Good Intentions: Adaptive Parameter Management via Intent Signaling

Alexander Renz-Wieland
Technische Universität Berlin

Andreas Kieslinger
Technische Universität Berlin

Robert Gericke
Technische Universität Berlin

Rainer Gemulla
Universität Mannheim

Zoi Kaoudi
IT University Copenhagen

Volker Markl
Technische Universität Berlin

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

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].

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
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 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 specify relocation 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

---

2Workers 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.
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). Such an approach integrates naturally with the data loader paradigm of common ML systems — such as Pytorch’s data loader [22], TensorFlow’s data loader thread(s):  

3 Applications may, of course, synchronize worker clocks.

3.1 Parameter access

A worker can signal multiple (potentially overlapping) intents for the same parameter, extend one intent by signaling another later on, etc. With this primitive, a worker signals that it intends to access a set of parameters in the time window between a start clock $c_{\text{start}}$ and an end clock $c_{\text{end}}$ (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 $\text{Intent}(P_2, 2, 3)$, the data loader signals that the corresponding worker intends to access parameters $P_2$ (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., $C^i < c_{\text{start}}$. We say that an intent is active if the worker clock is within the intent time window, i.e., $c_{\text{start}} \leq C^i < c_{\text{end}}$. And we say that an intent is expired when the worker clock has reached the end clock, i.e., when $c_{\text{end}} \leq C^i$. 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

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.

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

With this primitive, a worker signals that it intends to access a set of parameters in the time window between a start clock $c_{\text{start}}$ (inclusive) and an end clock $c_{\text{end}}$ (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 $\text{Intent}(P_2, 2, 3)$, the data loader signals that the corresponding worker intends to access parameters $P_2$ (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., $C^i < c_{\text{start}}$. We say that an intent is active if the worker clock is within the intent time window, i.e., $c_{\text{start}} \leq C^i < c_{\text{end}}$. And we say that an intent is expired when the worker clock has reached the end clock, i.e., when $c_{\text{end}} \leq C^i$. 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
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 ADAPTM PARAMETER MANAGER
Intent signaling opens a large design space for adaptive PM. Key design questions include: where 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 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 destroys 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...
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 $\text{Poisson}(\lambda_i^t)$ with expected rate $\lambda_i^t$. 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).
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 $\hat{\lambda}_i^{t-1}$, clock of worker $i$ at the start of round $t$ ($C_i^t$) and round $t-1$ ($C_i^{t-1}$), smoothing factor $\alpha$, quantile $p$.

1. $\Lambda \leftarrow C_i^t - C_i^{t-1}$
2. if $\Lambda > 0$ then
3. $\hat{\lambda}_i^t \leftarrow (1-\alpha)\hat{\lambda}_i^{t-1} + \alpha(\Lambda)$
4. else
5. $\hat{\lambda}_i^t \leftarrow \hat{\lambda}_i^{t-1}$
6. end if

return $C_{\text{start}} < C_i^t + Q_{\text{Pois}}(2 \cdot \max(\hat{\lambda}_i^t, \Lambda), p)$

**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 ($C_{\text{start}}$) before round $t+1$ finishes, i.e., roughly if $C_{\text{start}} < C_i^t + Q_{\text{Pois}}(2 \cdot \max(\hat{\lambda}_i^t, \Lambda), p)$, where $C_i^t$ is the current clock of worker $i$ at the start of round $t$ and $Q_{\text{Pois}}(\lambda, p)$ computes the $p$ quantile of a Poisson distribution with rate parameter $\Lambda$. 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 $\lambda_i^t$ is unknown. AdaPM estimates this rate from the number of clocks in past communication rounds, using exponential smoothing:

$\hat{\lambda}_i^t \leftarrow (1-\alpha)\hat{\lambda}_i^{t-1} + \alpha(C_i^t - C_i^{t-1})$

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

We consider two further aspects to improve the robustness of the estimate $\hat{\lambda}_i^t$. 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 $C_i^t - C_i^{t-1} = 0$).

Second, the observed number of clocks during round $t-1$ (i.e., $C_i^t - C_i^{t-1} = \Lambda$) is not independent of the estimate $\hat{\lambda}_i^{t-1}$: 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 $\max(\hat{\lambda}_i^t, \Lambda)$).

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 $\alpha$, the quantile $p$, and the initial rate estimate $\hat{\lambda}_0^t$. 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 $\hat{\lambda}_0^t = 10$). 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
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. 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. 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.

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 except the box, with 6.5x–7.0x effective speedups on 8 nodes.

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.

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
Graph neural networks (GNN)

- Single node (1x32)
- AdaPM
- Static parameter partitioning
- Static full replication
- AdaPM without relocation
- AdaPM without replication
- NuPS (best found config)
- NuPS (worst found config)
- NuPS (tuned by NuPS authors)

Matrix factorization (MF)

- Single node (1x32)
- AdaPM
- Static parameter partitioning
- Static full replication
- AdaPM without relocation
- AdaPM without replication
- NuPS (best found config)
- NuPS (worst found config)
- NuPS (tuned by NuPS authors)

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>
<thead>
<tr>
<th>Variant</th>
<th>Commun. (GB per node)</th>
<th>Mean replica staleness (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KGE</td>
<td>WV</td>
</tr>
<tr>
<td>AdaPM</td>
<td>490</td>
<td>415</td>
</tr>
<tr>
<td>AdaPM w/o relocation</td>
<td>685</td>
<td>607</td>
</tr>
</tbody>
</table>

Table 2: Per-epoch network communication and staleness.

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%,
With adaptive action timing, 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. 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.

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 were 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\(^6\) 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.

\( ^{6}\)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.


