Good Intentions:
Adaptive Parameter Management via Intent Signaling

Alexander Renz-Wieland
Technische Universität Berlin

Andreas Kieslinger
Technische Universität Berlin
BIFOLD

Robert Gericke
Technische Universität Berlin
BIFOLD

Rainer Gemulla
Universität Mannheim

Zoi Kaoudi
IT University Copenhagen

Volker Markl
Technische Universität Berlin
BIFOLD

ABSTRACT

Model parameter management is essential for distributed training of large machine learning (ML) tasks. Some ML tasks are hard to distribute because common approaches to parameter management can be highly inefficient. Advanced parameter management approaches—such as selective replication or dynamic parameter allocation—can improve efficiency, but they typically need to be integrated manually into each task's implementation and they require expensive upfront experimentation to tune correctly. In this work, we explore whether these two problems can be avoided. We first propose a novel intent signaling mechanism that integrates naturally into existing ML stacks and provides the parameter manager with crucial information about parameter accesses. We then describe AdaPM, a fully adaptive, zero-tuning parameter manager based on this mechanism. In contrast to prior parameter managers, our approach decouples how access information is provided (simple) from how and when it is exploited (hard). In our experimental evaluation, AdaPM matched or outperformed state-of-the-art parameter managers out of the box, suggesting that automatic parameter management is possible.

CCS CONCEPTS

• **Computer systems organization** → *Distributed architectures.*

KEYWORDS

machine learning systems, distributed training, parameter servers

ACM Reference Format:
Alexander Renz-Wieland, Andreas Kieslinger, Robert Gericke, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. 2023. Good Intentions: Adaptive Parameter Management via Intent Signaling. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3583780.3614895

1 INTRODUCTION

Distributed training is essential for large-scale machine learning (ML) tasks. It enables (i) training of models that exceed the memory capacity of a single node and (ii) faster training by leveraging the compute of multiple nodes of a cluster. Typically, each node accesses one (local) partition of the training data, but requires global read and write access to all model parameters. Thus, *parameter management* (PM) among nodes is a key concern in distributed training. We refer to any system that provides distributed PM, i.e., that provides global parameter access across a cluster, as *parameter manager.* Most ML systems include a parameter manager as core component [1, 6, 22].

Some large-scale ML tasks are particularly hard to distribute because standard PM approaches are infeasible or inefficient. Currently, the most widely used approach is full static replication, i.e., to maintain a copy of the entire model at each cluster node. Static full replication is infeasible when the model size exceeds the memory capacity of a single node. It is also inefficient when the task accesses model parameters *sparsely*, i.e., when each update step reads and writes only a small subset of all parameters. This is because it synchronizes updates for all parameters to all nodes, even though each node accesses only a small subset at each point in time. Such sparse access is common in natural language processing [9, 19, 24, 25], knowledge graph embeddings [3, 4, 21, 31], some graph neural networks [29, 30, 32], click-through-rate prediction [7, 10, 35, 41], and recommender systems [5, 12, 15]. [1] Another standard PM approach—static parameter partitioning—partitions model parameters and transfers parameters to where they are needed on demand. This approach is often inefficient because of the access latency that this ad-hoc transfer induces [27]. Figure 1 shows that the performance of both these approaches (blue and red lines) falls behind that of a single node baseline for an example ML task, thus necessitating more advanced PM.

Advanced PM approaches can improve efficiency [8, 11, 27, 39], but to do so, they require information about the underlying ML task. However, in current ML systems, this information is not readily available to the parameter manager. To work around this lack of information, existing advanced PM approaches require application developers to control critical PM decisions manually, by exposing configuration choices and tuning knobs. This requirement makes these PM approaches complex to use, such that their adoption in common ML systems remains limited. For example, multi-technique

[1]For example, sparsity may arise due to sparse features (e.g., binary features or embedding layers of tokens, entities, or vertices) or due to sampling (e.g., negative sampling for classification or neighborhood sampling in GNNs).

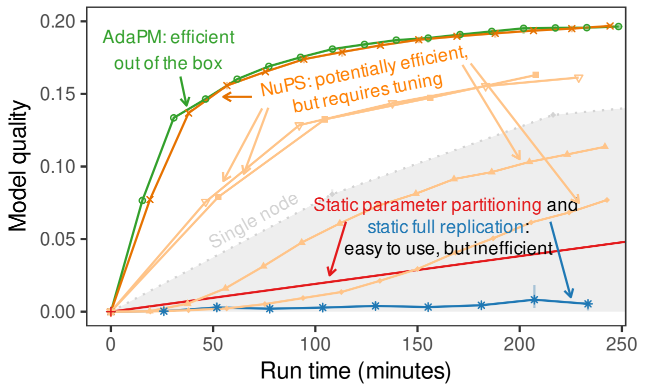


Figure 1: Performance for training large knowledge graph embeddings (ComplEx, dimension 500 on Wikidata5m) on an 8-node cluster. Static full replication and static parameter partitioning are easy to use, but inefficient. NuPS can be more efficient, but is hard to use. AdaPM is easy to use and efficient. Details in Section 5.1.

parameter managers [26, 39] use different PM techniques (e.g., replication and dynamic allocation) for different parameters. This can be more efficient than simple PMs (e.g., see the performance of NuPS in Figure 1), but application developers need to specify upfront which technique to use for which parameter, and—for optimal performance—need to tune these technique choices. The absence of a general-purpose approach that is both simple to use and efficient gives rise to a range of custom, task-specific approaches that tightly couple PM to one specific ML task or one class of ML tasks [2, 16, 18, 23, 38, 42]. These approaches can be efficient, but they are not general-purpose leading to similar techniques being re-developed and re-implemented many times.

In this paper, we propose *intent signaling*, a general-purpose mechanism for providing the information that is necessary to enable tuning-free advanced PM for training with (some) sparse parameter access. Intent signaling cleanly decouples the application from PM: the application passes information about which parameters it *intends* to access before it does so, and the parameter manager acts based on these signals. We designed intent signaling in a way that can naturally be integrated into existing ML systems: the data loader can signal intent during batch preparation. We show that intent signaling not only simplifies the use of existing advanced PM approaches, it also enables a range of novel, more fine-grained, parameter managers that so far were impractical because PM had to be controlled manually.

In addition, we develop AdaPM, a proof-of-concept parameter manager that is based on intent signaling. AdaPM is (i) *fully adaptive*, i.e., it dynamically adapts all critical decisions to the underlying ML task, and (ii) *zero-tuning*, i.e., it adapts automatically, based purely on intent signals. To do so, AdaPM continuously re-evaluates what is the most efficient way to manage a specific parameter in the current situation (e.g., whether relocating the parameter is more efficient than replicating it). As the timing of PM actions (e.g., when to start maintaining a replica at a node) significantly affects performance, AdaPM continuously optimizes its action timing by learning from past timing decisions. AdaPM currently provides automatic PM among the main memory of cluster nodes.[2]

We conducted an experimental evaluation on five ML tasks: training knowledge graph embeddings, word vectors, matrix factorization, click-through-rate prediction models, and graph neural networks. AdaPM was efficient across all tasks without requiring any tuning. In addition, it matched or even outperformed existing, manually tuned PM approaches. Figure 1 shows one example of this performance: AdaPM out of the box matched the performance of the best NuPS configuration, which was tuned specifically for this ML task.

In summary, our contributions are: (i) we analyze the complexity of existing PM approaches (Section 2), (ii) we propose intent signaling (Section 3), (iii) we describe AdaPM (Section 4), and (iv) we investigate the performance of AdaPM in an experimental study (Section 5).

2 EXISTING APPROACHES

Several general-purpose PM approaches aim to improve efficiency by adapting specific aspects to the underlying ML task. This adaptation requires information about the task. As this information is not available to the parameter manager in current ML systems, existing approaches rely on the application developers to manually control adaptation, which makes these approaches complex to use. In this section, we briefly discuss existing approaches, which aspects they adapt, and what makes them complex to use. Table 1 gives an overview. A more detailed analysis that also discusses efficiency can be found in Appendix A of the long version of this paper [28].

We discussed static full replication and static parameter partitioning in Section 1. These are implemented in many ML systems. For example, TensorFlow [1] implements the two in the *mirrored* and *parameter server* distribution strategies, respectively; PyTorch [22] implements them in the *distributed data parallel* and *fully sharded data parallel* wrappers, respectively. They are easy to use, but their efficiency is limited, as discussed in Section 1.

Selective replication [37] statically partitions parameters, and, to improve efficiency, selectively replicates parameters to further nodes during training. Existing approaches [8, 11, 36] are complex to use because they require applications to manually tune a staleness threshold parameter that affects both model quality and run time efficiency for each ML task.

Dynamic parameter allocation [27] partitions parameters to nodes and, to improve efficiency, dynamically relocates parameters to the nodes that access them during training. Existing approaches are complex to use because they require the application to initiate parameter relocations manually in the application code and to tune the *relocation offset* (i.e., how long before an actual access the relocation is initiated).

Multi-technique parameter managers [14, 26, 39] support multiple PM techniques (e.g., replication, static partitioning, or dynamic allocation) and use a suitable one for each parameter. They are complex to use because they require the application to specify upfront which technique to use for which parameter. For

[2]Workers may, of course, use GPUs to process batches. Direct support for GPU memory in AdaPM may provide further efficiency improvements and is a key direction for future work.

Table 1: Approaches to distributed PM: adaptivity, ease of use, and efficiency for sparse workloads. Existing approaches adapt only individual system aspects to the underlying ML task and require applications to manually control these adaptations.

	Adaptivity					
Approach	Replication	Parameter location	Choice of technique	Timing	Ease of use	Efficiency
Static full replication	static (full)	static	single	none	++	--
Static parameter partitioning (PS-Lite)	none	static	single	none	++	--
Selective replication (Petuum)	adaptive	static	single	by application	-	-
Dynamic allocation (Lapse)	none	adaptive	single	by application	--	-
Multi-technique PM (BiPS, Parallax)	static (partial)	static	static	none	--	+
Multi-technique PM (NuPS)	static (partial)	adaptive	static	by application	--	+
AdaPM (this paper)	adaptive	adaptive	adaptive	adaptive	+	++

optimal performance, application developers additionally need to tune these technique choices manually.

3 INTENT SIGNALING

To enable automatic adaptive PM, we propose *intent signaling*, a novel mechanism that naturally integrates into ML systems. It passes information about upcoming parameter access from the application to the parameter manager. It *decouples* information from action, with a clean API in between: the application provides information (intent signals); the parameter manager transparently adapts to the workload based on the intent signals. In other words, all PM-related decision-making and knob tuning is done transparently by the parameter manager. The application *only signals intent.*

An intent is a declaration by one worker that this worker intends to access a specific set of parameters in a specific (logical) time window in the future. A natural choice of time window is a training batch. For example, a worker may signal that it will require access to, say, only 1M out of 100M parameters in batch 2 after analyzing the examples in that batch (e.g., embeddings of categories that do not appear in the batch are not needed).[3] Such an approach integrates naturally with the data loader paradigm of common ML systems — such as Pytorch's data loader [22], TensorFlow's data sets [1], or the Gluon data loader [6] —, where training batches are constructed upfront in one or more separate threads and then queued until processed by the training thread(s) later on. This process is illustrated in Figure 2.

In general, it is important that intent is signaled before the parameter is actually accessed, such that the parameter manager can adapt proactively. We use logical clocks as a general-purpose construct for specifying the start and end points of intents. To ensure generality, each worker i has one logical clock C^i that is independent of other workers' clocks.[4] Each worker advances its clock with an `advanceClock()` primitive (as done in Petuum [37], but, in contrast to Petuum, invocation of our `advanceClock()` is cheap, as it only raises the clock). For example, a worker could advance its clock whenever it starts processing a new batch.

We propose the following primitive for signaling intent:

$$\texttt{Intent(parameters, } \texttt{C}_{\texttt{start}}\texttt{, } \texttt{C}_{\texttt{end}} \texttt{ [, type])}$$

[3]For example, in the click through-rate prediction task of Sec. 5, processing a batch size with 10000 examples requires accesses to less than 0.7% of all parameters on average.
[4]Applications may, of course, synchronize worker clocks.

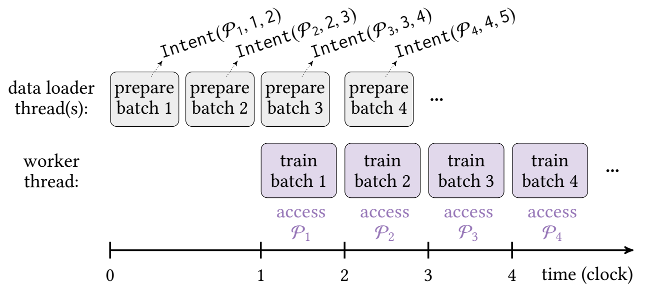


Figure 2: The data loader is a natural place to integrate intent signaling. After a batch i is constructed, the data loader signals intent for the parameters $\mathcal{P}_i$ that are accessed in batch i.

With this primitive, a worker signals that it intends to access a set of parameters in the time window between a start clock $\texttt{C}_{\texttt{start}}$ (inclusive) and an end clock $\texttt{C}_{\texttt{end}}$ (exclusive). The primitive allows to (optionally) specify intent type, e.g., *read*, *write*, *read+write*. Figure 2 illustrates an example of how the data loader can signal intent. With $\texttt{Intent}(\mathcal{P}_2, 2, 3)$, the data loader signals that the corresponding worker intends to access parameters $\mathcal{P}_2$ (e.g., 1M out of 100M parameters) when at clock 2 (i.e., while it trains on batch 2). We say that an intent is *inactive* if it is signaled, but the worker has not reached the start clock yet, i.e., $\texttt{C}^{\texttt{i}} < \texttt{C}_{\texttt{start}}$. We say that an intent is *active* if the worker clock is within the intent time window, i.e., $\texttt{C}_{\texttt{start}} \leq \texttt{C}^{\texttt{i}} < \texttt{C}_{\texttt{end}}$. And we say that an intent is *expired* when the worker clock has reached the end clock, i.e., when $\texttt{C}_{\texttt{end}} \leq \texttt{C}^{\texttt{i}}$. Invocation of the `Intent` primitive is meant to be cheap, i.e., it should not slow down the worker, even if the worker signals a large number of intents. Workers can flexibly combine intents: they can signal multiple (potentially overlapping) intents for the same parameter, extend one intent by signaling another later on, etc.

Intent signaling enables the PM to continuously adapt its parameter management strategy. This is in contrast to most existing PM approaches, where applications directly or indirectly control which or when actions such as selective replication are performed. As we will discuss in Section 4, intent signals allow AdaPM to choose suitable PM techniques dynamically during run time (as opposed to statically per parameter) and to time actions appropriately (as opposed to explicitly triggering actions). The key benefits of intent signaling are (i) ease of use, as applications do not need to make

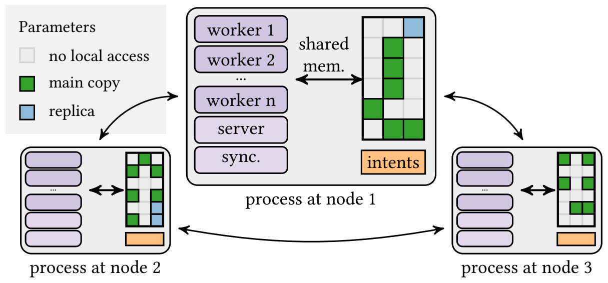


Figure 3: AdaPM architecture. For efficiency, AdaPM runs multiple worker threads in one process per node, and accesses locally available parameters via shared memory.

these hard choices (but only signal intent), and (ii) improved efficiency, as the PM can make better choices by carefully deciding which and when to perform actions.

4 THE ADAPM PARAMETER MANAGER

Intent signaling opens a large design space for adaptive PM. Key design questions include: when and where to maintain replicas, whether and when to change parameter allocation, when to act on intent signals, how to synchronize replicas, how to exchange intent signals, on which nodes to make decisions, and how to communicate efficiently. We explore this space and describe AdaPM, a parameter manager that—in contrast to previous ones—is *fully adaptive*, i.e., it adapts *all* critical aspects to the underlying ML task, see Table 1. It does so *automatically*, based purely on intent signals, thus requiring no user input or knob tuning. And it does so *dynamically*, continuously re-evaluating what the most efficient approach is for the current situation. AdaPM has low overhead —a major design goal—and it takes all management decisions off the critical path of workers.

We give a brief overview of key design choices before we discuss the most challenging ones in detail. Figure 3 illustrates the architecture of AdaPM. We assume an architecture that—for efficiency—co-locates the parameter store in the same processes as the worker threads, as parameter managers commonly do [11, 13, 27].

Adaptive choice of technique. AdaPM employs dynamic allocation and selective replication, and adaptively picks between the two per parameter. AdaPM dynamically chooses the most efficient technique for the current situation. Intuitively, AdaPM relocates a parameter if—at one point in time—only one node accesses the parameter. Otherwise, it creates replicas precisely where they are needed. We discuss AdaPM's choice of technique in Section 4.1.

Adaptive action timing. AdaPM learns automatically when the right time to act on an intent signal is. This ensures that applications do not need to fine-tune the timing of their intent signals. They can simply signal their intents early, without sacrificing performance. See Section 4.2 for details.

Responsibility follows allocation. In AdaPM, the node that currently holds the main copy of a parameter (owner node) takes on the main responsibility for managing this parameter: it decides how to act on intent signals and acts as a hub for replica synchronization. For efficiency, this responsibility moves with the parameter whenever the it is relocated. This strategy generally reduces network communication as the responsibility for a parameter is always at a node that "needs" it. Details can be found in Appendix B.1 of the long version of this paper [28].

Efficient communication. AdaPM communicates for exchanging intent signals, relocating parameters, and managing replicas. To communicate efficiently, AdaPM locally aggregates intents, groups messages when possible to avoid small message overhead, and employs location caches to improve routing. Details can be found in Appendix B.2 of the long version of this paper [28].

Optional intent. Intent signals are optional in AdaPM. An application can access any parameter at any time, without signaling intent. However, signaling intent makes access more efficient, as it allows AdaPM to avoid synchronous network communication.

4.1 Adaptive Choice of Technique

AdaPM receives intent signals from workers. Based on these intent signals, AdaPM tries to ensure that a parameter can be accessed locally at a node while this node has active intent for this parameter. To achieve this, AdaPM has to determine where to ideally allocate a parameter, if and where to create replicas, and for how long to maintain each replica.

To make parameters available locally, AdaPM employs (i) dynamic allocation and (ii) selective replication. That is, it (i) can relocate parameters among nodes and (ii) can selectively create replicas on subsets of nodes for specific time periods. AdaPM uses a simple heuristic rule to decide between the two: if, at one point in time, only *one* node has active intent for a parameter, AdaPM relocates the parameter to the node with active intent. After the node's intent expires, AdaPM keeps the parameter where it is until some other node signals intent. In contrast, when multiple nodes have active intent for one parameter concurrently, AdaPM selectively creates a replica at each of the nodes when the intent of that node becomes active. It destructs the replica when the intent of that node expires. The simplicity of the decision rule allows AdaPM to communicate intent among nodes efficiently. Figure 4a illustrates AdaPM's decision between relocation and selective replication. More involved approaches (or cost models) may provide further benefits, but this simple strategy performed remarkably well in our experimental study.

Let us consider three exemplary intent scenarios:

(1) Two nodes have intent for the same parameter, and the active phases of the intents do not overlap; see Figure 4b. AdaPM relocates the parameter from its initial allocation to the first node with intent and keeps it there even after the intent expires. AdaPM relocates the parameter to the second node with intent shortly before the node's intent becomes active.
(2) Two nodes have intent for the same parameter, and the active periods of the intents partially overlap; see Figure 4c. AdaPM relocates the parameter to the first node with intent, then creates a replica on the second node while the two active intents overlap, and finally relocates the parameter to the second node after the intent of the first node expires.
(3) Multiple nodes repeatedly have intent for the same parameter, see Figure 4d. AdaPM creates replicas on all nodes with

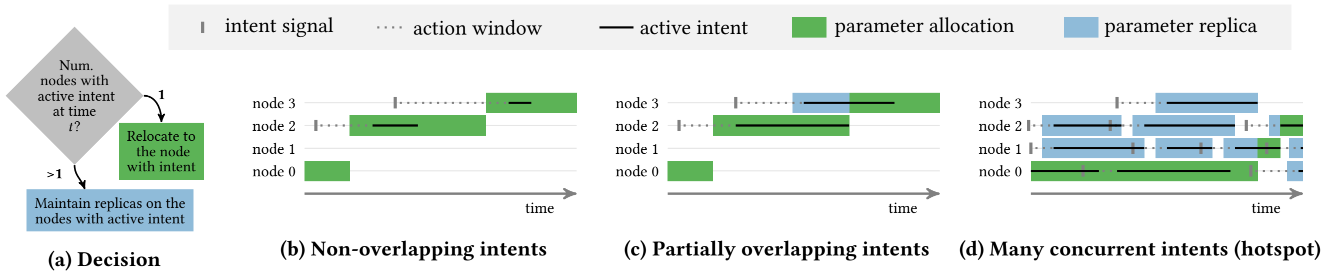

intent signal
action window
active intent
parameter allocation
parameter replica
Num. nodes with active intent at time t?
1
Relocate to the node with intent
>1
Maintain replicas on the nodes with active intent
(a) Decision
node 3
node 2
node 1
node 0
time
(b) Non-overlapping intents
(c) Partially overlapping intents
(d) Many concurrent intents (hotspot)

Figure 4: AdaPM decides automatically whether to relocate or replicate a parameter at any time t (a). Example scenarios (b–d).

active intent. Whenever there is exactly one node with active intent (and the parameter is not currently allocated at this node), AdaPM relocates the parameter to this node.

AdaPM combines parameter relocation and replication because previous research has shown that their combination is beneficial [26]. Relocation is efficient for parameters that are accessed infrequently, as in the first example above, because the parameter value is transferred over the network only once per access (from where the parameter currently is to where the intent is). In contrast, replication can significantly reduce network overhead for frequently accessed parameters. In addition, unlike previous multi-technique approaches [26, 39], AdaPM employs *selective* replication: AdaPM creates a replica precisely for the time during which it is needed. This increases efficiency as AdaPM does not need to maintain replicas while they are not needed. This also limits the impact of stale reads — which always arise when replication is used — since (i) AdaPM replicates as few parameters and as selectively as possible and continuously refreshes them (cf. Tab. 2) and (ii) no staleness occurs for reads of any non-replicated parameter. Further, unlike previous approaches, the choice of PM techniques for each parameter is dynamic: AdaPM can relocate the parameter at one point in time, and replicate it at another.

For making its decisions, AdaPM treats all intent types identically. More complex approaches could tailor their choices to intent type. For example, systems could choose to take different actions for *read* and *write* intents. In AdaPM, we keep the system simple and treat all intent types identically because we do not expect tailoring to improve performance for typical ML workloads: (i) applications typically both read *and* write a parameter and (ii) synchronous remote reads are so expensive that it is beneficial to provide a locally accessible value for a parameter even for a single read.

4.2 Adaptive Action Timing

AdaPM receives intent signals before the intents become active. I.e., there is an *action window* between the time the intent is signaled and the time the intent becomes active. AdaPM needs to determine at which point in this action window it should start to act on the intent signal, i.e., when it relocates the parameter or sets up a replica for this parameter. For example, consider the intent of node 3 in Figure 4c: AdaPM needs to figure out at which point in time it starts maintaining a replica on node 3.

Relocating a parameter or setting up a replica takes some time. Consequently, if AdaPM acts too late and relocation or replica setup is not finished in time, remote parameter access is required. On the other hand, if AdaPM acts too early, it might maintain a replica longer than needed, inducing unnecessary communication. Furthermore, if AdaPM acts too early, it might use replication in scenarios in which—with better timing—relocation would have been both possible and more efficient. However, acting on an intent signal (slightly) too early is much cheaper than acting too late because the remote accesses caused by too late action slow down training significantly. In contrast, acting slightly too early merely causes over-communication. Thus, it is better to err on the side of acting too early.

The key challenge is that both (i) the *preparation time* for relocation or replica setup and (ii) the length of the action window are unknown. The action window length is unknown because it is unclear when the worker will reach the intent's start clock. Both times are affected by many factors, e.g., by the application, the compute and network hardware, and the utilization of that hardware.

4.2.1 *Learning When to Act.*

AdaPM aims to *learn* when the right time is to act on an intent signal. A general approach would estimate both preparation time and action window length separately. However, AdaPM acts on intent signals in point-to-point communication rounds that take a fairly constant amount of time. AdaPM thus simplifies the general approach and directly estimates the number of worker clocks per communication round. This allows AdaPM to decide whether an intent signal should be included in the current round or if it suffices to include the signal in a later round. A later round suffices if the *next* round will finish before the worker reaches the start clock of the intent.

As acting (slightly) too early is much cheaper than acting too late, our goal is to estimate a soft upper bound for the number of clocks during one communication round. I.e., we want to be confident that the true number of clocks only rarely (ideally, never) exceeds this soft upper bound. To this end, we employ a probabilistic approach in AdaPM: we assume that the number of clocks follows a Poisson distribution, estimate the (unknown) rate parameter for the distribution from past communication rounds, and use a high quantile of this Poisson distribution as a soft upper bound (e.g., the 0.9999 quantile). In detail, we assume that the number of clocks by worker i in round t follows Poisson(λ_t^i) with expected rate λ_t^i. We choose a Poisson distribution because it is the simplest, most natural assumption, and it worked well in our experiments. Note that we assume a Poisson distribution for a short time period (one round t by one worker i), not one global distribution (a much stronger, unrealistic assumption, which, for example, would not account for changes in workload or system load).

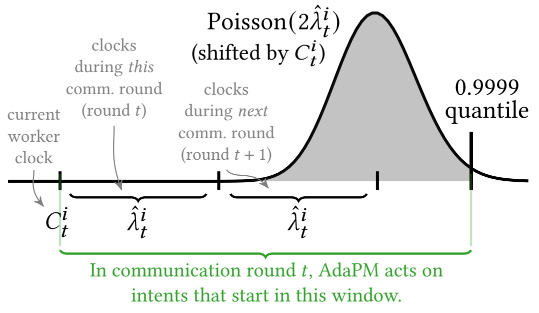

Poisson($2\hat{\lambda}_t^i$)
(shifted by C_t^i)
clocks during *this* comm. round (round t)
clocks during *next* comm. round (round $t+1$)
current worker clock
0.9999 quantile
C_t^i
$\hat{\lambda}_t^i$
$\hat{\lambda}_t^i$
In communication round t, AdaPM acts on intents that start in this window.

Figure 5: AdaPM learns automatically when to act on an intent signal, using a probabilistic model to estimate a soft upper bound.

```
Input: Intent start $C_{\text{start}}$, previous estimate $\hat{\lambda}_{t-1}^i$, clock of
  worker $i$ at the start of round $t$ ($C_t^i$) and round $t-1$ ($C_{t-1}^i$),
  smoothing factor $\alpha$, quantile $p$.
1: $\Delta \leftarrow C_t^i - C_{t-1}^i$
2: if $\Delta > 0$ then
3:     $\hat{\lambda}_t^i \leftarrow (1-\alpha)\hat{\lambda}_{t-1}^i + \alpha(\Delta)$ ;
4: else
5:     $\hat{\lambda}_t^i \leftarrow \hat{\lambda}_{t-1}^i$ ;
6: end if
   return $C_{\text{start}} < C_t^i + Q_{\text{Poiss}}(2 \cdot \max(\hat{\lambda}_t^i, \Delta), p)$;
```
Algorithm 1: Whether to act on a given intent in round t.

AdaPM acts on a given intent in round t if it estimates that the corresponding worker might reach the start clock of the intent (C_{start}) before round $t+1$ finishes, i.e., roughly[5] if

$$C_{\text{start}} < C_t^i + Q_{\text{Poiss}}(2 \cdot \lambda_t^i, p)$$

where C_t^i is the current clock of worker i at the start of round t and $Q_{\text{Poiss}}(\lambda, p)$ computes the p quantile of a Poisson distribution with rate parameter λ. Figure 5 illustrates this decision. Under our Poisson assumption, a $p = 0.9999$ quantile gives a 99.99% probability that the actual number of clocks during the two rounds is below our estimate.

4.2.2 *Estimating the Rate Parameter.*

Naturally, the true Poisson rate λ_t^i is unknown. AdaPM estimates this rate from the number of clocks in past communication rounds, using exponential smoothing:

$$\hat{\lambda}_t^i \leftarrow (1-\alpha)\hat{\lambda}_{t-1}^i + \alpha(C_t^i - C_{t-1}^i)$$

where $\hat{\lambda}_t^i$ is the estimate for the number of clocks by worker i in round t and α is the smoothing factor.

We consider two further aspects to improve the robustness of the estimate $\hat{\lambda}_t^i$. First, in ML training tasks, there commonly are periods in which the workers do not advance their clocks at all. For example, this is commonly the case at the end of an epoch, while training is paused for model evaluation. In such periods, the estimate would shrink. To keep it more constant, AdaPM does not update the estimate when the worker did not raise its clock during the previous communication round (i.e., if $C_t^i - C_{t-1}^i = 0$).

[5]The exact decision is given in Algorithm 1 in Section 4.2.2.

Second, the observed number of clocks during round $t-1$ (i.e., $C_t^i - C_{t-1}^i = \Delta$) is not independent of the estimate $\hat{\lambda}_{t-1}^i$: if the estimate was too low, AdaPM did *not* act on some intents that the worker reached in this round, so that the worker was potentially slowed down drastically by remote parameter accesses. Thus, the estimate could settle in a "slow regime". A large enough Poisson quantile (i.e., $p \gg 0.5$) ensures that the estimate grows out of such regimes over time. AdaPM further uses a simple heuristic to get out more quickly: if the number of clocks in the last round is larger than the current estimate, it uses this number rather than the estimate (i.e., it uses $\max(\hat{\lambda}_t^i, \Delta)$).

Algorithm 1 depicts precisely how AdaPM decides whether to act on a given intent and how it updates the estimate.

4.2.3 *Effect on Usability.*

AdaPM's action timing relieves applications from the need to signal intent "at the right time", as is, for example, required for initiating relocations in dynamic allocation PM approaches. It is important that applications signal intent early enough so that there is enough time for AdaPM to act on the signal. Below this lower limit, however, action timing makes AdaPM insensitive to when intent is signaled. Thus, applications can simply signal intent early, and rely on AdaPM to act at the right time.

AdaPM's adaptive action timing introduces three hyperparameters: the smoothing factor α, the quantile p, and the initial rate estimate $\hat{\lambda}_0^i$. However, these hyperparameters do not require task-specific tuning. We used the same configuration for all ML tasks and all experiments ($\alpha = 0.1$, $p = 0.9999$, and $\hat{\lambda}_0^i = 10$). This configuration worked well across all five tested tasks, see Section 5.2.

5 EXPERIMENTS

We conducted an experimental study to investigate whether and to what extent automatic fully adaptive PM is possible and beneficial. The source code, datasets, and information for reproducibility are available online.[6]

In our study, we evaluated the performance of AdaPM by comparing it to efficient single-node baselines (Section 5.2), a manually tuned state-of-the-art parameter manager (Section 5.3), and standard PM approaches (Section 5.4). Further, we investigated whether and to what extent AdaPM benefits from supporting multiple management techniques (Sections 5.5 and 5.6), how scalable it is (Section 5.7), whether action timing is crucial for its performance (Section 5.8), how its performance compares to GPU implementations (Section 5.9). and which decisions it makes in practice (Appendix E of the long version [28]). Our major insights are: (i) AdaPM provided good speedups out of the box, without any tuning, (ii) AdaPM matched or even outperformed a manually tuned state-of-the-art parameter manager, (iii) AdaPM scaled efficiently, and (iv) adaptive action timing is a key building block for AdaPM's efficiency. We conclude that *automatic PM is possible and can be efficient.*

5.1 Experimental Setup

Tasks. We considered five ML tasks: knowledge graph embeddings (KGE), word vectors (WV), matrix factorization (MF), click-through-rate prediction (CTR), and graph neural networks (GNN). The tasks differ in multiple ways, including the size of the models, the size

[6]https://github.com/alexrenz/AdaPM/

of the data set, with what rate workers advance their clocks, and in their access patterns. In particular, some workloads exhibit a large amount of dense accesses (e.g., 52% for CTR), some very few (e.g., 0% for MF). We used common practices for task training, e.g., for partitioning training data and measuring model quality. When necessary, we tuned hyperparameters for each task on a single node and used the best found setting in all systems and variants. Appendix C of the long version of this paper [28] describes each task in detail. Table 3 of the long version provides an overview.

Baselines. We compared the performance of AdaPM to efficient single-node implementations, to NuPS [26] (a state-of-the-art multi-technique parameter manager), to static parameter partitioning, to static full replication, and to three ablation variants. To ensure a fair comparison to NuPS, we ran six different hyperparameter configurations of NuPS for each task. Five configurations are designed to simulate a typical hyperparameter search by an application developer: a random search that is loosely informed by the NuPS heuristic and intuition, see Appendix D of the long version for details. In addition, we ran NuPS with the tuned hyperparameters from Renz-Wieland et al. [26]. These were tuned in a series of detailed experiments (but for a setting with 8 worker threads per node, not 32). Note that such detailed insights are not commonly available to application developers. As a single node baseline, we used an efficient shared memory implementation.

Implementation and cluster. We implemented AdaPM in C++, using ZeroMQ and Protocol Buffers for communication. We used a cluster of up to 16 Lenovo ThinkSystem SR630 computers, running Ubuntu Linux 20.04, connected with 100 Gbit/s 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. The CTR and GNN tasks are implemented using PyTorch 1.12.1 and ran with Python 3.9.13. All other tasks are implemented in C++. We consistently used 32 worker threads per node and, unless specified otherwise, 8 cluster nodes. In NuPS and AdaPM, we additionally used 3 ZeroMQ I/O threads per node. In AdaPM, we used 4 communication channels per node; in NuPS, we used 1 channel as it supports only 1. As prior work, we use asynchronous SGD throughout so workers do not block. Thus, reads to non-replicated parameters (e.g., in static partitioning, NuPS, AdaPM) are always current. Reads to replicated parameters, however, may be stale (e.g., in full replication, NuPS, AdaPM). This staleness is generally bounded by the time between refreshs; see [11, 17] for a convergence analysis under bounded staleness.

Measures. To keep costs controlled, we ran all variants with a fixed 4 h time budget. This time budget was sufficient for convergence for the fastest methods. We measured model quality over time and over epochs within this time budget. We conducted 3 independent runs of each experiment, each starting from a distinct randomly initialized model, and report the mean. For NuPS, we ran each of the 6 configurations once. We depict error bars for model quality and run time; they present the minimum and maximum measurements. In some experiments, error bars are not clearly visible because of small variance. Gray shading indicates performance that is dominated by the single node. We report two types of speedups: (i) *raw speedup* depicts the speedup in epoch run time, without considering model quality; (ii) *effective speedup* depicts the improvement w.r.t. time-to-quality. It is calculated from the time that each variant took to reach 90% of the best model quality that we observed in the single node. We chose this rather low threshold to determine speed-ups also for slower variants (which otherwise would not achieve the threshold in the time budget). The speed-ups for AdaPM at a higher accuracy, e.g., 99%, are near-identical to the ones at 90% (e.g., for CTR, 6.4x for both levels).

5.2 Overall Performance (Figure 6)

We ran AdaPM on all tasks, without any tuning, and compared its performance to the one of a single node. Figure 6 depicts the results. Figure 12 of the long version additionally shows model quality per epoch. **AdaPM achieved good speedups over the single node for all tasks out of the box, with 6.5x–7.0x effective speedups on 8 nodes.**

We measured these speedups against the efficient shared-memory single-node implementation to ensure that they are practically relevant. Not comparing the performance of distributed implementations to single-node baselines or comparing to inefficient single-node implementations can be misleading [27].

5.3 Comparison to Manually Tuned PM (Figure 6)

We further compared the performance of AdaPM to NuPS on the tasks for which NuPS implementations and tuned configurations are available (KGE, WV, and MF). Figures 6a, 6b, and 6c depict the results. **AdaPM matched or even outperformed the performance of NuPS across all tasks.**

NuPS required task-specific tuning to achieve good performance. The figures depict three of the six NuPS configurations: (i) the best and worst performing ones *per task* from our hyperparameter search and (ii) the ones tuned by the NuPS authors. Different configurations were efficient for different tasks. For example, configuration 4 was the best one for MF, but the worst one for WV.

AdaPM matched (MF) or outperformed (slightly in KGE[7] and drastically in WV) the best NuPS configurations. AdaPM can outperform NuPS because AdaPM manages parameters more precisely, e.g., it maintained replicas only while needed, allowing it to synchronize fewer replicas more frequently. Appendix E of the long version provides more insight into how AdaPM works and how it differs from NuPS.

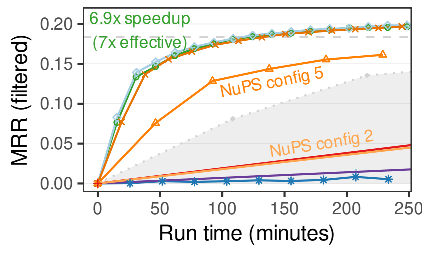
5.4 Comparison to Standard PM (Figure 6)

We compared AdaPM to static full replication and static parameter partitioning for all tasks. Again, Figure 6 shows the results. **AdaPM outperformed standard PM.**

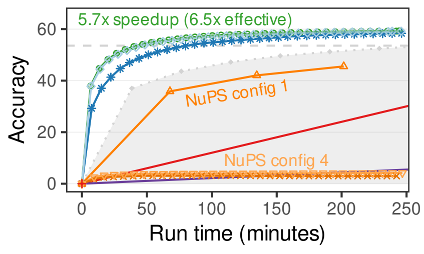
Static full replication provided poor performance for all tasks but WV. It provided poor model quality for KGE and CTR because synchronizing full replicas on all nodes allowed for only infrequent replica synchronization. It ran out of memory for MF and GNN because their models are large. It worked well for WV because the WV model is small (only 14 GB, see Table 3 of the long version) and the WV task is robust towards infrequent replica synchronization.

Static parameter partitioning was inefficient because it required synchronous network communication for the majority of

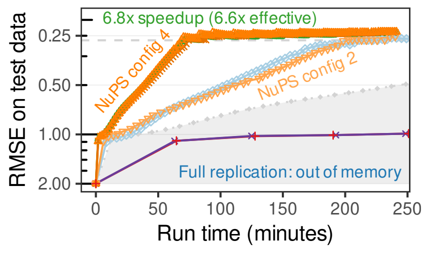
[7]Epoch were 20% faster in AdaPM than in tuned NuPS.



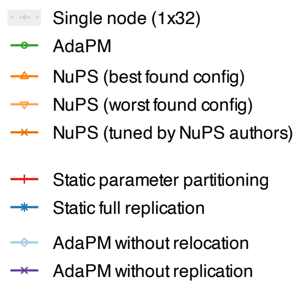
(a) Knowledge graph embeddings (KGE)

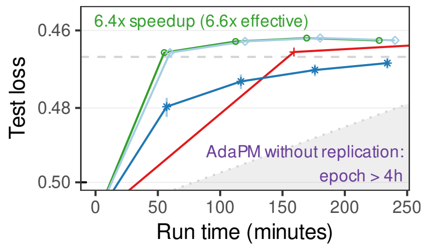


(b) Word vectors (WV)

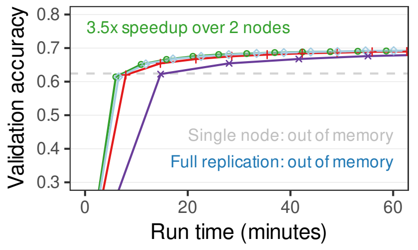


(c) Matrix factorization (MF)





(d) Click-through-rate predication (CTR)



(e) Graph neural networks (GNN)

Figure 6: Performance of AdaPM on 8 nodes (32 threads per node), compared to efficient single-node baselines (Section 5.2), manually tuned PM (Section 5.3), standard PM approaches (Section 5.4), and single-technique AdaPM variants (Section 5.5).

Table 2: Per-epoch network communication and staleness.

Variant	Commun. (GB per node)					Mean replica staleness (ms)				
	KGE	WV	MF	CTR	GNN	KGE	WV	MF	CTR	GNN
AdaPM	490	415	39	910	82	1.2	11.2	1.4	7.2	2.0
AdaPM w/o relocation	685	607	349	1066	171	1.4	13.8	7.7	7.7	2.6

parameter accesses. It was relatively efficient for GNN because this task accesses parameters in large groups, such that the impact of access latency is small.

5.5 Comparison to Single-Technique Adaptive PM (Figure 6)

We compared AdaPM to two ablation variants, see Figure 6. These variants are identical to AdaPM, but each variant is restricted to one management technique: *AdaPM without relocation* only replicates parameters, and *AdaPM without replication* only relocates parameters. **AdaPM without replication was *inefficient*; AdaPM without relocation was *efficient* for most tasks.**

AdaPM without replication performed poorly for all tasks because relocation is inefficient for hot spot parameters, as observed previously [26].

AdaPM without relocation was efficient for KGE, WV, CTR, and GNN. For MF, it was 3.0x slower than AdaPM because the MF task exhibits locality (due to row-partitioning, each row parameter is accessed by only one node) and replication is inefficient for managing locality.

5.6 The Benefit of Relocation (Table 2)

As replication-only AdaPM performed well on many tasks (see Section 5.5), we further investigated whether it is beneficial for AdaPM to employ relocation in addition to replication. To do so, we measured the amount of communicated data and replica staleness. For simplicity, we take the time since the last replica refresh (i.e., the last check for updates) as staleness and ignore whether or not the value has actually changed since then. Table 2 depicts the results. Besides improving performance for some tasks (see Section 5.5), **employing relocation reduced network overhead and decreased replica staleness for all tasks.**

The contrast in communication and staleness was particularly large for tasks with locality (MF and GNN due to data partitioning), where AdaPM communicated up to 9x fewer data. But supporting relocation also improved efficiency for all other tasks. For example, AdaPM without relocation communicated 40% more data for one KGE epoch because relocation is more efficient if two nodes access a parameter after each other: relocation sends the parameter directly from the first to the second node, whereas replication synchronizes via the owner node.

5.7 Scalability (Figure 7)

We investigated the scalability of AdaPM and compared it to NuPS. Figure 7 depicts raw and effective speedups over the single node for KGE, WV, and MF; Figure 13 in the long version does so for the other tasks. **AdaPM scaled efficiently, achieving near-linear raw and good effective speedups.**

AdaPM scaled more efficiently than NuPS because NuPS's scalability was limited by relocation conflicts (i.e., concurrent accesses to a relocation-managed parameter). For example, in KGE, the share of remote accesses in the best found NuPS configuration was 1.2%,

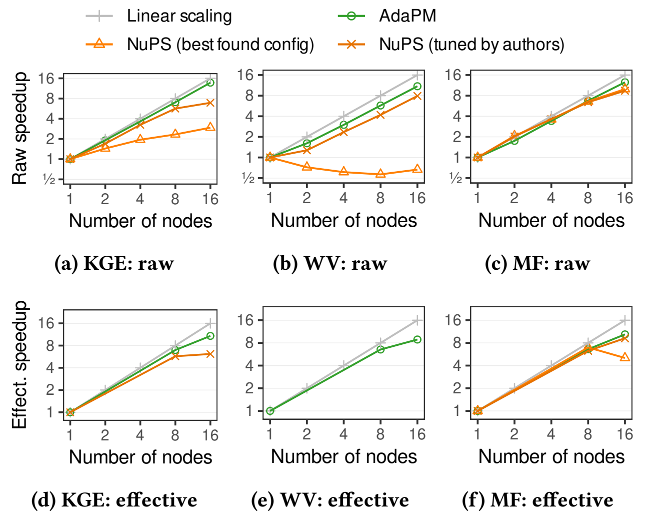


Figure 7: Scalability (logarithmic axes). Raw speedup (a-c), i.e., w.r.t. epoch run time, and effective speedup (d-f), i.e., w.r.t. reaching 90% of the best model quality observed on a single node. Runs that did not reach this threshold within the time limit are not shown.

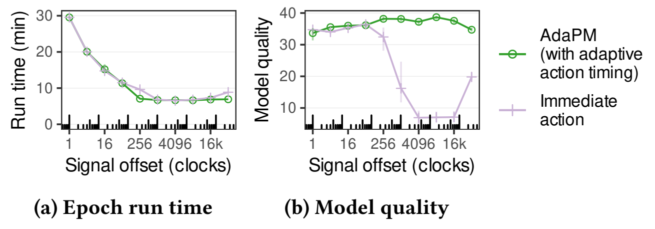


Figure 8: The effect of adaptive action timing for WV on epoch run time (a) and on model quality after one epoch (b).

2.4%, 3.4%, and 5.3% on 2, 4, 8, and 16 nodes, respectively; in AdaPM, it was <0.0001%. The effective speedups slightly dropped on 16 nodes because we tuned task hyperparameters (e.g., learning rate and regularization) for the single node and—to minimize the impact of hyperparameter tuning—used these settings throughout all experiments. These settings were not optimal for runs with 16x more parallelism. Other settings achieved better effective scalability.

5.8 Effect of Action Timing (Figure 8)

To investigate the effect of action timing, we compared AdaPM to an ablation variant that acts immediately after each intent signal, on workloads with varying signal offsets. Figure 8 depicts the results for WV, see Figure 14 of the long version for further tasks. **With adaptive action timing, AdaPM was efficient for any sufficiently large signal offset.**

Early signals. With adaptive action timing, AdaPM provided excellent performance for all large signal offset values. In contrast, with immediate action, performance was poor for large signal offsets: run time increased and model quality decreased. The reason for this was that the immediate action variant maintained replicas for longer than necessary (thus lowering synchronization frequency).

Late signals. Smaller relocation offsets improved performance for immediate action, but did not further improve performance for AdaPM. For both, epoch run time was poor when intent was signaled so late that the system did not have sufficient time for setting up replicas or relocating parameters. Thus, there was a task-specific optimum value for immediate action (necessitating tuning), but not for AdaPM.

5.9 Comparison to GPU Implementations

CTR and GNN models are typically trained on GPUs, whereas we used CPUs in our proof-of-concept implementation of AdaPM. To put our results in perspective, we briefly compared to recent GPU-based methods for these two tasks. Some overhead is expected because these methods are task-specific, i.e., they use specialized and tuned training algorithms that cannot be used for other tasks directly, whereas AdaPM is general-purpose. Although the CPU and GPU settings are very different and cannot be compared directly, we found that **the results obtained by AdaPM seem competitive**.

For CTR, Zheng et al. [40] ran one epoch on a 90% train split of our dataset, for the same model, and with comparable batch size[8] in 76.8 minutes on one V100 GPU. AdaPM took 57.5 minutes on 8 CPU nodes for a 85.7% train split. The key bottleneck of GPU-based methods is data transfer to and from GPU memory as the models do not fit in GPU memory. Very large batch sizes can mitigate this to some extent, but require careful work to avoid loss of accuracy [40].

For GNN, Min et al. [20] ran one epoch on the same dataset, but with a different model, on 2 RTX 3090 GPUs in 10.2 minutes. AdaPM ran 1 epoch in 5.3 minutes on 8 CPU nodes. MariusGNN [33] achieved a validation accuracy of 0.6638 in 8.2 minutes on the same dataset (with a slightly different model) using 1 V100 GPU. Waleffe et al. [33] further report that DGL [34] required 4 V100 GPUs to achieve the same accuracy in similar time. AdaPM achieved slightly better accuracy (0.665) after 15.5 minutes using 8 CPU nodes.

6 CONCLUSION

We proposed intent signaling, a novel mechanism that decouples providing information about parameter accesses (simple) from how and when it is exploited (hard). Intent signaling increases ease of use and efficiency in parameter management. We presented AdaPM, a parameter manager that automatically adapts to the underlying ML task based solely on intent signals. In our experimental study, AdaPM was efficient for many ML tasks out of the box, without requiring any tuning, and matched or even outperformed state-of-the-art (more complex to use) systems. Interesting directions for future work are how to better integrate co-processor (e.g, GPU) memory in an adaptive parameter manager such as AdaPM and how to support intent signaling directly in common ML systems.

ACKNOWLEDGEMENTS

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

[8]Zheng et al. [40] used a batch size of 1000 with synchronous parallel SGD. For AdaPM, we used a batch size of 128 with 8 workers of asynchronous SGD.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[2] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. Accelerating Recommendation System Training by Leveraging Popular Choices. *PVLDB* 15, 1 (Sept. 2021), 127–140. https://doi.org/10.14778/3485450.3485462

[3] Ivana Balazevic, Carl Allen, and Timothy Hospedales. 2019. TuckER: Tensor Factorization for Knowledge Graph Completion. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP '19)*. Association for Computational Linguistics, Hong Kong, China, 5185–5194. https://doi.org/10.18653/v1/D19-1522

[4] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems (NeurIPS '13)*. Curran Associates. https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

[5] Po-Lung Chen, Chen-Tse Tsai, Yao-Nan Chen, Ku-Chun Chou, Chun-Liang Li, Cheng-Hao Tsai, Kuan-Wei Wu, Yu-Cheng Chou, Chung-Yi Li, Wei-Shih Lin, Shu-Hao Yu, Rong-Bing Chiu, Chieh-Yen Lin, Chien-Chih Wang, Po-Wei Wang, Wei-Lun Su, Chen-Hung Wu, Tsung-Ting Kuo, Todd G. McKenzie, Ya-Hsuan Chang, Chun-Sung Ferng, Chia-Mau Ni, Hsuan-Tien Lin, Chih-Jen Lin, and Shou-De Lin. 2012. A Linear Ensemble of Individual and Blended Models for Music Rating Prediction. In *Proceedings of KDD Cup 2011 (Proceedings of Machine Learning Research, Vol. 18)*. PMLR, 21–60. https://proceedings.mlr.press/v18/chen12a.html

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[7] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (Boston, MA, USA) *(DLRS '16)*. Association for Computing Machinery, New York, NY, USA, 7–10. https://doi.org/10.1145/2988450.2988454

[8] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. 2015. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI '15)*. 79–87.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL '19)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine Based Neural Network for CTR Prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 1725–1731.

[11] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip Gibbons, Garth Gibson, Gregory Ganger, and Eric Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems* (Lake Tahoe, NV, USA) *(NeurIPS '13)*. Curran Associates, USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999611.2999748

[12] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *8th IEEE International Conference on Data Mining (ICDM '08)*. 263–272. https://doi.org/10.1109/ICDM.2008.22

[13] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. 2018. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. *PVLDB* 11, 5 (Jan. 2018), 566–579. https://doi.org/10.1145/3187009.3177734

[14] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-Aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the 14th European Conference on Computer Systems* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 43, 15 pages. https://doi.org/10.1145/3302424.3303957

[15] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37. https://doi.org/10.1109/MC.2009.263

[16] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of Machine Learning and Systems (MLSys '19)*. 120–131. https://proceedings.mlsys.org/paper/2019/file/e2c420d928d4bf8ce0ff2ec19b371514-Paper.pdf

[17] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *Advances in Neural Information Processing Systems* (Montreal, Canada) *(NeurIPS '15)*. MIT Press, Cambridge, MA, USA, 2737–2745.

[18] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: Scaling out Huge Embedding Model Training via Cache-Enabled Distributed Framework. *PVLDB* 15, 2 (Oct. 2021), 312–320. https://doi.org/10.14778/3489496.3489511

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations (ICLR '13)*. http://arxiv.org/abs/1301.3781

[20] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *PVLDB* 14, 11 (Oct. 2021), 2087–2100. https://doi.org/10.14778/3476249.3476264

[21] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. 2011. A Three-way Model for Collective Learning on Multi-relational Data. In *Proceedings of the 28th International Conference on Machine Learning* (Bellevue, Washington, USA) *(ICML '11)*. Omnipress, USA, 809–816. http://dl.acm.org/citation.cfm?id=3104482.3104584

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS '19)*. Curran Associates, Red Hook, NY, USA, Article 721, 12 pages.

[23] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *PVLDB* 15, 9 (May 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614

[24] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[25] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL '18)*. Association for Computational Linguistics, New Orleans, Louisiana, 2227–2237. https://doi.org/10.18653/v1/N18-1202

[26] Alexander Renz-Wieland, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. 2022. NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. In *Proceedings of the 2022 ACM International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA.

[27] Alexander Renz-Wieland, Rainer Gemulla, Steffen Zeuch, and Volker Markl. 2020. Dynamic Parameter Allocation in Parameter Servers. *PVLDB* 13, 12 (July 2020), 1877–1890. https://doi.org/10.14778/3407790.3407796

[28] Alexander Renz-Wieland, Andreas Kieslinger, Robert Gericke, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. 2022. Good Intentions: Adaptive Parameter Servers via Intent Signaling. *CoRR* abs/2206.00470 (2022). https://doi.org/10.48550/arXiv.2206.00470 arXiv:2206.00470

[29] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web (ESWC '18)*. Springer International Publishing, Cham, 593–607.

[30] Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, and Bowen Zhou. 2019. End-to-End Structure-Aware Convolutional Networks for Knowledge Base Completion. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI '19)*. 3060–3067. https://doi.org/10.1609/aaai.v33i01.33013060

[31] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex Embeddings for Simple Link Prediction. In *Proceedings of the 33rd International Conference on Machine Learning* (New York, NY, USA) *(ICML '16)*. JMLR.org, 2071–2080. http://dl.acm.org/citation.cfm?id=3045390.3045609

[32] Shikhar Vashishth, Soumya Sanyal, Vikram Nitin, and Partha Talukdar. 2020. Composition-based Multi-Relational Graph Convolutional Networks. In *Proceedings of the 8th International Conference on Learning Representations (ICLR '20)*. https://openreview.net/forum?id=BylA_C4tPr

[33] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2022. Marius++: Large-Scale Training of Graph Neural Networks on a Single Machine. *CoRR* abs/2202.02365 (2022). arXiv:2202.02365 https://arxiv.org/abs/2202.02365

[34] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[35] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17* (Halifax, NS, Canada) *(ADKDD'17)*. Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3124749.3124754

[36] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*. 381–394.

[37] Eric Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. ACM, New York, NY, USA, 1335–1344. https://doi.org/10.1145/2783258.2783323

[38] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Proceedings of Machine Learning and Systems (MLSys '20)*. 412–428. https://proceedings.mlsys.org/paper/2020/file/f7e6c85504ce6e82442c770f7c8606f0-Paper.pdf

[39] Qiming Zheng, Quan Chen, Kaihao Bai, Huifeng Guo, Yong Gao, Xiuqiang He, and Minyi Guo. 2021. BiPS: Hotness-aware Bi-tier Parameter Synchronization for Recommendation Models. In *35th IEEE International Parallel and Distributed Processing Symposium (ISDPS '21)*. IEEE, 609–618. https://doi.org/10.1109/IPDPS49936.2021.00069

[40] Zangwei Zheng, Pengtai Xu, Xuan Zou, Da Tang, Zhen Li, Chenguang Xi, Peng Wu, Leqi Zou, Yijie Zhu, Ming Chen, Xiangzhuo Ding, Fuzhao Xue, Ziheng Qing, Youlong Cheng, and Yang You. 2022. CowClip: Reducing CTR Prediction Model Training Time from 12 hours to 10 minutes on 1 GPU. *CoRR* abs/2204.06240 (2022). https://doi.org/10.48550/arXiv.2204.06240 arXiv:2204.06240

[41] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 1059–1068. https://doi.org/10.1145/3219819.3219823

[42] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *PVLDB* 12, 12 (Aug. 2019), 2094–2105. https://doi.org/10.14778/3352063.3352127