniform PS architecture. Figure 2 depicts an overview of this architecture. NuPS overcomes two key performance limitations of existing PSs. First, existing PSs are inefficient for managing skew because they employ one single management technique for all parameters. Using a single technique limits performance as none of the existing techniques is efficient for all access patterns. To overcome this limitation, NuPS introduces multi-technique parameter management, i.e., it integrates multiple parameter management techniques and chooses a suitable technique for each parameter. In particular, NuPS integrates both replication [14, 24] and relocation [57].

Second, existing PSs are inefficient for sampling because common parameter management techniques are ill-suited for randomly sampled access. To improve performance, applications can implement specialized sampling schemes manually, outside the PS [29, 37, 62, 73], but this limits the efficiency of some schemes, potentially produces incorrect samples, and causes repeated implementation effort. NuPS overcomes this limitation by integrating sampling schemes directly into the PS. To do so, NuPS extends the PS API with a sampling primitive that allows applications to request samples from a specific sampling distribution (rather than accessing specific parameters directly). NuPS’s sampling manager transparently chooses one of several sampling schemes to reduce communication overhead for sampling, according to a conformity level. Conformity levels provide a controlled trade-off between efficiency and sample quality.

In our experimental evaluation, NuPS outperformed state-of-the-art PSs by up to one order of magnitude and provided up to linear scalability across multiple ML tasks. Figure 1 exemplarily shows its performance for the task of training knowledge graph embeddings.

In summary, our contributions are as follows: (i) we evaluate the suitability of existing PSs under skew (Section 3.1), (ii) we propose multi-technique parameter management to handle skew efficiently (Section 3.2), (iii) we develop a hierarchy of conformity levels (Section 4.1) and analyze properties of common sampling schemes (Section 4.2), (iv) we argue for and propose a PS API extension for sampling (Section 4.3) and present how NuPS implements several schemes behind this API (Section 4.4), and (v) we experimentally investigate how these changes affect PS performance (Section 5).

2 NON-UNIFORM PARAMETER ACCESS

We study ML tasks that exhibit non-uniform parameter access. We identify two main sources of non-uniformity: skew (Section 2.1) and sampling (Section 2.2).

2.1 Skew

A workload exhibits skew non-uniformity when some parts of the model are accessed (much) more frequently than others. The main reason for this is that many real-world datasets have skewed frequency distributions [9, 12, 20, 21, 44, 46, 47]. For example, heavy skew is common in text corpora, because word frequencies are skewed [47], and in graph data, because in- and out-degree distributions are skewed [9, 20, 21, 46]. As many ML models associate specific parameters with specific data items (e.g. with words in a text or with the nodes of a graph) [22, 35, 45, 49], access to the parameters is heavily skewed, too: a small subset of hot spot parameters is accessed frequently, whereas the majority of parameters is accessed rarely. In the following, we will refer to the parameters that are not hot spots as long tail parameters.

We have measured the extent of skew for two real-world ML tasks: training knowledge graph embeddings and training word vectors. The left hand sides of Figures 3a and 3b show the number of reads per parameter over one epoch of these tasks, respectively. Access is heavily skewed: in the knowledge graph embeddings task, 18% of 12.9 trillion total reads go to only 0.02% of 4.8 billion parameters. In the word vectors task, 45% of 9 trillion total reads go to 0.17% of 1.9 billion parameters. Details on the tasks and datasets can be found in Section 5.1.

Note that skew is not always present in distributed training. For example, there is no skew in convolutional neural networks for image recognition [36] because model access is dense, i.e., every update step writes to all parameters. In contrast, in common neural network models for natural language processing [16, 25, 52], access is partially dense, and partially sparse and skewed: access to the first (embedding) layer and sometimes the last (classification) layer is based on word or token frequency (and thus sparse and skewed), and access to other layers is dense. The share of parameters with frequency-based access depends on the model architecture, but can be high, e.g., around 90% in ELMo [52]. In this paper, we investigate skew in shallow models, but conjecture that a non-uniform PS can also be beneficial for deeper models with partially skewed access.

2.2 Sampling

A workload exhibits sampling non-uniformity when, for a subset of parameter accesses, random sampling determines which parameters...
are read and written [3, 6, 40, 45, 53, 55, 58, 59]. I.e., the application randomly draws a parameter key from an application-specific sampling distribution over (all or a subset of) parameter keys. It then accesses the drawn key for training. We refer to such access as sampling access. In contrast, in direct access, the training data determines which parameters are accessed. Sampling access is common in many-class classification tasks, e.g., extreme classification [3], natural language processing [45], knowledge graph embeddings [40, 58], graph representations [22, 69], recommender systems [55], and when triplet loss is used [6, 59].

For example, knowledge graph embeddings and word vectors training tasks often use negative sampling to enable efficient training [45, 53, 58]. For each (positive) data point, a set of negative samples is drawn from a distribution. Each negative sample corresponds to a data item (e.g., a word) or a class. The corresponding parameters are subsequently accessed for training. For instance, the example knowledge graph embeddings task draws negative samples from a uniform distribution over all entities [40, 58]. The right hand side of Figure 3a shows the frequency distributions of direct and sampling accesses separately for this task. In our implementation (based on [45]) and with 200 negative samples for each subject–relation–object triple (100 negative samples for the subject and another 100 for the object), sampling accesses make up 31% of all accesses. In the word vectors task, negative samples correspond to words and the sampling distribution resembles the word frequencies in the training data [45], see Figure 3b. In the plot for direct access, parameters that belong to the output layer of the task’s neural network are visually distinct from the other parameters. The reason for this is that the task draws samples only from the output layer, and parameters in the plot are sorted by total access frequency. In our implementation (based on [45]) and with 3 negative samples for each word–word pair, sampling accesses make up 56% of all parameter accesses in this task.

3 MULTI-TECHNIQUE PARAMETER MANAGEMENT

In this section, we analyze the suitability of existing PSs for ML tasks with skewed parameter access (Section 3.1) and argue that existing PSs are inefficient for managing skew because they employ one single management technique for all parameters. Based on this analysis, we propose multi-technique parameter management and discuss NuPS’s implementation (Section 3.2).

3.1 Analysis of Common Parameter Management Techniques

PSs [2, 15, 24, 26, 27, 30, 31, 38, 57, 60, 72] partition the model parameters across a set of servers. The PS provides pull and push primitives for global reads and writes to model parameters, respectively. In the data-parallel setting, the training data are partitioned to a set of workers. During training, each worker processes its local part of the training data (often multiple times) and continuously reads and updates model parameters. To coordinate parameter accesses across workers, each parameter is assigned a unique key. Many PSs physically co-locate the (logically distinct) servers and workers on the same node for efficiency, either in multiple processes per node [27, 30, 38] or within one process [24, 26, 57].

Several techniques have been proposed for managing parameters among the cluster nodes in a PS. In the following, we discuss common techniques, briefly introducing each before analyzing its suitability for managing skew.

3.1.1 Classic PS. A classic PS allocates parameters to servers statically (e.g., via range partitioning of the parameter keys) and uses no replication [2, 38, 60]. Thus precisely one server holds the current value of a parameter, and this server is used for all pull and push operations on this parameter. Classic PSs typically guarantee sequential consistency for operations on the same key [57].

Analysis: The performance of a classic PS is limited for both hot spots and long tail parameters. The reason for this is that every parameter access uses the network: it incurs network latency for two messages (to and from the responsible server) and the parameter value is sent over the network once (from the server to the worker in a pull operation, in the other direction for a push operation). This network overhead is incurred for all parameters, i.e., hot spot and long tail ones. For hot spots, the overhead is incurred many times for a few parameters. In the long tail, the overhead is incurred a few times for each of many parameters.

3.1.2 Replication PS. A replication PS replicates parameters and tolerates some amount of staleness in the replicas [13, 14, 24, 26, 30]. Replication PSs provide weaker consistency guarantees, such as bounded staleness, and require applications to explicitly control staleness via special primitives (e.g., an “advance the clock” operation). There are two main protocols for creating and refreshing replicas in general-purpose PSs: SSP [24] creates a replica when a parameter is accessed and uses this replica until the staleness bound is reached (at which point the replica is terminated). ESSP [14] also creates a
replica when a parameter is (first) accessed, but then maintains this replica throughout the entire training task (by repeatedly propagating updates). In both SSP and ESSP, nodes accumulate replica updates locally and propagate them to the responsible server at the subsequent “advance the clock” invocation. A subset of replication PSs specifically target deep learning workloads in which each node holds replicas of all parameters and replicas are updated synchronously after each step of mini-batch stochastic gradient descent [23, 28, 31, 70]. In contrast to NuPS, these PSs focus on workloads in which (i) the model size does not exceed the memory capacity of a single node and (ii) synchronous replica updates are not prohibitively slow w.r.t. to computational cost; some further apply only to GPU-based training [31].

**Analysis:** A replication PS is efficient for hot spots, but its benefit for the long tail is limited. Replication reduces network overhead (compared to a classic PS) if a replicated parameter value is used more than once and multiple updates can be sent to the PS in aggregated form. Replication further reduces access latency if a parameter value (within the acceptable staleness bound) is already locally available when a read operation is issued. Both is typically the case for hot spot parameters, even within relatively tight staleness bounds (because hot spot parameters are accessed frequently at each node). In contrast, long tail parameters are accessed infrequently. So it is unlikely that a long tail parameter is accessed more than once within reasonable staleness bounds (large staleness bounds commonly deteriorate model convergence [24]). For the same reason, SSP (which creates replicas on demand) does not reduce access latency for long tail parameters, because replicas are mostly “cold”. With its eager replica maintenance, ESSP ensures that replicas are always “warm” (after the first access to a parameter), but at the cost of significant over-communication: ESSP constantly updates all replicas, although replicas for long tail parameters are accessed rarely.

### 3.1.3 Relocation PS

A relocation PS asynchronously re-allocates parameters among nodes during run time so that access operations can be processed locally, without network communication [57]. Relocation PSs require applications to control allocation via special primitives (e.g., a “localize” operation). As classic PSs, relocation PSs can provide per-key sequential consistency [57].

**Analysis:** A relocation PS is efficient for long tail parameters, but has limited benefit for hot spots. Relocation eliminates access latency if there is sufficient time to relocate a parameter between accesses at different nodes. It further reduces network overhead (compared to classic) if a parameter is accessed more than once between two relocations (which is common, most ML tasks at least read and write a parameter): a relocation takes three messages in Lapse [57] (including the parameter value once), whereas each remote access in a classic PS sends two messages (including the parameter value once). There is typically sufficient time for relocating long tail parameters between accesses by different nodes, as they are accessed infrequently. Hot spot parameters, however, are frequently accessed at multiple nodes concurrently [56]. Thus, there is not sufficient time for relocations between accesses, such that access latency is not eliminated. Further, a relocation PS incurs higher network overhead than a classic PS if a parameter is relocated so frequently that only one operation is processed locally.

### 3.1.4 Summary

Individual management techniques are efficient for either hot spot or long tail parameters (or neither of the two), but none is efficient for both. Consequently, managing all parameters with the same technique limits the performance of PSs for ML tasks with skewed parameter access.

### 3.2 Parameter Management in NuPS

From the above discussion, it follows naturally to explore whether combining multiple management techniques is beneficial for PS performance. The idea of combining multiple management techniques has been studied in other (non-PS) distributed data management systems, such as general-purpose distributed databases [11, 17, 19, 67] and distributed graph processing systems [42]. These systems combine static allocation with replication, but do not consider relocation. To the best of our knowledge, integrating multiple management techniques in PSs has not been explored before.

NuPS integrates two management techniques: replication and relocation. First, to manage hot spot parameters efficiently, NuPS integrates a lightweight variant of eager replication [14]. NuPS eagerly creates replicas for hot spot keys on all nodes and provides time-based staleness bounds. Basing the staleness bound on time rather than clocks alleviates the need for adding “advance the clock” operations to application code, but potentially complicates the analysis of convergence properties. We discuss these implications below. Second, to manage long tail parameters efficiently, NuPS integrates relocation. As Lapse [57], NuPS asynchronously relocates these parameters before they are accessed. This guarantees per-key sequential consistency for long tail parameters. NuPS picks a technique for each key based on the key’s access pattern: if the key is accessed frequently, NuPS replicates the key; if there are few accesses, NuPS employs relocation (see Section 5.1). The choice of management technique is transparent to the application, i.e., the application accesses all parameters in the same way, via the push and pull primitives. Our experimental evaluation shows that the combination of replication and relocation can be highly beneficial. Integrating other techniques (e.g., highly tailored ones) may further improve performance, but is beyond the scope of this paper. NuPS does not integrate the classic technique as it is dominated by replication for hot spots and by relocation for the long tail.

For efficiency, NuPS co-locates workers and servers in one process per node, and accesses replicas and locally allocated parameters via shared memory. Figure 4 depicts an overview. To access a key, a...
worker checks whether this key is managed by replication or relocation. If the key is managed by replication, the worker accesses the key via shared memory, without network communication. If the key is managed by relocation, the worker checks whether the key is currently allocated locally. If so, it accesses the key via shared memory. Otherwise, the worker accesses the parameter remotely, using the message protocol proposed in Lapse [57]: a request to the node that knows where the parameter is currently allocated, which then forwards the request to this node, which in turn processes the request and sends a response to the worker.

NuPS is designed to minimize the run time overhead of providing multiple management techniques. To do so, NuPS integrates the check for the management technique and the check for local allocation into one latch acquisition (i.e., a lock held for the duration of the API call). Further, NuPS can be reduced to a single-technique PS with no measurable run time overhead for providing more than one management technique: If replication is not used for any key, the replica synchronization background thread exits immediately, without sending any messages. If relocation is not used for any key, no messages are sent for relocation.

NuPS bases its staleness bounds on time rather than logical clocks because this makes the PS easier to use: time-based bounds alleviate the need for adding “advance the clock” operations to application code and for timing them appropriately. NuPS synchronizes the replicas periodically, using sparse all-reduce operations (i.e., only updated parameters are exchanged [64]). The synchronization is run by a background thread and uses the recursive doubling algorithm. However, time-based bounds potentially complicate the analysis of convergence properties. If only a bounded number of SGD steps can occur within one synchronization round, bounded staleness holds (as for clock-based staleness bounds) and the corresponding analysis carries over [24]. However, if the number of SGD steps within one synchronization round cannot be bounded, convergence analyses for asynchronous SGD apply [39, 71]. In our experiments, the effect of time-based bounds on performance was minimal because we used replication only for a small number of parameters and synchronized replicas frequently (see Section 5.6 and Section 5.7).

4 SAMPLING MANAGEMENT

Existing PSs provide no support for sampling. This means that applications manually sample keys and then access the corresponding parameters via direct access, which leads to significant communication overhead. To reduce this overhead, many applications implement a variety of sampling schemes [29, 37, 62, 73]. The key idea of such sampling schemes is that slightly (or sometimes rather significantly) deviating from the ideal of independent sampling from the desired target distribution might have only little or no effect on model quality, but can reduce communication overhead substantially (and consequently speed up model training). The lack of sampling support in PSs forces applications to implement such schemes in application code, outside the PS. This leads to repeated implementation effort and potentially produces incorrect samples. Further, this precludes schemes that require tight integration with parameter management.

In contrast to existing PSs, NuPS integrates sampling directly into the PS. In the following, we present the components of this integration. We first introduce a set of conformity levels that allow for a controlled trade-off between efficiency and sample quality (Section 4.1). We then analyze conformity and communication overhead of sampling schemes that are commonly used by applications (Section 4.2). Based on this analysis, we propose an API extension that enables sampling in PSs (Section 4.3) and discuss how NuPS implements several sampling schemes, within this API (Section 4.4).

4.1 Sampling Conformity Levels

Let \( \pi \) be a target distribution over parameter keys. We assume that \( \pi \) is specified by the application and remains fixed throughout run time.\(^1\) For example, in the word vectors training task of Section 2.2, the target distribution \( \pi \) roughly corresponds to relative word frequencies [45], cf. Figure 3b. When training knowledge graph embeddings, \( \pi \) is often a uniform distribution over all entities [38]; cf. Figure 3a. Denote by \( \mathcal{K} \) the set of parameter keys and by \( \pi_k \geq 0 \) the target probability for key \( k \in \mathcal{K} \), where \( \sum_{k=1}^{\mathcal{K}} \pi_k = 1 \). Workers repeatedly draw one or more samples from the target distribution \( \pi \). Denote by \( X_q \subseteq \mathcal{K} \) a random variable for the \( i \)-th sample obtained at node \( q \).\(^2\) We write \( N_q \) for the number of samples drawn at node \( q \) during the complete run time of some application. Set \( X_q = \{ X_{q1}, \ldots, X_{qN_q} \} \) and \( X = \bigcup_q X_q \).

We propose a hierarchy of four sampling conformity levels to control the trade-off between sample quality and efficiency. From the top (L1) to the bottom (L4) of this hierarchy, sample quality decreases, and potential efficiency increases:

**L1 CONFORM.** The sampling scheme produces mutually independent samples from the target distribution \( \pi \). I.e.,

\[
p(X_qi = k | S) = \pi_k
\]

for all \( q, i, k \) and \( S \subseteq X \setminus \{ X_{qi} \} \).

**L2 BOUNDED.** The samples at each node have dependencies on past samples, but these dependencies are limited and samples at different nodes are independent. In more detail, given a dependency bound \( B \in \mathbb{N} \), it holds

\[
p(X_qi = k | S^B_q, S_{-q}) = \pi_k
\]

for all \( q, i, k \), where \( S_{-q} \subseteq X \setminus X_q \) refers to samples at other nodes and \( S^B_q \subseteq \{ X_{qi1}, \ldots, X_{qi(B-1)} \} \) refers to samples at node \( q \) taken from all but the last \( B \) samples taken so far. Note that first-order inclusion probabilities match the target probabilities—i.e., \( p(X_qi = k) = \pi_k \)—even though subsequent samples may be dependent. For example, a sampling scheme that internally draws independent samples from \( \pi \) but uses each sample twice is BOUNDED with \( B = 1 \).

**L3 LONG-TERM.** The mean first-order inclusion probabilities match the target probabilities asymptotically at each node, i.e.,

\[
\lim_{N_q \to \infty} \frac{1}{N_q} \sum_{i=1}^{N_q} p(X_qi = k | X_{qi1}, \ldots, X_{qi(i-1)}) = \pi_k
\]

for all \( q, k \). Note that this does not imply \( p(X_qi = k) = \pi_k \). Also, arbitrary dependencies between samples within one or across

\(^1\)This is mainly to facilitate analysis; an application may use multiple different sampling distributions, each of which can be analyzed separately.

\(^2\)Depending on the implementation, there can be multiple workers on each node. We analyze sampling schemes at the node level to simplify exposition.
Table 1: Conformity levels of common sampling schemes.

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent sampling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sample reuse</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Local sampling</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Direct-access repurposing</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

multiple nodes are accepted as long as the asymptotic relative frequencies of the samples match the target. For example, a sequential sampling scheme that selects a random key order for the \(|K|\) keys and then draws samples in a round-robin fashion satisfies \textit{LONG-TERM} but not \textit{BOUNDED}: each key is selected equally often in the long run, but the knowledge of the first \(|K|\) samples allows to uniquely determine all future samples, so that no dependency bound can be established.

\textbf{(L1) NON-CONFORM}. No guarantees about the sampling probabilities or independence.

The levels are hierarchical in that L1 implies L2, and L2 implies L3. The first implication follows since we can set \(S = S_q^B \cup S_{-q}\) for any choice of \(S_q^B\) and \(S_{-q}\).

\textbf{Proof}. L2 implies L3. Starting from some offset \(1 \leq o \leq B\), fix some node \(q\) and consider the subset \(\{X_q(\pi + o)\}_{\pi \in \Omega}\) of every \(B\)-th sample on node \(q\), starting from the \(o\)-th sample. Using the definition of \textit{BOUNDED}, we obtain

\[
\frac{1}{((N_q - o)/B)} \sum_{\pi=1}^{[(N_q - o)/B]} p(X_q(\pi + o) = k | X_q1, \ldots, X_q(\pi + o - B)) = \pi_k
\]

for any choice of \(N_q\), i.e., the long-term relative frequencies of every \(B\)-th sample match if we start at offset \(o\). Since this holds for every offset \(o\), we conclude that Eq. (1) holds and L3 implies L1.

Note that we defined L3 via Eq. (1) rather than a simpler first-order probability condition such as \(p(X_qi = k) = \pi_k\), because correct first-order conditions are not sufficient to ensure that a sampling scheme is useful in practice. For example, a sampling scheme that internally draws one independent sample \(X\) from \(\pi\), and then uses solely this sample throughout (i.e., \(X_qi = X\) for all \(q, i\)) satisfies such a condition, but is clearly unsuitable in practice.

### 4.2 Analysis of Common Sampling Schemes

ML applications employ a variety of sampling schemes. In the following, we analyze schemes that are common in distributed training [29, 34, 37, 62, 73] w.r.t. their effect on (i) communication overhead and (ii) sampling quality, i.e., into which conformity level they fall. Table 1 provides an overview of the latter.

\textbf{Independent sampling}. Ideally, applications draw iid. samples from the target distribution and use each sample once. This scheme is \textit{CONFORM}, but can lead to significant communication overhead: for each sample, the corresponding parameter values need to be transferred to the node and, after an update is computed, updates need to be propagated to other nodes.

\textbf{Sample reuse}. Sample reuse reduces communication overhead by using each sample multiple times [5, 29, 37, 73]. For example, knowledge graph embeddings training can use shared sampling, i.e., reuse negative samples for all positive examples in a mini-batch [5]. Reusing a sample multiple times avoids the transfer of parameter values for another, fresh sample: using a sample \(U\) times can reduce the communication overhead by a factor of \(U\). We refer to this factor as the \textit{use frequency} and to a sample reuse scheme that uses each sample \(u\) times as \(U = u\) \textit{sample reuse}. Sample reuse does not provide \textit{CONFORM} since samples are not independent. However, it can provide \textit{BOUNDED}. For example, if each fresh sample is sampled iid. from \(\pi\) and then used exactly \(U\) times, then the scheme is \textit{BOUNDED} for all \(B \geq U\). Moreover, in mini-batch negative sample reuse as in [5, 29, 37], \textit{BOUNDED} also holds. Here samples are reused only within one mini-batch of gradient descent so that the mini-batch size provides a bound on the sample dependency.

\textbf{Local sampling}. In many distributed ML architectures [14, 24, 26, 57], at each node, a distinct subset of the model parameters—the \textit{local partition}—can be accessed without network communication. \textit{Local sampling} restricts sampling accesses to this local partition [73]. This scheme eliminates network overhead for sampling accesses entirely. However, local sampling is \textit{NON-CONFORM} as nodes see only samples from the local partition. Some implementations re-partition parameters periodically such that all nodes at least see all samples over time [73]. Careful re-partitioning might satisfy Eq. (1) for certain target distributions; e.g., if \(\pi\) is uniform and parameters are allocated uniformly and at random. In general, however, local sampling cannot provide \textit{LONG-TERM}. For example, consider any target distributions in which \(\pi_k > 1/Q\) for some \(k\) (with \(Q\) being the number of nodes). Local sampling cannot satisfy Eq. (1) for such a target since key \(k\) is available for sampling at only one node at a time. This implies that there is at least one node at which the long-term frequency of \(k\) is \(\leq 1/Q\).

\textbf{Direct-access repurposing}. Another sampling scheme is to repurpose direct-access parameters, i.e., to use them as negative samples. For example, DGL-KE [73] generates some of the samples by repurposing parameters that occur as positives in other data points of an SGD mini-batch. This requires no additional communication for sampling accesses, as the values for the direct access parameters are transferred to the node either way. In this scheme, the relative frequency of a seeing a key in a sample depends on the occurrence frequency of the key in the training data. As the training data occurrence distribution can be (and typically is [5, 45]) different from the target distribution, this scheme is \textit{NON-CONFORM}.

### 4.3 A Primitive for Sampling

It is impossible for PSs to integrate these sampling schemes within the push/pull API. The main problem is that sampling is done by application code: to conduct a sampling access, an ML application draws a sample of keys and accesses them via \textit{pul11} or \textit{push}. For instance, this makes it impossible for the PS to restrict sampling to the local partition. Further, the PS cannot even distinguish between direct access (for which it \textit{cannot} leverage sampling schemes) and sampling access (for which it \textit{can} leverage sampling schemes).

To overcome these limitations, we propose to extend the PS API with a sampling primitive that allows applications to access a
We now discuss the schemes in turn. The sampling manager is responsible for generating samples and localizes the corresponding examples iid. from the target distribution and localizes the corresponding parameters.

The argument \( \pi \) picks a scheme that is suitable for the specified conformity level. From the four implemented schemes, the sampling manager managing the corresponding parameters. The sampling manager \( \text{PrepareSample} \) returns.

When the background thread generates a new pool, it localizes the corresponding parameters. NuPS localizes the parameters again in \( \text{PrepareSample} \) if they have been relocated to another node since pool preparation. In \( \text{PullSample} \), NuPS accesses the parameters remotely if necessary. This sample reuse scheme is BOUNDDED because samples are drawn iid. from the target distribution \( \pi \), inter-sample dependency is bounded by \( U \cdot G \), and \( U \) is identical for all samples.

The background thread determines automatically when to prepare a new pool. Adding a new pool takes time and (for good performance) the localization should be finished when \( \text{PullSample} \) is called. This time depends on the ML task, the used hardware, and the system configuration. To estimate this time, we use a heuristic. In particular, the background thread keeps track of the duration of previous pool relocations. If the number of prepared, but unused samples is less than double of the current estimated relocation time, the preparation of another pool is triggered.

### Sample reuse (BOUNDDED).
NuPS implements a sample reuse scheme that reuses pools of keys. The pooling increases the temporal distance between the reused samples and thereby increases randomness. For a given pool size \( G \) and use frequency \( U \), NuPS repeatedly samples \( G \) keys iid. from \( \pi \) to form a sample pool and produces samples by traversing the sample pool \( U \) times, each time in a random order. For example, consider \( U = 2 \) and suppose that the iid. draws produce keys \( k_1, k_2, \text{and} \ k_3 \), respectively. With \( G = 1 \), we obtain sample sequence \( k_1k_2k_3k_3k_2 \). With \( G = 3 \), a sequence such as \( k_1k_2k_3k_2k_1k_3 \) is possible. The pools are prepared by a background thread. When the background thread generates a new pool, it localizes the corresponding parameters. NuPS localizes the parameters again in \( \text{PrepareSample} \) if they have been relocated to another node since pool preparation. In \( \text{PullSample} \), NuPS accesses the parameters remotely if necessary. This sample reuse scheme is BOUNDDED because samples are drawn iid. from the target distribution \( \pi \), inter-sample dependency is bounded by \( U \cdot G \), and \( U \) is identical for all samples.

The background thread determines automatically when to prepare a new pool. Adding a new pool takes time and (for good performance) the localization should be finished when \( \text{PullSample} \) is called. This time depends on the ML task, the used hardware, and the system configuration. To estimate this time, we use a heuristic. In particular, the background thread keeps track of the duration of previous pool relocations. If the number of prepared, but unused samples is less than double of the current estimated relocation time, the preparation of another pool is triggered.

### Sample reuse with postponing (LONG-TERM).
NuPS additionally implements sample reuse with sample postponing. This is identical to the described sample reuse scheme, but adds sample postponing: if sample \( i \) cannot be accessed locally in \( \text{PullSample} \), NuPS re-localizes the corresponding parameters, postpones sample \( i \) for later use, and uses sample \( i + 1 \) instead. To achieve LONG-TERM, it is crucial that, at some point, samples are used (and not re-postponed indefinitely). Thus, NuPS postpones only within the \( N \) samples of one invocation of \( \text{PrepareSample} \) (in other words, only within the samples of one handle). I.e., when NuPS finds a non-local sample (in \( \text{PullSample} \)), it moves the sample to the end of the \( N \) samples of this handle. NuPS postpones each sample maximally once. When it reaches samples that it has already postponed once (towards the end of the \( N \) samples), it accesses them remotely if necessary. This implementation of postponing reduces communication overhead only if the samples of one handle are pulled in groups smaller than \( N \) and there is some time between these partial pulls for the parameter relocation. Assuming that \( N \) is bounded from above, it provides LONG-TERM. It does not provide BOUNDDED because sampling probabilities depend on the current allocation of a key (i.e., keys can be postponed to a later sample if they are not local).

Fig. 5: Sampling scheme implementations in NuPS.
Local sampling (NON-COMFORM). NuPS implements local sampling without active re-partitioning. Instead, NuPS relies on the application to relocate parameters: in a relocation PS, the local partition usually changes constantly, as workers relocate the parameters that they work with (in direct access). The effect of this local sampling variant heavily depends on the relocations of the application. Generally, this approach cannot give any guarantees, as, for example, an application might not relocate parameters at all. Thus, it generally falls into the NON-COMFORM level. In an ideal setting, however, this approach could provide LONG-TERM. For example, this can be the case if an application partitions its training data randomly and continuously relocations all parameters (such that a parameter is equally likely to be on all nodes) and samples uniformly (such that \( n_k \sim \frac{1}{Q} \) for all \( k \)). To make local sampling efficient, NuPS employs a fast sampling implementation that does not sample independently.

5 EXPERIMENTS

We conducted an experimental study to investigate whether and to what extent a non-uniform PS is beneficial for PS performance. Source code, datasets, and information for reproducibility are available online.\(^5\)

In our study, we compared the performance of NuPS to several state-of-the-art PSs on three large-scale ML tasks (Section 5.2). Further, we conducted an ablation study (Section 5.3), investigated scalability (Section 5.4), evaluated different sampling schemes (Section 5.5), and explored specific components of NuPS (Sections 5.6 and 5.7). Our major insights are: (i) NuPS was more than an order of magnitude faster than existing PSs. (ii) NuPS achieved best performance when it replicated a small fraction of the model parameters, and relocated all other parameters. (iii) Both sample reuse and local sampling significantly reduced communication overhead for sampling access. We conclude that a non-uniform PS is key for high performance in ML tasks with non-uniform parameter access.

5.1 Experimental Setup

We considered three popular ML tasks that require long training: knowledge graph embeddings, word vectors, and matrix factorization. These tasks are representatives for shallow models that exhibit sparse and skewed access. The tasks differ in multiple ways, including the number of parameters, parameter access distributions, sampling distribution, and frequency of sampling accesses. Table 2 provides a summary. In the following, we briefly discuss each task.

Knowledge graph embeddings. Knowledge graph embedding (KGE) models learn algebraic representations of the entities and relations in a knowledge graph. For example, representations have been applied successfully to infer missing links in knowledge graphs [48]. This task, based on [40], trains ComplEx [65] (one of the most popular KGE models) embeddings using SGD with Adam [18] and negative sampling [40, 58]. Negative sampling creates sampling access in this task: to generate negative samples, both the subject and the object entity of a positive triple are perturbed \( n_{neg} \) times, by drawing random entities from a uniform distribution over all entities (we used a common setting of \( n_{neg} = 100 \) [58]). We used the Wikidata5M dataset [66], a real-world knowledge graph with 4 818 679 entities and 828 relations, and a common embedding size of 500 [58]. We partitioned the subject–relation–object triples of the dataset to the nodes randomly, as done in [34]. We used LibKGE [5] (commit 3146885) to evaluate models and report the mean reciprocal rank (filtered) (MRRF) as metric for model quality.

Word vectors. Word vectors (WV) are a language modeling technique in natural language processing: each word of a vocabulary is mapped to a vector of real numbers [45, 51, 52]. These vectors are useful as input for many natural language processing tasks, for example, syntactic parsing [61] or question answering [41]. This task, based on [45], uses SGD and negative sampling to train the skip-gram Word2Vec [45] model (dimension 1000) on the One Billion Word Benchmark [7] dataset (with stop words of the Gentisim [34] stop word list removed). The negative sampling creates sampling accesses: in this task, for each word pair, 3 negative samples are drawn from a distribution that is based on word frequencies (see Section 2.2). We used common model parameters [45] for window size (5), minimum count (1), and frequent word subsampling (0.01). We measured model accuracy using a common analogical reasoning task of 19 544 semantic and syntactic questions [45].

Matrix factorization. Low-rank matrix factorization (MF) is a common tool for analyzing and modeling dyadic data, e.g., in collaborative filtering for recommender systems [35]. This task, based on [63], uses SGD to factorize a synthetic, zipf 1.1 distributed 10m \( \times \) 1m dataset with 1b revealed cells, modeled after the Netflix Prize dataset.\(^6\) Data points were partitioned to nodes by row and to workers within a node by column. Each worker visited its data points by column (to create locality in column parameter accesses), with random order of columns and of data points within a column. There is no sampling access in this task. We report the root mean squared error (RMSE) on the test set as metric for model quality.

Baselines. We compared performance to a classic PS, to Petuum (a state-of-the-art replication PS), to Lapse (a state-of-the-art relocation PS), and to a single node implementation. As classic PS, we used Lapse with relocation disabled, which, according to [57], provides performance similar to PS-Lite. We ran both the SSP and

\(^5\)https://github.com/alexrenz/NuPS

\(^6\)See https://netflixprize.com/. We use a synthetic dataset because the largest openly available dataset that we are aware of is only 7.6 GB large.
ESSP protocols of Petuum [68], with different staleness thresholds. Petuum does not provide KGE or WV implementations. Thus, we implemented the KGE task described above in Petuum. We used version 1.1 of Petuum. We did not implement specific sampling schemes in application code, i.e., applications draw independent samples and access them via direct access. We used a shared memory implementation with 8 worker threads as single node baseline.

**Implementation and cluster.** We implemented NuPS in C++, using ZeroMQ and Protocol Buffers for communication, based on PS-Lite [38]. We used a local cluster of up to 16 Lenovo ThinkSystem SR630 computers, running Ubuntu Linux 20.04, connected with 100 Gbit Infiniband. Each node was equipped with two Intel Xeon Silver 4216 16-core CPUs, 512 GB of main memory, and one 2 TB D3-S4610 Intel SSD. We compiled code with g++ 9.3.0, except for Petuum, which we compiled with g++ 7.5.0, as the compilation with g++ 9.3.0 failed. Unless specified otherwise, we used 8 nodes and 8 worker threads per node. In Lapse and NuPS, we additionally used 1 server and 3 ZeroMQ I/O threads per node. In Petuum, we used 4 communication channels per node. To prevent exploding gradients, we used gradient norm clipping as suggested in [50] for replicated parameters in the WV and MF tasks (clipping updates that exceed the average norm by more than 2x). In the KGE task, the use of AdaGrad prevented exploding gradients. For each task, we tuned hyperparameters on the single node and used the best found hyperparameter setting in all systems and variants.

**NuPS**. We ran NuPS in two configurations: (i) a generally applicable **untuned** configuration that requires no task-specific tuning and (ii) a task-specific tuned configuration. The untuned configuration employs a heuristic to decide the management technique for each parameter: it replicates a parameter if its access frequency exceeds 100 times the mean access frequency. This heuristic is computed from dataset frequency statistics. The untuned configuration further employs sample reuse without postponing (BOUND) with a use frequency of U=16. To indicate the performance potential of task-specific insights, we included a tuned configuration by informing our configuration choices with the results of our detailed experiments in Sections 5.5 and 5.6. The tuned configuration for KGE replicates the 900 most frequently accessed keys (the same as the untuned setting), but uses local sampling (NON-CONFIRM). The tuned configuration for WV replicates the 209k most frequently accessed keys (64x more keys than the untuned configuration), and employs local sampling (NON-CONFIRM). For MF, the untuned configuration seemed to be near-optimal, such that we did not add a separate tuned configuration. Unless mentioned otherwise, we used the settings of the untuned configuration and a replica staleness threshold of 40 ms in all experiments. Throughout all experiments, we used a pool size of 250 in the sample reuse scheme.

**Measures.** Unless noted otherwise, we ran all variants with a fixed 6 h time budget. We measured model quality over time and over epochs within this time budget (using the quality metrics described above). We conducted 3 independent runs of each experiment, each starting from a distinct randomly initialized model, and report the mean. We depict error bars for model quality and run time; they present the minimum and maximum measurements. In

![Figure 6: End-to-end performance of different PSs on 8 nodes. NuPS outperformed Petuum (a state-of-the-art replication PS) and Lapse (a state-of-the-art replication PS) by up to one order of magnitude and provided up to linear scalability over the single node. The gray shaded area indicates performance that is dominated by the single node. Error bars depict minimum and maximum measurements for run time and model quality (but are often not visible due to low variance). The dashed gray line depicts the model quality threshold at which effective speedups are computed (90% of the best observed single-node quality).](image_url)
some experiments, error bars are not clearly visible because of small variance. Gray dotted lines indicate the performance of the single node baseline. Gray shading indicates performance that is dominated by the single node baseline. We report two types of speedups: (i) raw speedup depicts the speedup in epoch run time, without considering model quality; (ii) effective speedup is calculated from the time that each variant took to reach 90% of the best model quality that the single node baseline achieved. Unless specified otherwise, we report effective speedups.

5.2 Overall Performance

We investigated the overall effect of a non-uniform PS on PS performance. To do so, we compared the performance of NuPS to existing PSs and to the single node baseline. We ran each variant for the fixed time budget and measured model quality over this time. Figures 6a, 6b, and 6c show model quality over time, Figures 6d, 6e, and 6f show model quality over epoch. In summary, NuPS was 31–36x faster than a state-of-the-art replication PS (Petuum), 6–46x faster than a state-of-the-art relocation PS (Lapse), and 2.3–10.3x faster than the single node baseline.\(^7\)

The classic PS was inefficient (with epochs over 7x slower than the single node) because it accesses parameters over the network, which induced significant access latency. Lapse was faster than Classic, but still slower than the single node, because Lapse relocates all parameters, including hot spots. Hot spot parameters, however, are frequently accessed by multiple nodes simultaneously [56], such that some of these nodes had to wait for the relocation to finish or access the parameter remotely, which induced access latency. The per-epoch model quality of Classic and Lapse was indistinguishable from the single node in KGE and WV, as these systems provide sequential consistency for all parameters and employ no specialized sampling schemes. In MF, all distributed variants provided lower per-epoch quality than the single node, an effect that has been observed before [43]. The step pattern that is visible in MF training stems from the bold driver heuristic [4] that the MF implementation [43] that we adapted uses to tune the learning rate.

For KGE, we ran Petuum SSP and ESSP with staleness thresholds 1, 10, 100, 200, or 1000, and tried different frequencies for advancing the clock.\(^8\) None of the configurations completed the first epoch within the time budget of 6 hours. We observed the best performance for ESSP with staleness 10, which finished the first epoch after 13h with a model quality (MRRF) of 0.11. The best SSP run (staleness 200) finished the first epoch after 15h with a model quality of 0.10. The reasons for this performance are that Petuum is inefficient for long tail parameters (as discussed in Section 3.1) and that Petuum’s replica approach is inefficient for sampling because sampling access provides no locality: SSP replicas are mostly cold, ESSP over-communicates. Petuum’s MF implementation ran out of memory, because it stores the training matrix in dense format.

The untuned NuPS configuration outperformed existing PSs across all three tasks. For KGE and MF, it was also clearly faster than the single node, with up to 6.7x effective speedups over the single node and minimal negative effect on (per-epoch) model quality. For WV, however, it barely outperformed the single node (but still outperformed existing PSs). In contrast, the tuned configuration provided 4.6–10.3x effective speedups over the single node across all three tasks. For KGE, the tuned configuration of NuPS provided better per-epoch convergence than the single node. This was an effect of local sampling; see Section 5.5 for more details.

5.3 Ablation

NuPS introduces two novel features compared to existing PSs: (i) multi-technique parameter management and (ii) the integration of sampling into the PS. To investigate individual effects, we enabled each feature individually and measured model quality within the time budget. Figure 7 shows the results. We omit MF because there is no sampling access in MF, such that the entire performance improvement stems from multi-technique parameter management (which is visible in Figure 6c). We found that both multi-technique parameter management and sampling integration can be beneficial individually, and the individual benefits compounded when both were combined.

We compared the performance of four variants: (i) Lapse, a relocation PS without sampling integration; (ii) Relocation + Replication, a PS with multi-technique parameter management but without sampling integration; (iii) Relocation + Sampling, a relocation-only PS with sampling integration; (iv) NuPS, a multi-technique PS with sampling integration. Going from a single-technique relocation PS to a multi-technique PS made an epoch 67–73% faster with only small effect on model quality. Adding sampling support to the relocation PS made an epoch 17–62% faster, with a small negative effect on model quality. The combination of both made an epoch 94% faster, with a small negative effect on per-epoch model quality.

5.4 Scalability

To investigate scalability, we ran Lapse, the best Petuum SSP and ESSP configurations, and NuPS for one epoch on 1, 2, 4, 8, and 16 nodes and calculated the raw speedup. Figure 8 depicts the results. Further, we ran convergence experiments on 16 nodes for those systems that reached the 90% model quality threshold on 8 nodes. Figure 9 depicts the effective speedup for these systems. Overall,
NuPS scaled more efficiently than other PSs, with up to near-linear raw and up to superlinear effective speedups.

We first discuss raw scalability, i.e., the speedup w.r.t. epoch run time (Figure 8). On a single node, NuPS and Lapse were faster than Petuum because NuPS and Lapse access local parameters via shared memory, whereas Petuum sends intra-process messages to do so. Lapse provided poor scalability because the more nodes are used, the higher the chance that multiple nodes access a parameter at the same time and, thus, that they have to wait for a relocation to finish or to access parameters remotely. Neither Petuum ESSP nor SSP outperformed the shared-memory single-node baseline, even on 16 nodes. ESSP scaled poorly even when compared to its own (inefficient) run time on a single node (4.8x faster on 16 nodes) because its eager replication protocol over-communicates: after a short warm-up period, each node holds a replica of the full model. The more nodes, the more replicas had to be synchronized, such that replica synchronization became a bottleneck. The lazy replication protocol of SSP scaled better than ESSP compared to its own (inefficient) run time on a single node (12x faster on 16 nodes), but its overall performance was poor because its replicas were cold most of the time (and thus required synchronous replica refreshes).

NuPS scaled more efficiently than existing PSs because it (i) limits the bottleneck of eager replication by replicating only a small subset of hot spot parameters, (ii) prevents the majority of relocation conflicts by employing relocation only for long tail parameters, and (iii) employs sampling schemes to reduce sampling communication overhead. With 16 nodes, it provided up to 13.4x raw speedups over the shared memory single node. NuPS further provided up to 20x effective speedups for KGE and 8x for MF (see Figure 9). For WV, although the raw speedup on 16 nodes was 10.2x, the effective speedup was only 2.2x. The reason for this is that we used the hyperparameter configuration that worked best on the single node throughout all experiments. With other hyperparameters, we observed better effective speedups for WV.

5.5 Effect of Sampling Schemes

We investigated the effect of different sampling schemes in NuPS on run time and model quality. To do so, we ran KGE and WV with different sampling schemes: independent sampling (CONFORM), U=16 and U=64 sample reuse without postponing (BOUNDED) and with postponing (LONG-TERM), and local sampling (NON-CONFORM). Figures 10a and 10b show model quality over time, Figures 10c and 10d show model quality over epoch. We omit MF as it does not contain sample reuse. We further omit the results from sample reuse of independent sampling conflicts by employing relocation only for long tail parameters, and (iii) employs sampling schemes to reduce sampling communication overhead. With 16 nodes, it provided up to 13.4x raw speedups over the shared memory single node. NuPS further provided up to 20x effective speedups for KGE and 8x for MF (see Figure 9). For WV, although the raw speedup on 16 nodes was 10.2x, the effective speedup was only 2.2x. The reason for this is that we used the hyperparameter configuration that worked best on the single node throughout all experiments. With other hyperparameters, we observed better effective speedups for WV.

5.5 Effect of Sampling Schemes

We investigated the effect of different sampling schemes in NuPS on run time and model quality. To do so, we ran KGE and WV with different sampling schemes: independent sampling (CONFORM), U=16 and U=64 sample reuse without postponing (BOUNDED) and with postponing (LONG-TERM), and local sampling (NON-CONFORM). Figures 10a and 10b show model quality over time, Figures 10c and 10d show model quality over epoch. We omit MF as it does not contain sample reuse. We further omit the results from sample reuse with postponing as its results were within 10% of sample reuse without postponing.

We found that both sample reuse and local sampling led to significant speedups over independent sampling, with small negative or—in the case of local sampling—even positive effects on per-epoch model quality.

Postponing made no measurable difference in KGE, and sped up WV run times by 10%, with no measurable impact on model quality.
We investigated how the choice of management technique, i.e., the choice of whether to replicate or relocate a key, affects the performance of NuPS. The NuPS untuned heuristic replicates the 900 most frequent keys in KGE, the 3272 most frequent keys in WV, and the 755 most frequent column keys in MF. We varied these numbers by factors $\frac{1}{8}$, $\frac{1}{4}$, 1, 4, 16, 64, and 256. The leftmost columns of Table 3 depict what share of keys was replicated for each setting. We ran one epoch of each setting and measured epoch run time and model quality. Figure 11 depicts the results. We found that it was crucial for performance to replicate "enough" parameters such that the set of hot spot parameters is managed by replication, but not too many parameters, as replication created significant over-communication for long tail parameters.

This effect was visible for all tasks: starting from no replicated keys (i.e., all keys managed by relocation), increasing the number of replicated keys first improved run time, and had minimal effect on model quality. However, after some point, replicating more keys deteriorated model quality, and even slowed down run time for KGE and MF. The reason for the negative effect on model quality was that the replicas were stale, because the replica updates became too large to synchronize them frequently over the network of the cluster. We configured NuPS to provide the default 40 ms staleness bound (i.e., 25 synchronizations per second), but to not block operations when it did not reach this goal. Figure 11 includes the actual synchronization frequency if model quality was not within 10% of the model quality without replication. The middle columns of Table 3 provide the size of the replicated values for all settings. For example, the 64x WV setting replicated 799 MB of parameter values. Large numbers of replicated keys led to slower epoch run times for KGE and MF, because relocation operations competed with replica synchronization for network bandwidth. This effect was not visible for WV because, in WV, the majority of accesses went to replicated keys (and, thus, were fast despite network congestion). The share of accesses that went to replicas is depicted in the rightmost columns of Table 3. For example, 88% of all accesses went to replicas in the 64x WV setting.

### 5.7 Effect of Replica Staleness

We investigated the effect of replica staleness on epoch run time and model quality. To do so, we varied synchronization frequency: we synchronized replicas either 125, 25, 5, 1, or 0.2 times per second or not at all. We ran one epoch of each setting and measured epoch run time and model quality after this epoch. Note that without replica synchronization, nodes may hold different models. In these cases, we evaluated the model of the first node. Figure 12 reports the results. Overall, replication had only minimal effect on model quality when replica staleness was low.

Replication had only small effect on model quality when replicas were synchronized at least 5 times per second. In contrast, infrequent synchronization (less than once per second) deteriorated model quality drastically in KGE and WV. However, infrequent synchronization (or no synchronization at all) worked well in some settings (in particular in MF). We theorize that the reason for this was that NuPS employs replication for only a small subset of parameters, such that replication parameters are kept synchronized indirectly through the parameters that are managed by relocation.

### 5.8 Comparison to Task-Specific Implementations

In a general-purpose system, a performance overhead over optimized task-specific implementations is expected. To investigate the extent of this overhead in NuPS, we compared to specific implementations for each task. Each of these implementations is specialized and highly tuned for the respective task. In contrast to a general-purpose PS such as NuPS, these implementations cannot be used to run other ML tasks. Note that some of these implementations use different, more complex training algorithms than the implementations in NuPS. Overall, we found that NuPS was competitive to specialized and tuned task-specific implementations.
For MF, we compared to the highly tuned MPI implementations of DSGD and DSGD++ [63]. We ran convergence experiments on 8 and 16 nodes. We measured how long the implementations took to reach the 90% quality threshold. We used the same hyperparameters, model starting points, and learning rate schedule across DSGD, DSGD++, and NuPS. On 8 nodes, NuPS was 16% faster than DSGD and 15% slower than DSGD++. On 16 nodes, NuPS was 37% faster than DSGD and 16% faster than DSGD++.

For KGE, we compared to the highly specialized framework PyTorch-BigGraph [37]. Note that PyTorch-BigGraph is designed for a different training algorithm, with different hyperparameters: to reduce communication overhead, it uses mini-batch SGD, whereas the KGE implementation in NuPS employs regular SGD (i.e., batch size 1). To minimize the impact of algorithm hyperparameters in our comparison, we compared epoch run times. NuPS ran an epoch in 12 minutes on 16 nodes (24 minutes on 8 nodes). In this setting (i.e., batch size 1), PyTorch-BigGraph was much slower than NuPS: it took more than 5 hours to run one epoch, both on 8 and 16 nodes. Using a very large batch size led to faster epochs in PyTorch-BigGraph (up to 3x faster with batch size 1000 than NuPS with batch size 1), but can also be implemented in NuPS.

For WV, we are not aware of a highly tuned and publicly available distributed implementation, so we compared to two highly tuned single-node implementations: the original C implementation of Word2Vec [45] and Gensim [54]. The implementation in Gensim and the one in NuPS are both based on the original C implementation. For both single-node implementations, we achieved the fastest epoch run times with 64 threads. Gensim completed an epoch in 15 minutes, the original implementation in 12 minutes. With 8x8 threads, NuPS took 13.5 minutes for one epoch; with 16x8 threads, it took 8 minutes. One factor that limits the performance of NuPS compared to these task-specific implementations is that—as other general-purpose PSs [57]—NuPS provides per-key atomic updates. To achieve this, workers receive dedicated working copies of parameters. Creating these copies and writing updates back into the parameter store creates overhead compared to the task-specific WV implementations, which let workers read and write in the parameter store directly, without any consistency or isolation guarantees. Empirically, this works well for this particular task, but the effects for other tasks in a general-purpose system are unclear.

6 CONCLUSION

We explored how to extend the scope of PSs to ML with non-uniform parameter access. To this end, we presented NuPS, a non-uniform PS that employs multi-technique parameter management to efficiently handle skew and integrates sampling schemes to efficiently handle sampling. We found that a non-uniform PS can be highly beneficial: in our experimental study, NuPS outperformed existing PSs by up to one order of magnitude. These results open up several research directions for further improving PS performance: integrating more management techniques (e.g., highly specialized ones), developing further sampling schemes, and devising fine-grained methods for picking management techniques for parameters.

ACKNOWLEDGMENTS

This work was supported by the German Ministry of Education and Research in the Software Campus (01IS17052) and BIFOLD (01IS18037A) programs and by the German Research Foundation in the moreEVS (410830482) program.