Adaptive Parameter Servers

vorgelegt von
Florian Alexander Renz-Wieland, MSc

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doctor rerum naturalium
(Dr. rer. nat.)

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Klaus-Robert Müller
Gutachter: Prof. Dr. Volker Markl
Gutachter: Prof. Dr. Rainer Gemulla
Gutachter: Prof. Dr. Matthias Boehm
Gutachter: Prof. Dr. Tilmann Rabl

Tag der wissenschaftlichen Aussprache:
16. Dezember 2022

Berlin 2023

Abstract

Machine learning (ML) has become an essential tool for solving problems that have traditionally been challenging for computers, for example in the fields of natural language processing, computer vision, and recommender systems. For large ML tasks, distributed training has become a necessity for keeping up with increasing dataset sizes and model complexity. A key challenge in distributed training is to synchronize model parameters among cluster nodes and to do so efficiently. Parameter servers (PSs) facilitate the implementation of distributed training by providing cluster-wide read and write access to the parameters, and transparently handling partitioning and synchronization in the background (either among the cluster nodes directly or via physically separate server nodes).

In this thesis, we study the efficiency of PSs for ML tasks with sparse parameter access, i.e., tasks in which each update step reads and writes only a small (or tiny) part of the model. In a first step, we find that existing PSs are inefficient for such tasks: in our experiments, distributed implementations were slower than efficient single node implementations due to communication overhead. This inefficiency dramatically limits the utility of PSs and distributed training in general. With such inefficiency, distributed training in practice will be used only when it is indispensable, e.g., for very large models. And when distributed training is indeed employed, its inefficiency squanders hardware and energy resources.

Starting from this observation, we investigate whether and how PS efficiency can be improved. The main idea of this thesis is to increase efficiency by making the PS adapt to the underlying ML task. We first present and evaluate a series of potential performance improvements in this direction, each making the PS more adaptive. In particular, we explore (i) to dynamically adapt the allocation of model parameters, i.e., to relocate parameters among nodes during training, according to where they are accessed, (ii) to adapt the management technique of the PS to the access patterns of individual parameters, i.e., to employ a suitable management technique for each parameter, and (iii) to adapt to the type of a parameter access, by supporting sampling (i.e., randomized) access directly. We find empirically that each of these aspects can improve PS efficiency. However, each aspect also makes the PS more complex to use, because the application (i.e., the component that interacts with the PS) needs to control adaptivity manually.

To reduce complexity, we develop a mechanism that enables automatic adaptivity, i.e., adaptivity without requiring the application's manual control. With this mechanism, the application merely provides information about parameter accesses, in a way that naturally integrates into common ML systems. We describe a novel PS system—called AdaPS—that adapts to ML tasks automatically based on the information provided by this mechanism. AdaPS incorporates all adaptivity aspects presented in this thesis. It decides what to do (i.e., which management technique to use for a specific parameter and where to allocate each parameter) and when to do so. It does so automatically, i.e., without further user input, and dynamically, i.e., based on the current situation. In our experiments, AdaPS enabled efficient distributed ML training for multiple ML tasks: in contrast to previous PSs, it provided near-linear speed-ups over efficient single node implementations.

With these results, we argue that PSs can be efficient for sparse ML tasks, and that this efficiency can be reached with limited additional effort from application developers. Efficient and easy-to-use PSs make distributed training (i) attractive for a wide range of use cases—thus enabling solutions to challenging problems—and (ii) squander fewer of our planet's resources.

Zusammenfassung (German Abstract)

Maschinelles Lernen (ML) ist inzwischen ein essentielles Werkzeug, um Probleme zu lösen, die für Computer traditionell herausfordernd waren, zum Beispiel zur Verarbeitung natürlicher Sprache, für Bilderkennung und zum Generieren von Empfehlungen. Um mit steigenden Datenmengen und Modellkomplexitäten Schritt zu halten, ist für komplizierte ML-Aufgaben verteiltes Training notwendig, das heißt, paralleles Training auf mehreren Computern eines Clusters. Eine zentrale Herausforderung in verteiltem Training ist die effiziente Synchronisierung der Modellparameter zwischen den Computern des Clusters. Parameter Server erleichtern die Implementierung von verteiltem Training, indem sie clusterweiten Lese- und Schreibzugriff auf die Modellparameter ermöglichen, und im Hintergrund transparent Partitionierung und Synchronisation handhaben (entweder direkt zwischen den Computern des Clusters oder via physisch separate Computer).

In dieser Arbeit untersuche ich die Effizienz von Parameter Servern für ML-Aufgaben mit dünnbesetztem (engl. "sparse") Parameterzugriff, also Aufgaben, bei denen jeder Modell-Update-Schritt nur einen kleinen (oder winzigen) Teil des Modells liest und schreibt. In einem ersten Schritt stelle ich fest, dass bestehende Parameter Server für solche Aufgaben ineffizient sind: In meinen Experimenten waren verteilte Implementierungen (mit 8 Computern) langsamer als effiziente Implementierungen auf einem einzelnen Computer. Diese Ineffizienz limitiert den Nutzwert von Parameter Servern und verteiltem Training drastisch. Mit solch hoher Ineffizienz wird verteiltes Lernen in der Praxis nur eingesetzt wenn es unvermeidbar ist, zum Beispiel für sehr große Modelle. Und immer wenn verteiltes Training eingesetzt wird, vergeudet es Hardware- und Energieressourcen.

Auf der Grundlage dieser Beobachtung untersuche ich, ob und wie die Effizienz von Parameter Servern verbessert werden kann. Die zentrale Idee dieser Arbeit ist, die Effizienz zu verbessern indem der Parameter Server sich an die zugrunde liegende ML-Aufgabe anpasst. Ich präsentiere und evaluiere zunächst eine Reihe von potenziellen Verbesserungen in dieser Richtung, die den Parameter Server jeweils adaptiver machen. Insbesondere versuche ich, (i) die Allokation der Modellparameter dynamisch anzupassen, das heißt, die Parameter während des Trainings zwischen den Knoten zu verschieben, je nachdem, wo auf sie zugegriffen wird, (ii) die Parameter-Management-Techniken des Parameter Servers an die Zugriffsmuster der einzelnen Parameter anzupassen, also für jeden Parameter eine für den Parameter geeignete Technik einzusetzen, und (iii) den Parameter Server an die Art des Parameterzugriffs anzupassen, indem er stichprobenartige (randomisierte) Zugriffe direkt unterstützt. In einer Reihe von Experimenten stelle ich fest, dass jeder dieser Aspekte die Effizienz von Parameter Servern verbessern kann. Allerdings verkompliziert jeder Aspekt auch die Nutzung des Parameter Servers, weil die Anwendung (die Komponente, die mit dem Parameter Server interagiert) die Anpassungen manuell steuern muss.

Aus diesem Grund entwickle ich einen Mechanismus, der automatische Anpassung ermöglicht. Die Anwendung stellt über diesen Mechanismus Informationen über die Parameterzugriffe bereit, und zwar auf eine Art, die sich nahtlos in gängige ML-Systeme integriert. Darüber hinaus präsentiere ich einen neuartigen Parameter Server (AdaPS), der sich basierend auf den von diesem Mechanismus bereitgestellten Informationen automatisch an ML-Aufgaben anpasst. AdaPS vereint alle in dieser Arbeit vorgestellten Aspekte der Adaptivität. Der Parameter Server entscheidet, was zu tun ist (das heißt, welche Technik für einen bestimmten Parameter zu verwenden ist und wo jeder Parameter alloziert werden soll) und wann dies zu tun ist. Dies geschieht automatisch, also ohne zusätzliche Informationen, und dynamisch, also basierend auf der aktuellen Situation. In Experimenten ermöglicht AdaPS effizientes verteiltes ML-Training: Im Gegensatz zu früheren Parameter Servern liefert AdaPS nahezu lineare Geschwindigkeitssteigerungen gegenüber Implementierungen für einzelne Computer.

Aufgrund dieser Ergebnisse argumentiere ich, dass Parameter Server für dünnbesetzte ML-Aufgaben effizient sein können, und dass diese Effizienz mit begrenztem zusätzlichem Aufwand für Anwendungsentwickler:innen erreicht werden kann. Effiziente und einfach zu verwendende Parameter Server sorgen dafür, dass verteiltes Training (i) für ein breites Spektrum von Anwendungsfällen attraktiv wird (und so Lösungen für anspruchsvolle Probleme ermöglicht) und (ii) weniger der Ressourcen unseres Planeten verschwendet.

Acknowledgments

First of all, I would like to thank my thesis advisors, Volker Markl and Rainer Gemulla. Thank you Volker for providing me with a great research environment, for always trusting my skills, and for giving me the freedom and helping me to find my own way of doing things. Thank you Rainer for inspiring me to enter academia (by teaching an outstanding lecture on database systems), for instilling in me a wealth of learnings about approaching, thinking about, and solving problems, and for always providing me with excellent feedback on my research, my writing, and my talks. I also want to thank Matthias Böhm and Tilmann Rabl for being part of my committee and for providing valuable feedback on this thesis. I want to thank Steffen Zeuch and Zoi Kaoudi, for advising me in my research.

I want to thank all others who have been guiding and supporting me, in particular in the Database Systems and Information Management group at TU Berlin. Thanks to Jonas for always creating a friendly atmosphere in our office, for teaching me how to navigate the research group, saving me countless hours of figuring things out, and for many interesting discussions on research, politics, and life. Thank you Claudia, Melanie, and Lutz, for keeping the place running and for always being helpful. And thank you to all the other PhD students for making the research group a friendly and supportive place: Andreas, Gabor, Philipp, Haralampos, Kajetan, Sergey, Viktor, Clemens, Behrouz, Felix, Ventura, Makis, Rudi, Lennart, Ariane, Xenofon, Martin, Anastasiia, and Dimitrios. Thank you to Adrian Kochsiek for an inspiring collaboration on training knowledge graph embeddings. And thank you to the students whom I had the honor to supervise, in particular Andreas, Robert, and Tobias. Thank you for many interesting discussions and teaching me plenty about supervision.

None of this would have been possible without the constant support of my family, my friends, and my partner (who doubles as a fantastic office mate when a global pandemic forces you to work from home). Thank you for all your support and for the great time that we are having together, through life's ups and downs. My life is better because of you.

Contents

Chapter 1

Introduction

> *Science is a way of trying not to fool yourself. The principle is that you must not fool yourself, and you are the easiest person to fool.*
>
> — Richard Feynman

Machine learning (ML) studies algorithms that "learn" to carry out certain tasks from example data. ML algorithms provide state-of-the-art performance in multiple domain areas, sometimes drastically outperforming traditional (non-ML) approaches, for example in natural language processing (Bengio et al. 2000; Hochreiter and Schmidhuber 1997; Sutskever et al. 2014; Mikolov et al. 2013; Devlin et al. 2019; Brown et al. 2020), computer vision (LeCun et al. 1989; Krizhevsky et al. 2012; Simonyan and Zisserman 2015; Ramesh et al. 2021; Riquelme et al. 2021) and recommender systems (Koren et al. 2009; Das et al. 2007; P.-L. Chen et al. 2012). During *training*, the *parameters* of one specific ML model are adjusted from example training data in iterative update steps. Model training can be time-consuming because (i) models usually become better the more training data they have been trained on, such that there is a large number of such update steps, and (ii) larger models commonly achieve better model quality, but typically require more complex (i.e., more time-consuming) individual update steps.

Key to keeping up with increasing dataset sizes and model complexity is *distributed* model training, i.e., to use multiple nodes of a compute cluster to parallelize training. Distributed training allows (i) for training faster by leveraging distributed compute resources, (ii) for training models and data that exceed the memory capacity of a single node, and (iii) for potentially using a larger number of cheaper nodes rather than one expensive high-end node. During training, each node typically accesses a local partition of the training data, but requires global read and write access to all model parameters. Thus, parameter management among these nodes is a key concern in distributed training. *Parameter servers* (PS) facilitate the implementation of distributed training by providing such distributed parameter management: they transparently partition and synchronize parameters across the nodes behind primitives for reading and writing parameters. Most ML systems include such a PS as a core component (Abadi et al. 2016; Paszke et al. 2019; T. Chen et al. 2015; Lerer et al. 2019; J. K. Kim et al. 2016; Chilimbi et al. 2014), and there are many standalone PSs (Mu Li et al. 2014a; Ho et al. 2013; Dai et al. 2015; Sergeev and Balso 2018; J. Jiang et al. 2017; Yuzhen Huang et al. 2018; Jagerman et al. 2017; Z. Zhang et al. 2019; Y. Jiang et al. 2020). Early PSs stored the parameters on a set of physically separate server nodes (Smola and Narayanamurthy 2010; Ahmed et al. 2012). In this thesis, we use the PS term broadly to refer to systems that provide distributed parameter management, including ones that co-locate the parameters on the nodes that run the training (Ho et al. 2013; Dai et al. 2015; Yuzhen Huang et al. 2018).

In many ML tasks, the model is large, but accessed *sparsely*, i.e., each update step reads and writes only a (small) subset of a large number of model parameters. For example, such sparse access is common in natural language processing tasks (Mikolov et al. 2013; Pennington et al. 2014; Peters et al. 2018; Howard and Ruder 2018), knowledge graph embeddings (Trouillon et al. 2016; Nickel et al. 2011; Balazevic et al. 2019; Bordes et al. 2013; Kazemi and Poole 2018), some graph neural networks (Schlichtkrull et al. 2018; Shang et al. 2019; Vashishth et al. 2020), click-through prediction (H.-T. Cheng et al. 2016; Guo et al. 2017; R. Wang et al. 2017; Zhou et al. 2018), and recommender systems (Koren et al. 2009; P.-L. Chen et al. 2012; Hu et al. 2008). A key property of sparse ML tasks is that their parameter access pattern is dynamic, i.e., each node accesses different parameters at different points in time.

In this thesis, we study the efficiency of PSs for ML tasks with such sparse parameter access. In a first step, we find that existing PSs are inefficient: they barely outperform efficient single node implementations. The reason for this underwhelming performance is communication overhead for synchronizing model parameters among the cluster nodes. Based on this observation, we work on making PSs more efficient. To this end, we present and evaluate several potential PS performance improvements. The common theme among these potential improvements is *adaptivity*, i.e., that the PS adapts to the underlying ML task. Building on each other, these improvements step by step make the PS more and more adaptive, and—based on what we see in our empirical evaluations—more efficient. We first introduce these adaptivity aspects as PS features that are manually controlled by the application (i.e., the component that interacts with the PS). Therefore, as we make the PS more and more adaptive, we also make it more and more complex to use. In a final step, we drastically reduce usage complexity by presenting an approach for PSs to adapt automatically, without the application's manual control.

Contributions

In more detail, our main contributions are:

1. We investigate the efficiency of existing PSs. We find that existing PSs are scalable, but inefficient. Their performance can even fall behind that of efficient single node implementations. For example, in several experiments (see Section 3.3.2), 8 nodes were slower than a single node. We argue that it is crucial to compare PS performance to *efficient* single node baselines to detect and quantify such inefficiency.

2. We investigate whether dynamically adapting parameter allocation to the ML task—i.e., to relocate model parameters during run time—can improve PS efficiency. We find that dynamic allocation allows for exploiting *locality* in parameter access, and drastically improves efficiency for some ML tasks by reducing communication overhead.

3. We investigate whether adapting the management techniques of the PS to the access patterns of individual parameters can improve PS efficiency. I.e., the PS picks a suitable management technique per parameter. We find that such technique adaptation improves PS efficiency for ML tasks with non-uniform—i.e., skewed—access frequency distributions.

4. We investigate whether supporting sampling (i.e., randomized) parameter access directly in PSs can improve efficiency. We find that direct sampling support significantly improves PS efficiency for a range of ML tasks, in particular tasks that employ negative sampling for many-class classification or to mitigate an absence of negative training data.

5. We propose *intent signaling*, a novel mechanism for passing parameter access information from the application to the PS in a way that integrates naturally with common ML systems. Intent signaling allows PSs to adapt to ML tasks automatically, thus relieving applications from the need to control adaptation manually.

6. We present and evaluate a fully adaptive, zero-tuning PS called AdaPS. AdaPS dynamically adapts its management techniques and parameter allocation to the underlying ML task. It does so automatically, based only on intent signals. In our experiments, AdaPS was efficient out of the box (i.e., with zero tuning), providing near-linear speed-ups over efficient single node implementations.

Throughout this thesis, we present three prototype PS systems: Lapse, NuPS, and finally AdaPS. Each builds on the previous one(s), and explores and evaluates additional aspects of adaptivity. AdaPS is the final system that incorporates all the ideas of this thesis. All of our work is publicly available as open-source software.[1]

With the above results, we argue that distributed training for sparse ML tasks can be efficient, and that this efficiency can be reached with limited additional effort from application developers. Using *inefficient* PSs makes distributed training uneconomical for all ML tasks that can be trained with single node implementations. Consequently, with inefficient PSs, distributed training is employed in practice mostly when its scalability is indispensable due to very large datasets or models. The adaptations presented in this thesis drastically improve PS efficiency for sparse ML tasks, reaching near-linear speed-ups over efficient single node implementations. This makes distributed training attractive for many more ML uses cases. In addition, *automatic* adaptation makes these efficiency gains accessible without expertise in distributed systems and without extensive configuration and tuning effort. Together, these contributions make efficient distributed training of sparse ML tasks attractive and accessible for a wide range of ML use cases.

Publications

The work presented in this thesis is based on the following publications:

- A. Renz-Wieland, R. Gemulla, S. Zeuch, V. Markl. Dynamic Parameter Allocation in Parameter Servers. PVLDB, 13(11): 1877–1890. 2020. (Renz-Wieland et al. 2020)

- A. Renz-Wieland, R. Gemulla, Z. Kaoudi, V. Markl. NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. In *Proceedings of the 2022 International Conference of Management of Data*. ACM, New York, NY, USA. 2022. (Renz-Wieland et al. 2022a)

- A. Renz-Wieland, A. Kieslinger, R. Gericke, R. Gemulla, Z. Kaoudi, V. Markl. Good Intentions: Adaptive Parameter Servers via Intent Signaling. CoRR, abs/2206.00470. 2022. (Renz-Wieland et al. 2022b)

The thesis also draws material from the following publication:

- A. Renz-Wieland, T. Drobisch, R. Gemulla, S. Zeuch, V. Markl. Just Move It! Dynamic Parameter Allocation in Action. PVLDB, 14(12): 2707-2710, 2021. (Renz-Wieland et al. 2021)

[1] Available at `https://github.com/alexrenz/AdaPS`.

Outline

Chapter 2 introduces basic concepts. Chapter 3 investigates the efficiency of existing PSs and describes dynamic parameter allocation. In Chapter 4, we present our work on adapting management techniques and direct sampling support. Chapter 5 describes automatic adaptivity, i.e., intent signaling and AdaPS. Chapter 6 concludes this thesis with a summary and a discussion of open research problems.

Chapter 2

Background

In this chapter, we introduce concepts that are crucial for understanding this thesis (Section 2.1), and give an overview of existing work on distributed parameter management (Sections 2.2) and related work in general (Section 2.3).

2.1 Machine Learning

2.1.1 Basics

ML studies algorithms that "learn" to carry out certain tasks from example data (Mitchell 1997). This stands in contrast to conventional algorithms, in which software developers explicitly encode how to carry out a task. ML algorithms are given a set of training *examples*, commonly referred to as the *training data*. The ML algorithm uses these training data to construct a model. This process is referred to as *model training*. The goal is that—after training—the model can be applied to examples that are not included in the training data. This step is referred to as *inference*. This thesis focuses on model training.

ML models store their learned information in *model parameters*, typically one or multiple vectors, matrices, or tensors of real numbers. The goal of model training is to adjust these model parameters such that the model can carry out the desired task. This training is typically done iteratively, i.e., there are many consecutive *update steps* that adjust the model parameters.

2.1.2 Stochastic Gradient Descent

Currently, the most widely used training algorithms are gradient-based ones (Ruder 2016). These algorithms aim to minimize a given *cost function*. During training, these algorithms iteratively adjust the model parameters based on the gradient with respect to this cost function. *Gradient descent* updates the model parameters based on the exact gradient. To compute this gradient, the model is evaluated on all examples of the training dataset.

For large datasets, *stochastic gradient descent* (SGD) and *mini-batch SGD* are popular because they can learn faster than full gradient descent (Bottou et al. 2016). SGD learning algorithms use a stochastic approximation of the exact gradient to update parameters. In SGD, this approximation is computed from one single example of the training dataset. In mini-batch SGD, this approximation is computed from a set—a *mini-batch*—of examples. SGD methods commonly learn faster than gradient descent on large datasets because they can do many—approximate, but cheap—update steps in the time that it takes to do one gradient descent update step.

One update step of a gradient-based training algorithm has the following access pattern. It reads the relevant training examples, reads the parameters that it requires to compute the gradient for these training examples, computes the gradient, and writes updates to the parameters. Usually, training involves multiple passes over the training dataset. One pass over the training dataset is commonly referred to as one *epoch*.

Algorithm 2.1: Sequential mini-batch SGD.

```
Data: $\mathscr{D}$: training dataset,
      num_epochs: number of epochs to run,
      batch_size: batch size,
      $W$: model parameters
1 for epoch ← 1 to num_epochs do
2     b = num_batches ( $\mathscr{D}$, batch_size )
      // data loading (pipeline parallel with training, in separate thread(s))
3     $\mathscr{B} = []$
4     for i ← 1 to b do
5         $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch )
      // training
6     for i ← 1 to b do
7         $\Delta W$ = compute_update ( $\mathscr{B}_i$, $W$ )
8         $W \leftarrow W + \Delta W$
```

Algorithm 2.1 depicts a simplified example implementation of mini-batch SGD.[1] The algorithm runs multiple epochs (line 1). In each epoch, a worker thread iterates over the training data in a sequence of batches. Each batch is prepared before it is trained on (lines 3–5) (in pipeline parallel fashion, see below for details). For example, the preparation could consist of reading and annotating sentences, or loading and cropping images. The worker thread

[1] SGD can be seen as a special case of mini-batch size SGD with a batch size of 1.

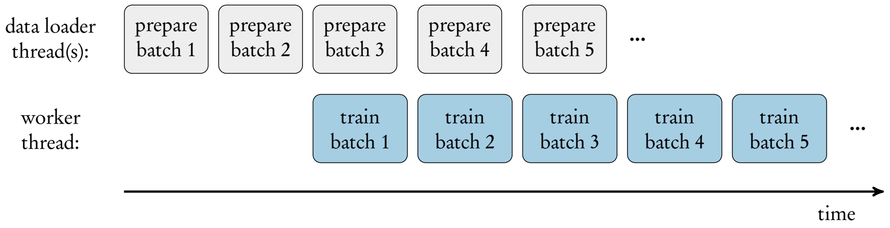


Figure 2.1: Pipeline parallel preparation and training of batches.

iterates through the batches (line 6). It computes an update from each batch, using the current model parameters (line 7) and applies this update to the model parameters (line 8).

We depict batch preparation (lines 3–5) as a separate loop because it is commonly run pipeline-parallel with the actual training, in one or multiple separate data loader threads (Paszke et al. 2019; Abadi et al. 2016; T. Chen et al. 2015). The data loader threads prepare each batch shortly before it is trained on by the worker thread. Figure 2.1 illustrates this pipeline parallelism.

2.1.3 Sparse Parameter Access

This thesis focuses on ML tasks in which parameter access is *sparse*, i.e., each update step reads and writes only a (small, and potentially tiny) subset of all model parameters. For example, such sparse access is common in natural language processing tasks (Mikolov et al. 2013; Pennington et al. 2014; Peters et al. 2018; Devlin et al. 2019; Howard and Ruder 2018), knowledge graph embeddings (Trouillon et al. 2016; Nickel et al. 2011; Balazevic et al. 2019; Bordes et al. 2013; Kazemi and Poole 2018), some graph neural networks (Schlichtkrull et al. 2018; Shang et al. 2019; Vashishth et al. 2020), click-through prediction (H.-T. Cheng et al. 2016; Guo et al. 2017; R. Wang et al. 2017; Zhou et al. 2018), and recommender systems (Koren et al. 2009; P.-L. Chen et al. 2012; Hu et al. 2008). Parameter access is sparse in these tasks because the models' predictions for one training example depend only on a subset of the model parameters. For example, consider an ML model that associates one parameter with each word of the English language. A training example is one sentence. In each sentence, only a fraction of all English words occur. Thus, only a fraction of parameters is accessed for one training example. Figure 2.2 illustrates this example.

In more detail, the existence and extent of sparsity depends on the ML model, the training dataset, and the cost function. Parameter access is sparse if (i) the model associates specific model parameters with specific properties of the training examples, (ii) the training data are sparse, i.e., each training example includes only a subset of all properties, (iii) and there is no update for

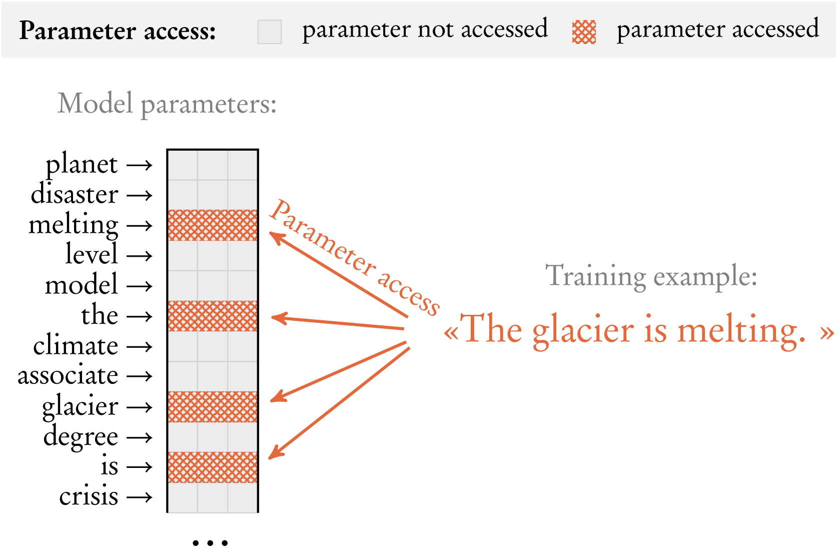


Figure 2.2: An example for sparse parameter access. Each square represents one model parameter. The example ML model associates a few parameters with each word of the English language. One training example, i.e., one sentence of a text corpus, accesses only the parameters that are associated with the words that occur in the sentence.

parameters that are associated with non-present properties (typically because the gradient for these parameters is zero).

The opposite of sparse access is *dense* parameter access, i.e., each update step reads and writes *all* model parameters. For example, dense access is common in computer vision (LeCun et al. 1989; Ramesh et al. 2021). For instance, in convolutional neural networks (LeCun et al. 1989), all parameters are required to classify a training example (i.e., one image), so that each update step accesses all parameters, see Figure 2.3.

In some models, parameter access is partially dense and partially sparse (Peters et al. 2018; Devlin et al. 2019; Howard and Ruder 2018). Typically, access to the first (embedding) layer and sometimes the last (classification) layer is sparse, and access to other layers is dense. The share of parameters that are accessed sparsely depends on the model architecture, but can be high, e.g., around 90% in ELMo (Peters et al. 2018).

2.1.4 Parallel Stochastic Gradient Descent

SGD is a sequential algorithm: an update step depends on the updates done by all previous update steps. In practice, two approaches are used to parallelize SGD methods. The first one, *synchronous (parallel) SGD* (Zinkevich et al. 2010; J. Chen et al. 2016) parallelizes the computation of one mini-batch.

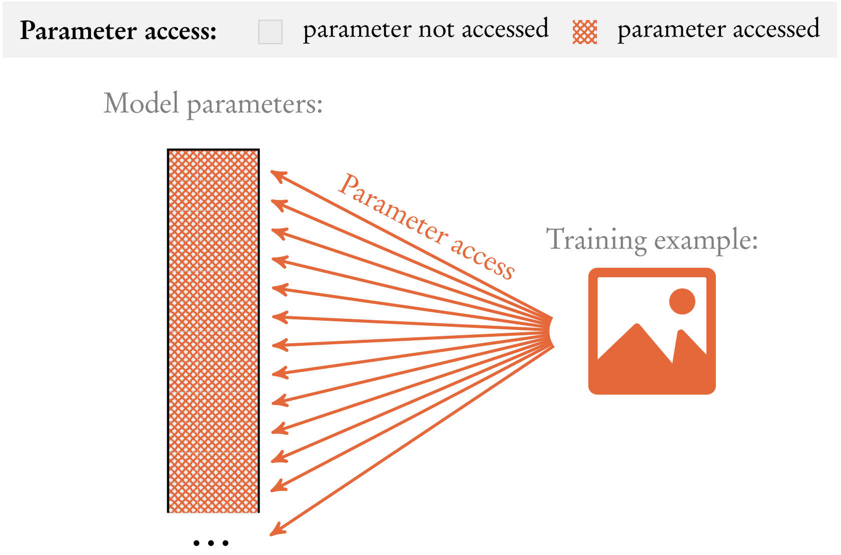


Figure 2.3: An example for dense parameter access. All model parameters are required to classify one training example (e.g., one image).

I.e., the gradients for the examples of a mini-batch are computed by different workers in parallel. The gradients are then aggregated among the workers and the update is applied to the parameters. Synchronous SGD is equivalent to sequential execution. However, it requires a global barrier at the end of each update step to wait until all participating workers have finished gradient computation. This can be inefficient for ML tasks in which gradient computation is relatively cheap (Niu et al. 2011). The overhead of the barrier can be reduced to some extent by launching additional *backup workers* to compute gradients for more training examples than needed and then using only the gradients of the fastest workers (J. Chen et al. 2016). This method decreases overall efficiency, as additional gradients are computed, but ignored.

In the second approach, *asynchronous SGD* (Bengio et al. 2000; Niu et al. 2011; Dean et al. 2012), workers carry out update steps independent of each other, potentially reading stale parameter values and potentially overwriting updates by other workers. Asynchronous SGD is *not equivalent* to sequential execution. Nevertheless, asynchronous SGD is popular—in particular for tasks with relatively cheap update steps—because it does not require a barrier after each update step. It is especially suitable for ML tasks with sparse updates because updates are overwritten less frequently (Niu et al. 2011).

Algorithm 2.2 depicts an example shared memory implementation of asynchronous parallel SGD. Multiple worker threads execute the depicted program in parallel. The worker threads access the model parameters W via shared memory. In contrast to the sequential implementation (see Algorithm 2.1), the set of all batches is divided up among all worker threads, such that each batch is processed only once. The batches are assigned to workers based on an identifier for the worker thread (t) and the total number of worker threads T (lines 2 and 5).

Algorithm 2.2: Asynchronous parallel mini-batch SGD. The program is run by multiple worker threads in parallel. Differences to a sequential implementation (Algorithm 2.1) are highlighted in blue.

```
Data: $\mathscr{D}$: training dataset,
      num_epochs: number of epochs to run,
      batch_size: batch size,
      $W$: model parameters (shared memory),
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads
1 for epoch ← 1 to num_epochs do
2     b = num_batches ( $\mathscr{D}$, batch_size, t, T )
      // data loading (pipeline parallel with training, in separate thread(s))
3     $\mathscr{B} = []$
4     for i ← 1 to b do
5         $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch, t, T )
      // training
6     for i ← 1 to b do
7         $\Delta W$ = compute_update ( $\mathscr{B}_i$, $W$ )
8         $W \leftarrow W + \Delta W$
```

The worker threads write updates to the model parameters W concurrently. There are implementations of asynchronous SGD that use no concurrency control for these updates (Mikolov et al. 2013), such that updates can be overwritten. Other implementations use locking mechanisms or atomics to ensure that there are no lost updates (Niu et al. 2011). For simpler notation, we depict that the update step reads (line 7) and writes (line 8) all model parameters W. In practice, an update step for an ML task with sparse parameter access reads and writes only a subset of all model parameters. I.e., the parameter update ΔW consists mostly of zeros. The frequency of concurrent updates decreases with increasing sparsity (Niu et al. 2011).

The program in Algorithm 2.2 is an implementation of asynchronous parallel SGD. Analogously, synchronous SGD could be implemented. The main difference is that in synchronous SGD workers wait after each update step for all other workers to complete the update step. I.e., there would be a barrier after each iteration of the training loop.

2.2 Distributed Parameter Management

To keep up with increasing training data size and model complexity, model training is often not only parallelized among multiple compute units (e.g., CPU cores, or hardware accelerators such as GPUs or TPUs) of one node, but distributed to the compute units of multiple nodes of a cluster. The main advantages of distributed training are that (i) one can potentially train models that exceed the memory capacity of a single node and (ii) training can potentially be accelerated, as the compute units of several computers can work in parallel.

A common approach to distributed training is the *data parallel* one: the training dataset is partitioned to the nodes of the cluster, i.e., each node holds one partition of the training dataset. Each node accesses only its local partition of the training data, but potentially reads and updates all model parameters. Thus, *distributed parameter management*, i.e., providing global read and write access to parameters and handling synchronization among the nodes, is a key concern.

The term *parameter server* is used inconsistently in current literature to refer to systems that provide such distributed parameter management. Authors use it to refer to at least two different concepts. First, the term is used to refer to architectures in which gradients and/or parameters are communicated exclusively via physically separate, dedicated *server* nodes, rather than by direct communication among the worker nodes. Second, the term is used in a broader sense to describe the class of all systems that provide distributed parameter management, i.e., that provide global parameter access across a cluster. This broader definition includes systems that achieve this via separate server nodes *and* systems that achieve this via direct communication among the nodes. Throughout this thesis, we will use the second meaning.

I.e., a PS is any system that provides global parameter access across a cluster. To this end, it provides primitives to read and write parameters (commonly referred to as `pull` and `push`, respectively). The read and write operations can be performed synchronously or asynchronously. Behind the scenes, the PS transparently handles parameter management: for example, some PSs partition parameters to nodes and, if necessary, send appropriate messages to process read and write operations. To coordinate parameter accesses across nodes, the PS assigns unique *keys* to parameters. The push operation is usually cumulative, i.e., the client sends an update term to the PS, which then adds this term to the parameter value. Many ML stacks use PSs as a component, e.g., TensorFlow (Abadi et al. 2016), MXNet (T. Chen et al. 2015), PyTorch BigGraph (Lerer et al. 2019), STRADS (J. K. Kim et al.

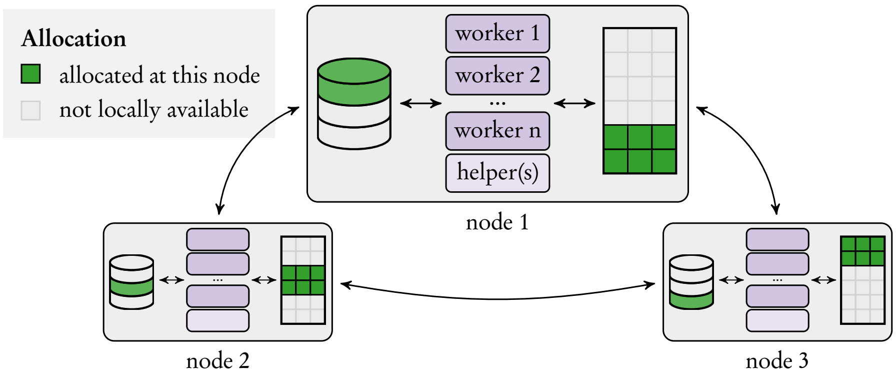


Figure 2.4: The distributed architecture that we assume in this thesis: multiple network-connected nodes, each holding a partition of the training dataset (⛁), several worker threads, potentially helper threads, and some model parameters (▦). The parameter allocation depends on the chosen PS; the figure exemplarily depicts the one of a classic PS.

2016), STRADS-AP (J. K. Kim et al. 2019), or Project Adam (Chilimbi et al. 2014), and there exist multiple standalone PSs, e.g., Petuum (Ho et al. 2013), PS-Lite (Mu Li et al. 2014a), Angel (J. Jiang et al. 2017), FlexPS (Yuzhen Huang et al. 2018), Glint (Jagerman et al. 2017), PS2 (Z. Zhang et al. 2019), and BytePS (Y. Jiang et al. 2020).

Throughout this thesis, we assume the following distributed architecture (see Figure 2.4). There are multiple cluster nodes, connected by network links. Each node holds one partition of the training dataset. On each node, there is one or multiple worker threads. Additionally, there can be one or multiple helper threads at each node, e.g., for synchronizing parameters among nodes in the background or for processing parameter access operations of other nodes. We further assume that each of these nodes holds some of the model parameters, either through (exclusive) allocation or through replication. Such co-location of the model parameters on the same physical nodes as the worker threads is commonly done in recent PSs for efficiency (Jagerman et al. 2017; J. Jiang et al. 2017; Ho et al. 2013; Yuzhen Huang et al. 2018).[2] Which node holds which parameters depends on the approach for distributed parameter management. We will discuss different approaches in the following. Figure 2.4 depicts the approach of a Classic PS: parameters are partitioned to the nodes (see Section 2.2.1 for details).

Access to local parameters is cheaper than access to remote parameters. For example, local parameters could be accessed via shared memory (see Section 3.2.3). In contrast, remote access is more expensive because the

[2] In contrast, early PSs stored the parameters on physically separate server nodes (Smola and Narayanamurthy 2010; Ahmed et al. 2012).

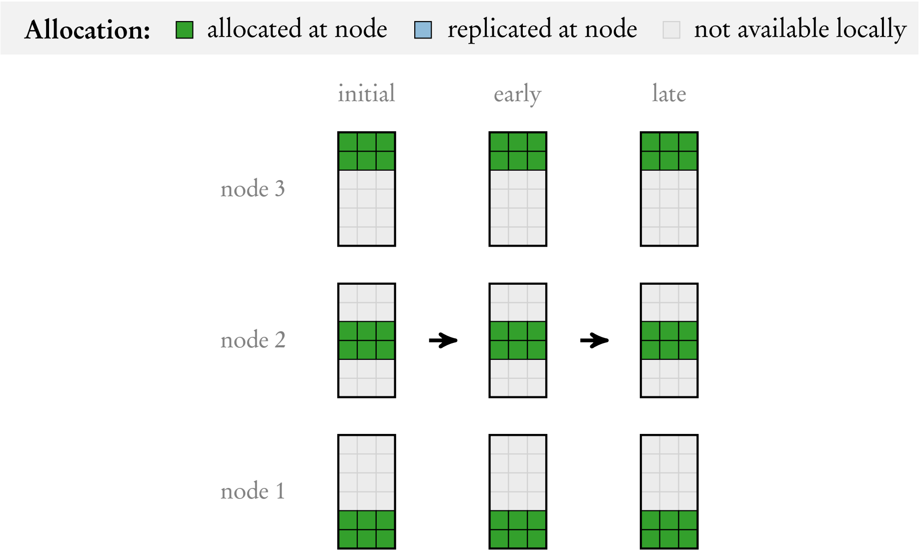


Figure 2.5: An example for parameter allocation in a classic PS. The parameters are partitioned to the nodes. Parameter allocation is static, i.e., it does not change throughout training.

parameter value (or the parameter update) needs to be transferred over the network between the node that accesses the parameter and a node that holds the parameter (or a replica). The cost of remote parameter access is visible throughout the experiments in this thesis, for example in Section 3.3.2.

In the following, we discuss fundamental approaches for providing global parameter access in such a distributed architecture.

2.2.1 Classic Parameter Server

A *classic PS* (Smola and Narayanamurthy 2010; Ahmed et al. 2012; Mu Li et al. 2014a) such as PS-Lite partitions the model parameters to the nodes of the cluster. Thus, precisely one node holds the current value of a parameter, and no replicas are created. To read a parameter, the classic PSs sends messages over the network to retrieve the parameter value from the node that holds this parameter. Analogously, it sends the update to the corresponding node for a write operation. Figure 2.5 depicts an example for parameter allocation in a classic PS. The example depicts parameter allocation for a three-node cluster at three different points in time: (i) initially, before training starts, (ii) at some early time of training, and (iii) at some late time of training. For a classic PS, parameter allocation is the same during these three points in time. In other words, parameter allocation in a classic PS is static. We will see non-static approaches below and throughout this thesis.

One advantage of Classic PSs is that they can provide sequential consistency (Lamport 1979) (see Section 3.2.4 for our analysis). That is, (1) each worker's operations are executed in the order specified by the worker, and (2) the result of any execution is equivalent to an execution of the operations of all workers in some sequential order. This consistency guarantee ensures that classic PSs have no or only small negative effect on the convergence of distributed training compared to sequential implementations. In contrast, more relaxed consistency can slow down convergence (Ho et al. 2013; Dai et al. 2015). A second advantage of classic PSs is that there is no need to determine when and how often replicas should be synchronized (because there are no parameter replicas). This makes classic PSs relatively easy to use. A third advantage is that the maximum possible model size scales linearly with the number of nodes.

The main disadvantage of a classic PS is that its performance can be very limited because most parameter accesses induce access latency for one network round trip. For example, to read a parameter, the accessing node contacts the node that holds the parameter, which then transfers the parameter value to the accessing node. This overhead can slow down distributed training dramatically. For example, in our experiments in Section 3.3.2, distributed implementations with a classic PS on 8 nodes were up to 22x slower than an efficient single node implementation.

Let us now consider how an application (i.e., the component that interacts with the PS) can leverage a classic PS to implement distributed training. Algorithm 2.3 depicts an example implementation of distributed asynchronous mini-batch SGD in a classic PS. As in parallel SGD (see Algorithm 2.2), multiple workers run the depicted program in parallel. In the distributed setup, these workers are spread across physically separate cluster nodes. The workers access the parameters via the pull and push primitives of the PS. For each batch, the worker reads the subset of parameters that is required to process this batch (line 7), computes an update (line 8), and writes the computed update back into the PS (line 9). We write w for the subset of model parameters that is required to process a batch (i.e., $w \subseteq W$) and `keys(`$\mathscr{B}_i$`)` for an application-specific function that returns the keys of the parameters that are required to process a given batch $\mathscr{B}_i$.

2.2.2 Static Full Replication

Static full replication parameter management statically replicates all parameters to all cluster nodes throughout training. The replicas are synchronized periodically, either synchronously (triggered by the application) or asyn-

Algorithm 2.3: Distributed asynchronous SGD with a classic PS. The program is run by many distributed worker threads in parallel. Differences to the shared-memory parallel implementation (Algorithm 2.2) are highlighted in orange.

```
Data: $\mathscr{D}$: training dataset,
      $num\_epochs$: number of epochs to run,
      $batch\_size$: batch size,
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads
for epoch ← 1 to num_epochs do
    b = num_batches ( $\mathscr{D}$, batch_size, t, T )
    // data loading (pipeline parallel with training, in separate thread(s))
    $\mathscr{B} = []$
    for i ← 1 to b do
        $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch, t, T )
    // training
    for i ← 1 to b do
        $w$ = pull ( keys($\mathscr{B}_i$) )
        $\Delta w$ = compute_update ( $\mathscr{B}_i$, $w$ )
        push ( keys($\mathscr{B}_i$), $\Delta w$ )
```

chronously in the background. Figure 2.6 illustrates parameter allocation in static full replication. Again, parameter allocation is static: it does not change throughout training.

The advantage of static full replication is that all parameters can be accessed without synchronous network communication because all nodes hold a local copy of all model parameters. This is one reason why static full replication is popular for distributed training of models with dense parameter access (LeCun et al. 1989; Sergeev and Balso 2018; S. Li et al. 2020). As every update step accesses all parameters, it is desirable to provide fast access to all parameters at all times. The static full replication approach is currently often paired with synchronous parallel SGD. There are PSs that are specialized for this setting. For example, BytePS (Y. Jiang et al. 2020) assumes a full model replica on all nodes, and focuses on efficiently synchronizing these replicas after each update step. The approach of BytePS is to synchronize the replicas via physically separate, cheaper ("server") nodes.

For ML tasks with sparse parameter access, however, static full replication is communication-inefficient. The problem is that it maintains replicas for all parameters on all nodes throughout training, even though each node accesses only a small subset of these replicas at each point in time. Thus, static full replication sends replica updates that are never read. This over-

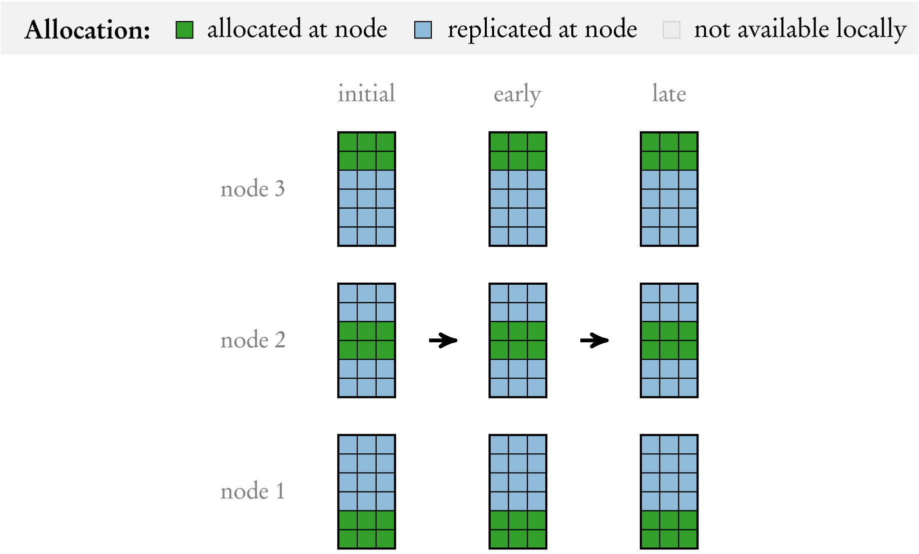


Figure 2.6: An example for parameter allocation with full static replication. Each node holds a local copy of each model parameter (either the main copy of the parameter or a replica).

communication can drastically slow down distributed training. For example, in our experiments in Section 5.4.4, full static replication on 8 nodes was significantly slower than efficient single node implementations.

A further disadvantage is that, with static full replication, the maximum model size does not increase with the number of nodes. On the contrary, the size of the ML model is limited by the memory capacity of a single node, as each node holds a replica of the entire model. Thus, static full replication is infeasible for very large models. Another disadvantage is that there potentially is staleness among replicas. Common ML systems (S. Li et al. 2020; Abadi et al. 2016; T. Chen et al. 2015) prevent such staleness by explicitly synchronizing after every step of synchronous parallel mini-batch SGD, which can create significant network overhead.

2.2.3 Selective Replication

A *replication PS* sets up and tears down replicas selectively (Ho et al. 2013; Yuzhen Huang et al. 2018; J. Jiang et al. 2017; Dai et al. 2015; Cui et al. 2014). I.e., parameters are still allocated statically, as in a classic PS, but the PS may dynamically replicate subsets of the parameters to additional nodes to reduce communication overhead (Ho et al. 2013) (see Figure 2.7 for an example). These subsets differ from node to node and from time to time. There are different protocols for deciding which subset of parameters should be replicated at which node at which point in time (see below). Replication

PSs provide weaker consistency guarantees than Classic PSs, such as *bounded staleness*. For example, Petuum (Ho et al. 2013), a popular replication PS, lets the application specify a logical clock (one per worker) and guarantees that local parameter values are not older than an application-specified staleness bound B with respect to this logical clock. Workers maintain their logical clock via an `advanceClock()` operation. Different staleness bounds are suitable for different ML tasks (Ho et al. 2013). It is the responsibility of the application to find a suitable staleness bound for the desired ML task.

Algorithm 2.4 depicts an example for how asynchronous mini-batch SGD can be implemented in Petuum. The program is similar to the one for a Classic PS, with two additions: (i) the application configures the staleness bound (line 1) and (ii) each worker advances its clock after each batch (line 11).

There are two main protocols for deciding which subset of the model parameters should be replicated at which node at which point in time: *stale synchronous parallel* (SSP) (Ho et al. 2013) and *eager stale synchronous parallel* (ESSP) (Dai et al. 2015).

Algorithm 2.4: Distributed asynchronous SGD with Petuum. Differences to a Classic PS implementation (Algorithm 2.3) are highlighted in green.

```
Data: $\mathscr{D}$: training dataset,
      $num\_epochs$: number of epochs to run,
      $batch\_size$: batch size,
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads ,
      $staleness\_bound$: staleness bound
1  set_staleness_bound ( $staleness\_bound$ )
2  for epoch ← 1 to num_epochs do
3      b = num_batches ( $\mathscr{D}$, batch_size, t, T )
       // data loading (pipeline parallel with training, in separate thread(s))
4      $\mathscr{B} = []$
5      for i ← 1 to b do
6          $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch, t, T )
       // training
7      for i ← 1 to b do
8          $w$ = pull ( keys($\mathscr{B}_i$) )
9          $\Delta w$ = compute_update ( $\mathscr{B}_i$, $w$ )
10         push ( keys($\mathscr{B}_i$), $\Delta w$ )
11         advanceClock ()
```

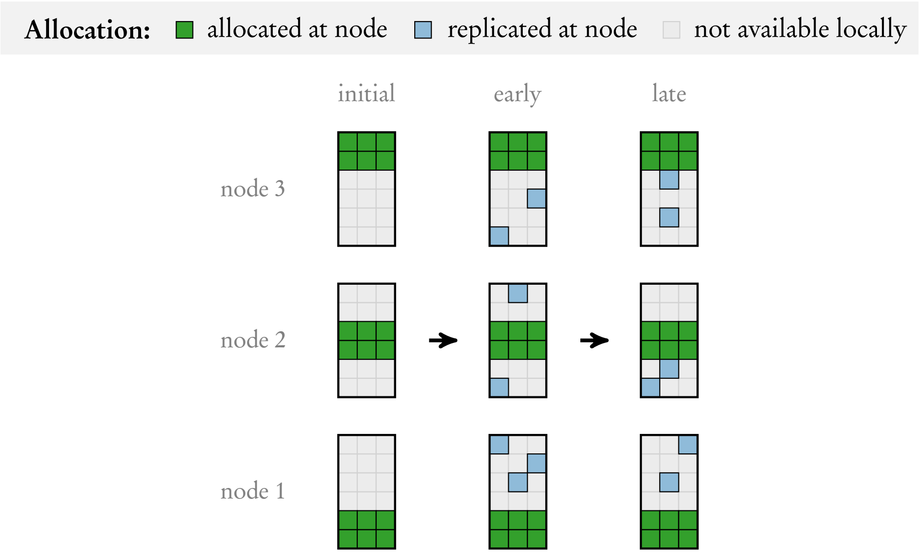


Figure 2.7: An example for parameter allocation in an SSP replication PS. The parameters are partitioned to the nodes as in a classic PS, but the PS selectively replicates subsets of parameters to the different nodes. An SSP replication PS terminates a replica when an application-specified staleness bound is reached.

Stale synchronous parallel (SSP)

SSP sets up a replica for a parameter p at a node n when node n accesses parameter p. It continues to use this replica for parameter reads until the local worker has advanced its clock B (the staleness bound) times since replica setup. At this point, the PS terminates the replica. It sets up a replica again when node n accesses parameter p again. Figure 2.7 illustrates an example for parameter allocation in an SSP replication PS. Replica updates are accumulated locally and propagated to the node that holds the main copy of the parameter at the subsequent `advanceClock` invocation.

The main advantage of SSP is that only a small number of replicas exists at each point in time.[3] Only this small number of replicas has to be maintained. The main disadvantage of SSP is that replicas are set up reactively: the PS sets up the replica when the worker reads the parameter. I.e., the worker has to wait synchronously for the replica to set up (Dai et al. 2015). This can severely limit the performance of SSP. For example, in our experiments in Section 4.4.2, SSP on 8 nodes was more than 5x slower than an efficient single node implementation.

[3] Assuming a realistic setting for the staleness bound. With excessively large staleness bounds, an excessively large number of replicas will be maintained at each point in time.

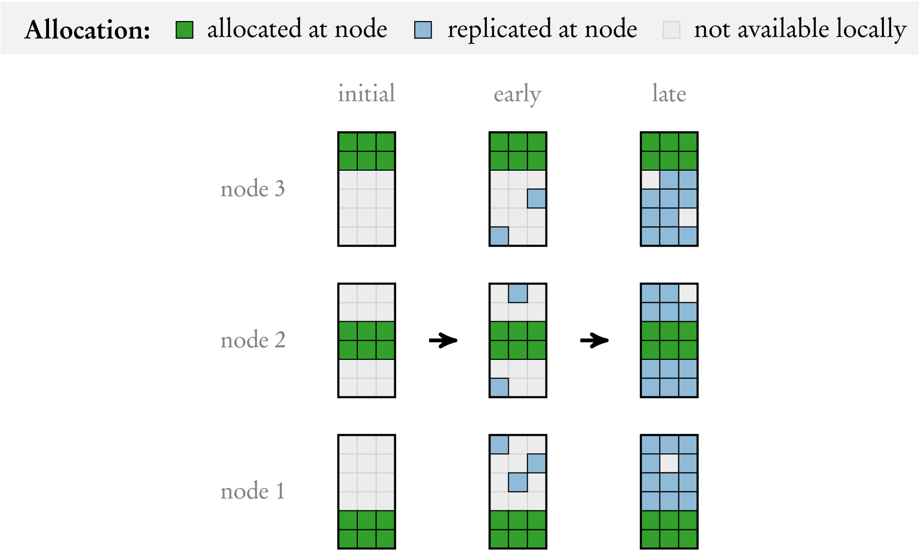


Figure 2.8: An example for parameter allocation in an ESSP replication PS. In contrast to SSP (Figure 2.7), ESSP does not terminate replicas. Once a replica is set up, ESSP maintains the replica throughout training (by continuously propagating updates). Thus, the number of replicas typically increases until each node holds replicas of (almost) all parameters.

Eager stale synchronous parallel (ESSP)

ESSP also creates a replica when a parameter is (first) accessed, but then maintains this replica throughout the entire training task. I.e., after a node n accesses a parameter p once, the PS will continuously propagate all updates for parameter p to node n. Figure 2.8 illustrates an example for parameter allocation in an ESSP replication PS. Petuum ESSP guarantees that a read parameter value is not older than the configured staleness threshold. Behind the scenes, however, ESSP aims to send updates as soon as possible, so that the typical actual staleness (ideally) is lower than the configured staleness bound (Dai et al. 2015). As in SSP, replica updates are accumulated locally and propagated to the node that holds the main copy of the parameter at the subsequent `advanceClock` invocation.

The main advantage of ESSP over SSP is that—after the initial creation of a replica for parameter p upon first access of node n—no further waiting for synchronous replica setup is necessary for parameter p at node n, as the replica is maintained throughout training. However, this eager replica setup comes at the cost of over-communication: for common ML tasks, the ESSP protocol is similar to full static replication after a short period of time: it maintains replicas for (almost) all parameters on all nodes. The reason for this is that, in common ML tasks, each node accesses almost all parameters over time. As for static full replication, this can make distributed implementations inefficient. For example, in our experiments in Section 4.4.2, ESSP on 8 nodes was more than 4x slower than an efficient single node implementation.

2.3 Related Work

We discuss the most closely related work (on PSs) in the previous section. In the following, we discuss other related work on efficient distributed ML.

2.3.1 Update Compression

Several forms of (lossy) compression have been proposed to reduce the size of the updates that need to be exchanged over the network. One direction is lower-precision updates, i.e., to reduce the number of bits for each element of the update (Seide et al. 2014; Wen et al. 2017; Alistarh et al. 2017; H. Zhang et al. 2017; Hubara et al. 2017; Bernstein et al. 2018). Another direction is sparsification, i.e., to selectively communicate only some elements of the update (Mu Li et al. 2014b; Aji and Heafield 2017; Lin et al. 2018; H. Wang et al. 2018). Some of these techniques accumulate un-synchronized updates locally and transfer them at a later point in time, such that most (or all) updates are eventually synchronized. Recent research has drawn into question how beneficial compression techniques are in recent distributed ML settings, as encoding and decoding take time and especially sparsification prevents optimizations for dense communication (Agarwal et al. 2022). Nevertheless, in general, work on update compression is orthogonal to the work in this thesis, and integrating these directions into PSs might make for interesting directions for future work.

2.3.2 Higher Compute Intensity

One way to reduce the impact of communication in distributed ML is to increase the computation-to-communication ratio of the update steps. This is commonly done by increasing the batch size of mini-batch SGD (Goyal et al. 2017; You et al. 2017b). With a larger batch size, more computation is necessary in one update step as the model is evaluated for each training example. At the same time, for dense ML tasks, the amount of communication remains constant as the gradients for the different training examples are summed up before communication. However, depending on model and data complexity, increasing the batch size potentially impairs model quality (specifically, generalization error) (Keskar et al. 2017; Masters and Luschi 2018; Shallue et al. 2019), giving rise to a field of research on maintaining high model quality with large batch sizes (You et al. 2017a; You et al. 2020;

Huo et al. 2021). Unfortunately, increasing batch size has only limited effect on compute intensity for ML tasks with sparse access—the focus of this thesis—because the amount of communication for an update step *increases* with increasing batch size for sparse ML tasks. The reason for this is that an update step in a sparse ML tasks communicates gradients only for the affected parameters (typically a tiny subset of all parameters). Each training example typically affects different parameters, such that an update step with more training examples affects more parameters. Thus, the larger the batch size, the more communication is required for this batch.

2.3.3 Deep Learning

Deep learning has become a highly popular subfield of ML, with a large body of research focusing on efficiently training multi-layer networks with synchronous SGD, with mostly dense access into large parameter tensors. The key challenge in this field is to efficiently exchange (entire) large tensors among nodes. One direction in this field is the development of efficient all-reduce methods (Sergeev and Balso 2018; Awan et al. 2017; S. Wang et al. 2019), e.g., by exchanging tensors in ring topologies, and alternative dense communication methods (Y. Jiang et al. 2020). Another direction is to overlap communication with computation of different layers of the model (Yanping Huang et al. 2019; Narayanan et al. 2019; Fan et al. 2021; Bowen Yang et al. 2021), e.g., to start exchanging the gradients for higher layers already while the backward pass of lower layers is still running (Yanping Huang et al. 2019). These research directions are not directly applicable to sparse ML tasks because all-reduce is not efficient for synchronizing sparsely updated tensors and there typically is no or only little potential for overlapping communication and computation synchronously within one batch in sparse ML models.

2.3.4 Other Task-Specific Approaches

There is a range of approaches to increase efficiency of distributed parameter management for specific classes of ML tasks other than deep learning. These approaches, too, are tailored to the respective class of tasks, and are not directly applicable to other tasks. For example, there are many approaches specifically for distributed training of large deep learning recommender systems (W. Zhao et al. 2020; Adnan et al. 2021; Yuzhen Huang et al. 2021; Miao et al. 2022). And there are many approaches specifically for distributed training of graph neural networks (R. Zhu et al. 2019; D. Zhang et al. 2020; Min et al. 2021; D. Zheng et al. 2020a; J. Peng et al. 2022). In contrast, we focus on general-purpose distributed parameter management in this thesis.

2.3.5 Faster Interconnect Hardware

Another direction to reduce the impact of communication is to develop and use faster interconnect hardware. Fast network interconnects, such as InfiniBand, enable data exchange in the order of 10s of gigabytes per second between nodes (e.g., 10.2 GB per second in a InfiniBand EDR 4x cluster (Ziegler et al. 2022)). Faster interconnects help to reduce the impact of communication overhead in distributed ML. However, existing interconnects do not eliminate the communication overhead bottleneck. For example, we run our experiments in Sections 4.4 and 5.4 on a cluster with a modern 100 GBit InfiniBand network. Nevertheless, existing approaches to parameter management (e.g., full replication, a classic PS, and existing replication PSs) are inefficient. In particular, modern interconnects do not eliminate the latency hierarchy, i.e., that, for example, access to local DRAM is faster (in the order of 100 nanoseconds) than access to remote memory (even with RDMA over InfiniBand, at least a few microseconds (Ziegler et al. 2020; Kalia et al. 2016)). This hierarchy is unlikely to disappear any time soon: latency tends to improve slower than bandwidth with new interconnects and is inherently limited by the speed of light (Patterson 2004).

Communication links that connect the GPUs of one node directly, such as NVLink, allow GPU-to-GPU transfers (within one node) in the order of 100s of gigabytes per second (A. Li et al. 2020). An interesting line of research aims to build PSs for the resulting heterogeneous communication topology (Y. Jiang et al. 2020; Miao et al. 2021; G. Wang et al. 2020; J. Jiang et al. 2017). It is a particularly interesting area for future work to apply the ideas of this thesis to such heterogeneous topologies (see also our discussion of future work in Chapter 6).

2.3.6 Decentralized Training

A further direction for reducing communication overhead is decentralized training approaches (Lian et al. 2017; Tang et al. 2018; Assran et al. 2019). The key idea is that the individual nodes of the cluster learn independently, and exchange (some of) their learning progress (i.e., the model parameters) only from time to time, and potentially only to some (and not all) other nodes. The main difference of decentralized approaches to the distributed parameter management approaches discussed above is that there is no ("central") point of truth for a parameter that is guaranteed to eventually receive all updates for this parameter. Often, these approaches draw ideas from gossip communication algorithms (Demers et al. 1987). They introduce novel training algorithms that differ from the popular and (comparably) well-understood

SGD training. In particular, these training algorithms have different convergence properties and guarantees (Lian et al. 2017). In contrast, we focus on SGD training in this thesis.

2.3.7 Programming Abstractions and Scheduling

There are many systems that introduce programming abstractions and/or ways to schedule computation for ML (often as parts of larger system stacks). Some of these are purpose-built for ML, such as PyTorch (Paszke et al. 2019; S. Li et al. 2020), TensorFlow (Abadi et al. 2016), MXNet (T. Chen et al. 2015), and SystemDS (Ghoting et al. 2011; Boehm et al. 2016; Boehm et al. 2020). Others are more general, but also applicable to ML tasks, such as data flow systems (Zaharia et al. 2010; Carbone et al. 2015) and graph-based systems (Low et al. 2012; Malewicz et al. 2010). Distributed parameter management is one component in the stack of these systems. In contrast to these broader systems, we specifically focus on this single component in this thesis. I.e., we treat the sequence of parameter access operations as set by the application and try to make PSs more efficient for common sequences. Such improvements to the PS component can potentially be leveraged by programming abstractions and scheduling approaches.

2.3.8 Key–Value Stores

PSs are key–value stores that are specialized for storing parameters of ML models. In fact, early work on PSs (Smola and Narayanamurthy 2010) used memcached,[4] an off-the-shelf general-purpose key–value store, to manage parameters. Since then, several different types of PSs, i.e., systems that are built for the specific purpose of storing ML parameters, have been developed (Mu Li et al. 2014a; Ho et al. 2013; J. Jiang et al. 2017; Yuzhen Huang et al. 2018; Jagerman et al. 2017; Z. Zhang et al. 2019; Y. Jiang et al. 2020). In contrast to general-purpose key–value stores, PSs (1) are designed for storing multi-dimensional arrays, such as vectors and tensors (Mu Li et al. 2014b), (2) are often co-located on the same nodes as the application processes to enable low-latency access (because ML applications access the PS frequently) (Jagerman et al. 2017; J. Jiang et al. 2017; Ho et al. 2013; Yuzhen Huang et al. 2018), (3) often work with relaxed consistency schemes, such as bounded staleness (Ho et al. 2013; Dai et al. 2015) or fully asynchronous updates (Dean et al. 2012), and (4) integrate ML-specific functionality directly into the PS, such as update filtering or compression (Mu Li et al. 2014b).

[4] `https://memcached.org/`

Chapter 3

Exploiting Locality: Dynamic Parameter Allocation

A set of implementations of distributed ML training reduce communication overhead by employing specific techniques to create and/or exploit *parameter access locality* (PAL). *PAL* refers to the tendency that different nodes access different subsets of parameters at specific points in time. There exist several different such *PAL techniques* (Gemulla et al. 2011; Yun et al. 2014; Teflioudi et al. 2012; Beutel et al. 2014; Lerer et al. 2019; Raman et al. 2019; Yu et al. 2015; B. Peng et al. 2017; Low et al. 2012; Gonzalez et al. 2012; Nakandala et al. 2019). Some techniques explicitly create locality, e.g., by clustering training data according to the parameters that training examples access, or by restricting nodes to specific parameter subsets at specific points in time. Another approach is to exploit locality that is inherent in common ML tasks.

Existing PSs provide only limited support for PAL techniques: some techniques are tedious to implement in PSs (because they require knowledge of PS internals and careful key design), others are entirely impossible to implement in existing PSs (because existing PSs allocate parameters statically). Due to this limited support, PAL techniques have to be implemented manually—outside PSs—, using low-level distributed programming primitives (Gemulla et al. 2011; Yun et al. 2014; Teflioudi et al. 2012; Lerer et al. 2019). This low-level programming requirement makes PAL techniques complex to implement and, consequently, hinders their adoption.

In this chapter, we explore whether and to what extent PSs can exploit locality, and whether that is beneficial. First, we discuss common PAL techniques and analyze what is necessary for PSs to support them (Section 3.1). Based on this analysis, we propose *dynamic parameter allocation* (DPA), i.e., that the PS adapts its parameter allocation to the ML task. DPA allows the PS to dynamically relocate parameters to where they are currently accessed, while providing location transparency and PS consistency guarantees, i.e., sequential consistency. We discuss design options for PSs with DPA and describe a prototype implementation of such a PS called Lapse (Section 3.2). Our experimental evaluation shows (i) that existing PSs—without DPA—barely outperform efficient single node baselines and (ii) that DPA drastically improves efficiency for some ML tasks (Section 3.3).

3.1 The Case for Dynamic Parameter Allocation

We outline common PAL techniques used in distributed ML (Section 3.1.1). For each technique, we discuss to what extent it is supported in existing PSs and identify what features are required to enable or improve support. Finally, we propose DPA, which enables PSs to exploit PAL techniques directly (Section 3.1.2).

3.1.1 PAL Techniques

We consider three common PAL techniques, i.e., techniques that applications employ to create and/or exploit access locality: data clustering, parameter blocking, and latency hiding.

Data Clustering

One method to reduce communication cost is to exploit structure in training data (Low et al. 2012; Gonzalez et al. 2012; R. Chen et al. 2015; Smola and Narayanamurthy 2010; Ahmed et al. 2012). For example, consider a training dataset that consists of documents written in two different languages and an ML model that associates a parameter with each word (e.g., a bag-of-words classifier or word embeddings). When processing a document during training, only the parameters for the words contained in the document are relevant. This property can be exploited using *data clustering*. For example, if a separate worker is used for the documents of each language, different workers access mostly separate parameters. This is an example of PAL: different workers access different subsets of the parameters at a given time. This locality can be exploited by allocating parameters to the worker machines that access them. Figure 3.1 depicts an example; here, rows correspond to documents, dots to words, and each parameter is allocated to the node that accesses the parameter most frequently.

Data clustering can be exploited in existing PSs in principle, although it is often painful to do so because PSs provide no direct control over the allocation of the parameters. Instead, parameters are typically partitioned

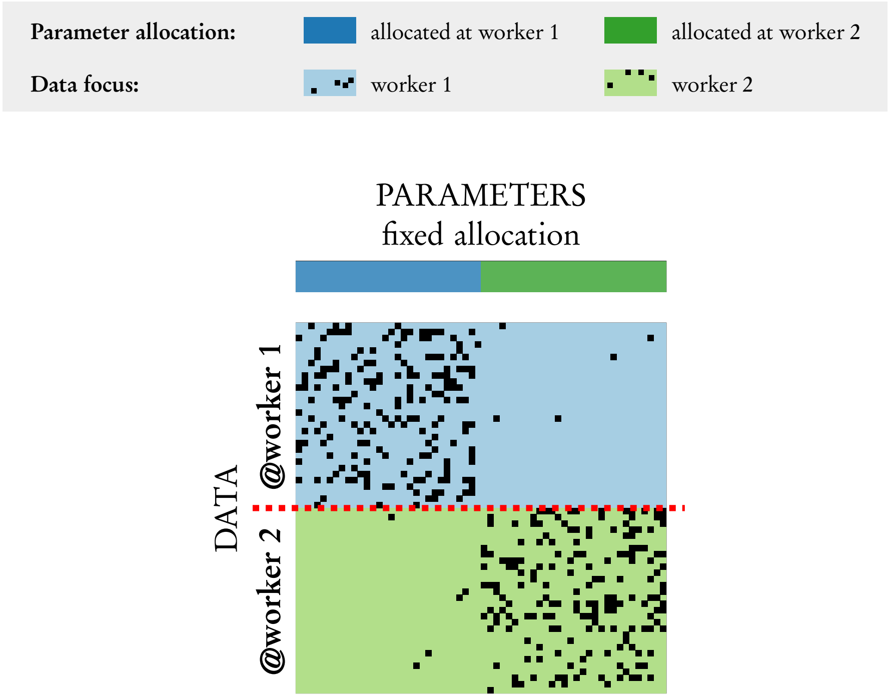


Figure 3.1: The *data clustering* PAL technique: the training data are clustered such that each worker accesses mostly a separate subset of parameters. Rows correspond to data points, columns to parameters, and black dots to parameter access.

using either hash or range partitioning. To exploit data clustering, applications may manually enforce the desired allocation by *key design*, i.e., by explicitly assigning keys to parameters such that the parameters are allocated to the desired node. Such an approach requires knowledge of PS internals, preprocessing of the training data, and a custom implementation for each task. To improve support for data clustering, PSs should provide support for explicit **parameter location control**.

To exploit data clustering, it is essential that the PS provides **fast access to local parameters**; e.g., by using shared memory as in manual implementations (Gemulla et al. 2011; Yun et al. 2014). However, to the best of our knowledge, all existing PSs access parameters either through inter-process (Mu Li et al. 2014a; Jagerman et al. 2017; J. Jiang et al. 2017) or inter-thread communication (Xing et al. 2015; Yuzhen Huang et al. 2018), leading to overly high access latency. We will see in the experiments in Section 3.3.2 that such inter-process and inter-thread communication are not sufficient to provide fast parameter access.

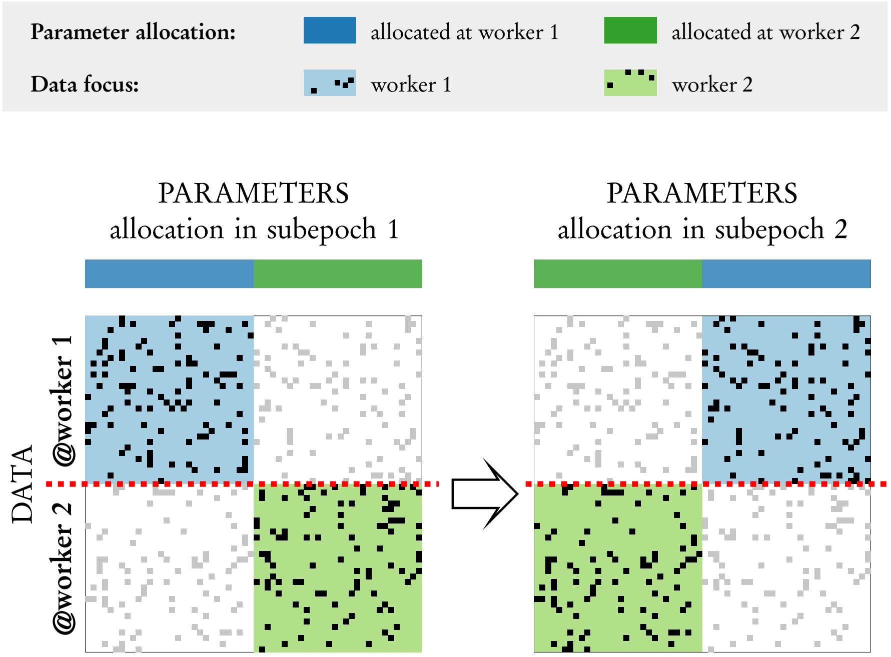


Figure 3.2: The *parameter blocking* PAL technique: within each subepoch, each worker is restricted to one block of parameters; which worker has access to which block changes from subepoch to subepoch. Rows correspond to data points, columns to parameters, black dots to parameter access in the current subepoch, and gray dots to parameter access in another subepoch.

Parameter Blocking

An alternative approach to provide PAL is to divide the model parameters into blocks. Training is split into *subepochs* such that each worker is restricted to one block of parameters within each subepoch. Which worker has access to which block changes from subepoch to subepoch, however. Such *parameter blocking* approaches have been developed for many ML algorithms, including matrix factorization (Gemulla et al. 2011; Yun et al. 2014; Teflioudi et al. 2012), tensor factorization (Beutel et al. 2014), latent dirichlet allocation (Yu et al. 2015; B. Peng et al. 2017), multinomial logistic regression (Raman et al. 2019), and knowledge graph embeddings (Lerer et al. 2019). They were also proposed for efficient multi-model training (Nakandala et al. 2019).

Manual implementations exploit parameter blocking by allocating parameters to the node where they are currently accessed (Gemulla et al. 2011; Teflioudi et al. 2012; Yun et al. 2014), thus eliminating network communication for individual parameter accesses. Communication is required only between subepochs. Figure 3.2 depicts a simplified example for a matrix factorization task (Gemulla et al. 2011). In the first subepoch, worker 1 (worker 2) focuses on the left (right) block of the model parameters. It does so by only processing the corresponding part of its data and ignoring the remainder. In the second subepoch, each worker processes the other block and the other part of its data. This process is repeated multiple times.

Existing PSs offer limited support for parameter blocking because they allocate parameters *statically*. This means that a parameter is assigned to one server and stays there throughout training. It is therefore not possible to dynamically allocate a parameter to the node where it is currently accessed. Parameter blocking can be emulated to some extent in replication PS architectures, however. This requires the creation of replicas for each block and forced refreshes of replicas between subepochs. Such an approach is limited to synchronous parameter blocking approaches, requires significant changes to the implementation, and induces unnecessary communication (because parameters are transferred via their server instead of directly from worker to worker). To exploit parameter blocking efficiently, PSs need to support **parameter relocation**, i.e., the ability to move model parameters among nodes during run time.

Latency Hiding

Latency hiding techniques reduce communication overhead (but not communication itself). For example, prefetching is commonly used when there is a distinction between local and remote data, such as in processor caches (A. Smith 1982) or distributed systems (Steen and Tanenbaum 2017). In distributed ML, the latency of parameter access can be reduced by ensuring that a parameter value is already present at a worker when it is accessed (Dai et al. 2015; Cui et al. 2014; Teflioudi et al. 2012). Such an approach is beneficial when parameter access is sparse, i.e., each worker accesses few parameters at a time. Note that latency hiding does not require the ML algorithm to explicitly create locality (as data clustering and parameter blocking do). Thus, latency hiding is both easier to apply and more widely applicable to ML tasks than data clustering and parameter blocking.

Prefetching can be implemented by pulling a parameter *asynchronously* before it is needed. The disadvantage of this approach is that an application needs to manage prefetched parameters, and that updates that occur between prefetching a parameter and using it are not visible. Therefore, such an approach provides neither sequential nor causal consistency (we discuss and analyze consistency guarantees of PSs in Section 3.2.4). Moreover, the exchange of parameters between different workers always involves the server, which may be inefficient. An alternative approach is the ESSP consistency protocol of Petuum (Dai et al. 2015), which proactively replicates all previ-

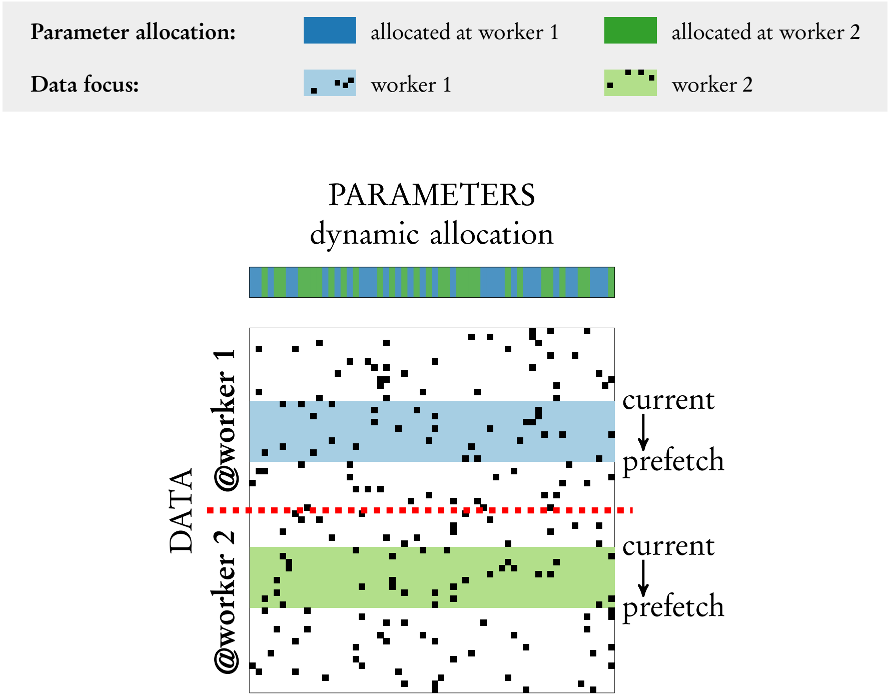


Figure 3.3: The *latency hiding* PAL technique: asynchronously prefetching (or prelocalizing) of parameter values, such that they can be accessed locally, hides access latency. Rows correspond to data points, columns to parameters, and black dots to parameter access.

ously accessed parameters at a node. This approach avoids manual parameter management, but does not provide sequential consistency. ESSP also causes over-communication, as typically, after a short warm-up time, nodes will hold more replicas than necessary (we discuss this effect of ESSP in more detail in Section 4.2).

An alternative to prefetching is to *prelocalize* a parameter before access, i.e., to reallocate the parameter from its current node to the node where it is accessed and to *keep it there* afterward (until it is prelocalized by some other worker). This approach is illustrated in Figure 3.3. Note that, in contrast to prefetching, the parameter is not replicated. Consequently, parameter updates by other workers are immediately visible after prelocalization. Moreover, there is no need to write local updates back to a remote location as the parameter is now stored locally. To support prelocalization, PSs need to support **parameter relocation with consistent access** before, during, and after relocation. For example, no updates should be lost if a parameter is accessed during relocation. Ideally, a PS with relocation would maintain the strong consistency guarantees of classic PSs.

3.1.2 Dynamic Parameter Allocation

As discussed above, existing PSs offer limited support for the PAL techniques of distributed ML. The main obstacles are that existing PSs provide limited control over parameter allocation and perform allocation statically. In more detail, we identified the following requirements to enable or improve support:

Fast local access. PSs should provide low-latency access to local parameters.

Parameter location control. PSs should allow applications to control where a parameter is stored.

Parameter relocation. PSs should support relocating parameters between servers during run time.

Consistent access. Parameter access should be consistent before, during, and after a relocation.

To satisfy these requirements, the PS must support DPA, i.e., it must be able to change the allocation of parameters during run time. While doing so, the PS semantics must not change: `pull` and `push` operations need to be oblivious of a parameter's current location and provide correct results whether or not the parameter is currently being relocated. This requires the PS to manage parameter locations, to transparently route parameter accesses to the parameter's current location, to handle reads and writes correctly during relocations, and to provide to applications new primitives to initiate parameter relocations.

A DPA PS enables support for PAL techniques roughly as follows: each worker instructs the PS to *localize* the parameters that it will access frequently in the near future, but otherwise uses the PS as it would use any other PS, i.e., via the `pull` and `push` primitives. For data clustering, applications control parameter locations once in the beginning: each node localizes the parameters that it accesses more frequently than the other nodes. Subsequently, the majority of parameter accesses (using `pull` and `push`) is local. For parameter blocking, at the beginning of each subepoch, applications move parameters of a block to the node that accesses them during the subepoch. Parameter accesses (both reads and writes) within the subepoch then require no further network communication. Finally, for latency hiding, workers prelocalize parameters before accessing them. When the parameter is accessed, latency is low because the parameter is already local (unless another worker localized the parameter in the meantime). Concurrent updates by other workers are seen locally, because the PS routes them to the parameter's current location.

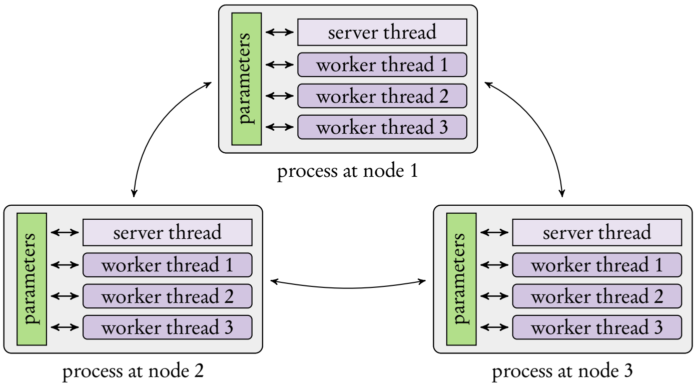


Figure 3.4: PS architecture with server and worker threads co-located in one process per node.

3.2 The Lapse Parameter Server

To explore the suitability of PSs with DPA as well as architectural design choices, we created Lapse. Lapse is based on PS-Lite (Mu Li et al. 2014a) and aims to fulfill the requirements established in the previous sections. In particular, Lapse provides fast access to local parameters, consistency guarantees similar to classic PSs, and efficient parameter relocation. We start with a brief overview of Lapse and subsequently discuss individual components, including parameter relocation, parameter access, consistency, location management, granularity, and important implementation aspects.

3.2.1 Overview

Lapse co-locates worker and server threads in the same process, as illustrated in Figure 3.4, because this architecture facilitates low-latency local parameter access (see below).

API

Lapse adds a single primitive called `localize` to the API of the PS; see Table 3.1. The primitive takes the keys of one or more parameters as arguments. When a worker issues a `localize`, it requests that all provided parameters are relocated to its node. Lapse then transparently relocates these parameters and future accesses by the worker require no further network communication. Algorithm 3.1 depicts one example of how the primitive can be used, for implementing the asynchronous distributed SGD example from Section 2.2. In this example, the data loader initiates the parameter relocation (line 6) when the batch is prepared. Batch preparation runs pipeline parallel with the training (see Figure 2.1 on page 9). Thus, the relocation for a batch is

Table 3.1: Primitives of Lapse, a PS with dynamic parameter allocation. The push primitive is cumulative. All primitives can run synchronously or asynchronously. Compared to classic PSs, Lapse adds one primitive to initiate parameter relocations.

Primitive	Support for		Description
	sync.	async.	
`pull(parameters)`	✓	✓	Read the values of `parameters` from the corresponding servers.
`push(parameters, updates)`	✓	✓	Send `updates` for `parameters` to the corresponding servers.
`localize(parameters)`	✓	✓	Request local allocation of `parameters`.

triggered shortly before the training on this batch starts, so that (ideally) relocation finishes before the worker thread reads and writes the parameters (lines 8 and 10, respectively).

We opted for the `localize` primitive—instead of a more general primitive that allows for relocation among arbitrary nodes—because it is simpler and sufficiently expressive to support PAL techniques. Furthermore, `localize` preserves the PS property that two workers logically interact only via the servers (and not directly) (Mu Li et al. 2014a). Workers access localized parameters in the same way as non-localized parameters. This allows Lapse to relocate parameters without affecting workers that use them.

Location Management

Lapse manages parameter locations with a decentralized home node approach (Steen and Tanenbaum 2017): for each parameter, there is one *owner node* that stores the current parameter value and one *home node* that knows the parameter's current location. The home node is assigned statically as in existing PSs, whereas the owner node changes dynamically during run time. We further discuss location management in Section 3.2.5.

Parameter Access

Lapse ensures that local parameter access is fast by accessing local parameters via shared memory. For non-local parameter access, Lapse sends a message to the home node, which then forwards the message to the current owner of a parameter. Lapse optionally supports location caches, which eliminate the message to the home node if a parameter is accessed repeatedly while it is not relocated. See Section 3.2.3 for details.

Algorithm 3.1: Distributed asynchronous SGD with Lapse. Differences to the Classic PS implementation (Algorithm 2.3 on page 17) are highlighted in green.

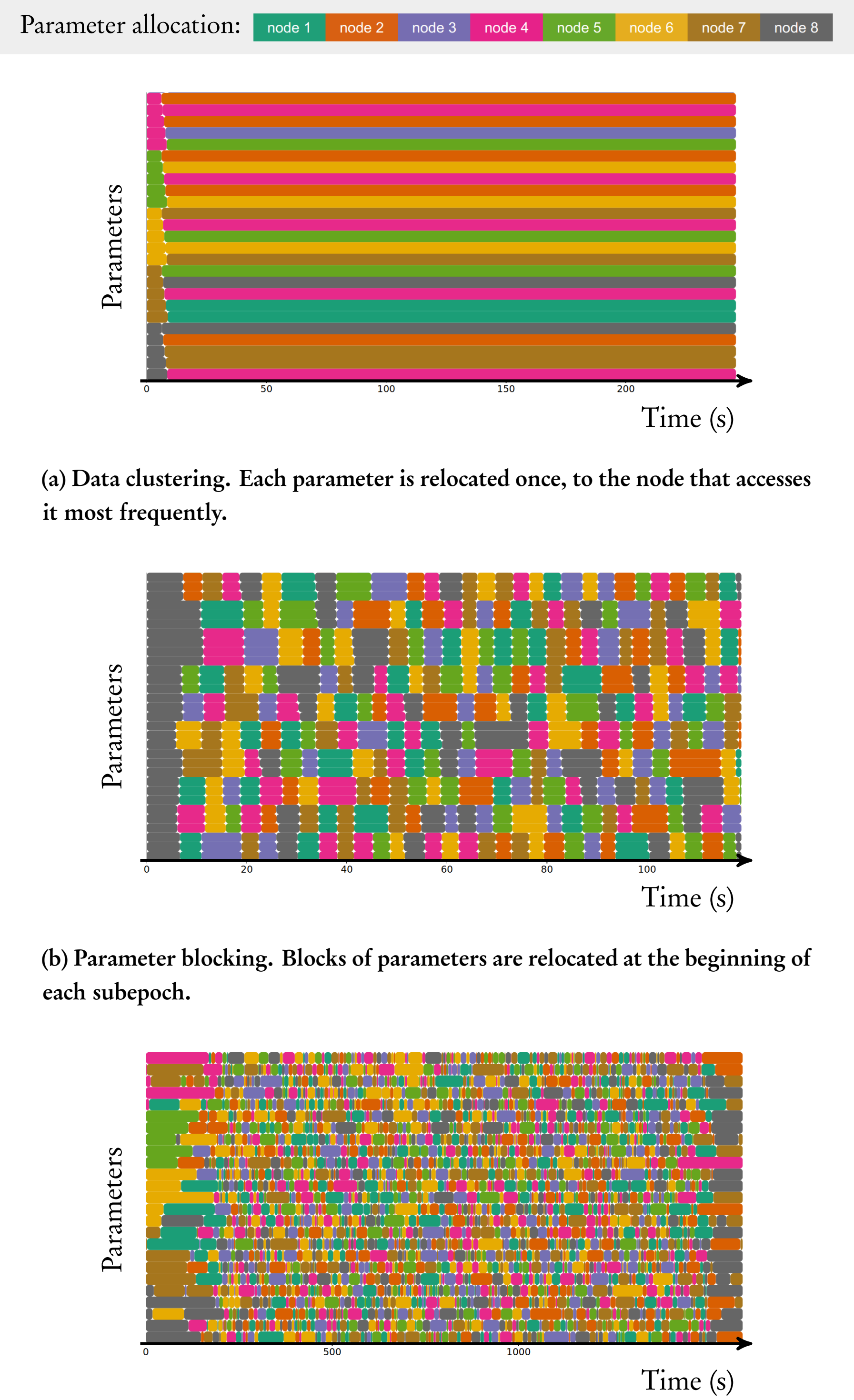
```
Data: $\mathscr{D}$: training dataset,
      $num\_epochs$: number of epochs to run,
      $batch\_size$: batch size,
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads
for epoch ← 1 to num_epochs do
    b = num_batches ( $\mathscr{D}$, batch_size, t, T )
    // data loading (pipeline parallel with training, in separate thread(s))
    $\mathscr{B}$ = []
    for i ← 1 to b do
        $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch, t, T )
        localize ( keys($\mathscr{B}_i$) )
    // training
    for i ← 1 to b do
        $w$ = pull ( keys($\mathscr{B}_i$) )
        $\Delta w$ = compute_update ( $\mathscr{B}_i$, $w$ )
        push ( keys($\mathscr{B}_i$), $\Delta w$ )
```

Parameter Relocation

A localize call requires Lapse to relocate the parameter to the new owner and update the location information on the home node. Care needs to be taken that push and pull operations that are issued while the parameter is relocated are handled correctly. Lapse ensures correctness by forwarding all operations to the new owner immediately, possibly before the relocation is finished. The new owner simply queues all operations until the relocation is finished. Lapse sends at most three messages for a relocation of one parameter and pauses processing for the relocated parameter only for the time that it takes to send one network message. The entire protocol is described in Section 3.2.2.

Consistency

In general, Lapse provides the sequential consistency guarantees of classic PSs even in the presence of relocations. We show in Section 3.2.4 that location caches may impact consistency guarantees. In particular, when location caches are used, Lapse still provides sequential consistency for synchronous operations, but only eventual consistency for asynchronous operations.



(a) Data clustering. Each parameter is relocated once, to the node that accesses it most frequently.

(b) Parameter blocking. Blocks of parameters are relocated at the beginning of each subepoch.

(c) Latency hiding. A parameter is relocated to a node whenever that node accesses the parameter.

Figure 3.5: Parameter allocation in Lapse for three example workloads with different PAL techniques. Each row corresponds to one parameter, the x-axis depicts time, and colors indicate the current allocation of a parameter.

3.2.2 Parameter Relocation

A key component of Lapse is the relocation of parameters. It is important that this relocation is efficient because PAL techniques may relocate parameters frequently (up to 36 000 keys and 289 million parameter values per second in our experiments). Figure 3.5 illustrates how Lapse relocates parameters for example workloads with the data clustering, parameter blocking, and latency hiding PAL techniques. In the following, we discuss how Lapse relocates parameters, how it manages operations that are issued during a relocation, and how it handles simultaneous relocation requests by multiple nodes.

During a localize operation, (1) the home node needs to be informed of the location change, (2) the parameter needs to be moved from its current owner to the new owner, and (3) Lapse needs to stop processing operations at the current owner and start processing operations at the new owner. Key decisions are what messages to send and how to handle operations that are issued during parameter relocation. Lapse aims to keep both the relocation time and the blocking time for a relocation short. We use *relocation time* to refer to the time between issuing a localize call and the moment when the new owner starts answering operations locally. By *blocking time*, we mean the time in which Lapse cannot immediately process operations for the parameter (but instead queues operations for later processing).[1] The two measures usually differ because the current owner continues to process operations for some time after the localize call is issued at the requesting node, i.e., the relocation time may be larger than the blocking time.

We refer to the node that issued the localize operation as the *requester node*. The requester node is the new owner of the parameter after the relocation has finished. Lapse sends three messages in total to relocate a parameter, see Figure 3.6. ❶ The requester node informs the home node of the parameter about the location change. The home node updates the location information immediately and starts routing parameter accesses for the relocated parameter to the requester node. ❷ The home node instructs the old owner to stop processing parameter accesses for the relocated parameter, to remove the parameter from its local storage, and to transfer it to the requester node. ❸ The old owner hands over the parameter to the requester node. The requester node inserts the parameter into its local storage and starts processing parameter accesses for the relocated parameter.

During the relocation, the requester node queues all parameter accesses that involve the relocated parameter. It queues both local accesses (i.e., accesses

[1] Both pull and push operations need to be queued during this period. However, as push operations are typically run asynchronously (i.e., the application continues execution before the push returns), the blocking time affects pull operations more directly.

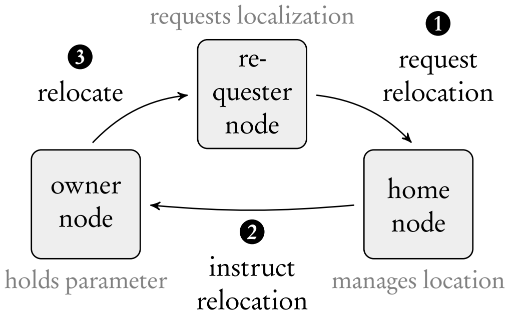


Figure 3.6: A worker requests to localize a parameter. Lapse transparently relocates the parameter from the current owner to the requester node and informs the home node of the location change.

by workers at the requester node) and remote accesses that are routed to it before the relocation is finished. Once the relocation is completed, it processes the queued operations in order and then starts handling further accesses as the new owner. As discussed in Section 3.2.4, this approach ensures that sequential consistency is maintained.

In the absence of other operations, the relocation time for this protocol is approximately the time for sending three messages over the network, and the blocking time is the time for sending one message (because operations are queued at the requester and the home node starts forwarding to the requester immediately). One may try to reduce blocking time by letting the old owner process operations until the relocation is complete (and forwarding all updates to the new owner). However, such an approach would require additional communication and would increase relocation time. The protocol used by Lapse strikes a balance between short relocation and short blocking time.

If multiple nodes simultaneously localize the same parameter, there is a *localization conflict*: without replicas, a parameter resides at only one node at a time. In the case of a localization conflict, the above protocol transfers the parameter to each requesting node once (in the order the relocation requests arrive at the home node). This gives each node a short opportunity to process the parameter locally, but also causes communication overhead for frequently localized parameters (because it repeatedly transfers the parameter value, potentially in cycles). A short localize moratorium, in which further localize requests are ignored, may reduce this cost, but would change the semantics of the localize primitive, increase complexity, and may impact overall efficiency. Additionally, a centralized protocol, i.e., a protocol in which workers *request* localization (rather than explicitly triggering relocations) and a central instance decides where to allocate the parameter, could avoid frequent relocations, e.g., by keeping the parameter at the node with the most requests in some time period. We did not consider these approaches in Lapse. We do, however, pick up these considerations in the following chapters of this thesis: we (i) explore how localization conflicts can be avoided by replicating (some of) the parameters rather than relocating them (at the cost of introducing staleness and, thus, falling back to weaker consistency guarantees) (Chapter 4) and (ii) develop an approach in which workers indeed *request* parameters (we call these requests *intent signals*) and the requests for one parameter are coordinated centrally to decide where this parameter should be allocated and replicated (Chapter 5).

3.2.3 Parameter Access

When effective PAL techniques are used, the majority of parameter accesses are processed locally. Nevertheless, remote access to all parameters may arise at all times and needs to be handled appropriately. We now discuss how Lapse handles local access, remote access, location caches, and access to a parameter that is currently relocating.

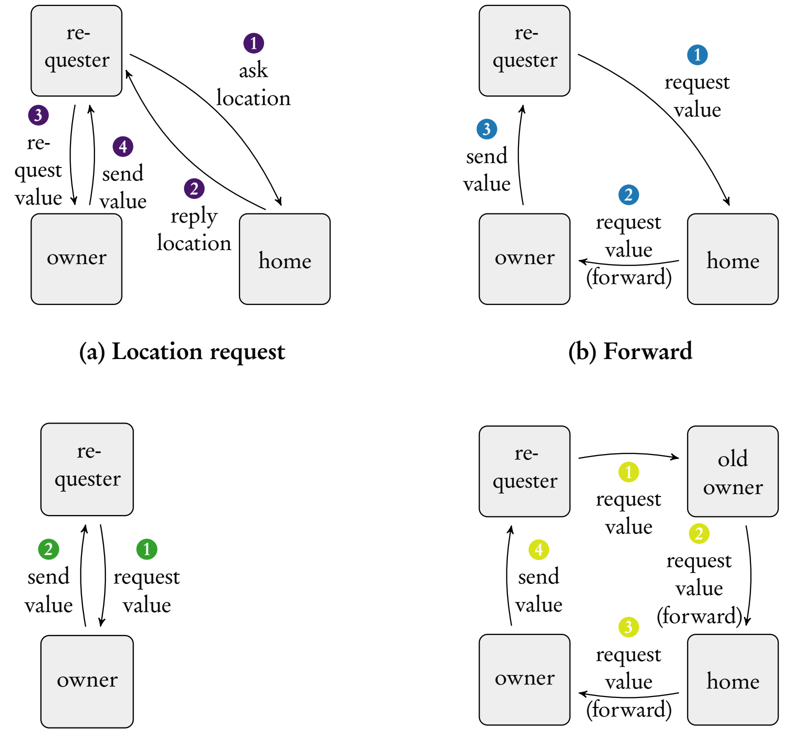
Local Access

Lapse provides fast local parameter access by accessing locally stored parameters via shared memory directly from the worker threads, i.e., without involving the PS thread (see Figure 3.4) or other nodes. In our experiments, accessing the parameter storage via shared memory provided up to 6x lower latency than access via a PS thread using queues (as implemented in Petuum (Xing et al. 2015), for example). As other PSs, Lapse guarantees per-key atomic reads and writes;[2] it does so using latches (i.e., locks held for the time of the operation) for local accesses (see Section 3.2.7).

Remote Access

We now discuss remote parameter access and first assume that there is no location caching. There are two basic strategies. In the *location request* strategy, the worker retrieves the current owner of the parameter from the home node and subsequently sends the pull or push request to that owner (Figure 3.7a). In the *forward* strategy, the worker sends the request itself to the home node, which then forwards it to the current owner (Figure 3.7b). Lapse employs the *forward* strategy because (i) it always uses up-to-date location information for routing decisions and (ii) it requires one message less than *location request*. The *forward* strategy uses the latest location information for routing because the home node, which holds the location information, sends

[2] Such per-key atomic access is common even in asynchronous SGD (Niu et al. 2011).



(a) Location request

(b) Forward

(c) Correct location cache

(d) Stale cache: double-forward

Figure 3.7: Routing for non-local parameter access. If the location of a parameter is unknown, Lapse employs the *forward* strategy (Figure b), requiring 3 messages. Lapse optionally supports location caches. A correct cache reduces the number of messages to 2 (Figure c), a stale cache increases it to 4 (Figure d). The figures depict labels for `pull` messages. Labels for `push` are analogously: *request value* → *send update*, *send value* → *confirm update*, *request value (forward)* → *send update (forward)*

the request to the owner (message ❷). In contrast, in *location request*, the requester node sends the request (message ❸) based on the location obtained from the home node. This location may be outdated if another worker requests a relocation after the home node replied (message ❷). In this case, the requester node may send message ❸ to an outdated owner; such a case would require special handling.

Location Caching

Lapse provides the option to cache the locations of recently accessed parameters. This allows workers to contact the current owner directly (Figure 3.7c), reducing the number of necessary messages to two. To avoid managing cached locations and sending invalidation messages, the location caches are updated only after `push` and `pull` operations and after parameter relocations (i.e., without any additional messages). As a consequence, the cache may hold stale entries. If such an entry is used, Lapse uses a *double-forward* approach, which increases the number of sent messages by one (Figure 3.7d).

Access During Relocation

Workers can issue operations for any parameter at any time, including when a parameter is relocating. In the following, we discuss how Lapse handles different possible scenarios of operations on a relocating parameter. First, suppose that the requester node (to which the parameter is currently relocating) accesses the parameter. Lapse then locally queues the request at the requester node and processes it when the relocation is finished. Second, suppose that the old owner accesses the parameter. Lapse processes the parameter access locally if it occurs before the parameter leaves the local store. Otherwise, Lapse sends the operation to the new owner and processes it there. Finally, consider that a third node (neither the requester nor the old owner) accesses the parameter. If location caches are disabled, there are two cases. (1) The access arrives at the home node before the relocation. Then Lapse forwards the access to the old owner and processes it there (before the relocation). (2) The access arrives at the home node after the relocation. Then Lapse forwards and processes it at the new owner. If necessary, the new owner queues the access until the relocation is finished. With location caches, Lapse additionally processes the request at the old owner if the parameter's location is cached correctly at the requester node and the access arrives at the old owner before the relocation.

Table 3.2: Per-key consistency guarantees of PS architectures, using representatives for types: PS-Lite (Mu Li et al. 2014a) for classic and Petuum (Xing et al. 2015) for replication.

Parameter Server	Classic		Lapse			Replication
Synchronous	sync	async	sync	async		sync, async
Location caches				off	on	
Eventual	✓	✓	✓	✓	✓	✓
PRAM[a] (Lipton and Sandberg 1988)	✓	✓[b]	✓	✓[b]	×	✓
Causal (Hutto and Ahamad 1990)	✓	✓[b]	✓	✓[b]	×	×
Sequential (Lamport 1979)	✓	✓[b]	✓	✓[b]	×	×
Serializability	×	×	×	×	×	×

[a] I.e., monotonic reads, monotonic writes, and read your writes

[b] Assuming that the network layer preserves message order (which is true for Lapse and PS-Lite)

3.2.4 Consistency

In this section, we analyze the consistency properties of Lapse and compare them to classic PSs, i.e., to PS-Lite (Mu Li et al. 2014a). Table 3.2 provides a summary. Consistency guarantees affect the convergence of ML algorithms in the distributed setting; in particular, relaxed consistency can slow down convergence (Ho et al. 2013; Dai et al. 2015). The extent of this impact differs from task to task (Ho et al. 2013). None of the existing PSs guarantee serializability, as pull and push operations of different workers can overlap arbitrarily; neither do PSs give consistency guarantees across multiple keys. PSs can, however, provide per-key sequential consistency. Sequential consistency provides two properties (Lamport 1979): (1) each worker's operations are executed in the order specified by the worker, and (2) the result of any execution is equivalent to an execution of the operations of all workers in some sequential order. In the following, we study per-key sequential consistency for synchronous and asynchronous operations. Replication PSs do not provide sequential consistency. They provide weaker forms, specifically PRAM consistency and eventual consistency (Ho et al. 2013). We assume in the following that nodes process messages in the order they arrive (which is true for PS-Lite and Lapse).

Synchronous Operations

A classic PS guarantees sequential consistency: it provides property (1) because workers block during synchronous operations, preventing reordering, and (2) because all operations on one parameter are performed sequentially by its owner.

Theorem 1. *Lapse guarantees sequential consistency for synchronous operations.*

Proof. In the absence of relocations, Lapse provides sequential consistency, analogously to classic PSs: it provides property (1) because workers block during synchronous operations and property (2) because all operations on one parameter are performed sequentially by the owner of this parameter. In the presence of relocations, it provides property (1) because synchronous operations also block the worker if a parameter relocates. It provides property (2) because, at each point in time, only one node processes operations for one parameter. During a relocation, the old owner processes operations until the parameter leaves the local store (at this time, no further operations for this key are in the old owner's queue). Then the parameter is transferred to the new owner, which then starts processing. The new owner queues concurrent operations until the relocation is finished and then processes them in sequence. Lapse enforces sequential execution among local threads with latches and blocking operations. □

Asynchronous Operations

A classic PS such as PS-Lite provides sequential consistency for asynchronous operations.[3] Property (1) requires that operations reach the responsible server in program order (as the worker does not block during an asynchronous operation). This is the case in PS-Lite as it sends each message directly to the responsible server. Property (2) is given as for synchronous operations.

Theorem 2. *Lapse without location caches guarantees sequential consistency for asynchronous operations.*

Proof. For property (1), first suppose that there is no concurrent relocation. Lapse routes the operations of a worker on one parameter to the parameter's home node and from there to the owner. Message order is preserved in both steps under our assumptions. Now suppose that the parameter is relocated in-between operations. In this case, the old owner processes all operations that arrive at the home node before the relocation. Then the parameter is moved to the new owner, which then takes over and processes all operations that arrive at the home node after the relocation. Again, message order is preserved in all steps, such that Lapse provides property (1). It provides property (2) by the same argument as for Theorem 1. □

Theorem 3. *Lapse with location caches does not provide sequential consistency for asynchronous operations.*

[3] We assume that the network layer preserves message order. This is the case in PS-Lite and Lapse because they use TCP and send operations of a thread over the same connection.

Table 3.3: Location management strategies. N is the number of nodes, K is the number of parameter keys. In practice, if one operation accesses multiple keys, the total number of messages potentially scales sub-linearly because Lapse groups messages when possible, see Section 3.2.7.

Strategy	Storage	Number of messages for	
	(per node)	remote access	relocation
Static partition	0	2	n/a
Broadcast operations	0	N	0
Broadcast relocations	K	2	N
Home node	K/N	3[a]	3

[a] 3 messages if uncached, 2 with correct cache, 4 with stale cache

Proof. Lapse does not provide property (1) because a location cache change can cause two operations to be routed differently, which can change message order at the recipient. For example, consider two operations O_1 and O_2 of one worker. The worker first issues operation O_1, then operation O_2. It is possible that operation O_1 is sent to the currently cached, but outdated, owner. Then the location cache is updated (by another returning operation) and operation O_2 is sent directly to the current owner. With this, it is possible that operation O_2 is processed before operation O_1, because operation O_1 has to be double-forwarded to the current owner. This breaks sequential, causal, and PRAM consistency. □

3.2.5 Location Management

There are several strategies for managing location information in a PS with DPA. Key questions are how to store and communicate knowledge about which server is currently responsible for a parameter. Table 3.3 contrasts several possible strategies. For reference, we include the static partitioning of existing PSs (which does not support DPA). In the following, we discuss the different strategies. We refer to the number of nodes as N and to the number of keys as K.

Broadcast Operations

One strategy is to avoid storing any location information and instead broadcast the request to all nodes for each non-local parameter access. Then, only the server that currently holds the parameter responds to the request (all other servers ignore the message). This requires no storage but sends N messages per parameter access ($N - 1$ messages to all other nodes, one reply back to the requester). This high communication cost is not acceptable within a PS.

Broadcast Relocations

An alternative strategy is to replicate location information to all nodes. This requires to store K locations on each node (one for each of K keys). An advantage of this approach is that only two messages are required per remote parameter access (one request to the current owner of the parameter and the response). However, storage cost may be high when there is a large number of parameters and each location change has to be propagated to all nodes. The simplest way to do this is via direct mail, i.e., by sending $N-2$ additional messages to inform all nodes that were not involved in a parameter relocation. Gossip protocols (Demers et al. 1987) could be used to reduce this communication overhead.

Home Node

Lapse uses a home node strategy, inspired by distributed hash tables (Ratnasamy et al. 2001; Stoica et al. 2001; Rowstron and Druschel 2001; B. Zhao et al. 2004) and home-based approaches in general (Steen and Tanenbaum 2017). The home node of a parameter knows which node currently holds the parameter. Thus, if any node does not know the location of a parameter, it sends a request to the home node of that parameter. As discussed in Section 3.2.3, this requires at least one additional message for remote parameter access. A home node is assigned to each parameter using static partitioning, e.g., using range or hash partitioning. A simpler, but not scalable variant of this strategy is to have one centralized home node that knows the locations of all parameters. We discard this strategy because it limits the number of model parameters to the size of one node and creates a scalability bottleneck at the central home node.

Lapse employs the (decentralized) home node strategy because it requires little storage overhead and sends few messages for remote parameter access, especially when paired with location caches.

3.2.6 Granularity of Location Management

Location can be managed at different granularities, e.g., for each key or for ranges of keys. Lapse manages parameter location per key and allows applications to localize multiple parameters in a single localize operation.

This provides high flexibility, but can cause overhead if applications do not require fine-grained location control. For example, parameter blocking algorithms (Gemulla et al. 2011; Teflioudi et al. 2012; Yun et al. 2014) relocate parameters exclusively in static (pre-defined) blocks. For such algorithms, a possible optimization would be to manage location on group level. This would reduce storage requirements and would allow the system to optimize for communication of these groups. We do not consider such optimizations because Lapse aims to support many PAL methods, including ones that require fine-grained location control, such as latency hiding.

3.2.7 Important Implementation Aspects

In this section, we discuss implementation aspects that are key for the performance or the consistency of Lapse.

Message Grouping

If a single push, pull, or localize operation includes more than one parameter, Lapse groups messages that go to the same node to reduce network overhead. For example, consider that one localize call relocates multiple parameters. If two of the parameters are managed by the same home node, Lapse sends only one message from requester to this home node. If the two parameters then also currently reside at the same location, Lapse again sends only one message from home node to the current owner and one back from the current owner to the requester. Message grouping adds system complexity, but is highly beneficial when clients access or localize sets of parameters at once.

Local Parameter Store

As other PSs (Ho et al. 2013; Mu Li et al. 2014a), Lapse provides two variants for the local parameter store: dense arrays and sparse maps. Dense parameter storage is suitable if parameter keys are contiguous; sparse storage is suitable when they are not. Lapse uses a list of L latches to synchronize parameter access, while allowing parallel access to different parameters. A parameter with key k is protected by latch $k \bmod L$. Applications can customize L. A default value of $L = 1000$ latches worked well in our experiments.

No Message Prioritization

To reduce blocking time, Lapse could have opted to prioritize the processing of messages that belong to parameter relocations. However, this prioritization would break most consistency guarantees for asynchronous operations (i.e., sequential, causal, and PRAM consistency). The reason for this is that an "instruct relocation" message could overtake a parameter access message at the old owner of a relocation. The old owner would then reroute the parameter access message, such that it potentially arrives at the new owner after parameter access messages that were issued later. Therefore, Lapse does not prioritize messages.

3.3 Experiments

We conducted an experimental study to investigate the efficiency of classic PSs (Section 3.3.2) and whether it can be beneficial to integrate PAL techniques into PSs (Section 3.3.3). Further, we investigated how efficient Lapse is in comparison to a task-specific low-level implementation (Section 3.3.4) and replication PSs (Section 3.3.5), and conducted an ablation study (Section 3.3.6). Our source code, datasets, and information on reproducing our experiments are available online.[4]

Our major insights are: (i) Classic PSs suffered from severe communication overhead compared to a single node: using Classic PSs, 2–8 nodes were slower than 1 node in all tested tasks. (ii) Integrating PAL techniques into the PS reduced this communication overhead: Lapse was 4–203x faster than a classic PS, with 8 nodes outperforming 1 node by up to 9x. (iii) Lapse scaled better than a state-of-the-art replication PS (8 nodes were 9x vs. 2.9x faster than 1 node).

3.3.1 Experimental Setup

Tasks

We considered three popular ML tasks that require long training: matrix factorization, knowledge graph embeddings, and word embeddings. Table 3.4 summarizes details about the models and the datasets that we used for these tasks. We employ varied PAL techniques for the tasks. In the following, we briefly discuss each task.

Matrix factorization Low-rank matrix factorization is a common tool for analyzing and modeling dyadic data, e.g., in collaborative filtering for recommender systems (Koren et al. 2009). We employed a parameter blocking approach (Gemulla et al. 2011) to create and exploit PAL: communication happens only between subepochs; within a subepoch, all parameter access is local. We implemented this algorithm in PS-Lite (a classic PS), Petuum (a replication PS), and Lapse. Further, we compared to a task-specific and tuned low-level implementation of this parameter blocking approach.[5] We used two synthetic datasets from (Makari et al. 2015), because the largest openly available dataset that we are aware of is only 7.6 GB large.[6] For both datasets, we ran

[4] `https://github.com/alexrenz/lapse-ps/tree/vldb20/`

[5] `https://github.com/uma-pi1/DSGDpp`

[6] We adopt these datasets from (Makari et al. 2015). Note that the revealed cells in both datasets are distributed uniformly across rows and columns, whereas real-world datasets often have skewed distributions (Meka et al. 2009). For experiments with skewed MF datasets, see

Table 3.4: ML tasks, models, and datasets. The rightmost columns depict the number of key accesses and the size of read parameters (per second, for a single thread), respectively.

Task	Model parameters				Data			Param. Access	
	Model	Keys	Values	Size	Dataset	Data points	Size	Keys/s	MB/s
Matrix factorization	Latent factors, rank 100	6.4 M	640 M	4.8 GB	3.4m × 3m matrix	1 000 M	31 GB	414 k	315
	Latent factors, rank 100	11.0 M	1 100 M	8.2 GB	10m × 1m matrix	1 000 M	31 GB	316 k	241
Knowledge graph embeddings	ComplEx, dim. 100	0.5 M	98 M	0.7 GB	DBpedia-500k	3 M	47 MB	312 k	476
	ComplEx, dim. 4 000	0.5 M	3 929 M	29.3 GB	DBpedia-500k	3 M	47 MB	11 k	643
	RESCAL, dim. 100	0.5 M	110 M	0.8 GB	DBpedia-500k	3 M	47 MB	12 k	614
Word embeddings	Word2Vec, dim. 1 000	1.1 M	1 102 M	4.1 GB	1b word benchm.	375 M	3 GB	17 k	65

a factorization of rank 100. In all PSs, we ran a global barrier after each subepoch to ensure consistency. In Petuum, to ensure consistent replicas, we issued one clock after each subepoch and set a staleness threshold of 1. Petuum's own matrix factorization implementation ran out of memory because it stores dense matrices.

Knowledge graph embeddings Knowledge graph embedding (KGE) models learn algebraic representations of the entities and relations in a knowledge graph. For example, these representations have been applied successfully to infer missing links in knowledge graphs (Nickel et al. 2016a). A vast number of KGE models has been proposed (Nickel et al. 2011; Bordes et al. 2013; Nickel et al. 2016b; Bishan Yang et al. 2015; H. Liu et al. 2017), with different training techniques (Ruffinelli et al. 2020). We studied two models as representatives: RESCAL (Nickel et al. 2011) and ComplEx (Trouillon et al. 2016). We employed data clustering and latency hiding to create and exploit PAL. We used the DBpedia-500k dataset (Shi and Weninger 2018), a real-world knowledge graph that contains 490 598 entities and 573 relations of DBpedia (Auer et al. 2007). We ran the common setting of SGD with AdaGrad (Duchi et al. 2011) and negative sampling (Ruffinelli et al. 2020; H. Liu et al. 2017). We stored the AdaGrad metadata in the PS. In all experiments, we generated negative samples by perturbing both subject and object of positive triples 10 times. We set the initial learning rate for AdaGrad to 0.1. We used data clustering to create and exploit PAL for relation parameters, and latency hiding for entity parameters. For the relation parameters, we partitioned the training dataset by relation and allocated each relation parameter at the node that uses it, such that

our studies in Sections 4.4 and 5.4.

all accesses to relation parameters are local. Regarding entity parameters, each worker pre-localizes all parameters that it requires for the batch that follows the current one (where one batch consists of one subject—relation—object training example and 10 subject-perturbed and 10 object-perturbed negative samples). The transfer of these parameters then overlaps with the computation for the current batch. We tried looking further into the future, e.g., localizing the parameters of a batch 2, 3, 10, or 100 batches into the future. We observed similar speed-ups for 2 and 3 and lower speed-ups for 10 and 100.

Word embeddings Word embeddings are a language modeling technique in natural language processing: each word of a vocabulary is mapped to a vector of real numbers (Mikolov et al. 2013; Pennington et al. 2014; Peters et al. 2018). These vectors are useful as input for many natural language processing tasks, for example, syntactic parsing (Socher et al. 2013) or question answering (X. Liu et al. 2018). In our experimental study, we used the skip-gram Word2Vec (Mikolov et al. 2013) model and employed latency hiding to create and exploit PAL. We used the One Billion Word Benchmark (Chelba et al. 2013) dataset, with stop words of the Gensim (Řehůřek and Sojka 2010) stop word list removed. We used common model parameters (Mikolov et al. 2013) of embedding size 1 000, window size 5, minimum count 2, negative sampling with 25 samples, and 1e-5 frequent word subsampling. We used a latency hiding approach that pre-localizes parameters for the words of a sentence when it reads the sentence. As negative samples, our approach chooses only parameters that are (currently) available locally. This pool of locally available parameters changes constantly as parameters are relocated when they occur in sentences. This approach changes the local sampling distribution of negative examples at one node. However, it mostly preserves the global sampling distribution, as each parameter is local at exactly one node (except that frequent parameters are sampled underproportionately, because they relocate more often and are sampled nowhere during a transfer). To implement this scheme, Lapse exposes an additional `PullIfLocal` primitive. We will discuss such sampling access, available schemes to reduce communication overhead (such as the *local sampling* scheme used here), and a principled approach to sampling support in PSs in detail in Section 4.3.

Implementation and Cluster

We implemented Lapse in C++, using ZeroMQ and Protocol Buffers for communication, drawing from PS-Lite (Mu Li et al. 2014a). We ran version 1.1 of Petuum[7] (Xing et al. 2015) and the version of Sep 1, 2019 of PS-Lite (Mu Li et al. 2014a). We used a local cluster of 8 Dell PowerEdge R720 computers, running CentOS Linux 7.6.1810, connected with 10 GBit Ethernet. Each node was equipped with two Intel Xeon E5-2640 v2 8-core CPUs, 128 GB of main memory, and four 2 TB NL-SAS 7200 RPM hard disks. We compiled all code with gcc 4.8.5.

Settings and Measures

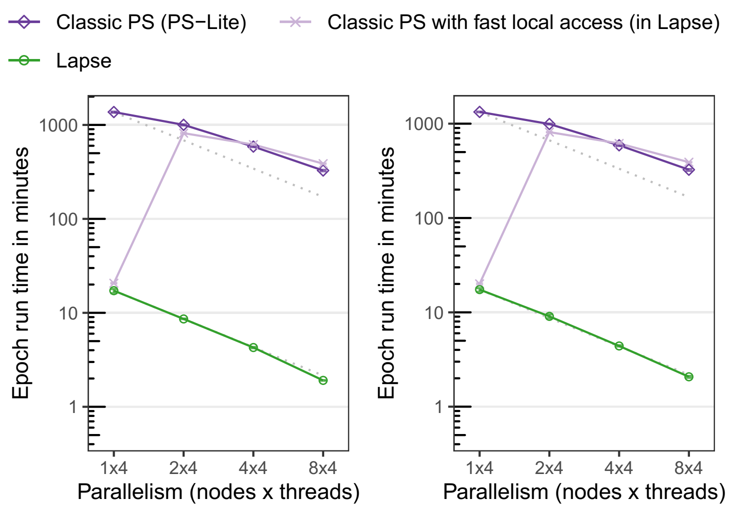
In all experiments, we used 1 server and 4 worker threads per node[8] and stored all model parameters in the PS, using dense storage. Each key held a vector of parameter values. We report Lapse run times without location caches, because they had minimal effect in Lapse. The reason for this is that Lapse localizes parameters and location caches are not beneficial for local parameters (see Section 3.3.6 for details). For all tasks but word embeddings, we measured epoch run time, because the different variants that we run are identical (or near-identical) with respect to convergence, such that the only difference among these variants is in epoch run time. This allowed us to conduct experiments in more reasonable time. For word embeddings, epochs are not identical because the chosen latency hiding approach changes the sampling distribution of negative samples. Thus, we measure model accuracy over time. We calculated model accuracy using a common analogical reasoning task of 19 544 semantic and syntactic questions (Mikolov et al. 2013). We conducted 3 independent runs of each experiment and report the mean. Error bars depict the minimum and maximum. In some experiments, error bars are not clearly visible because of small variance. Gray dotted lines indicate run times of linear scaling.

3.3.2 Performance of Classic Parameter Servers

We investigated the performance of classic PSs and how it compares to the performance of efficient single node implementations. To this end, we measured the performance of a classic PS on 1–8 nodes for matrix factorization (Figure 3.8), knowledge graph embeddings (Figure 3.9), and word embeddings (Figure 3.10). Besides PS-Lite, we ran Lapse as a classic PS (with shared

[7] In consultation with the Petuum authors, we fixed an issue in Petuum that prevented Petuum from running large models on a single node.

[8] For higher degrees of parallelism, see the experiments in Sections 4.4 and 5.4.



(a) 10m × 1m matrix, 1b entries (b) 3.4m × 3m matrix, 1b entries

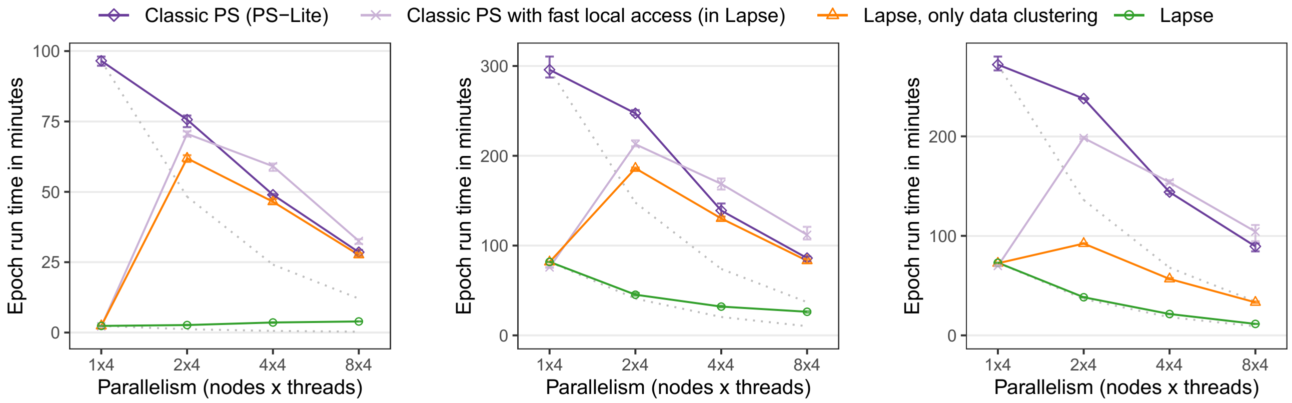
Figure 3.8: Performance for matrix factorization. Lapse scaled linearly because it exploits PAL. Classic PS approaches displayed significant communication overhead over the single node. The classic PS approach in Lapse drops in performance because it is efficient on a single node, see Section 3.3.2. The gray dotted lines indicate linear scaling. Error bars depict minimum and maximum run time (hardly visible here because of low variance).

memory access to local parameters). To do so, we disabled DPA, such that parameters are allocated statically. We used random keys for the parameters in both implementations.[9] We omitted PS-Lite from the word embeddings task due to its long run time.

Multi-Node Performance

The performance of classic PSs was dominated by communication overhead: in none of the tested ML tasks did 2–8 nodes outperform a single node. Instead, 2–8 nodes were 22–47x slower than 1 node for matrix factorization, 1.4–30x slower for knowledge graph embeddings, and 11x slower for word embeddings. The two classic PS implementations displayed similar performance on multiple nodes. With smaller numbers of nodes (e.g., on 2 nodes), the variant with fast local access can access a larger part of parameters with low latency, and thus has a performance benefit. Further performance differences stem from differences in the system implementations.

[9] The performance of classic PSs depends on the (static) assignment of parameters. Both implementations range partition parameters, which can be suboptimal if algorithms assign keys to parameters non-randomly. Manually assigning random keys improved performance for most tasks (and never deteriorated performance).



(a) **ComplEx-Small (dim.** 100/100**)** (b) **ComplEx-Large (dim.** 4 000/4 000**)** (c) **RESCAL-Large (dim.** 100/10 000**)**

Figure 3.9: Performance for training knowledge graph embeddings. Distributed training using the classic PS approach did not outperform a single node in any task. Lapse scaled well for the large tasks (b and c), but not for the small task (a).

Single-Node Performance

On a single node, the run times of PS-Lite and Lapse differed significantly (e.g., see Figure 3.8), because they access local parameters (i.e., all parameters when using 1 node) differently. Lapse accesses local parameters via shared memory. This was 71–91x faster than PS-Lite, which accesses local parameters via inter-process communication.[10] The Classic PS with fast local access displayed the same single-node performance as Lapse, as all parameters are local if only one node is used (even without relocations). This single-node efficiency is the reason for its performance drop from 1 to 2 nodes. Comparing distributed run times only against inefficient single node implementations can be misleading.

Communication Overhead

The extent of communication overhead depended on the communication-to-computation ratios of the different tasks. The two rightmost columns of Table 3.4 give an indication of this ratio. They depict the number of key accesses and size of read parameter data per second, respectively, measured for a single thread on a single node for the respective task. For example, ComplEx-Small (Figure 3.9a) accessed the PS frequently (312k accessed keys per second) and displayed high communication overhead (8 nodes were 14x

[10] PS-Lite provides an option to explicitly speed up single node performance by using memory copy inter-process communication. In our experiments, this was still 47–61x slower than shared memory.

Table 3.5: Parameter reads, relocations, and relocation times in ComplEx-Large. In this task, each key holds a vector of 8 000 doubles. All parallelism levels read 196 million keys in one epoch. On 2 nodes, mean RT is short as every relocation involves only 2 nodes (instead of 3).

Nodes	Reads (keys/s)			Relocations	Mean RT[a]
	Total	Local	Non-local	(keys/s)	(ms)
1	36 k	36 k	0.0 k	0 k	-
2	72 k	72 k	0.0 k	12 k	2.4
4	104 k	102 k	1.6 k	27 k	6.9
8	121 k	118 k	2.5 k	36 k	7.7

[a] Relocation time, see Section 3.2.2

slower than 1 node). ComplEx-Large (Figure 3.9b) accessed the PS less frequently (11k accesses keys per second) and displayed lower communication overhead (8 nodes were 1.4x slower than 1 node).

3.3.3 Effect of Dynamic Parameter Allocation

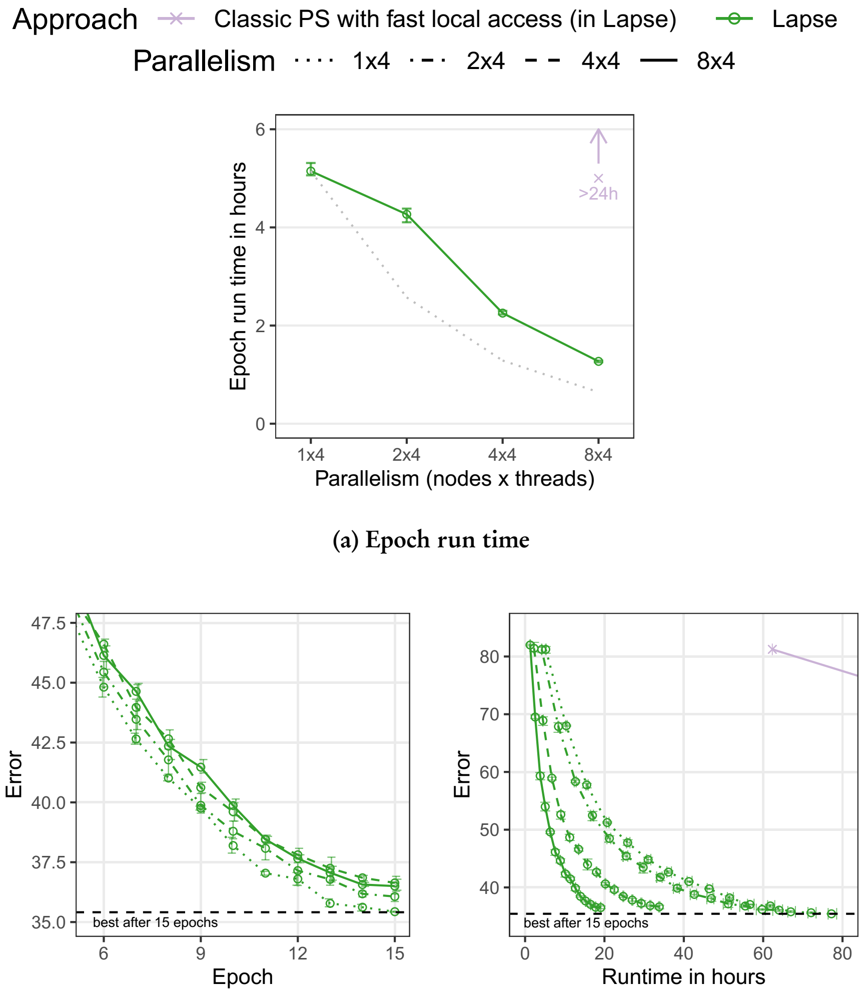
We compared the performance of Lapse to a classic PS approach for matrix factorization (Figure 3.8), KGE (Figure 3.9), and word embeddings (Figure 3.10). Lapse was 4–203x faster than classic PSs. **Lapse outperformed the single node in all but one tasks (ComplEx-Small), with speed-ups of 3.1–9x on 8 nodes (over 1 node).**

Matrix Factorization

In matrix factorization, Lapse was 90–203x faster than classic PSs and achieved linear speed-ups over the single node (see Figure 3.8). The reason for this speed-up is that classic PSs (e.g., PS-Lite) cannot exploit the PAL of the parameter blocking algorithm. Thus, their run time was dominated by network latency.

Knowledge Graph Embeddings

In knowledge graph embeddings, Lapse was 4–26x faster than a classic PS (see Figure 3.9). It scaled decently for the large tasks (ComplEx-Large and RESCAL-Large). In particular for ComplEx-Large, scalability was limited by localization conflicts on frequently accessed parameters. The probability of a localization conflict, i.e., that two or more nodes localize the same parameter at the same time, increases with the number of workers, see Table 3.5. In the table, the number of localization conflicts is indicated by the number of non-local parameter reads (which are caused by localization conflicts). For



(a) Epoch run time

(b) Error over epochs 6–15

(c) Error over run time (all epochs)

Figure 3.10: Performance for training word embeddings. The classic PS approach did not scale (8 nodes were >4x slower than 1 node). In Lapse, 8 nodes reached (for example) 39% error 3.9x faster than 1 node. The dashed horizontal line indicates the best observed error after 15 epochs.

ComplEx-Small, distributed execution in Lapse did not outperform the single node because of communication overhead.

We additionally measured performance of running Lapse with only data clustering (i.e., without latency hiding). This approach accesses relation parameters locally and entity parameters remotely. It improved performance for RESCAL (Figure 3.9c) more than for ComplEx (Figures 3.9a and 3.9b), because in RESCAL, relation embeddings have higher dimension (10 000 for RESCAL-Large) than entity embeddings (100), whereas in ComplEx, both are the same size (100 in ComplEx-Small and 4 000 in ComplEx-Large).

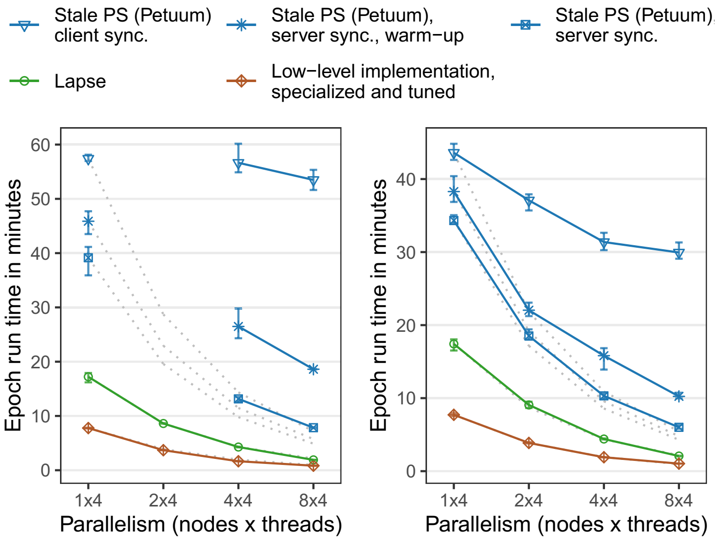
Word Embeddings

For word embeddings, Lapse executed an epoch 44x faster than a classic PS (Figure 3.10). Further, 8 nodes reached, for example, 39% error 3.9x faster than a single node. The speed-up for word embeddings is lower than for knowledge graph embeddings, because word embeddings training exhibits strongly skewed access to parameters: few parameters are accessed frequently (Mikolov et al. 2013). This stronger skew lead to more frequent localization conflicts in the latency hiding approach than in knowledge graph embeddings, where at least negative samples are sampled uniformly (Ruffinelli et al. 2020; H. Liu et al. 2017). We will investigate how PSs can be efficient for such strongly skewed access patterns in the subsequent chapter of this thesis (see Section 4.2).

3.3.4 Comparison to Manual Management

We compared the performance of Lapse to a highly specialized and tuned low-level implementation of the parameter blocking approach for matrix factorization, see Figure 3.11. This low-level implementation cannot be used for other ML tasks; i.e., it is a competitive baseline specifically for this one ML task. **Both the low-level implementation and Lapse scaled linearly.**

Lapse had only 2.0–2.6x generalization overhead over the low-level implementation. The reason for the overhead is that the low-level implementation exploits task-specific properties that a PS cannot exploit in general if it aims to provide PS consistency and isolation guarantees for a wide range of ML tasks. I.e., the task-specific implementation lets workers work directly on the data store, without copying data and without concurrency control. This works for this particular algorithm, because each worker focuses on a separate part of the model (at a time), but is not applicable in general. In contrast, Lapse and other PSs copy parameter data out of and back into the server, causing overhead over the task-specific implementation. Additionally, the low-level



(a) 10m × 1m matrix, 1b entries (b) 3.4m × 3m matrix, 1b entries

Figure 3.11: Performance comparison to manual parameter management and to Petuum, a state-of-the-art replication PS, for matrix factorization. Lapse and manual parameter management (using a specialized and tuned low-level implementation) scale linearly, in contrast to Petuum. For the 10m × 1m matrix, Petuum crashed with a network error on 2 nodes.

implementation focuses on and optimizes for communication of blocks of parameters (which Lapse does not) and does not use a key–value abstraction for accessing keys.

Implementing the parameter blocking approach was significantly easier in Lapse than using low-level programming. The low-level implementation manually moves parameters from node to node, using MPI communication primitives. This manual parameter allocation required 100s of lines of MPI code. In contrast, in Lapse, the same parameter allocation required only 4 lines of additional code.

3.3.5 Comparison to Replication PSs

We compared Lapse to Petuum, a popular replication PS, for matrix factorization (see Figure 3.11). **We found that the replication PS was 2–28x slower than Lapse and did not scale linearly, in contrast to Lapse.**

Petuum provides bounded staleness consistency. As discussed in Section 3.1.1, this can support synchronous parameter blocking algorithms, such as the one we test for matrix factorization. We compared separately to the two synchronization protocols that Petuum provides: SSP (Ho et al. 2013) and ESSP (Dai et al. 2015). On the 10m × 1m dataset, Petuum crashed on two nodes with a network error.

SSP

Petuum with the SSP synchronization protocol outperformed the classic PS, but was 2.5–28x slower than Lapse. The main reason for the overhead was network latency for synchronizing parameters when their value became too stale. This approach did not scale because the number of synchronizations per worker was constant when increasing the number of workers (due to the increasing number of subepochs).

ESSP

Petuum with the ESSP synchronization protocol outperformed the classic PS, but was 2–4x slower than Lapse, and only 2.9x faster on 8 nodes than Lapse on 1 node. The reason for this is that after every global clock advance, Petuum's ESSP synchronization eagerly synchronizes to a node all parameters that this node accessed previously. On the one hand, this eliminated the network latency overhead of ESSP synchronization. On the other hand, this caused significant unnecessary communication, causing the overhead over Lapse and preventing linear scale-out: in each subepoch, each node accesses only a subset of all parameter blocks, but Petuum replicates all blocks. Petuum "learns" which parameters to replicate to which node in a slower *warm-up* epoch, depicted separately in Figure 3.11.

3.3.6 Ablation Study

DPA and Fast Local Access

Lapse differs from classic PSs in two ways: (1) DPA and (2) shared memory access to local parameters. To investigate the effect of each difference separately, we compared the run time of three different variants: *Classic PS (PS-Lite)* (neither DPA nor shared memory), *Classic PS with fast local access (in Lapse)* (no DPA, but shared memory), and *Lapse* (DPA and shared memory). The run times of these variants can be compared in Figures 3.8 and 3.9. Without DPA, shared memory had limited effect, as many parameters were non-local and access times were thus dominated by network latency (except for the single-node case, in which all parameters are local). Combining DPA and shared memory yielded better performance: DPA ensures that parameters are local and shared memory ensures that access to local parameters is fast.

Location Caching

All figures report run times of Lapse *without* location caching. We investigated the effect of location caching, which Lapse supports optionally. We observed similar run times *with* location caching. For example, for KGE (Figure 3.9), Lapse was max. 3% faster and max. 2% slower with location caching than without. The reason for this is that location caches speed up only remote parameter accesses (see Section 3.2.3). The latency hiding approach in KGE, however, localizes all parameters before they are used, such that the vast majority of parameter accesses are local (see Table 3.5). For matrix factorization, location caching had no effect at all, because all parameter accesses were local (due to the parameter blocking approach). In the Classic PS variant of Lapse, location caches had no effect because parameters remained at their home nodes throughout training (as they do in other classic PSs).

3.4 Related Work

We discuss related work on distributed parameter management in Section 2.2 and other related work in Section 2.3. In the following, we further discuss work that is specifically related to dynamic parameter allocation.

3.4.1 Dynamic Parallelism

FlexPS (Yuzhen Huang et al. 2018) reduces communication overhead by executing different phases of an ML task with different levels of parallelism and moving parameters to active nodes. However, it provides no location control, moves parameters only between phases, and pauses the training process for the move. This reduces communication overhead for some ML tasks, but is not applicable to many others, e.g., the tasks that we consider in this thesis. FlexPS cannot be used for PAL techniques because it does not provide fine-grained control over the location of parameters. Lapse, in contrast, is more general: it supports the FlexPS setting, but also provides fine-grained location control and moves parameters without pausing workers.

3.4.2 Dynamic Allocation in Key–Value Stores

DAL (Németh et al. 2017), a general-purpose key–value store, dynamically allocates a data item at the node that accesses it most (after an adaptation period). In theory, DAL could exploit data clustering and synchronous parameter blocking PAL techniques, but no others (due to the adaptation period). However, DAL accesses data items via inter-process communication, such that access latency is too high to exploit PAL in ML algorithms. Husky (F. Yang et al. 2016) allows applications to move data items among nodes and provides fast access to local data items. However, local data items can be accessed by only one worker (i.e., Husky does not provide global read and write access to the data items). Thus, Husky can exploit only PAL techniques in which each parameter is accessed by only one worker, i.e., parameter blocking, but not data clustering or latency hiding.

3.5 Summary

In this chapter, we explored whether and to what extent PSs can exploit locality, and whether doing so can be beneficial. To this end, we proposed that the PS adaptively relocates parameters during run time, according to where the parameters are currently accessed. This allows PSs to support PAL techniques directly. We found that dynamic allocation can reduce communication overhead of PSs significantly for some ML tasks.

In particular, dynamic allocation allows efficient distributed training for algorithms that explicitly create locality, i.e., for the data clustering or parameter blocking PAL techniques. However, for some tasks, no such algorithms exist, either because they have not been developed yet or because they are infeasible for the task. For such tasks, only the latency hiding PAL technique is available. However, as seen in Section 3.3, the efficiency of the latency hiding technique can be limited by localization conflicts, especially when parameter access frequency is strongly skewed towards a few frequently accessed parameters. In the experimental study of this chapter, we saw initial evidence of this limitation, even though we were using relatively small degrees of parallelism (only 4 worker threads per node). In the upcoming Chapter 4, we will observe stronger impacts of this limitation (by evaluating PS efficiency in harder-to-scale conditions, e.g., with more skew and higher levels of parallelism), will explore support for the latency hiding technique further, and try to improve efficiency for real-world ML tasks.

We showed that Lapse can be more efficient than a classic PS. However, Lapse is also more complex to use than a classic PS. In particular, applications need to manually trigger relocations by adding `localize` invocations to application code (see Algorithm 3.1 on page 36 for an example). They need to decide *which* parameters to relocate and *when* to do so (i.e., how large the *relocation offset* between the actual parameter access and the initiation of the relocation should be). For the data clustering and parameter blocking PAL techniques, both decisions are usually straightforward. In contrast, for the latency hiding technique, the timing is often not straightforward, as it directly affects the number of localization conflicts (and, thus, performance). Relocation should be early enough to finish before the actual access, but not much earlier than necessary because that would increase the chance of localization conflicts. For optimal performance, applications might need to tune these decisions, making Lapse complex to use. In Chapter 5, we discuss how adaptive PSs can be made easier to use.

Chapter 4

Handling Diversity: Non-Uniform Parameter Management

In this chapter, we further investigate the efficiency of PSs for tasks without explicit locality (e.g., through data clustering or parameter blocking). The latency hiding technique is applicable to such tasks, but its efficiency can be limited by localization conflicts (see Section 3.3). We observe that a key cause for limited performance in real-world ML tasks is *non-uniform parameter access*. We identify two main sources of non-uniformity: *skew* and *sampling*.

First, in a workload that exhibits skew, a (typically small) subset of parameters is accessed frequently (e.g., up to 100 000 times per second), whereas a large part of the parameters is accessed rarely (e.g., only once every minute) (Meka et al. 2009; W. Cheng et al. 2016; Gonzalez et al. 2012; Faloutsos et al. 1999; Moreno-Sánchez et al. 2016; Clauset et al. 2009). The main reason for skew is that real-world datasets often have skewed frequency distributions (e.g., graphs (W. Cheng et al. 2016; Gonzalez et al. 2012; Faloutsos et al. 1999), texts (Moreno-Sánchez et al. 2016), and others (Clauset et al. 2009; Meka et al. 2009)), and many ML models associate specific parameters with specific data items (e.g., with the tokens in a text document or with the vertices of a graph (Mikolov et al. 2013; Nickel et al. 2011; Koren et al. 2009)).

Existing PSs are inefficient for managing skew because they employ one single management technique for all parameters. Using a single technique limits performance as none of the existing techniques is efficient for all access patterns. To overcome this limitation, we introduce *multi-technique parameter management*, i.e., to adapt the parameter management technique to the access pattern of a parameter. The PS provides multiple management techniques and chooses a suitable technique *for each parameter*. Our prototype implementation NuPS integrates parameter relocation (as presented in Chapter 3) and parameter replication (Ho et al. 2013; Dai et al. 2015).

The second source of non-uniformity is *sampling*: for a subset of parameter accesses, random sampling (rather than training data) determines which parameters are read and written (Mikolov et al. 2013; Ruffinelli et al. 2020; H. Liu et al. 2017; Rendle et al. 2009; Bamler and Mandt 2020; Chechik et al. 2010; Schroff et al. 2015). One common reason for this access pattern is negative sampling (Mikolov et al. 2013; Bamler and Mandt 2020; Ruffinelli et al. 2020; Grover and Leskovec 2016), which, for example, is used to reduce the cost of many-class classification tasks or to mitigate an absence of negative training data (e.g., in recommender systems with only positive feedback or in knowledge graphs that contain only positive edges). For example, when training knowledge graph embeddings, it is common to randomly perturb either the subject, the relation, or the object of subject–relation–object training examples to obtain negative training examples (Ruffinelli et al. 2020). For instance, from the training example *Marie Curie—is a—scientist*, you could obtain the negative training example *Marie Curie—is a—butterfly* by randomly perturbing the object of the training example (*scientist*).

Existing PSs are inefficient for sampling because common parameter management techniques are ill-suited for randomly sampled access. To improve performance, applications can implement specialized *sampling schemes* manually, outside the PS (Lerer et al. 2019; D. Zheng et al. 2020b; Ji et al. 2019; Stergiou et al. 2017), but this limits the efficiency of some schemes, potentially produces incorrect samples, and causes repeated implementation effort. We propose to overcome this limitation by integrating sampling schemes directly into the PS. To do so, we extend the PS API with a sampling primitive that allows applications to request samples from a specific sampling distribution (rather than accessing specific parameters directly). A *sampling manager* transparently chooses one of several sampling schemes to reduce communication overhead for sampling, according to a *conformity level*. Conformity levels provide a controlled trade-off between efficiency and sample quality.

NuPS, our implementation of a non-uniform PS, implements both multi-technique parameter management and sampling support, see Figure 4.1. In our experimental evaluation, NuPS outperformed state-of-the-art PSs by up to one order of magnitude and provided good scalability across multiple tasks.

We begin this chapter by introducing skew and sampling in detail in Section 4.1. Section 4.2 explores how PSs can efficiently handle skew. Section 4.3 explores how PSs can efficiently handle sampling. Finally, Section 4.4 experimentally investigates how non-uniform parameter management affects PS performance. Section 4.5 concludes the chapter with an interim summary.

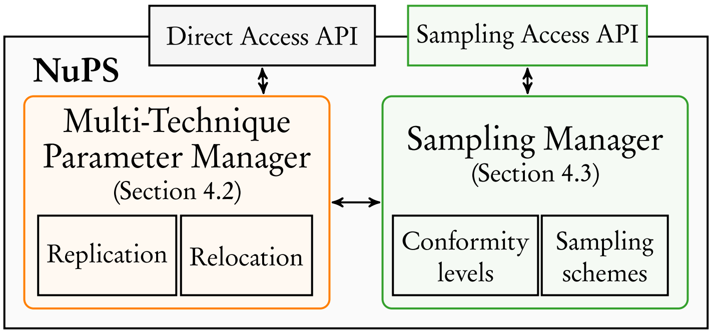


Figure 4.1: NuPS components. NuPS differs from existing PSs in two main ways: it introduces (i) multi-technique parameter management to handle skew and (ii) a sampling manager and API to handle sampling.

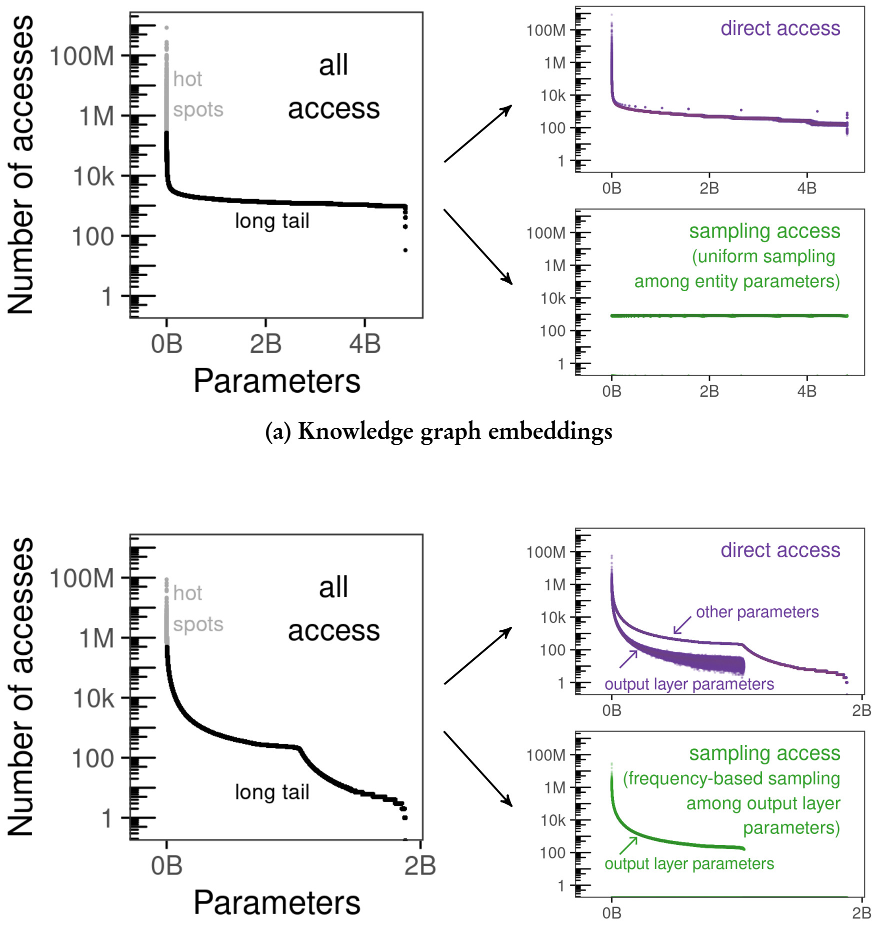
4.1 Non-Uniform Parameter Access

We study ML tasks that exhibit non-uniform parameter access. We identify two main sources of non-uniform parameter access: skew (Section 4.1.1) and sampling (Section 4.1.2).

4.1.1 Skew

A workload exhibits skew non-uniformity when some parts of the model are accessed (much) more frequently than others. The main reason for this is that many real-world datasets have skewed frequency distributions (Meka et al. 2009; W. Cheng et al. 2016; Mohamed et al. 2020; Gonzalez et al. 2012; Faloutsos et al. 1999; Moreno-Sánchez et al. 2016; Clauset et al. 2009). For example, heavy skew is common in text corpora, because word frequencies are skewed (Moreno-Sánchez et al. 2016), and in graph data, because in- and out-degree distributions are skewed (W. Cheng et al. 2016; Mohamed et al. 2020; Gonzalez et al. 2012; Faloutsos et al. 1999). As many ML models associate specific parameters with specific data items (e.g, with words in a text or with the nodes of a graph) (Mikolov et al. 2013; Nickel et al. 2011; Koren et al. 2009; Grover and Leskovec 2016), access to the parameters is heavily skewed, too: a small subset of *hot spot parameters* is accessed frequently, whereas the majority of parameters is accessed rarely. In the following, we will refer to the parameters that are not hot spots as *long tail parameters*.

We have measured the extent of skew for two real-world ML tasks: training knowledge graph embeddings and training word embeddings. The left hand sides of Figures 4.2a and 4.2b show the number of reads per parameter over one epoch of these tasks, respectively. Access is heavily skewed: in the knowledge graph embeddings task, 18% of 12.9 trillion total reads go to only



(a) Knowledge graph embeddings

(b) Word embeddings

Figure 4.2: Number of accesses per parameter in one epoch. Parameters are sorted by decreasing total number of accesses. See Section 4.4.1 for details on tasks and experimental setup.

0.02% of 4.8 billion parameters. In the word embeddings task, 45% of 9 trillion total reads go to 0.17% of 1.9 billion parameters. Details on the tasks and datasets can be found in Section 4.4.1.

Note that skew is not always present in distributed training. For example, there is no skew in convolutional neural networks for image recognition (LeCun et al. 1989) because model access is dense, i.e., every update step writes to all parameters.[1,2] In contrast, in common neural network models for natural language processing (Peters et al. 2018; Devlin et al. 2019; Howard and Ruder 2018), access is partially dense, and partially sparse and skewed: access to the first (embedding) layer and sometimes the last (classification) layer is based

[1] Thus, with dense access, essentially, all parameters are hot spots.

[2] However, sparsity and skew can be introduced to the training of such dense models by specific training methods, e.g., by update sparsification (see Section 2.3.1).

on word or token frequency (and thus sparse and skewed), and access to other layers is dense. The share of parameters with frequency-based access depends on the model architecture, but can be high, e.g., around 90% in ELMo (Peters et al. 2018). In this chapter, we investigate skew in shallow models, but conjecture that a non-uniform PS can also be beneficial for deeper models with partially skewed access.

4.1.2 Sampling

A workload exhibits sampling non-uniformity when, for a subset of parameter accesses, random sampling determines which parameters are read and written (Mikolov et al. 2013; Ruffinelli et al. 2020; H. Liu et al. 2017; Rendle et al. 2009; Rawat et al. 2021; Bamler and Mandt 2020; Chechik et al. 2010; Schroff et al. 2015). I.e., the application randomly draws a parameter key from an application-specific sampling distribution over (all or a subset of) parameter keys. It then accesses the drawn parameter for training. We refer to such access as *sampling access*. In contrast, in *direct access*, the training data determines which parameters are accessed. Sampling access is common in many-class classification tasks, e.g., extreme classification (Bamler and Mandt 2020), natural language processing (Mikolov et al. 2013), knowledge graph embeddings (Ruffinelli et al. 2020; H. Liu et al. 2017), graph representations (Grover and Leskovec 2016; Z. Yang et al. 2020), recommender systems (Rendle et al. 2009), and when triplet loss is used (Chechik et al. 2010; Schroff et al. 2015).

For example, knowledge graph embeddings and word embeddings training tasks often use *negative sampling* to enable efficient training (Mikolov et al. 2013; Ruffinelli et al. 2020; Rawat et al. 2021). For each (positive) data point, a set of *negative samples* is drawn from a distribution. Each negative sample corresponds to a data item (e.g., a word) or a class. The corresponding parameters are subsequently accessed for training. For instance, the example knowledge graph embeddings task draws negative samples from a uniform distribution over all entities (Ruffinelli et al. 2020; H. Liu et al. 2017). The right hand side of Figure 4.2a shows the frequency distributions of direct and sampling accesses separately for this task. In our implementation (based on (H. Liu et al. 2017), see Section 4.4.1) and with 200 negative samples for each subject–relation–object triple (100 negative samples for the subject and another 100 for the object), sampling accesses make up 31% of all accesses. In the word embeddings task, negative samples correspond to words and the sampling distribution resembles the word frequencies in the training data (Mikolov et al. 2013), see Figure 4.2b. In the plot for direct access, parameters that belong to the output layer of the task's neural network are visually distinct from the other parameters. The reason for this is that the task draws samples only from the output layer, and parameters in the plot are sorted by *total* access frequency. In our implementation (based on (Mikolov et al. 2013), see Section 4.4.1) and with 3 negative samples for each word–word pair, sampling accesses make up 56% of all parameter accesses in this task.

4.2 Multi-Technique Parameter Management

In this section, we analyze the suitability of existing PSs for ML tasks with skewed parameter access (Section 4.2.1) and argue that existing PSs are inefficient for managing skew because they employ one single management technique for all parameters. Based on this analysis, we propose multi-technique parameter management and discuss NuPS's implementation (Section 4.2.2).

4.2.1 Analysis of Common Parameter Management Techniques

As discussed in Section 2.2, several different techniques are used in PSs to manage parameters. In the following, we analyze the suitability of existing techniques *for managing skew*. We briefly recap each technique before starting our analysis.

Classic Parameter Server

A classic PS allocates parameters to servers statically (e.g., via range partitioning of the parameter keys) and uses no replication (Smola and Narayanamurthy 2010; Ahmed et al. 2012; Mu Li et al. 2014a). Thus precisely one server holds the current value of a parameter, and this server is used for all pull and push operations on this parameter.

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

Replication Parameter Server

A replication PS replicates parameters and tolerates some amount of staleness in the replicas (Ho et al. 2013; Yuzhen Huang et al. 2018; J. Jiang et al. 2017; Dai et al. 2015; Cui et al. 2014). The SSP protocol creates a replica when a parameter is accessed and uses this replica until the staleness bound is reached (at which point the replica is terminated). The ESSP protocol also creates a replica when a parameter is (first) accessed, but then maintains this replica throughout the entire training task (by repeatedly propagating updates to the nodes that hold replicas).

Analysis: A replication PS is efficient for hot spots, but its benefit for the long tail is limited. Replication reduces network overhead (compared to a classic PS) if a replicated parameter value is used more than once and multiple updates can be sent to the PS in aggregated form. Replication further reduces access latency if a parameter value (within the acceptable staleness bound) is already locally available when a read operation is issued. Both is typically the case for hot spot parameters, even within relatively tight staleness bounds (because hot spot parameters are accessed frequently at each node). In contrast, long tail parameters are accessed infrequently. So it is unlikely that a long tail parameter is accessed more than once within reasonable staleness bounds (large staleness bounds commonly deteriorate model convergence (Ho et al. 2013)). For the same reason, SSP (which creates replicas on demand) does not reduce access latency for long tail parameters, because replicas are mostly “cold”. With its eager replica maintenance, ESSP ensures that replicas are always “warm” (after the first access to a parameter), but at the cost of significant over-communication: ESSP constantly updates all replicas, although replicas for long tail parameters are accessed rarely.

Relocation Parameter Server

A relocation PS such as Lapse (see Section 3.2) asynchronously re-allocates parameters among nodes during run time so that access operations can be processed locally, without network communication.

Analysis: A relocation PS is efficient for long tail parameters, but has limited benefit for hot spots. Relocation eliminates access latency if there is sufficient time to relocate a parameter between accesses at different nodes.[3] It further reduces network overhead (compared to classic) if a parameter is accessed more than once between two relocations (which is common,

[3] We assume the general-purpose latency hiding technique here, i.e., that there is no explicit locality through data clustering or parameter blocking. If there is explicit locality, parameter relocation is highly efficient, as discussed in Section 3.1.2.

most ML tasks at least read and write a parameter): a relocation takes three messages in Lapse (including the parameter value once, see Section 3.2.2), whereas each remote access in a classic PS sends two messages (including the parameter value once). There is typically sufficient time for relocating long tail parameters between accesses by different nodes, as they are accessed infrequently. Hot spot parameters, however, are frequently accessed at multiple nodes concurrently. Thus, there is not sufficient time for relocations between accesses, such that access latency is not eliminated. Further, a relocation PS incurs higher network overhead than a classic PS if a parameter is relocated so frequently that only one operation is processed locally.

Summary

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

4.2.2 Parameter Management in NuPS

From the above discussion, it follows naturally to explore whether combining multiple management techniques is beneficial for PS performance. The idea of combining multiple management techniques has been studied in other distributed data management systems, such as general-purpose distributed databases (Dowdy and Foster 1982; Wolfson and Jajodia 1992; El Abbadi 1991; Ciciani et al. 1990), distributed graph processing systems (Low et al. 2012), and PSs (S. Kim et al. 2019; Q. Zheng et al. 2021). However, existing systems combine static allocation with replication, and do not consider relocation.

NuPS combines replication and relocation. First, to manage hot spot parameters efficiently, NuPS integrates a lightweight variant of eager replication (Dai et al. 2015). NuPS eagerly creates replicas for hot spot keys on all nodes and provides time-based staleness bounds. Basing the staleness bound on time rather than clocks alleviates the need for adding “advance the clock” operations to application code, but potentially complicates the analysis of convergence properties. We discuss these implications below. Second, to manage long tail parameters efficiently, NuPS integrates relocation. As Lapse, NuPS asynchronously relocates these parameters before they are accessed, thus guaranteeing per-key sequential consistency for long tail parameters. The support for multiple management techniques in NuPS enables applications to pick a technique for each key based on the key’s access pattern: if the key is accessed frequently, NuPS can replicate the key; if there are few accesses, NuPS can employ relocation. During training, the choice of management technique is transparent to the application, i.e., the application accesses all parameters in the same way, via the `push` and `pull` primitives. Our experimental evaluation shows that the combination of replication and relocation can be highly beneficial. Integrating other techniques (e.g., highly tailored ones) may further improve performance, but is beyond the scope of this thesis. NuPS does not integrate the classic technique as it is dominated by replication for hot spots and by relocation for the long tail.

Algorithm 4.1 gives an example for how the multi-technique parameter management of NuPS can be used in the distributed SGD example that we introduced in Section 2.2. The differences to a Classic PS implementation are: (i) the application (i.e., the component that interacts with the PS) picks a management technique for each parameter key (lines 1–2), and (ii) the application initiates parameter relocation when a batch is prepared (line 8), as done for Lapse (see Algorithm 3.1 on page 36). NuPS ignores requests to relocate parameters that are managed by replication to ensure that the management technique is transparent to the application.

Algorithm 4.1: Distributed asynchronous SGD with NuPS. Differences to the Classic PS implementation (Algorithm 2.3 on page 17) are highlighted in green.

```
Data: $\mathcal{D}$: training dataset,
      $num\_epochs$: number of epochs to run,
      $batch\_size$: batch size,
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads ,
      $\mathcal{T}$: list of desired management techniques (one for each key)
1  foreach $k \in$ all_keys() do
2      set_technique ( $k$, $\mathcal{T}_k$ )
3  for epoch ← 1 to num_epochs do
4      b = num_batches ( $\mathcal{D}$, batch_size, t, T )
       // data loading (pipeline parallel with training, in separate thread(s))
5      $\mathcal{B}$ = []
6      for i ← 1 to b do
7          $\mathcal{B}_i$ = prepare_batch ( i, $\mathcal{D}$, batch_size, epoch, t, T )
8          localize ( keys($\mathcal{B}_i$) )
       // training
9      for i ← 1 to b do
10         $w$ = pull ( keys($\mathcal{B}_i$) )
11         $\Delta w$ = compute_update ( $\mathcal{B}_i$, $w$ )
12         push ( keys($\mathcal{B}_i$), $\Delta w$ )
```

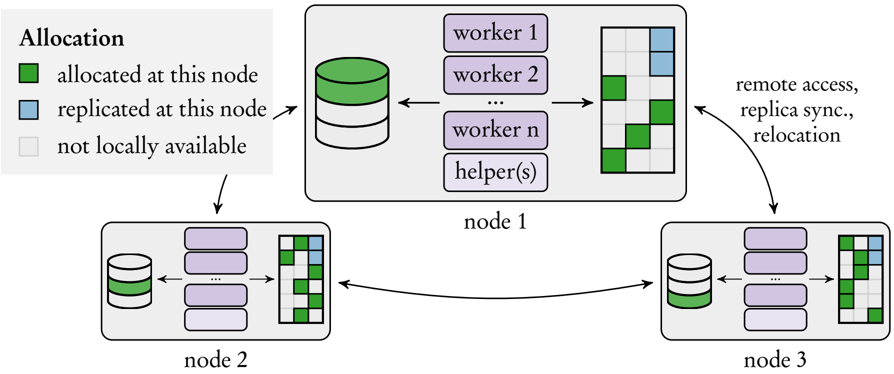


Figure 4.3: Parameter management in NuPS. NuPS replicates hot spots and relocates long tail parameters. It accesses replicated and current local parameters via shared memory.

For efficiency, NuPS co-locates workers and servers in one process per node (as Lapse does), and accesses replicas and locally allocated parameters via shared memory. Figure 4.3 depicts an overview. To access a key, a worker checks whether this key is managed by replication or relocation. If the key is managed by replication, the worker accesses the key via shared memory, without network communication. If the key is managed by relocation, the worker checks whether the key is currently allocated locally. If so, it accesses the key via shared memory. Otherwise, the worker accesses the parameter remotely, using the message protocol used by Lapse: a request to the node that knows where the parameter is currently allocated, which then forwards the request to this node, which in turn processes the request and sends a response to the worker (as described in Section 3.2.3).

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

NuPS bases its staleness bounds on time rather than logical clocks because this makes the PS easier to use: time-based bounds alleviate the need for adding "advance the clock" operations to application code and for timing them appropriately. NuPS synchronizes the replicas periodically, using sparse all-reduce operations (i.e., only updated parameters are exchanged (Träff 2010)). The synchronization is run by a background thread and uses recursive doubling all-reduce (Thakur and Gropp 2003). However, time-based staleness bounds potentially complicate the analysis of convergence properties. If only a bounded number of SGD steps can occur within one synchronization round, bounded staleness holds (as for clock-based staleness bounds) and the corresponding analysis carries over (Ho et al. 2013). However, if the number of SGD steps within one synchronization round cannot be bounded, convergence analyses for asynchronous SGD apply (Lian et al. 2015; X. Zhang et al. 2018). In our experiments, the effect of time-based bounds on performance was minimal because we used replication only for a small number of parameters and synchronized replicas frequently (see Section 4.4.6 and Section 4.4.7).

In NuPS, the decision between replication and relocation is *static*. I.e., the application chooses one management technique for each parameter at the beginning of training and then uses this technique throughout training. This static decision is a first step towards exploring and evaluating multi-technique parameter management. One limitation of a static decision is that NuPS cannot support approaches that require switching techniques dynamically during run time. Based on our findings with NuPS in this chapter, we will take a further step and explore dynamic technique switching in the subsequent chapter (Chapter 5).

4.3 Sampling Management

Existing PSs provide no support for sampling. This means that applications manually sample keys and then access the corresponding parameters via direct access, which leads to significant communication overhead. To reduce this overhead, many applications implement a variety of sampling schemes (Lerer et al. 2019; D. Zheng et al. 2020b; Ji et al. 2019; Stergiou et al. 2017). The key idea of such sampling schemes is that slightly (or sometimes rather significantly) deviating from the ideal of independent sampling from the desired target distribution might have only little or no effect on model quality, but can reduce communication overhead substantially (and consequently speed up model training). The lack of sampling support in PSs forces applications to implement such schemes in application code, outside the PS. This leads to repeated implementation effort and potentially produces incorrect samples. Further, this precludes sampling schemes that require tight integration with parameter management.

In contrast to existing PSs, NuPS integrates sampling directly into the PS. In the following, we present the components of this integration. We first introduce a set of conformity levels that allow for a controlled trade-off between efficiency and sample quality (Section 4.3.1). We then analyze conformity and communication overhead of sampling schemes that are commonly used by applications (Section 4.3.2). Based on this analysis, we propose an API extension that enables sampling in PSs (Section 4.3.3) and discuss how NuPS implements several sampling schemes, within this API (Section 4.3.4).

4.3.1 Sampling Conformity Levels

Let π be a *target distribution* over parameter keys. We assume that π is specified by the application and remains fixed throughout run time.[4] For example, in the word embeddings training task of Section 4.1.2, the target distribution π roughly corresponds to relative word frequencies (Mikolov et al. 2013); cf. Figure 4.2b. When training knowledge graph embeddings, π is often a uniform distribution over all entities (Ruffinelli et al. 2020); cf. Figure 4.2a. Denote by $\mathscr{K}$ the set of parameter keys and by $\pi_k \geq 0$ the target probability for key $k \in \mathscr{K}$, where $\sum_{k=1}^{|K|} \pi_k = 1$. Workers repeatedly draw one or more samples from the target distribution π. Denote by $X_{qi} \in \mathscr{K}$ a random variable for the i-th sample obtained at node q.[5] We write N_q for the number of samples drawn at node q during the complete run time of some application. Set $\mathscr{X}_q = \{X_{q1}, \ldots, X_{qN_q}\}$ and $\mathscr{X} = \bigcup_q \mathscr{X}_q$.

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

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

$$p(X_{qi} = k | \mathscr{S}) = \pi_k$$

for all q, i, k and $\mathscr{S} \subseteq \mathscr{X} \setminus \left\{X_{qi}\right\}$.

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

$$p(X_{qi} = k | \mathscr{S}_q^{-B}, \mathscr{S}_{-q}) = \pi_k$$

for all q, i, k, where $\mathscr{S}_{-q} \subseteq \mathscr{X} \setminus \mathscr{X}_q$ refers to samples at other nodes and

[4] This is mainly to facilitate analysis; an application may use multiple different sampling distributions, each of which can be analyzed separately.

[5] Depending on the implementation, there can be multiple workers on each node. We analyze sampling schemes at the node level to simplify exposition.

$\mathcal{S}_q^{-B} \subseteq \{X_{q1}, \ldots, X_{q(i-B-1)}\}$ refers to the samples at node q except the last B samples. Note that first-order inclusion probabilities match the target probabilities—i.e., $p(X_{qi} = k) = \pi_k$—even though subsequent samples may be dependent. For example, a sampling scheme that internally draws independent samples from π but uses each sample twice is `BOUNDED` with $B = 1$.

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

$$\lim_{N_q \to \infty} \frac{1}{N_q} \sum_{i=1}^{N_q} p(X_{qi} = k | X_{q1}, \ldots, X_{q(i-1)}) = \pi_k \tag{4.1}$$

for all q, k. Note that this does *not* imply $p(X_{qi} = k) = \pi_k$. Also, arbitrary dependencies between samples within one or across multiple nodes are accepted as long as the asymptotic relative frequencies of the samples match the target. For example, a sequential sampling scheme that selects a random key order for the $|K|$ keys and then draws samples in a round-robin fashion satisfies `LONG-TERM` but not `BOUNDED`: each key is selected equally often in the long run, but the knowledge of the first $|K|$ samples allows to uniquely determine all future samples, so that no dependency bound can be established.

(L4) `NON-CONFORM`**.** No guarantees about the sampling probabilities or independence.

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

Proof (L2 implies L3). Starting from some offset $1 \leq o \leq B$, fix some node q and consider the subset $\left\{ X_{q(aB+o)} \right\}_{a \in \mathbb{N}}$ of every B-th sample on node q, starting from the o-th sample. Using the definition of `BOUNDED`, we obtain

$$\frac{1}{\lfloor (N_q - o)/B \rfloor} \sum_{a=1}^{\lfloor (N_q - o)/B \rfloor} p(X_{q(aB+o)} = k | X_{q1}, \ldots, X_{q(aB+o-B)}) = \pi_k$$

for any choice of N_q, i.e., the long-term relative frequencies of every B-th sample match if we start at offset o. Since this holds for every offset o, we conclude that Eq. (4.1) holds and L2 implies L3.

Note that we defined L3 via Eq. (4.1) rather than a simpler first-order probability condition such as $p(X_{qi} = k) = \pi_k$, because correct first-order

Table 4.1: Conformity levels of common sampling schemes.

	L1 CONFORM	L2 BOUNDED	L3 LONG-TERM
Independent sampling	✓	✓	✓
Sample reuse	×	✓	✓
Local sampling	×	×	×
Direct-access repurposing	×	×	×

conditions are not sufficient to ensure that a sampling scheme is useful in practice. For example, a sampling scheme that internally draws one independent sample X from π, and then uses solely this sample throughout (i.e., $X_{qi} = X$ for all q, i) satisfies such a condition, but is clearly unsuitable in practice.

4.3.2 Analysis of Common Sampling Schemes

ML applications employ a variety of sampling schemes. In the following, we analyze schemes that are common in distributed training (Lerer et al. 2019; D. Zheng et al. 2020b; Ji et al. 2019; Stergiou et al. 2017; Kochsiek and Gemulla 2021) with respect to their effect on (i) communication overhead and (ii) sampling quality, i.e., into which conformity level they fall. Table 4.1 provides an overview of the latter. In this section, we focus on theoretical analyses, an empirical evaluation follows in Section 4.4.5.

Independent Sampling

Ideally, applications draw i.i.d. samples from the target distribution and use each sample once. This scheme is `CONFORM`, but can lead to significant communication overhead: for each sample, the corresponding parameter values need to be transferred to the node and, after an update is computed, updates need to be propagated to other nodes.

Sample Reuse

Sample reuse reduces communication overhead by using each sample multiple times (Ji et al. 2019; D. Zheng et al. 2020b; Lerer et al. 2019; Broscheit et al. 2020). For example, knowledge graph embeddings training can use shared sampling, i.e., reuse negative samples for all positive examples in a minibatch (Broscheit et al. 2020). Reusing a sample multiple times avoids the transfer of parameter values for another, fresh sample: using a sample U times can reduce the communication overhead by a factor of U. We refer to this factor as the *use frequency* and to a sample reuse scheme that uses each sample u times as *U=u sample reuse*. Sample reuse does not provide `CONFORM` since samples are not independent. However, it can provide BOUNDED. For example, if each fresh sample is sampled i.i.d. from π and then used exactly U times, then the scheme is BOUNDED for all $B \geq U$. Moreover, in mini-batch negative sample reuse as in (Ji et al. 2019; Lerer et al. 2019; Broscheit et al. 2020), BOUNDED also holds. Here samples are reused only within one mini-batch of gradient descent so that the mini-batch size provides a bound on the sample dependency.

Local Sampling

In many distributed ML architectures (Ho et al. 2013; Dai et al. 2015; Yuzhen Huang et al. 2018), at each node, a distinct subset of the model parameters—the *local partition*—can be accessed without network communication. *Local sampling* restricts sampling accesses to this local partition (D. Zheng et al. 2020b). This scheme eliminates network overhead for sampling accesses entirely. However, local sampling is NON-CONFORM as nodes see only samples from the local partition. Some implementations re-partition parameters periodically such that all nodes at least see all samples over time (D. Zheng et al. 2020b). Careful re-partitioning might satisfy Eq. (4.1) for certain target distributions; e.g., if π is uniform and parameters are allocated uniformly and at random. In general, however, local sampling cannot provide LONG-TERM (i.e., local sampling is only NON-CONFORM). For example, consider any target distributions in which $\pi_k > 1/Q$ for some k (with Q being the number of nodes). Local sampling cannot satisfy Eq. (4.1) for such a target since key k is available for sampling at only one node at a time. This implies that there is at least one node at which the long-term frequency of k is $\leq 1/Q$.

Direct-Access Repurposing

Another sampling scheme is to repurpose direct-access parameters, i.e., to use them as negative samples. For example, DGL-KE (D. Zheng et al. 2020b) generates some of the samples by repurposing parameters that occur as positives in other data points of an SGD mini-batch. This requires no additional communication for sampling accesses, as the values for the direct access parameters are transferred to the node either way. In this scheme, the relative frequency of a seeing a key in a sample depends on the occurrence frequency of the key in the training data. As the training data occurrence distribution can be (and typically is (Broscheit et al. 2020; Mikolov et al. 2013)) different from the target distribution, this scheme is NON-CONFORM.

4.3.3 A Primitive for Sampling

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

To overcome these limitations, we propose to extend the PS API with a sampling primitive that allows applications to access a sample from a target distribution, under a specific sampling conformity level. The sampling manager in NuPS transparently chooses a sampling scheme that conforms with the chosen conformity level and applies the scheme for all sampling accesses. We propose one operation dist = register_distribution(π, L) to register a specific sampling distribution π under a specific sampling conformity level L, and a combination of two operations to draw samples:

$$\begin{aligned}\texttt{handle} &= \texttt{prepare\_sample}(\texttt{dist}, \texttt{N})\\ \texttt{keys}, \texttt{values} &= \texttt{pull\_sample}(\texttt{handle}[, \texttt{n}_\texttt{j}])\end{aligned}$$

The argument N is the number of desired samples. The prepare_sample operation is intended to return instantaneously (and run preparatory work in the background), whereas pull_sample blocks if called synchronously. After pull_sample returns, the corresponding keys are stored in keys and corresponding values are copied to values. Applications can call pull_sample once to obtain all N samples at once or multiple times to obtain the N samples in smaller portions (by passing $n_0, n_1, ... < N$ to multiple invocations of pull_sample such that $\sum n_j = N$). Such partial pulls give the PS more flexibility, and, thus, may result in better performance.

Algorithm 4.2 illustrates how the sampling primitive could be used in the distributed SGD example of Section 2.2. The worker registers the desired sampling distributing (line 1), requests a sample when the batch is prepared (line 8),[6] and pulls the sample keys and values during training (line 10). It then uses the sample keys and values together with the direct access keys and values (lines 11–13). In the algorithm, we use $\oplus$ to depict concatenation (of, e.g., two vectors).[7]

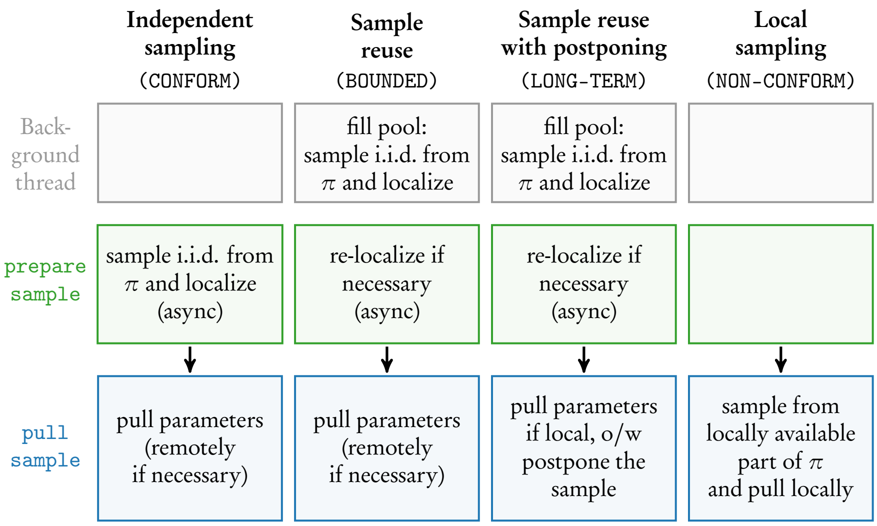
This extension provides sufficient flexibility for implementing a wide range of sampling schemes, as we describe in the following Section 4.3.4.

[6] The num_samples($\mathcal{B}_i$) function determines the number of samples required in batch $\mathcal{B}_i$.

[7] For example, with two vectors $a = \begin{pmatrix} 0 & 1 \end{pmatrix}$ and $b = \begin{pmatrix} 2 & 3 & 4 \end{pmatrix}$, we have $a \oplus b = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \end{pmatrix}$.

Algorithm 4.2: Sampling support in distributed asynchronous SGD. Differences to the Classic PS implementation (Algorithm 2.3 on page 17) are highlighted in green. We write ⊕ to depict concatenation (e.g., of two vectors).

```
Data: $\mathscr{D}$: training dataset,
      num_epochs: number of epochs to run,
      batch_size: batch size,
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads ,
      $\pi$: a sampling distribution,
      $L$: a sampling conformity level
dist = register_distribution ( $\pi$, $L$ )
for epoch ← 1 to num_epochs do
    b = num_batches ( $\mathscr{D}$, batch_size, t, T )
    // data loading (pipeline parallel with training, in separate thread(s))
    $\mathscr{B} = []$
    $\mathscr{S} = []$
    for i ← 1 to b do
        $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch, t, T )
        $\mathscr{S}_i$ = prepare_sample ( dist, num_samples($\mathscr{B}_i$) )
    // training
    for i ← 1 to b do
        $\text{keys}_{\text{samples}}, w_{\text{samples}}$ = pull_sample ( $\mathscr{S}_i$ )
        $w$ = pull ( keys($\mathscr{B}_i$) ) $\oplus\ w_{\text{samples}}$
        $\Delta w$ = compute_update ( $\mathscr{B}_i$, $w$ )
        push ( keys($\mathscr{B}_i$) $\oplus$ $\text{keys}_{\text{samples}}$, $\Delta w$ )
```



	Independent sampling (CONFORM)	Sample reuse (BOUNDED)	Sample reuse with postponing (LONG-TERM)	Local sampling (NON-CONFORM)
Background thread		fill pool: sample i.i.d. from π and localize	fill pool: sample i.i.d. from π and localize	
prepare sample	sample i.i.d. from π and localize (async)	re-localize if necessary (async)	re-localize if necessary (async)	
pull sample	pull parameters (remotely if necessary)	pull parameters (remotely if necessary)	pull parameters if local, o/w postpone the sample	sample from locally available part of π and pull locally

Figure 4.4: Sampling scheme implementations in NuPS.

The extension derives its flexibility from three key design choices. First, the extension transfers sampling from the application to the PS. Second, the extension provides the PS with a hook for doing preparatory work, such as pre-fetching parameter values, modifying partitions, or coordinating among nodes. Third, the extension does not force final decisions (e.g., about the sampled keys) before `pull_sample` returns.[8]

4.3.4 The Sampling Manager in NuPS

The sampling manager is responsible for generating samples and managing the corresponding parameters. The sampling manager of NuPS currently supports four sampling schemes behind the sampling API. Figure 4.4 provides an overview. Schemes implement `prepare_sample` and `pull_sample`, and optionally a background thread. From the four implemented schemes, the sampling manager picks a scheme that is suitable for the specified conformity level. We now discuss the schemes in turn.

Independent Sampling (`CONFORM`)

In this scheme, NuPS samples i.i.d. from the target distribution and localizes the corresponding parameters in `prepare_sample` (such that they can be accessed locally when `pull_sample` is called). In `pull_sample`, NuPS accesses parameters remotely if they have been relocated to another node between the invocation of `prepare_sample` and the invocation of `pull_sample` (this can

[8] For this reason, the `prepare_sample` operation returns a handle rather than the parameter keys directly.

happen because other nodes can independently work on the same parameters). This approach is `CONFORM` because each worker samples i.i.d. from π.

Sample Reuse (`BOUNDED`)

NuPS implements a sample reuse scheme that reuses pools of keys. The pooling increases the temporal distance between the reused samples and thereby increases randomness. For a given pool size G and use frequency U, NuPS repeatedly samples G keys i.i.d. from π to form a *sample pool* and produces samples by traversing the sample pool U times, each time in a random order. For example, consider $U = 2$ and suppose that the i.i.d. draws produce keys k_1, k_2, and k_3, respectively. With $G = 1$, we obtain sample sequence $k_1k_1k_2k_2k_3k_3$. With $G = 3$, a sequence such as $k_1k_2k_3k_2k_1k_3$ is possible. The pools are prepared by a separate background thread. When the background thread generates a new pool, it localizes the corresponding parameters. NuPS re-localizes the parameters in `prepare_sample` if they have been relocated to another node since pool preparation. In the `pull_sample` operation, NuPS accesses the parameters remotely if necessary. This sample reuse scheme is `BOUNDED` because samples are drawn i.i.d. from the target distribution π, inter-sample dependency is bounded by $U \cdot G$, and U is identical for all samples.

The background thread determines automatically when to prepare a new pool. Adding a new pool takes time and (for good performance) the localization should be finished when `pull_sample` is called. This time depends on the ML task, the used hardware, and the system configuration. To estimate this time, we use a heuristic. Note that while the heuristic may affect performance, it does not affect correctness. In particular, the background thread keeps track of the duration of previous pool relocations. If the number of prepared, but unused samples is less than double of the current estimated relocation time, the preparation of another pool is triggered.

Sample Reuse With Postponing (`LONG-TERM`)

NuPS additionally implements sample reuse with *sample postponing*. This is identical to the described sample reuse scheme, but adds sample postponing: if sample i cannot be accessed locally in `pull_sample`, NuPS re-localizes the corresponding parameters, postpones sample i for later use, and uses sample $i+1$ instead. To achieve `LONG-TERM`, it is crucial that, at some point, samples are used (and not re-postponed indefinitely).[9] Thus, NuPS postpones

[9] If samples could be re-postponed indefinitely, some samples may never be used because they are constantly being relocated. In such cases, Eq. (4.1) would not hold.

only within the N samples of one invocation of `prepare_sample` (in other words, only within the samples of one handle). I.e., when NuPS finds a non-local sample (in `pull_sample`), it moves the sample to the end of the N samples of this handle. NuPS postpones each sample maximally once. When it reaches samples that it has already postponed once (towards the end of the N samples), it accesses them remotely if necessary. This implementation of postponing reduces communication overhead only if the samples of one handle are pulled in groups smaller than N and there is some time between these partial pulls for the parameter relocation. Assuming that N is bounded from above, it provides `LONG-TERM`. It does not provide `BOUNDED` because sampling probabilities depend on the current allocation of a key (i.e., keys can be postponed to a later sample if they are not local).

Local Sampling (`NON-CONFORM`)

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

4.4 Experiments

We conducted an experimental study to investigate whether and to what extent non-uniformity is beneficial for PS performance. Our source code, datasets, and information on reproducing experiments are available online.[10]

In this study, we compared the performance of NuPS to several state-of-the-art PSs on three large-scale ML tasks (Section 4.4.2). Further, we conducted an ablation study (Section 4.4.3), investigated the scalability of different approaches (Section 4.4.4), evaluated different sampling schemes (Section 4.4.5), and explored specific components of NuPS (Sections 4.4.6

[10] `https://github.com/alexrenz/NuPS/tree/sigmod22`

Table 4.2: ML tasks, models, and datasets.

Task	Model parameters				Data		
	Model	Keys	Values	Size	Dataset	Data points	Size
Knowledge graph embeddings	ComplEx, dim. 500	4.8 M	4.8 B	35.9 GB	Wikidata5M	21 M	317 MB
Word embeddings	Word2Vec, dim. 1 000	1.9 M	1.9 B	7.0 GB	1b word benchm.	375 M	3 GB
Matrix factorization	Latent factors, rank 1 000	11.0 M	11 B	82.0 GB	10m × 1m matrix	1 000 M	31 GB

and 4.4.7). Our major insights are: (i) NuPS was more than an order of magnitude faster than existing PSs. (ii) NuPS achieved best performance when it replicated a small fraction of the model parameters, and relocated all other parameters. (iii) Both sample reuse and local sampling significantly reduced communication overhead for sampling access. We conclude that *a non-uniform PS is key for high performance in ML tasks with non-uniform parameter access.*

4.4.1 Experimental Setup

Tasks

We considered three popular ML tasks that require long training: knowledge graph embeddings (KGE), word embeddings (WE), and matrix factorization (MF). The tasks differ in multiple ways, including the number of parameters, parameter access distributions, sampling distribution, and frequency of sampling accesses. Table 4.2 provides a summary of the models and datasets. Table 4.3 depicts the share of direct and sampling access for each task. These tasks are similar to the ones used in Section 3.3, but with some important differences. The tasks in this section are "harder" to distribute efficiently than the ones in Section 3.3 because they contain less explicit locality, more skew, and we use faster compute hardware and higher degrees of parallelism (see a detailed discussion at the end of this section). In the following, we briefly discuss each of the three tasks.

Knowledge graph embeddings This KGE task, based on (H. Liu et al. 2017), trains ComplEx (Trouillon et al. 2016) (one of the most popular KGE models) embeddings using SGD with AdaGrad (Duchi et al. 2011) and negative sampling (Ruffinelli et al. 2020; H. Liu et al. 2017). Negative sampling creates sampling access in this task: to generate negative samples, both the subject and the object entity of a positive triple are perturbed n_{neg} times, by drawing random entities from a uniform distribution over all entities (we used a common setting of $n_{\text{neg}} = 100$ (Ruffinelli et al. 2020)). We used the Wikidata5M dataset (X. Wang et al. 2019), a real-

Table 4.3: Share of direct and sampling access for each ML task.

Task	Parameter access	
	Direct	Sampling
Knowledge graph embeddings	69%	31%
Word embeddings	44%	56%
Matrix factorization	100%	0%

world knowledge graph with 4 818 679 entities and 828 relations, and a common embedding size of 500 (Ruffinelli et al. 2020). We partitioned the subject–relation–object triples of the dataset to the nodes randomly, as done in (Kochsiek and Gemulla 2021). We used LibKGE (Broscheit et al. 2020) (commit 3146885) to evaluate models and report the *mean reciprocal rank (filtered)* (MRRF) as metric for model quality.

Word embeddings This WE task, based on (Mikolov et al. 2013), uses SGD and negative sampling to train the skip-gram Word2Vec (Mikolov et al. 2013) model (dimension 1 000) on the One Billion Word Benchmark (Chelba et al. 2013) dataset (with stop words of the Gensim (Řehůřek and Sojka 2010) stop word list removed). The negative sampling creates sampling accesses: in this task, for each word pair, 3 negative samples are drawn from a distribution that is based on word frequencies (see Section 4.1.2). We used common model parameters (Mikolov et al. 2013) for window size (5), minimum count (1), and frequent word subsampling (0.01). We measured model accuracy using a common reasoning task of 19 544 semantic and syntactic questions (Mikolov et al. 2013).

Matrix factorization This MF task, based on (Teflioudi et al. 2012), uses SGD to factorize a synthetic, zipf-1.1 distributed 10m × 1m dataset with 1b revealed cells, modeled after the Netflix Prize dataset.[11] We generated this dataset to model real-world datasets more closely than the uniformly distributed datasets that we adapted from (Makari et al. 2015) for our study in the previous chapter (see Section 3.3). Data points were partitioned to nodes by row and to workers within a node by column. Each worker visited its data points by column (to create locality in column parameter accesses), with random order of columns and of data points within a column. There is no sampling access in this task. We report the *root mean squared error* (RMSE) on the test set as metric for model quality.

[11] See `https://netflixprize.com/`. We use a synthetic dataset because the largest openly available dataset that we are aware of is only 7.6 GB large.

Baselines

We compared performance to a classic PS, to Petuum (a state-of-the-art replication PS), to Lapse (the state-of-the-art relocation PS), and to a single node implementation. As classic PS, we used Lapse with relocation disabled, which, as seen in Section 3.3.2, provides performance similar to PS-Lite. We ran both the SSP and ESSP protocols of Petuum (Xing et al. 2015), with different staleness thresholds. Petuum does not provide KGE or WE implementations. Thus, we implemented the KGE task described above in Petuum. We used version 1.1 of Petuum. We did not implement specific sampling schemes in application code, i.e., applications draw independent samples and access them via direct access. We used a shared memory implementation with 8 worker threads as single node baseline.

Implementation and Cluster

We implemented NuPS in C++, using ZeroMQ and Protocol Buffers for communication, based on PS-Lite (Mu Li et al. 2014a) and Lapse. We used a local cluster of up to 16 Lenovo ThinkSystem SR630 computers, running Ubuntu Linux 20.04, connected with 100 Gbit Infiniband. Each node was equipped with two Intel Xeon Silver 4216 16-core CPUs, 512 GB of main memory, and one 2 TB D3-S4610 Intel SSD. We compiled code with g++ 9.3.0, except for Petuum, which we compiled with g++ 7.5.0, as the compilation with g++ 9.3.0 and g++ 8.4.0 failed. Unless specified otherwise, we used 8 nodes and 8 worker threads per node. In Lapse and NuPS, we additionally used 1 server and 3 ZeroMQ I/O threads per node. In Petuum, we used 4 communication channels per node. To prevent exploding gradients, we used gradient norm clipping as suggested in (Pascanu et al. 2013) for replicated parameters in the WE and MF tasks (clipping updates that exceed the average norm by more than 2x). In the KGE task, the use of AdaGrad prevented exploding gradients. For each task, we tuned hyperparameters on the single node and used the best found hyperparameter setting throughout all systems and variants.

NuPS

We ran NuPS in two configurations: (i) a generally applicable *untuned* configuration that requires no task-specific tuning and (ii) a task-specific tuned configuration. The untuned configuration employs a heuristic to decide the management technique for each parameter: it replicates a parameter if its access frequency exceeds 100 times the mean access frequency. This heuristic is computed from dataset frequency statistics. The untuned configuration further employs sample reuse without postponing (`BOUNDED`) with a use frequency of U=16. To indicate the performance potential of task-specific insights, we included a tuned configuration by informing our configuration choices with the results of our detail experiments in Sections 4.4.5 and 4.4.6. The tuned configuration for KGE replicates the 900 most frequently accessed keys (the same as the untuned setting), but uses local sampling (`NON-CONFORM`). The tuned configuration for WE replicates the 209 k most frequently accessed keys (64x more keys than the untuned configuration), and employs local sampling (`NON-CONFORM`). For MF, the untuned configuration seemed to be near-optimal, such that we did not add a separate tuned configuration. Unless mentioned otherwise, we used the settings of the untuned configuration and a replica staleness threshold of 40 ms in all experiments. Throughout all experiments, we used a pool size of 250 in the sample reuse scheme.

Measures

Unless noted otherwise, we ran all variants with a fixed 6 h time budget. We measured model quality over time and over epochs within this time budget (using the quality metrics described above). As in Section 3.3, we conducted 3 independent runs of each experiment, each starting from a distinct randomly initialized model, and report the mean. We depict error bars for model quality and run time; they present the minimum and maximum measurements. In some experiments, error bars are not clearly visible because of small variance. Gray dotted lines indicate the performance of the single node baseline. Gray shading indicates performance that is dominated by the single node baseline. We report two types of speedups: (i) *raw speedup* depicts the speedup in epoch run time, without considering model quality; (ii) *effective speedup* is calculated from the time that each variant took to reach 90% of the best model quality that the single node baseline achieved. Unless specified otherwise, we report effective speedups.

Differences to the Experiments in Chapter 3

Overall, the experimental setup of this study seems similar to the setup in Section 3.3. However, there is a series of important differences that make the tasks in this section harder to scale. First, we used simpler (more general-purpose) algorithms in this study, with (much) less explicit locality. In particular, the KGE and WE tasks rely exclusively on latency hiding. In the MF task, no parameter blocking approach is used. Instead, the data points were partitioned by rows and latency hiding is used otherwise. In contrast, in Section 3.3, we evaluated whether PSs could support algorithms that deliberately use PAL techniques to create locality (e.g., the DSGD (Gemulla et al. 2011) parameter blocking approach for MF). Simpler algorithms require less work for from the application developer: e.g., there is no need to develop a blocking approach (MF) or to partition the dataset (KGE).

Second, we examined performance for higher degrees of parallelism in this study. We doubled both the number of worker threads per node and the maximum number of nodes. A higher number of workers leads to a higher chance of localization conflicts (as already observed in Section 3.3).

Third, we used more recent cluster hardware. In particular, the CPUs of these machines were much faster (around 2x faster for our tasks). This made single node baselines significantly faster and thus makes it harder to "hide" communication overhead. (On the other hand, the new cluster is also equipped with faster network hardware.)

Fourth, the datasets in this study are more skewed. In MF, we used a dataset that more realistically models real-world datasets, which means it is skewed. And in KGE, the larger Wikidata5M dataset also happens to be more skewed than the smaller DBpedia-500k dataset in the previous study.

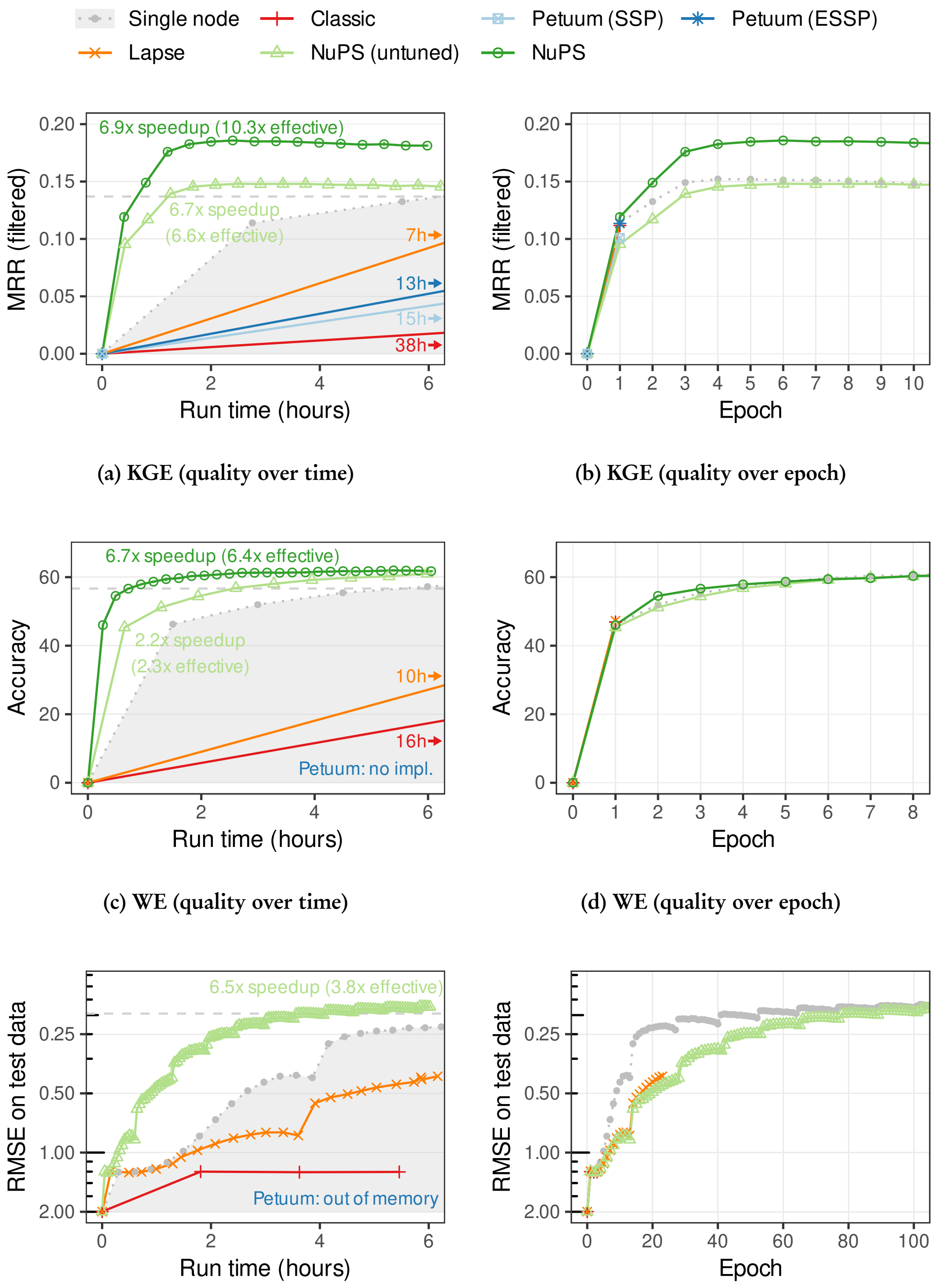
4.4.2 Overall Performance

We investigated the overall effect of a non-uniform PS on PS performance. To do so, we compared the performance of NuPS to existing PSs and to the single node baseline. We ran each variant for the fixed time budget and measured model quality over this time. Figures 4.5a, 4.5c, and 4.5e show model quality over time, Figures 4.5b, 4.5d, and 4.5f show model quality over epoch. **In summary, NuPS was 31–36x faster than a state-of-the-art replication PS (Petuum), 6–46x faster than Lapse (the state-of-the-art relocation PS), and 2.3–10.3x faster than the single node baseline.**[12]

Classic PS and Lapse

The classic PS was inefficient (with epochs over 7x slower than the single node) because it accesses parameters over the network, which induced significant access latency. Lapse was faster than Classic, but still slower than the single node, because Lapse relocates all parameters, including hot spots. Hot spot parameters, however, are frequently accessed by multiple nodes simultaneously, such that some of these nodes had to wait for the relocation to finish or access the parameter remotely, which induced access latency. The key reasons for Lapse performing worse in this study than in our previous

[12] The comparisons to Petuum and Lapse report raw speedups, because Petuum and Lapse did not reach the 90% thresholds within the time budget.



(a) KGE (quality over time)

(b) KGE (quality over epoch)

(c) WE (quality over time)

(d) WE (quality over epoch)

(e) MF (quality over time)

(f) MF (quality over epoch)

Figure 4.5: End-to-end performance of different PSs on 8 nodes. NuPS outperformed Petuum and Lapse by up to one order of magnitude. The gray shaded area indicates performance that is dominated by the single node baseline. The dashed gray line depicts the model quality threshold at which effective speedups are computed.

study in Section 3.3 are (i) that we used simpler algorithms that exhibit less locality,[13] (ii) that we used more worker threads, (iii) that the compute of the used cluster is faster (i.e., each thread is faster individually), and (iv) that the tasks contain more skew than in our previous study.

The per-epoch model quality of Classic and Lapse was indistinguishable from the single node in KGE and WE, as these systems provide sequential consistency for all parameters and employ no specialized sampling schemes. In MF, all distributed variants provided lower per-epoch quality than the single node, an effect that has been observed before (Makari et al. 2015). The step pattern that is visible in MF training stems from the bold driver heuristic (Battiti 1989). The MF implementation (Makari et al. 2015) that we adapted for our experiments uses this heuristic to tune the learning rate.

Petuum SSP and ESSP

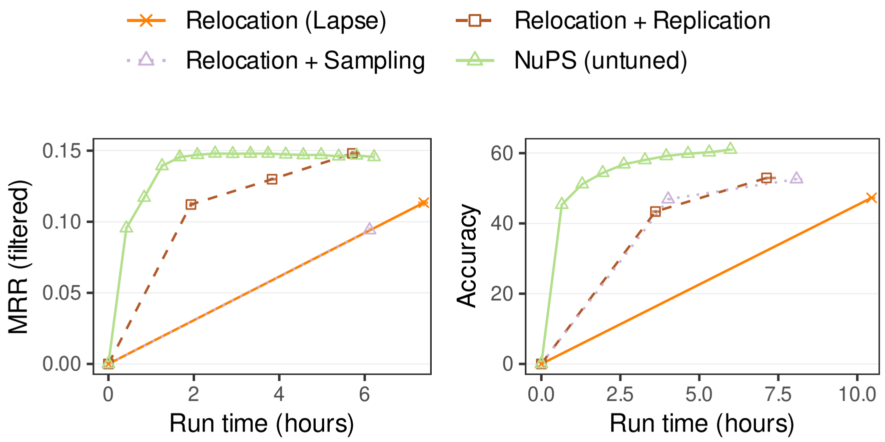
For KGE, we ran Petuum SSP and ESSP with staleness thresholds 1, 10, 100, 200, or 1 000, and tried different frequencies for advancing the clock.[14] None of the configurations completed the first epoch within the time budget of 6 hours. We observed the best performance for ESSP with staleness 10, which finished the first epoch after 13h with a model quality (MRRF) of 0.11. The best SSP run (staleness 200) finished the first epoch after 15h with a model quality of 0.10. The reasons for this performance are that Petuum is inefficient for long tail parameters (as discussed in Section 4.2.1) and that Petuum's replica approach is inefficient for sampling because sampling access provides no locality: SSP replicas are mostly cold, ESSP over-communicates. Petuum's MF implementation ran out of memory, because it stores the training matrix in dense format.

NuPS

The untuned NuPS configuration outperformed existing PSs across all three tasks. For KGE and MF, it was also clearly faster than the single node, with up to 6.7x effective speedups over the single node and minimal negative effect on (per-epoch) model quality. For WE, however, it barely outperformed the single node (but still outperformed existing PSs). In contrast, the tuned configuration provided 4.6–10.3x effective speedups over the single node across all three tasks. For KGE, the tuned configuration of NuPS provided

[13] Such simpler algorithms are harder to scale for the PS, but require less additional effort from application developers.

[14] We tried to advance the clock after every 1, every 10, and every 100 data points. We observed best performance for clocking after every 10th data point. Due to the high run times of Petuum, we ran each configuration only once.



(a) KGE

(b) WE

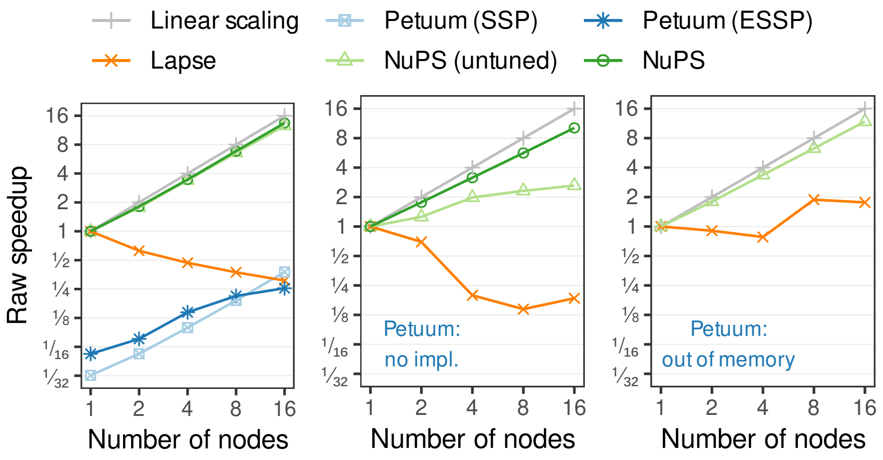
Figure 4.6: Ablation. Both (i) combining replication and relocation and (ii) integrating specialized sampling access management techniques improved performance individually, and it was beneficial to combine the two.

better per-epoch convergence than the single node. This was an effect of local sampling; see Section 4.4.5 for more details.

4.4.3 Ablation

NuPS introduces two novel features compared to existing PSs: (i) multi-technique parameter management and (ii) direct sampling support. To investigate individual effects, we enabled each feature individually and measured model quality within the time budget. Figure 4.6 shows the results. We omit MF because it contains no sampling access, such that the entire performance improvement stems from multi-technique parameter management (which is visible in Figure 4.5e). **We found that both multi-technique parameter management and sampling integration can be beneficial individually, and the individual benefits compounded when both were combined.**

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



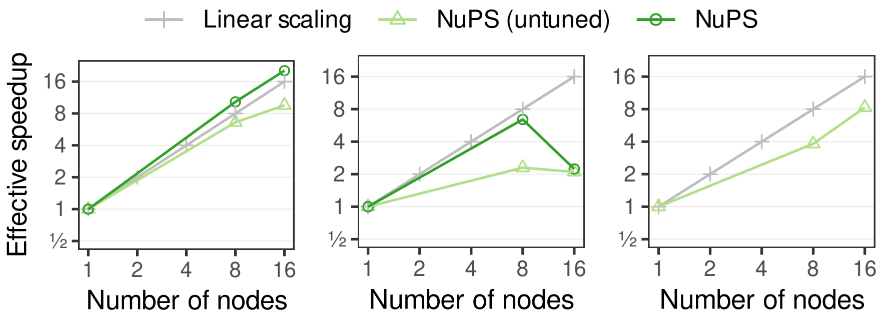
(a) KGE (b) WE (c) MF

Figure 4.7: Strong scaling (logarithmic axes). The y-axis depicts raw speedup, i.e., speedup with respect to epoch run time over the shared-memory single node baseline. NuPS scaled more efficiently than other PSs, with up to near-linear speedups over the single node baseline.

4.4.4 Scalability

To investigate scalability, we ran Lapse, the best Petuum SSP and ESSP configurations, and NuPS for one epoch on 1, 2, 4, 8, and 16 nodes and calculated the raw speedup. Figure 4.7 depicts the results. Further, we ran convergence experiments on 16 nodes for those systems that reached the 90% model quality threshold on 8 nodes. Figure 4.8 depicts the effective speedup for these systems. **Overall, NuPS scaled more efficiently than other PSs, with up to near-linear raw and up to superlinear effective speedups.**

We first discuss raw scalability, i.e., the speedup with respect to epoch run time (Figure 4.7). On a single node, NuPS and Lapse were faster than Petuum because NuPS and Lapse access local parameters via shared memory, whereas Petuum sends intra-process messages to do so. Lapse provided poor scalability because (with the latency hiding technique) the more nodes are used, the higher the chance that multiple nodes access a parameter at the same time and, thus, that they have to wait for a relocation to finish or to access parameters remotely. Neither Petuum ESSP nor SSP outperformed the shared-memory single node baseline, even on 16 nodes. ESSP scaled poorly even when compared to its own (inefficient) run time on a single node (4.8x faster on 16 nodes) because its eager replication protocol overcommunicates: after a short warm-up period, each node holds a replica of the full model. The more nodes, the more replicas had to be synchronized,



(a) KGE

(b) WE

(c) MF

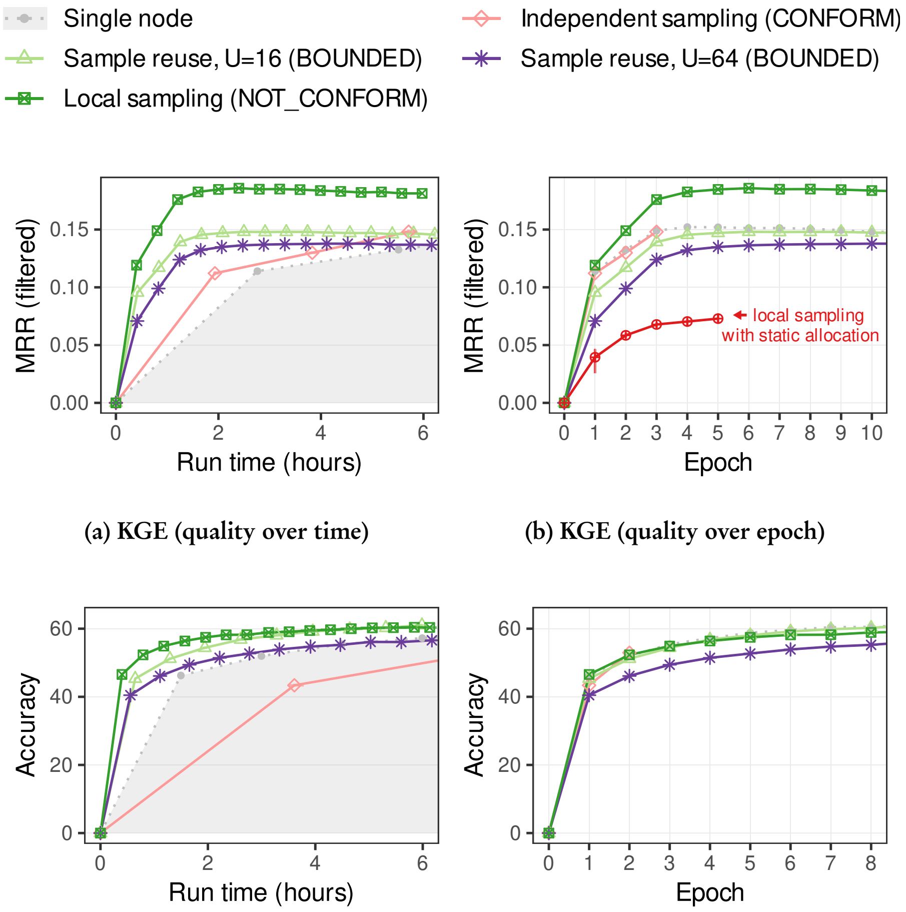
Figure 4.8: Effective scalability (logarithmic axes). The y-axis depicts effective speedup, i.e., speedup with respect to reaching 90% of the best model quality observed on a single node.

such that synchronization became a bottleneck. The lazy SSP protocol scaled better than ESSP compared to its own (inefficient) single node run time (12x faster on 16 nodes), but its overall performance was poor because its replicas were cold most of the time (and thus required synchronous replica refreshes).

NuPS scaled more efficiently than existing PSs because it (i) limits the bottleneck of eager replication by replicating only a small subset of hot spot parameters, (ii) prevents the majority of relocation conflicts by employing relocation only for long tail parameters, and (iii) employs sampling schemes to reduce sampling communication overhead. With 16 nodes, it provided up to 13.4x raw speedups over the shared memory single node. NuPS further provided up to 20x effective speedups for KGE and 8x for MF (see Figure 4.8). For WE, although the raw speedup on 16 nodes was 10.2x, the effective speedup was only 2.2x. The reason for this is that we used the hyperparameter configuration that worked best on the single node throughout all experiments. With other hyperparameters, we observed better effective speedups for WE.

4.4.5 Effect of Sampling Schemes

We investigated the effect of different sampling schemes in NuPS on run time and model quality. To do so, we ran KGE and WE with different sampling schemes: independent sampling (`CONFORM`), U=16 and U=64 sample reuse without postponing (`BOUNDED`) and with postponing (`LONG-TERM`), and local sampling (`NON-CONFORM`). Figures 4.9a and 4.9c show model quality over time, Figures 4.9b and 4.9d show model quality over epoch. We omit MF as it does not contain sampling access. We further omit the results from sample reuse with postponing as its results were within 10% of sample



(a) KGE (quality over time)

(b) KGE (quality over epoch)

(c) WE (quality over time)

(d) WE (quality over epoch)

Figure 4.9: Performance of different sampling access management techniques. Both sample reuse and local sampling led to significant speedups over independent sampling.

reuse without postponing.[15] **We found that both sample reuse and local sampling led to significant speedups over independent sampling, with small negative or—in the case of local sampling—even positive effects on per-epoch model quality.**

Independent Sampling and Sample Reuse

Independent sampling provided per-epoch quality near-identical to the single node, but was slowest, because it induced high communication overhead for each sample. *Sample reuse* had lower communication overhead (and, thus, faster epoch run times), but at the cost of a (small) negative effect on per-epoch model quality. The higher the use frequency, the faster an epoch and the larger the negative effect on quality. The U=16 variant provided a good compromise, with minimal effect on model quality and fast run times.

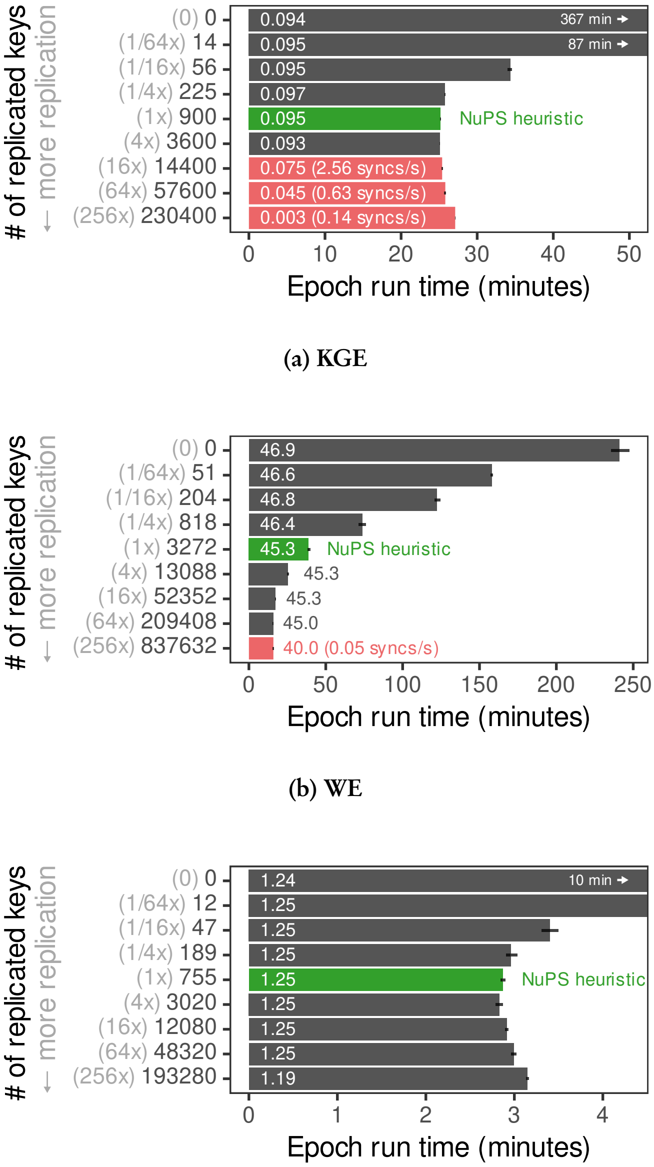
Local Sampling

Local sampling exhibited excellent performance, despite providing no guarantees on sampling quality: it was fast and per-epoch model quality was as good as the single node in WE, and was *better* in KGE. We hypothesize that this mainly is because NuPS combines local sampling with dynamic allocation: both tasks continuously relocate model parameters, such that the local parameter partitions contain many different parameters over time. To evaluate this hypothesis, we ran local sampling with *static allocation* in KGE. Figure 4.9b includes the results: with static allocation, model quality deteriorated drastically. We further conjecture that the reason for the better-than-single-node quality of local sampling in KGE was that relocation led to local samples that were more informative than global samples. Similar effects have been observed previously (D. Zheng et al. 2020b).

4.4.6 Choice of Management Technique

We investigated how the choice of management technique, i.e., the choice of whether to replicate or relocate a key, affects the performance of NuPS. The NuPS untuned heuristic replicates the 900 most frequent keys in KGE, the 3 272 most frequent keys in WE, and the 755 most frequent column keys in MF. We varied these numbers by factors $\frac{1}{64}$, $\frac{1}{16}$, $\frac{1}{4}$, 4, 16, 64, and 256. The leftmost columns of Table 4.4 depict what share of keys was replicated for each setting. We ran one epoch of each setting and measured epoch run time and model quality. Figure 4.10 depicts the results. **We found that it was**

[15] Postponing made no measurable difference in KGE, and sped up WE run times by 10%, with no measurable impact on model quality.



(a) KGE

(b) WE

(c) MF

Figure 4.10: Impact of the management technique on epoch run time and model quality. The numbers in the plots depict model quality. A run is marked red if the resulting model quality was not within 10% of the model quality without replication. For these runs, the numbers in the plots additionally depict the actual synchronization frequency.

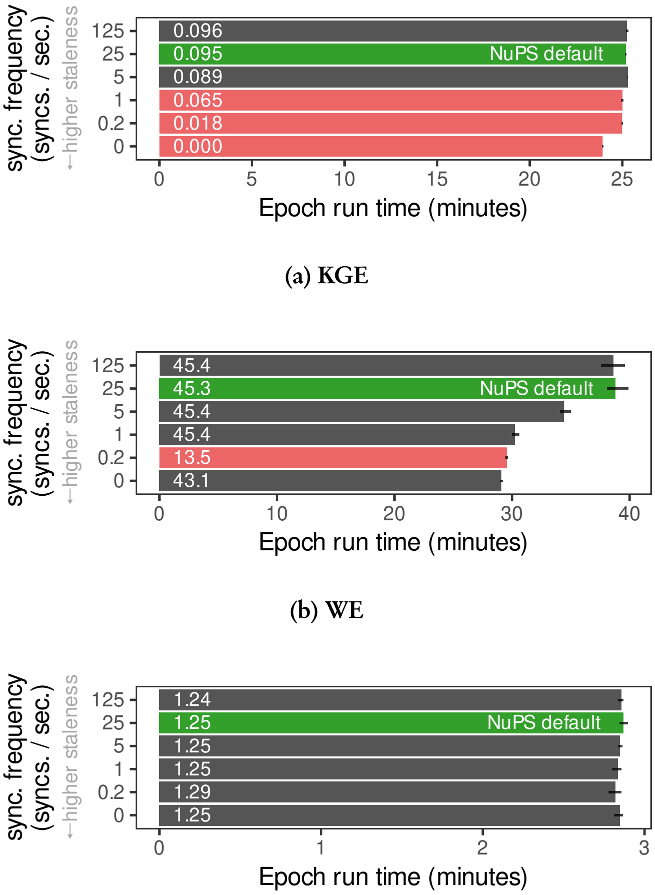
crucial for performance to replicate "enough" parameters such that the set of hot spot parameters is managed by replication, but not too many parameters, as replication created significant over-communication for long tail parameters.

This effect was visible for all tasks: starting from no replicated keys (i.e., all keys managed by relocation), increasing the number of replicated keys first improved run time, and had minimal effect on model quality. However, after some point, replicating more keys deteriorated model quality, and even slowed down run time for KGE and MF. The reason for the negative effect on model quality was that the replicas were stale, because the replica updates became too large to synchronize them frequently over the network of the cluster. We configured NuPS to provide the default 40 ms staleness bound (i.e., 25 synchronizations per second), but to *not* block operations when it did not reach this goal. Figure 4.10 includes the actual synchronization frequency if model quality was not within 10% of the model quality without replication. The middle columns of Table 4.4 provide the size of the replicated values for all settings. For example, the 64x WE setting replicated 799 MB of parameter values. Large numbers of replicated keys led to slower epoch run times for KGE and MF, because relocation operations competed with replica synchronization for network bandwidth. This effect was not visible for WE because, in WE, the majority of accesses went to replicated keys (and, thus, were fast despite network congestion). The share of accesses that went to replicas is depicted in the rightmost columns of Table 4.4. For example, 88% of all accesses went to replicas in the 64x WE setting.

4.4.7 Effect of Replica Staleness

We investigated the effect of replica staleness on epoch run time and model quality. To do so, we varied synchronization frequency: we synchronized replicas either 125, 25, 5, 1, or 0.2 times per second or not at all. We ran one epoch of each setting and measured epoch run time and model quality after this epoch. Note that without replica synchronization, nodes may hold different models. In these cases, we evaluated the model of the first node. Figure 4.11 reports the results. **Overall, replication had only minimal effect on model quality when replica staleness was low.**

Replication had only small effect on model quality when replicas were synchronized at least 5 times per second. In contrast, infrequent synchronization (less than once per second) deteriorated model quality drastically in KGE and WE. However, infrequent synchronization (or no synchronization at all) worked well in some settings (in particular in MF). We speculate that the



(a) KGE

(b) WE

(c) MF

Figure 4.11: Effect of replica staleness on epoch run time and model quality. The numbers in the bars depict model quality after one epoch. Bars are marked red if model quality was not within 90% of the quality produced by a setting with no replication.

Table 4.4: Share of replicated keys, replica size, and share of accesses to replicas for different extents of replication. A cell is marked red if the resulting model quality was not within 10% of the quality without replication.

	Replicated keys (%)			Size of replicated values (MB)			Accesses to replicas (%)		
Factor	KGE	WE	MF	KGE	WE	MF	KGE	WE	MF
0	0.0000	0.0000	0.0000	0	0	0	0	0	0
1/64 x	0.0003	0.0027	0.0001	0	0	0	23	7	3
1/16 x	0.0012	0.0108	0.0004	0	1	0	33	13	5
1/4 x	0.0047	0.0435	0.0017	2	3	1	38	25	9
1 x (heuristic)	0.0187	0.1740	0.0069	7	12	6	41	45	14
4 x	0.0747	0.6958	0.0275	27	50	23	44	67	19
16 x	0.2988	2.7832	0.1098	110	200	92	45	82	24
64 x	1.1951	11.1330	0.4393	439	799	369	47	88	30
256 x	4.7806	44.5319	1.7571	1758	3195	1475	52	92	37

reason for this was that NuPS employs replication for only a small subset of parameters, such that replication parameters are kept synchronized indirectly through the parameters that are managed by relocation.

4.4.8 Comparison to Task-Specific Implementations

In a general-purpose system, a performance overhead over optimized task-specific implementations is expected. To investigate the extent of this overhead in NuPS, we compared to specific implementations for each task. Each of these implementations is specialized and highly tuned for the respective task. In contrast to a general-purpose PS such as NuPS, these implementations cannot be used to run other ML tasks. Note that some of these implementations use different, more complex training algorithms than the implementations in NuPS. **Overall, we found that NuPS was competitive to specialized and tuned task-specific implementations.**

Matrix Factorization

For MF, we compared to the highly tuned MPI implementations of DSGD and DSGD++ (Teflioudi et al. 2012). We ran convergence experiments on 8 and 16 nodes. We measured how long the implementations took to reach the 90% quality threshold. We used the same hyperparameters, model starting points, and learning rate schedule across DSGD, DSGD++, and NuPS. On 8 nodes, NuPS was 16% faster than DSGD and 15% slower than DSGD++. On 16 nodes, NuPS was 37% faster than DSGD and 16% faster than DSGD++.

Knowledge Graph Embeddings

For KGE, we compared to the highly specialized PyTorch-BigGraph framework (Lerer et al. 2019). Note that PyTorch-BigGraph is designed for a different training algorithm, with different hyperparameters: to reduce communication overhead, it uses mini-batch SGD, whereas the KGE implementation in NuPS employs regular SGD (i.e., batch size 1).[16] To minimize the impact of algorithm hyperparameters in our comparison, we compared epoch run times. NuPS ran an epoch in 12 minutes on 16 nodes (24 minutes on 8 nodes). In this setting (i.e., batch size 1), PyTorch-BigGraph was much slower than NuPS: it took more than 5 hours to run one epoch, both on 8 and 16 nodes. Using a very large batch size led to faster epochs in PyTorch-BigGraph (up to 3x faster with batch size 1 000 than NuPS with batch size 1), but can also be implemented in NuPS.

Word Embeddings

For WE, we are not aware of a highly tuned and publicly available distributed implementation, so we compared to two highly tuned single node implementations: the original C implementation of Word2Vec (Mikolov et al. 2013) and Gensim (Řehůřek and Sojka 2010). The implementation in Gensim and the one in NuPS are both based on the original C implementation. For both single node implementations, we achieved the fastest epoch run times with 64 threads. Gensim completed an epoch in 15 minutes, the original implementation in 12 minutes. With 8x8 threads, NuPS took 13.5 minutes for one epoch; with 16x8 threads, it took 8 minutes. One factor that limits the performance of NuPS compared to these task-specific implementations is that—as other general-purpose PSs (Mu Li et al. 2014a)—NuPS provides per-key atomic updates. To achieve this, workers receive dedicated working copies of parameters. Creating these copies and writing updates back into the parameter store creates overhead compared to the task-specific WE implementations, which let workers read and write in the parameter store directly, without any consistency or isolation guarantees. Empirically, this works well for this particular task, but the effects for other tasks in a general-purpose system are unclear.

[16] This batch size stems from the C++ KGE implementation that we adopted for our experiments (H. Liu et al. 2017).

4.5 Summary

In this chapter, we explored how PSs can be efficient for tasks that exhibit non-uniform parameter access. We discussed two major improvements for PS efficiency in such tasks. First, we found that PSs can be more efficient if they adapt their parameter management techniques to the access patterns of individual parameters. Second, we found that PSs can be more efficient if they account for different types of parameter access, i.e., direct and sampling. Sampling support allows NuPS to transparently use suitable sampling schemes to reduce communication overhead for sampling access.

An important limitation of the multi-technique parameter management described in this chapter is that the technique for each parameter is picked (i) by the application and (ii) statically, before the start of training. Picking suitable techniques in this way requires either domain knowledge or hyperparameter tuning. The heuristic that we used in our experimental evaluation provided decent results for some, but not all, ML tasks. In addition, similar to Lapse, NuPS requires applications to manually initiate parameter relocation, and to potentially tune the timing of these initiations. In the upcoming Chapter 5, we will describe how adaptive PSs can be efficient for many ML tasks out of the box, without prior tuning.

Chapter 5

Attaining Ease of Use: Automatic Adaptivity

In the previous chapters, we found that adapting individual aspects of the PS to the underlying ML task can improve PS efficiency for ML tasks with sparse parameter access. For example, Lapse dynamically adapts parameter allocation (see Chapter 3). NuPS (see Chapter 4) adapts by combining different parameter management techniques (e.g., replication and relocation) and picking a suitable one for each parameter; this allows NuPS to manage parameters with different access patterns efficiently. Further, replication PSs, such as Petuum (Ho et al. 2013; Dai et al. 2015), adaptively replicate parameters on specific nodes when the nodes access these parameters.

However, to be efficient, these approaches require the application to choose the right approach and to specify suitable performance hyperparameters. These choices depend on the ML task, the workload, and even individual parameters. Making efficient choices often requires domain knowledge and/or expensive upfront experimentation. These requirements make existing approaches complex to use for applications. For example, in Lapse, applications need to modify their application code to initiate parameter relocations and tune the timing of these relocations. In multi-technique PSs such as NuPS, applications need to specify upfront which technique to use for which parameter, and—for optimal performance—need to tune these choices (even if the access frequency distribution can be computed relatively easily, see Section 4.4.6). In Petuum, applications need to tune a staleness threshold specifically for each ML task.

In this chapter, we explore whether PSs can adapt to the underlying ML task automatically, i.e., without any prior tuning. To enable such automatic adaptation, we propose *intent signaling*, a novel mechanism for passing information about parameter access from the application to the PS. Intent signaling decouples information from action: the application signals which parameters it *intends* to access before it does so; the PS adapts automatically based on these signals. Intent signaling is easier to use than existing approaches because it requires no upfront information, requires no tuning, and integrates naturally with the construction of training batches in common ML systems. Intent signaling does not only allow for implementing most existing approaches, it also enables more precise—and more efficient—parameter management.

Intent signaling opens a large design space for adaptive parameter management. We explore the main aspects of this space and describe AdaPS, a novel PS that automatically (i.e., without user input) and adaptively (i.e., based on the current situation) decides what to do and when to do it. AdaPS is easy to use as it requires no information besides intent signals and no knob tuning. Behind the scenes, AdaPS dynamically picks a management technique that is currently suitable for the parameter: if, at one point in time t, only one node accesses the parameter, AdaPS relocates the parameter to this node; otherwise, it replicates the parameter to precisely the nodes that have active intent at t. Furthermore, AdaPS learns automatically *when* to act on an intent signal, so that applications do not need to tune when they signal intent. In our experimental evaluation, AdaPS was efficient across multiple large ML tasks without requiring any tuning. It matched or even outperformed existing (more complex to use) approaches out of the box.

This chapter is structured as follows. We begin by analyzing efficiency and complexity of existing PSs (Section 5.1). We then propose intent signaling (Section 5.2), describe AdaPS (Section 5.3), and investigate the performance of AdaPS in an experimental study (Section 5.4).

5.1 Efficiency and Complexity of Existing Approaches

In this section, we briefly recap existing approaches for distributed parameter management and analyze them with respect to ease of use and efficiency for ML tasks with sparse parameter access. For adaptive approaches, we additionally analyze in which dimensions they are adaptive. Figure 5.1 illustrates existing approaches. Table 5.1 summarizes our analyses.

5.1.1 Static Full Replication

Static full replication (Figure 5.1a) replicates the full model to all cluster nodes statically (i.e., throughout training) and synchronizes the replicas periodically, either synchronously (triggered by the application) or asynchronously in the background. As this approach is entirely static, it requires no run time information from the application. So it is relatively easy to use. (It does, however, require the application to either trigger replica synchronization or to set the frequency of background synchronization.) Parameter access is fast, as every worker can access every parameter locally, without synchronous network communication. However, the full replication approach is communication-inefficient for sparse workloads (as visible in Section 4.4.6, for example), as it maintains the replicas of all parameters on all nodes throughout the training task, even though each node accesses only a small subset of these replicas at each point in time. Also, full replication limits model size to the memory capacity of a single node. **In summary, full replication is very easy to use, but inefficient for sparse workloads because it over-communicates.**

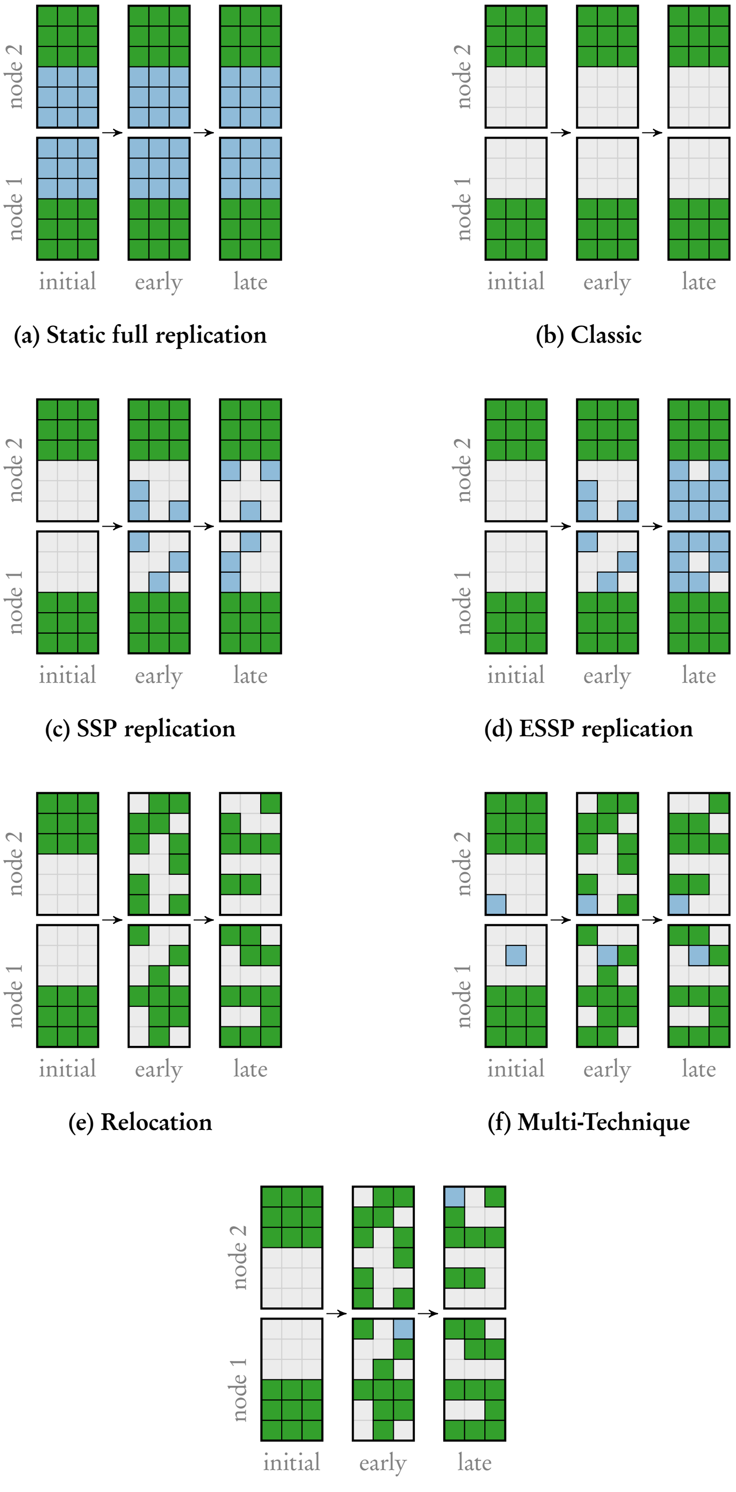
5.1.2 Classic PS

A classic PS (Figure 5.1b) statically partitions the model parameters to the cluster nodes and processes reads and writes by transparently sending messages to the corresponding nodes. A Classic PS is easy to use: it requires no information from the application and no hyperparameter tuning. However, the classic PS approach is inefficient because the vast majority of parameter accesses involve synchronous network communication for sending messages to the node that holds the parameter (see Sections 3.3.2 and 4.4.2). **In summary, a classic PS is very easy to use, but inefficient due to synchronous network communication.**

5.1.3 Replication PS

A replication PS, such as Petuum (Xing et al. 2015), partitions parameters as a Classic PS does. During training, it adaptively replicates a subset of the parameters to nodes that access these parameters. Petuum sets up replicas reactively when a worker on a node accesses the parameter. Thus, the workers have to wait for replicas to be set up synchronously.

The SSP protocol (Figure 5.1c) maintains a replica for an application-specified number of logical clocks, the so-called *staleness bound*. Applications have to tune this staleness bound specifically for each task, as the staleness bound impacts both model quality and run time efficiency and these effects differ from task to task (Ho et al. 2013). This tuning makes SSP complex to use. Further, SSP is inefficient for many tasks because, for realistic staleness bounds, no replicas are set up for the majority of parameter accesses, so that workers have to wait for synchronous replica setup frequently. **In summary, an SSP replication PS is complex to use because it requires tuning and inefficient because of synchronous replica setup.**



(a) Static full replication

(b) Classic

(c) SSP replication

(d) ESSP replication

(e) Relocation

(f) Multi-Technique

(g) AdaPS

Figure 5.1: Parameters held by different nodes at different times (initially; and early and late during training) in common parameter management approaches. One square depicts one parameter. A node either (i) cannot access the parameter locally (⬜), (ii) holds the main copy of the parameter (🟩), or (iii) holds a replica of the parameter (🟦).

Table 5.1: Approaches to distributed parameter management: adaptivity, ease of use, and efficiency for sparse workloads. Some approaches adaptively set up and destruct replicas, and some adaptively change the main location of parameters. However, existing approaches are mostly static with respect to choice of management technique, and require the application to time adaptation.

	Adaptivity					
Approach	Replication	Parameter location	Choice of technique	Timing	Ease of use	Efficiency
Full replication	static (full)	static	single	none	++	--
Classic PS (PS-Lite)	none	static	single	none	++	--
Replication PS (SSP)	adaptive	static	single	by application	-	-
Replication PS (ESSP)	adaptive	static	single	by application	-	--
Relocation PS (Lapse)	none	adaptive	single	by application	--	-
Multi-Technique PS (BiPS, Parallax)	static (partial)	static	static	none	--	+
Multi-Technique PS (NuPS)	static (partial)	adaptive	static	by application	--	+
AdaPS (this thesis)	adaptive	adaptive	adaptive	adaptive	+	++

The ESSP protocol (Figure 5.1d) maintains the replica *throughout the entire training task*. This mitigates the inefficiency of reactive replica creation (as each replica is set up only once), at the cost of over-communication. For many workloads, after a short setup phase, ESSP is essentially equivalent to full replication. This makes ESSP inefficient for sparse workloads. **In summary, an ESSP replication PS is complex to use, and inefficient due to over-communication.**

5.1.4 Relocation PS

A relocation PS, such as Lapse (Section 3.2), adaptively changes the location of parameters during training, such that parameters can then be accessed locally at the respective nodes, and no replica synchronization is required.[1] Key for the efficiency of a relocation PS is that parameter relocation is *proactive*, i.e., that parameter relocation runs asynchronously and is finished before the parameter is accessed. The PS lacks the necessary information to trigger relocations proactively. Thus, Lapse requires the application to trigger parameter relocations manually via the additional `localize` primitive. Application developers are required to add appropriate invocations to their application code and (for optimal performance) tune the relocation *offset*, i.e., how long before the actual access parameter relocation is triggered. Relocation should be early enough such that it is finished when the parameter is accessed, but not too early to minimize the probability of relocating the parameter away from other nodes that are still accessing it. Offloading these performance-critical decisions to the application makes relocation PSs complex to use and leads to potentially sub-optimal performance. In addition, relocation PSs are inefficient for many real-world ML tasks because they are inefficient for hot spots, i.e., parameters that are frequently accessed by multiple nodes concurrently. **In summary, relocation PSs are complex to use, and inefficient for many real-world ML tasks.**

5.1.5 Multi-technique PS

Multi-technique PSs, such as NuPS, support multiple parameter management techniques (e.g., replication, classic, or relocation) and let the application pick a suitable one for each parameter. Parallax (S. Kim et al. 2019) and BiPS (Q. Zheng et al. 2021) support static replication (i.e., creating replicas on all nodes) and classic. NuPS supports static replication and relocation (Section 4.2.2). The choice of technique in existing multi-technique PSs is static: for each

[1] As before, we assume the general-purpose latency hiding technique here, i.e., that there is no explicit locality through data clustering or parameter blocking. If there is explicit locality, parameter relocation is highly efficient, as discussed in Section 3.1.2.

parameter, the application picks one technique before training, which is then used throughout. Using a suitable technique for each parameter can improve PS efficiency. However, it requires information about the workload. As these PSs decide on a technique for each parameter before training, they require this information upfront. There are heuristics that pick a technique for each parameter (e.g., see Section 4.4.1 or (S. Kim et al. 2019)). These heuristics require access frequency statistics and do not consistently achieve optimal performance. Thus, manual tuning can be required to achieve high efficiency. These information and tuning requirements make multi-technique PSs very complex to use and—if not tuned appropriately—lead to sub-optimal performance. NuPS additionally requires the application to manually trigger relocations (as in Lapse), further complicating its use. **In summary, multi-technique PSs are efficient, but very complex to use.**

5.1.6 Summary

Adaptivity is key for efficient distributed parameter management for sparse ML tasks. However, existing approaches adapt only with respect to a few dimensions, see Table 5.1. Adaptivity also requires information about the underlying workload. In existing ML systems, this information is not available to the PS. Thus, current adaptive approaches place key parameter management decisions (which technique/PS to use, when to relocate, etc.) in the hand of the application and require the application to tune these performance knobs. This interdependence makes current adaptive approaches complex to use: developers need to learn about PSs and their performance knobs, modify application code, and run training multiple times for tuning. This interdependence also limits efficiency: applications can make sub-optimal decisions and neglect the tuning of performance knobs. The interdependence also hinders the development of more adaptive approaches (e.g., combining several types of adaptivity, or picking management techniques dynamically during run time), as more adaptivity would further increase complexity.

5.2 Intent Signaling

To enable easy-to-use adaptive parameter management, we propose *intent signaling*, a novel mechanism that naturally integrates into common ML systems. Intent signaling passes information about upcoming parameter access from the application to the PS. It *decouples* information from action, with a clean API in between: the application provides information (intent signals); the PS transparently adapts to the workload based on the intent signals. I.e., all parameter management related decision-making and knob tuning is done transparently by the PS. 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 time window in the future. A typical choice for the time window would be one training batch. For example, a worker could signal *I will access parameters 13 and 16 in batch 5*. We use logical clocks as a general way to specify the start and end points of an intent. Each worker i has one logical clock C^i that is independent of other workers' clocks. (These clocks are used only to define time steps, i.e., no clock synchronization among workers is necessary.)[2] Each worker advances its clock with an `advanceClock()` primitive (as done in Petuum (Xing et al. 2015), 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. Key is that intent is signaled before the parameter is actually accessed, such that the PS has time to adapt proactively.

Intent signaling integrates naturally with the data loader paradigm of common ML systems: in common ML systems, there is one component—usually in one or multiple separate threads—that prepares training batches before they are processed by the worker thread(s). Examples are the data loader in PyTorch (Paszke et al. 2019), TensorFlow (Abadi et al. 2016) datasets, and the Gluon data loader in MXNet (T. Chen et al. 2015). While preparing the training data for the batch, this component could signal intent for the batch, before the training thread later accesses the corresponding parameters.

We propose the following primitive for signaling intent:

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

With this primitive, a worker signals that it intends to access a set of parameters in the time window between a start clock C_{start} (inclusive) and an end clock C_{end} (exclusive). The primitive allows to (optionally) specify intent type, e.g., *read*, *write*, or *read+write*. Figure 5.2 illustrates an example: with `Intent({13, 16}, 5, 6, read+write)`, a worker signals that it intends to read and write parameters 13 and 16 while it is at clock 5 (i.e., in batch 5 if the worker advances its clock at the start of each batch). 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

[2] Applications can, of course, choose to synchronize the clocks of different workers.

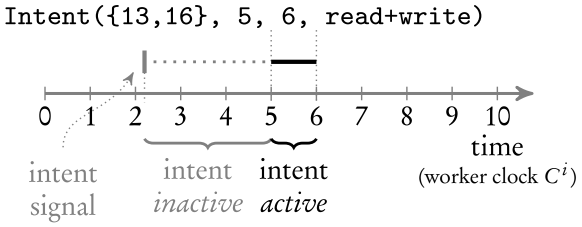


Figure 5.2: Example intent (for parameters 13 and 16).

$C_{\text{end}} \leq C^i$. Invocation of the `Intent` primitive is meant to be cheap, i.e., it should not slow down the worker, even if the worker signals many intents. Workers can flexibly combine intents: they can signal multiple (potentially overlapping) intents for the same parameter, extend one intent by signaling another intent later on, etc.

Algorithm 5.1 illustrates how intent signaling can be used in the distributed SGD example of Section 2.2. The program is similar to the one for a Classic PS (Algorithm 2.3 on page 17), with two additions: (i) the data loader signals intent when it prepares the batch (line 7) and (ii) the worker advances its clock after each batch (line 13).

Intent signals allow for precise adaptation, more precise than existing approaches. For example, a replication PS could use the signals to set up a replica exactly while intent is active. In contrast to existing approaches, it would not need to rely on heuristics to decide how long to maintain the replica. Additionally, as intents are signaled before the actual access, the PS could set up the replica proactively, before the worker accesses the parameter. Intent signaling also enables more adaptive approaches. For example, as we will describe in Section 5.3, intent signals allow AdaPS to choose suitable management techniques dynamically during run time (rather than statically using one technique per parameter, as existing approaches do), and to time actions appropriately (opposed to applications explicitly triggering relocations, as done in Lapse and NuPS).

5.3 The AdaPS Parameter Server

Intent signaling opens a large design space for adaptive PSs. Key design questions include: how to change parameter allocation, when and where to maintain replicas, 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 among nodes efficiently. We explore this design space and describe AdaPS. AdaPS is a PS that requires no input beyond intent signals and no knob tuning. It automatically decides how to act on intent signals

Algorithm 5.1: Distributed asynchronous SGD with intent signaling. Differences to the Classic PS implementation (Algorithm 2.3 on page 17) are highlighted in green.

```
Data: $\mathscr{D}$: training dataset,
      num_epochs: number of epochs to run,
      batch_size: batch size,
      $t$: the ID of this worker thread,
      $T$: the number of total worker threads
$c = 0$ // batch counter
for epoch ← 1 to num_epochs do
    b = num_batches ( $\mathscr{D}$, batch_size, t, T )
    // data loading (pipeline parallel with training, in separate thread(s))
    $\mathscr{B} = []$
    for i ← 1 to b do
        $\mathscr{B}_i$ = prepare_batch ( i, $\mathscr{D}$, batch_size, epoch, t, T )
        intent ( keys($\mathscr{B}_i$), $c$, $c+1$ )
        $c = c + 1$
    // training
    for i ← 1 to b do
        $w$ = pull ( keys($\mathscr{B}_i$) )
        $\Delta w$ = compute_update ( $\mathscr{B}_i$, $w$ )
        push ( keys($\mathscr{B}_i$), $\Delta w$ )
        advanceClock ()
```

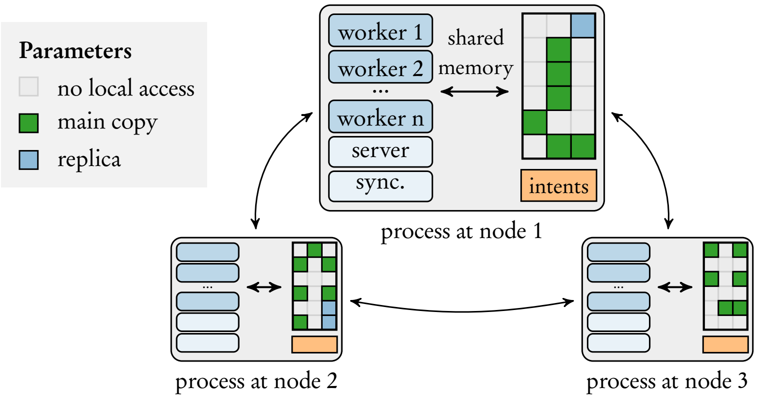


Figure 5.3: AdaPS architecture. For efficiency, AdaPS runs multiple worker threads in one process per node, and accesses locally available parameters via shared memory.

and when to do so. We give a brief overview of key design features before we detail each one in the following subsections. Figure 5.3 illustrates the architecture of AdaPS.

Automatic Choice of Technique

AdaPS employs relocation and replication, and automatically picks between the two. This choice can change over time: AdaPS dynamically chooses the right technique for the current situation. Intuitively, AdaPS 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 AdaPS's choice of technique in Section 5.3.1.

Automatic Action Timing

AdaPS 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 5.3.2 for details.

Responsibility Follows Allocation

In AdaPS, the node that currently holds the main copy of a parameter 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 parameter is relocated. We describe details in Section 5.3.3.

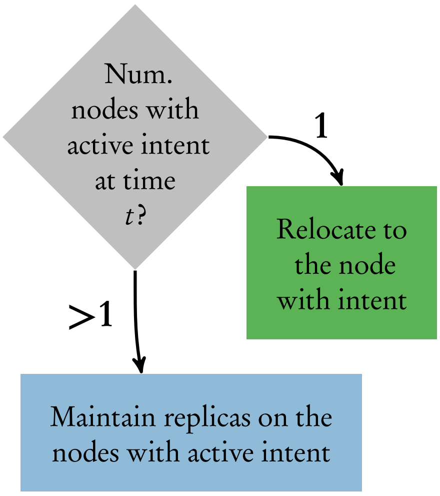

Num. nodes with active intent at time t?
1
Relocate to the node with intent
>1
Maintain replicas on the nodes with active intent

Figure 5.4: AdaPS decides automatically whether to relocate or replicate a parameter at any time t.

Efficient Communication

AdaPS communicates for exchanging intent signals, relocating parameters, and managing replicas. To communicate efficiently, AdaPS locally aggregates intents and sends aggregated intents over the network, groups messages when possible to avoid the overhead of small messages, and employs location caches to improve routing; see Section 5.3.4 for details.

Optional Intent

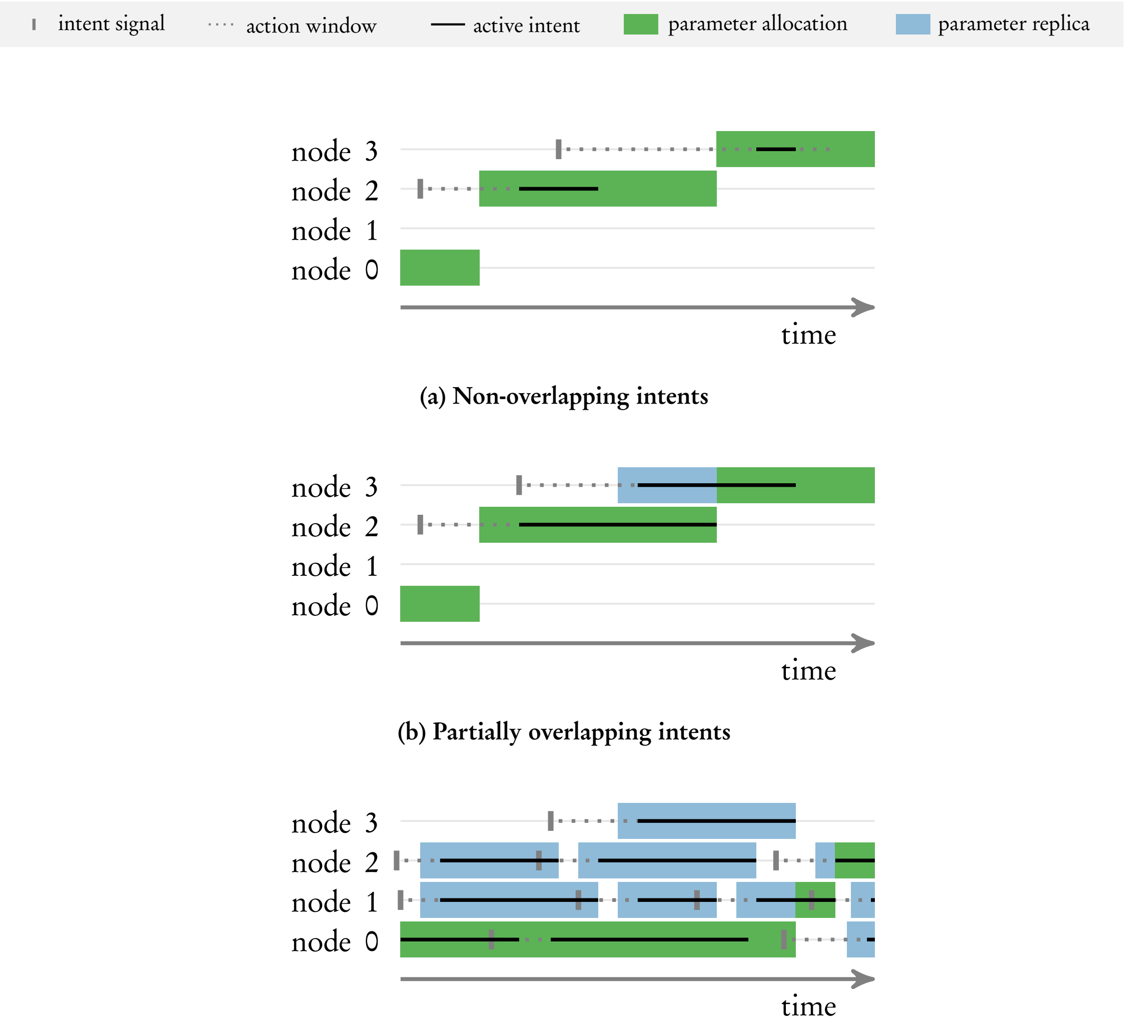
Intent signals are optional in AdaPS. An application can access any parameter at any time and any node, without signaling intent. However, signaling intent potentially makes access more efficient, as it allows AdaPS to avoid synchronous network communication.

5.3.1 Automatic Choice of Technique

AdaPS receives intent signals from workers. Based on these intent signals, AdaPS 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, AdaPS has to work out 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, AdaPS employs (i) relocation and (ii) selective replication. I.e., it (i) can relocate parameters from one node to another, and (ii) can selectively create replicas on subsets of all nodes for specific periods of time.[3] If, at one point in time, only *one* node has active

[3] Selective replication is also used by SSP in Petuum. The main difference is that intent



(a) Non-overlapping intents

(b) Partially overlapping intents

(c) Many concurrent intents

Figure 5.5: Examples for parameter management in AdaPS.

intent for a parameter, AdaPS relocates the parameter to the node with active intent. After the node's intent expires, AdaPS keeps the parameter where it is until some other node signals intent. In contrast, when multiple nodes have active intent for one parameter at the same time, AdaPS 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. Figure 5.4 illustrates AdaPS's decision between relocation and selective replication.

Let us consider three exemplary intent scenarios to understand how AdaPS manages parameters.

signals allow AdaPS to set up a replica before it is accessed and maintain it precisely while it is needed. And, in contrast to AdaPS, Petuum employs only replication.

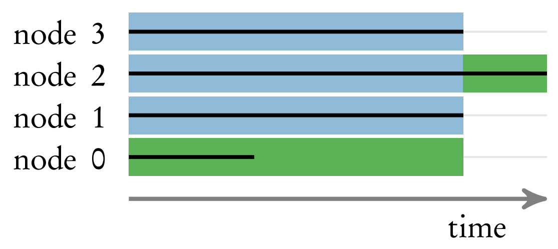
1. Two nodes have intent for the same parameter, and the active phases of the intents do not overlap; see Figure 5.5a. AdaPS relocates the parameter from its initial allocation (node 0) to the first node with intent (node 2). AdaPS keeps the parameter at this node even after the intent expires. Before the intent of the second node becomes active, AdaPS relocates the parameter to the second node with intent (node 3).

2. Two nodes have intent for the same parameter, and the active periods of the intents partially overlap; see Figure 5.5b. AdaPS relocates the parameter to the first node with intent, then creates a replica on the second node while the two active intents overlap, and relocates the parameter to the second node after the intent of the first node expires.

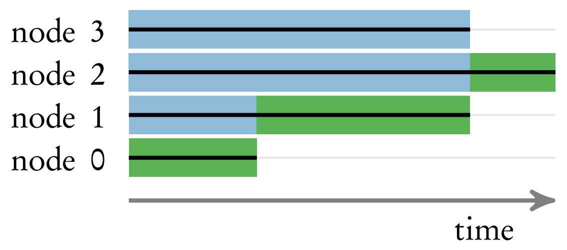
3. Multiple nodes repeatedly have intent for the same parameter, see Figure 5.5c. AdaPS creates replicas on all nodes with active intent. Whenever there is exactly one node with active intent (and the parameter is not currently allocated at this node), AdaPS relocates the parameter to this node.

AdaPS combines parameter relocation and replication because they complement each other well, as described in Section 4.2.2: relocation is efficient for parameters that are accessed rarely, 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 contrast to previous approaches combining these two (e.g., NuPS or BiPS (Q. Zheng et al. 2021)), AdaPS employs *selective* replication: intent signals allow AdaPS to create a replica precisely for the time during which a replica is needed. This increases efficiency as AdaPS does not need to maintain replicas while they are not needed. Further, in contrast to previous systems, the choice of parameter management techniques for each parameter is dynamic: depending on the intent signals, AdaPS can relocate the parameter at one point in time, and replicate it at another.

For making its decisions, AdaPS 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 AdaPS, 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.



(a) Relocate only when exactly one node has active intent



(b) Relocate immediately when the owner's intent expires

Figure 5.6: AdaPS relocates a parameter only when there is exactly one node with active intent (and the parameter is currently not allocated at this node).

AdaPS relocates parameters only when there is (at one point in time) *exactly one* node with active intent, and this node does currently not hold the parameter. It does not relocate a parameter while multiple nodes have active intent, even if the intent of the current owner expires, see Figure 5.6. AdaPS employs this approach because relocating in the presence of replicas would require (i) to update routing information on each replica holder (see Section 5.3.4) and (ii) to transfer intent information from the current owner to another node (see Section 5.3.3).

5.3.2 Automatic Action Timing

AdaPS 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. AdaPS needs to work out 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 5.5b: AdaPS 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 AdaPS acts too late, relocation or replica setup is not finished in time, such that the parameter is not available locally and, instead, has to be accessed remotely, slowing down training. On the other hand, if AdaPS acts too early, it might maintain a replica longer than needed. For example, imagine AdaPS would set up the replica on node 3 in Figure 5.5b immediately after the intent is signaled. This would cause AdaPS to over-communicate, as it would send updates to a replica that are never read. Furthermore, if AdaPS acts too early, it might use replication in scenarios in which—with better timing—relocation would have been both possible and more efficient. For example, consider the scenario in Figure 5.5a: if AdaPS acts on the intent of node 3 immediately after the intent signal, it would need to maintain a replica on node 3 while the intent of node 2 is still active. I.e., in that period, it would send any update of node 2 to node 3 unnecessarily.

However, acting on an intent signal (slightly) too early is much cheaper than acting too late. The reason for this is that acting too late slows down training significantly because the worker is forced to access the parameter remotely. In contrast, acting slightly too early merely causes over-communication. Thus, it is desirable to err on the side of acting too early.

The key challenge for AdaPS is that both (i) the *preparation time*, i.e., the time it takes to relocate or set up a replica and (ii) the length of the action window are unknown. The length of the action window is unknown because AdaPS does not know when the worker will reach the start clock of the intent. Both times are affected by many factors, e.g., by the application, the compute hardware, the network hardware, and the utilization of that hardware.

Learning When to Act

AdaPS aims to *learn* when the right time is to act on an intent signal. A general approach would estimate both preparation time and the length of the action window separately. However, AdaPS acts on intent signals in point-to-point communication rounds (see Section 5.3.4) that take a fairly constant amount of time. AdaPS thus simplifies the general approach and directly estimates the number of worker clocks per communication round. This allows AdaPS to decide whether an intent signal should be included in the current round or whether it suffices to include the signal in a later round. The intent can be included in a later round 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 worker clocks only rarely (ideally, never) exceeds this soft upper bound. To this end, we employ a probabilistic approach in AdaPS: we assume that the

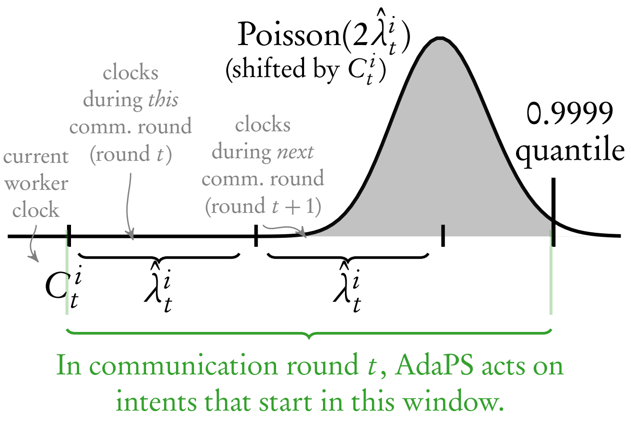

Poisson($2\hat{\lambda}_t^i$)
(shifted by C_t^i)
clocks during *this* comm. round (round t)
current worker clock
clocks during *next* comm. round (round $t+1$)
0.9999 quantile
C_t^i
$\hat{\lambda}_t^i$
$\hat{\lambda}_t^i$
In communication round t, AdaPS acts on intents that start in this window.

Figure 5.7: AdaPS learns automatically when to act on an intent signal. It employs a probabilistic model to estimate a soft upper bound for the number of clocks by a worker.

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 period of time (one round t by one worker i), not one global Poisson distribution (a much stronger, unrealistic assumption, which, for example, would not account for changes in workload or system load).

AdaPS acts on a given intent in round t if it estimates that the corresponding worker might reach the start clock of the intent (I_{start}) before round $t+1$ finishes, i.e., roughly[4] if

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

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 5.7 illustrates this decision. Throughout our experiments, we used the quantile $p = 0.9999$. Under our Poisson assumption, this gives a 99.99% probability that the actual number of clocks by the worker during the two communication rounds is below our estimate.

[4] The exact decision is given in Algorithm 5.2, which follows in Section 5.3.2.

Estimating the Rate Parameter

Naturally, the true Poisson rate λ_t^i is unknown. AdaPS estimates this rate from the number of clocks in past 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 used $\alpha = 0.1$ throughout our experiments).

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 for the number of clocks per round would shrink. To keep the estimate more constant during such periods, AdaPS does not update the estimate when the worker did not raise its clock during the previous communication round (i.e., if $C_t^i - C_{t-1}^i = 0$).

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, AdaPS did *not* act on some intents that the worker reached in this round, such that the worker needed to access the corresponding parameters remotely, which typically slows down the worker drastically. 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. AdaPS further uses a simple heuristic to get out of such regimes 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 5.2 depicts precisely how AdaPS decides whether to act on a given intent and how it updates the estimate.

Effect on Usability

AdaPS's automatic action timing relieves applications from the need to signal intent "at the right time", as applications need to do for triggering relocations in Lapse and NuPS. It is important that applications signal intent early enough so that there is enough time for AdaPS to act on the intent signal. Above this lower limit, however, automatic action timing makes AdaPS insensitive to when intent is signaled. Thus, applications can simply signal intent early, without sacrificing performance, relying on AdaPS to figure out when it should act on the signal.

Algorithm 5.2: Automatic action timing in AdaPS. Should AdaPS act on a given intent in communication round t?

```
Data: Intent start $I_{\text{start}}$, 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 $p$.
$\Delta \leftarrow C^i_t - C^i_{t-1}$
if $\Delta > 0$ then
    $\hat{\lambda}^i_t \leftarrow (1-\alpha)\hat{\lambda}^i_{t-1} + \alpha(\Delta)$
else
    $\hat{\lambda}^i_t \leftarrow \hat{\lambda}^i_{t-1}$
return $I_{\text{start}} < C^i_t + Q_{\text{Poiss}}(2 \cdot \max(\hat{\lambda}^i_t, \Delta), p)$
```

5.3.3 Responsibility Follows Allocation

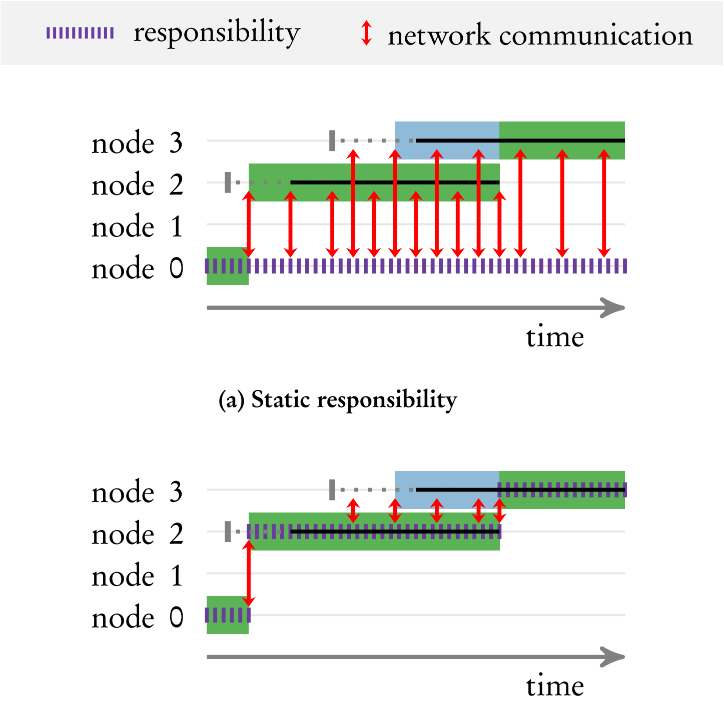
Based on intent signals, AdaPS decides when to relocate a parameter and when to maintain a replica on which node. Two important design decisions to enable this precise management are: (i) which node makes these decisions and (ii) when replicas exist, how to keep them synchronized (efficiently)? A key feature of AdaPS is that the node at which a parameter is currently allocated—the *owner node*[5] of the parameter—takes the main responsibility for both. The owner node decides whether to relocate a parameter and where to maintain replicas (Section 5.3.3); and the owner node acts as a hub for replica synchronization (Section 5.3.3). Placing responsibility at the owner node reduces network overhead and can reduce processing load because the owner node changes whenever the parameter is relocated, such that responsibility is close to where the parameter and the associated processing is.

Choice of Management Technique

AdaPS chooses a technique based on the intent signals of all nodes for a parameter. Thus, the intent signals of all nodes for one parameter need to come together at one node, such that this node can make this decision.

This decision could be made by a node that is *statically* assigned to a parameter (e.g., by hash partitioning). Adapting the terminology that we used in Section 3.2, we refer to such a statically assigned node as the *home node*. In the static approach, nodes continuously send their intent signals for parameter k to the parameter's home node. The home node decides whether to relocate or replicate the parameter, and instructs the current owner of parameter k to act accordingly. The advantage of this static approach is that

[5] We adapt the terminology that we used to describe a similar concept in Lapse in Section 3.2.



(a) Static responsibility

(b) Dynamic responsibility

Figure 5.8: Network communication for placing management responsibility (a) on the statically assigned home node (node 0) or (b) on the dynamically changing owner node.

it is straightforward to route intent signals, as the signals for one key are sent to the same node throughout training. Its disadvantage is that the home node is always involved, even when it does not have intent itself (a common disadvantage of home-based approaches in distributed systems (Steen and Tanenbaum 2017)). Figure 5.8a illustrates the communication for the static approach. Even while there is intent only at node 2 and the parameter is allocated at node 2, node 2 needs to communicate its intent to the home node (node 0), such that the home node can decide to keep the parameter allocated at node 2. While there is intent at nodes 2 and 3, both nodes need to communicate their intent to node 0.

To overcome this problem, AdaPS makes these decisions on the owner node of the parameter, i.e., the node where the parameter is currently allocated. This node changes whenever the parameter is relocated. The key advantage of this approach is that the home node does not need to be involved, reducing network traffic and processing load. Figure 5.8b illustrates the communication for this dynamic approach. After an initial communication of intent (and a subsequent relocation), no further communication between node 2 and node 0 is necessary, as node 2 makes any decisions about the parameter locally. While there is intent on node 2 and node 3, node 3 communicates its intent directly to node 2, without involving node 0. A disadvantage of the dynamic approach is that routing becomes more complex, as the owner node changes throughout training. To overcome this disadvantage, AdaPS employs location caches, which enables nodes to send their intent signals to the current owner node directly, most of the time, see Section 5.3.4.

Replica Synchronization

While a parameter is replicated, multiple nodes hold a copy of the value of the parameter and write updates to this local copy. For convergence, it is crucial that these copies are synchronized, i.e., that updates of one node are propagated to the replicas on other nodes. AdaPS employs relocation and selective replication (as discussed in Section 5.3.1). Consequently, for the majority of parameters, at a given time, only few nodes (if at all) hold a replica for a given parameter. Thus, replica updates have to be propagated only to a small subset of all nodes. Further, this subset is different for each parameter and changes constantly and potentially rapidly. These two properties make all-reduce or gossip-based synchronization approaches unattractive for AdaPS. Instead, AdaPS propagates replica updates via the owner node of a parameter: replica holders send updates to the parameter's owner, which then propagates them to other replica holders.

To use the network efficiently, AdaPS batches replica updates (as, e.g., Petuum (Xing et al. 2015) does). That is, AdaPS does not immediately send replica updates when they are received. Instead, AdaPS slightly delays individual updates so that it can potentially send multiple updates together. As Petuum, AdaPS does so for both directions of update propagation: from replica holders to the owner node and from the owner node to the replica holders. To further improve efficiency, AdaPS versions parameter values and communicates deltas: when nodes request updates for their replicas, they include the version number that is locally available; the owner node sends only updates that the requesting node has not received before.

5.3.4 Efficient Communication

AdaPS communicates to exchange intent signals, to relocate parameters, to set up and destruct replicas, and to synchronize replicas. Key for the overall efficiency of AdaPS is that this communication is efficient. In this section, we discuss several design aspects of AdaPS that improve communication efficiency: AdaPS locally aggregates intent signals and sends aggregated intent signals over the network (Section 5.3.4), groups messages (Section 5.3.4), and employs location caches for more efficient routing (Section 5.3.4).

Aggregated Intent

As discussed in Section 5.3.3, for each parameter, there is one (dynamically changing) node that decides whether to relocate or replicate a specific parameter. To enable this decision, all other nodes need to continuously send their intent signals for this specific parameter to this decision-making node.

A naive approach to intent communication is that all workers eagerly send each intent (i.e., a tuple of parameter, start clock, end clock, intent type,[6] and worker id[7]) to the node that makes decisions immediately after the intent is signaled. Additionally, workers regularly send the state of their clocks (potentially by piggybacking on other messages). The decision-making node stores all intents, determines which of them are active, and decides based on this perfect view of all intents for this specific parameter. However, this naive approach induces significant network overhead, as a large number of intent signals have to be sent over the network constantly. Especially for hot spot parameters, for which there are many intent signals, this can be prohibitive.

[6] AdaPS does not need to communicate intent type, as it does not require to know the intent type for its decisions, see Section 5.3.1.

[7] So that the system knows which worker's clock the start and end clocks refer to, and on which node the intent was signaled.

AdaPS employs a more communication-efficient approach: each node stores inactive intents locally, determines which ones should be treated as active, and sends aggregated information about active intents to the decision-making node. More precisely, each node communicates to the decision-making node when intent becomes active and it communicates when intent expires. The node does not communicate which or how many workers have active intent. This requires significantly less network communication, especially for hot spot parameters. The disadvantage is that the decision-making node has less information. Precisely, it does not know about inactive intents or how long active intent will last. For the decisions that AdaPS makes, this information is not required, such that AdaPS adopts the more communication-efficient approach.

Message Grouping

Automatic action timing ensures that a parameter is relocated or a replica is set up asynchronously, i.e., before a worker accesses the parameter. This allows AdaPS to improve network efficiency by grouping messages for communicating (aggregated) intent signals, for relocating parameters, and for creating, destructing, and synchronizing replicas into one request–response message protocol.

In more detail, to send a synchronization request, a dedicated thread (the *sync.* thread in Figure 5.3) at a node collects (i) a list of parameters for which local workers have active intent and (ii) all updates to local replicas. The node sends these to the owners of the corresponding parameters in a *synchronization request*. Each owner (i) merges the replica updates into its parameter store and (ii) responds to each intent signal (with parameter relocation or replica setup). It does so in one (grouped) *synchronization response*. By default, AdaPS triggers a synchronization request as soon as the last communication round has finished. To reduce the network and CPU load for synchronization, AdaPS allows for limiting the number of communication rounds per second.

Routing

In AdaPS, nodes send intent signals and replica updates to the current owner node. This owner node can change dynamically during run time. To route messages, AdaPS adapts the home node forwarding approach (with location caches) of Lapse. We briefly recap this approach below. See Section 3.2.3 for more details.

As fallback, there is one *home node* for each parameter. This home node is assigned statically to each parameter (by hash partitioning). The home node knows which (other) node is currently the owner of the parameter. If any node does not know where a parameter is currently allocated, it sends its message to the home node, which then forwards the message to the current owner. Whenever a parameter is relocated, the old owner node informs the home node of this relocation. These location updates are piggybacked onto synchronization messages.

To increase efficiency, AdaPS additionally employs location caches. I.e., each node locally stores the last known location for parameters that it accessed in the past. This allows for sending updates and intent signals directly to the current owner. AdaPS uses synchronization responses, outgoing parameter relocations, and responses to remote parameter accesses to update location caches. It does not explicitly invalidate location caches. Instead, it tolerates that messages can be routed based on stale ownership information and relies on the receiving nodes to forward the messages to the current owner (via the home node, see Section 3.2.3 for a more detailed discussion). Location caches are more important in AdaPS than in Lapse as there are scenarios in AdaPS in which nodes repeatedly send messages to the owner node. In particular for hot spot parameters, the owner node changes rarely (see, e.g., Figure 5.5c), such that nodes send their signals and updates to the same owner node repeatedly.

5.4 Experiments

We conducted an experimental study to investigate whether and to what extent fully adaptive parameter management is beneficial for the performance of distributed ML. The source code, datasets, and information on reproducing our experiments are available online.[8]

In this study, we evaluated the performance of AdaPS and compared it to the performance of state-of-the-art PSs (Section 5.4.2). Further, we evaluated its scalability (Section 5.4.3), the efficiency of different management techniques (Section 5.4.4), whether action timing is crucial for the performance of AdaPS (Section 5.4.5), and which decisions AdaPS makes when applied to real-world ML tasks (Section 5.4.6). Our major insights are: (i) AdaPS was efficient without any tuning, (ii) AdaPS even outperformed state-of-the-art PSs on multiple tasks, (iii) AdaPS was more scalable than state-of-the-art PSs, and (iv) automatic action timing made AdaPS efficient for early intent signals. We conclude that *PSs can be efficient and easy to use*.

[8] `https://github.com/alexrenz/AdaPS/tree/review`

Table 5.2: ML tasks, models, and datasets.

Task	Model parameters				Data		
	Model	Keys	Values	Size	Dataset	Data points	Size
Knowledge graph embeddings	ComplEx, dim. 500	4.8 M	4.8 B	35.9 GB	Wikidata5M	21 M	317 MB
Word embeddings	Word2Vec, dim. 1 000	1.9 M	1.9 B	14.0 GB	1b word benchm.	375 M	3 GB
Matrix factorization	Latent factors, rank 1 000	11.0 M	11 B	163.9 GB	10m × 1m zipf 1.1	1 000 M	31 GB

5.4.1 Experimental Setup

Tasks

We used the same three knowledge graph embeddings (KGE), word embeddings (WE), and matrix factorization (MF) tasks as in our previous study in Section 4.4 (see the task descriptions on pages 83 to 84). The only difference is that we used AdaGrad (Duchi et al. 2011) consistently through all three tasks (rather than plain SGD for WE and MF). The tasks differ in multiple ways, including the size of the models, the size of the dataset, with what rate workers advance their clocks, and in their access patterns. Table 5.2 provides a summary.

Baselines

We compared to a classic PS, to NuPS, and to a single node implementation. As classic PS, we used AdaPS without intent signals, which provided performance similar to PS-Lite (Mu Li et al. 2014a). We used a shared memory implementation with 32 worker threads as the single node baseline. To achieve good performance, NuPS required tuning for its main performance hyperparameters: (i) choosing a management technique for each parameter and (ii) specifying a relocation offset (i.e., how many steps ahead of time to relocate a parameter).

To ensure a fair comparison, we ran six different configurations of NuPS. Five of these six configurations are designed to simulate a typical hyperparameter search by an application developer: they represent a random search that is loosely informed by the NuPS heuristic and intuition. In detail, we generated five configurations quasi-randomly using the Sobol sequence implementation of Ax (Bakshy et al. 2018). For choosing techniques, we narrowed the search range using the NuPS heuristic presented in Section 4.4.1 (based on pre-computed dataset frequency statistics) for each task. We generated a set of configurations that replicate 0.01x–100x as many parameters (because in the experiments in Section 4.4.6, up to 64x deviation from the heuristic were beneficial). For the relocation offset, we set the search space to 1–1000 as we found that offsets of up to 512 can be beneficial (see Section 5.4.5). We use the same set of configurations for the three tasks. We provide more details on the search and the exact configurations online.

In addition to these five quasi-random configurations, we ran NuPS with the hyperparameters of the *tuned* variant of Section 4.4. Note that these hyperparameter choices are informed by a series of detail experiments (see Sections 4.4.5 and 4.4.6). Such detailed insights are not commonly available to application developers. However, note that this tuning has been done for a setting with a lower level of parallelism: in Section 4.4, we used 4x fewer worker threads (8 per node) than in this section (32 per node, see below).

Implementation and Cluster

We implemented AdaPS in C++, using ZeroMQ and Protocol Buffers for communication, based on PS-Lite (Mu Li et al. 2014a), Lapse, and NuPS. We used the same cluster as in Section 4.4, i.e., a cluster of up to 16 Lenovo ThinkSystem SR630 computers, running Ubuntu Linux 20.04, connected with 100 Gbit Infiniband. Each node was equipped with two Intel Xeon Silver 4216 16-core CPUs, 512 GB of main memory, and one 2 TB D3-S4610 Intel SSD. We compiled code with g++ 9.3.0. Unless specified otherwise, we used 8 nodes and 32 worker threads per node. In NuPS and AdaPS, we additionally used 3 ZeroMQ I/O threads per node. In AdaPS, we used 4 communication channels per node; in NuPS, we used 1 communication channel, as NuPS does not support multiple communication channels.

Task Hyperparameters

We tuned the task hyperparameters for each task on a single node and used the best found hyperparameter setting in all systems and variants. For the tasks that use negative sampling (KGE and WE), we used local sampling in NuPS and AdaPS. In the classic PS, we used independent sampling, as local sampling provides poor sampling quality in a classic PS (see Section 4.4.5). In AdaPS, we used arbitrary large values for the intent signal offset (1 000 data points in KGE, 2 000 sentences in WE, and 10 000 data points in MF). If chosen large enough, the offset did not affect AdaPS's performance (see Section 5.4.5 for details).

Measures

We ran all variants with a fixed 4 h time budget. 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.[9] 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 baseline. 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 in quality over time. The effective speedup is calculated from the time that each variant took to reach 90% of the best model quality that we observed in the single node baseline.

Differences to Chapter 4

The main difference between this experimental study and the one in Section 4.4 is that we used a higher degree of parallelism: we used 32 worker threads per node rather than 8. This 4x increase makes localization conflicts more likely.

5.4.2 Overall Performance

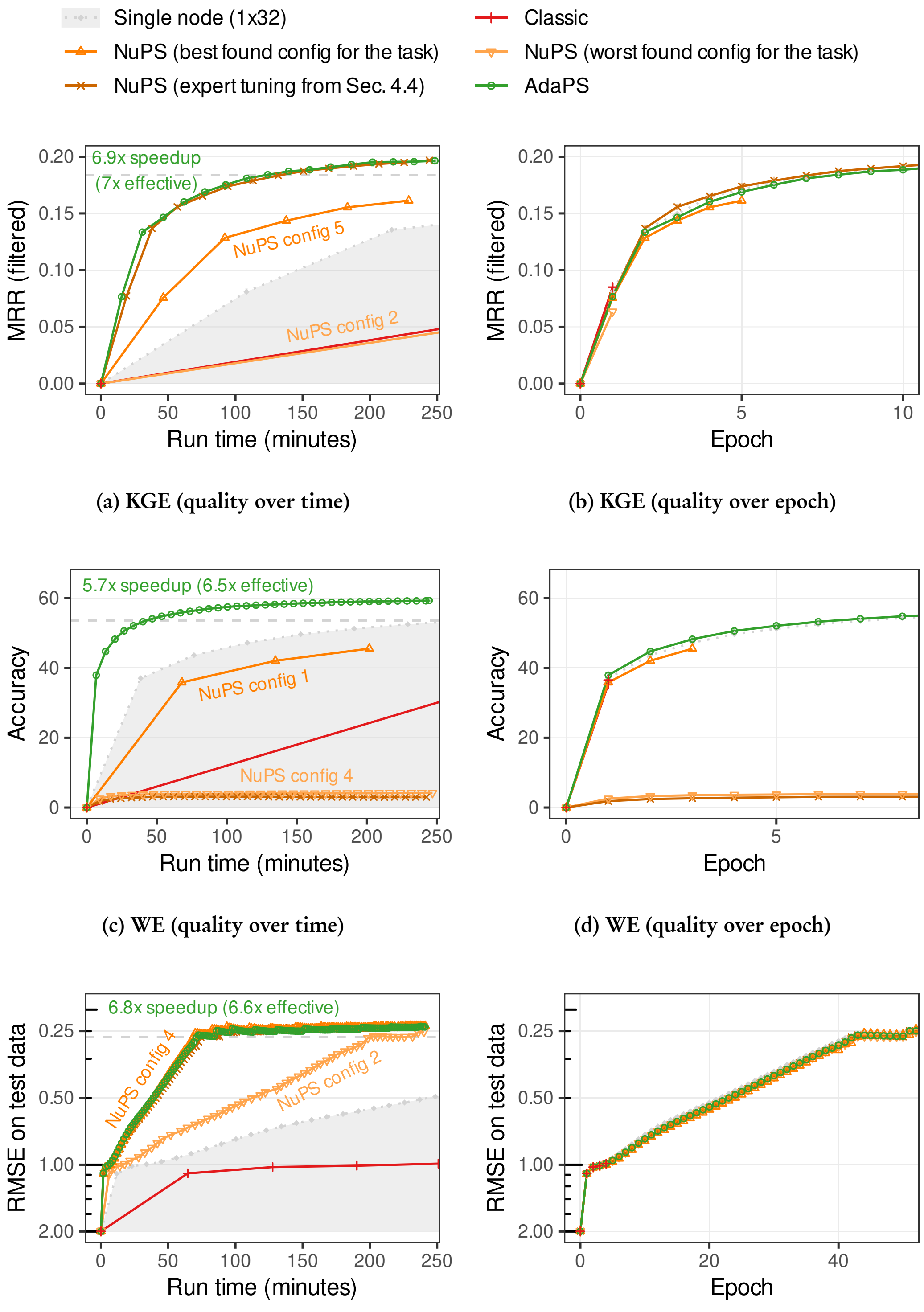
We compared the overall performance of AdaPS to existing PSs (classic and NuPS) and to the single node baseline. We ran each variant for the fixed time budget and measured model quality. Figure 5.9 depicts the results. **In summary, AdaPS matched or even outperformed the state-of-the-art PS NuPS out of the box.**

NuPS achieved good performance for KGE and MF (but not WE), but required tuning to do so. Figure 5.9 shows three of the six NuPS configurations that we ran: (i) the best and worst performing NuPS configurations *per task* from our quasi-random hyperparameter search and (ii) the expertly tuned hyperparameter configuration from Section 4.4. These results make it clear that NuPS requires task-specific tuning: different configurations were efficient for different tasks. For example, quasi-random configuration 4 was the best one for the MF task, but the worst one for the WE task.

In contrast, AdaPS achieved good performance for all three tasks *out of the box*, i.e., without requiring any tuning. AdaPS either matched (MF) or outperformed (slightly in KGE[10] and drastically in WE) the performance of even the tuned configuration of NuPS. AdaPS was able to outperform NuPS because (i) AdaPS did not suffer from relocation conflicts (i.e., remote parameter access caused by multiple nodes concurrently accessing a parameter that is managed by relocation) and (ii) AdaPS maintained replicas only while

[9] For NuPS, we ran each of the 5 configurations once.

[10] With an epoch 20% faster in AdaPS than the tuned configuration of NuPS.



(a) KGE (quality over time)

(b) KGE (quality over epoch)

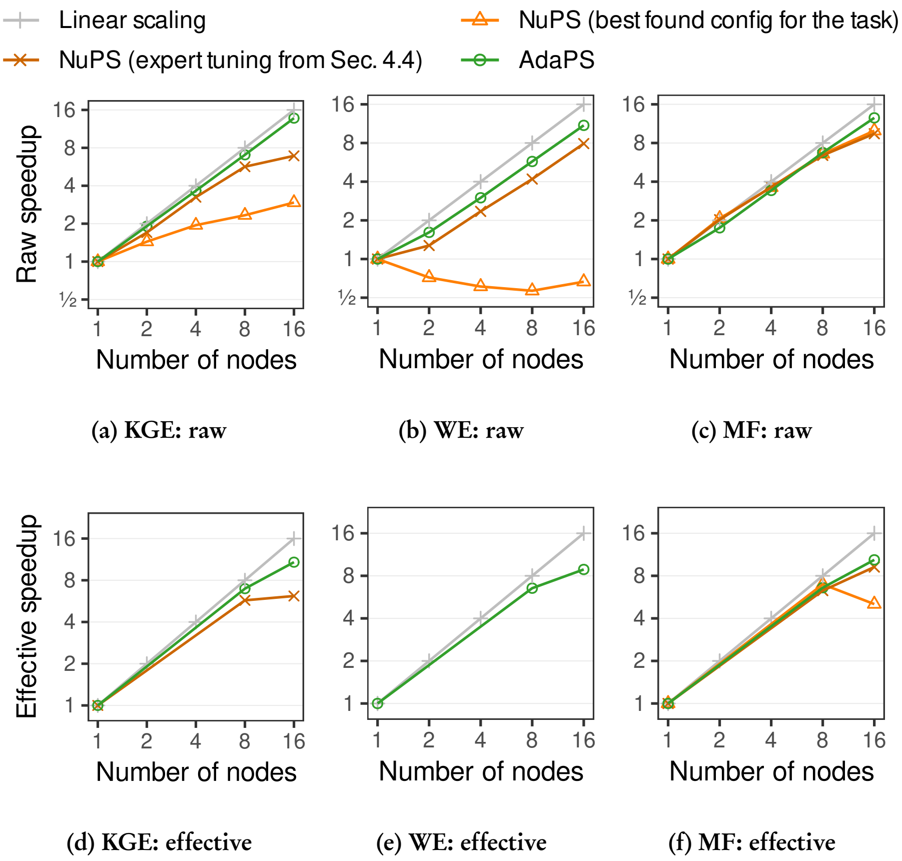
(c) WE (quality over time)

(d) WE (quality over epoch)

(e) MF (quality over time)

(f) MF (quality over epoch)

Figure 5.9: Performance of AdaPS and existing PSs on 8 nodes (32 threads per node). AdaPS matched or even outperformed (tuned) NuPS out of the box and provided good speedups over the single node baseline.



(a) KGE: raw

(b) WE: raw

(c) MF: raw

(d) KGE: effective

(e) WE: effective

(f) MF: effective

Figure 5.10: Strong scaling (logarithmic axes). Raw speedup (a-c), i.e., with respect to epoch run time, and effective speedup (d-f), i.e., with respect to 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.

they were needed, allowing it to synchronize the fewer maintained replicas more frequently than NuPS. Section 5.4.6 provides detailed insights into how AdaPS manages parameters and how that differs from NuPS.

The classic PS was inefficient because it requires synchronous network communication for the majority of parameter accesses.

5.4.3 Scalability

We investigated the scalability of AdaPS and compared it to the best found and the expertly tuned NuPS configurations. We ran on 2–16 nodes and measured raw and effective speedups over the shared-memory single node baseline. Figure 5.10 depicts the results. **AdaPS scaled more efficiently than NuPS, achieving near linear raw and good effective speedups.**

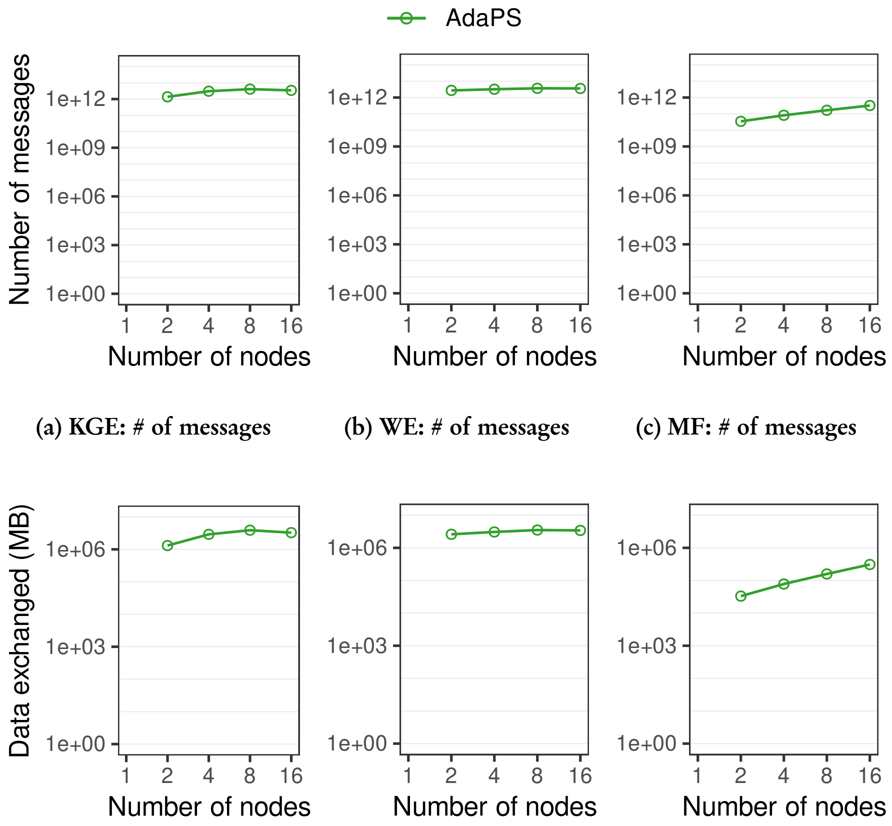
AdaPS scaled more efficiently than NuPS because increasing the number of nodes increased the number of relocation conflicts in NuPS. This was reflected in the share of remote accesses. In KGE, the share of remote accesses in the best quasi-random NuPS configuration was 1.2%, 2.4%, 3.4%, and 5.3% on 2, 4, 8, and 16 nodes, respectively; in WE, it was 2.4%, 5.4%, 9.9%, and 15.8%, respectively. NuPS scaled much better for MF because the task's locality resulted in a small number of relocation conflicts: 0.3%, 0.3%, 0.4%, and 1.6%, respectively. In contrast, there were almost no remote parameter accesses (e.g., <0.0001% for KGE) in AdaPS because AdaPS dynamically created replicas when multiple nodes concurrently accessed the same parameter.

The measured effective speedups slightly dropped on 16 nodes because we tuned task hyperparameters (learning rate, regularization, etc.) for the single node and—to minimize the impact of hyperparameter tuning in our experiments—used the best single-node hyperparameter settings throughout all experiments. However, these hyperparameter settings were not optimal for the 16-node setting (as 16x more worker threads are used). With other hyperparameters, we observed better effective scalability.

Communication Overhead

In addition, to gain further insights into scalability, we measured the number of messages and the amount of data that AdaPS exchanges in these experiments. Figure 5.11 depicts the measurements. It depicts (i) the total number of messages sent among all nodes during one epoch and (ii) the total amount of data exchanged among all nodes during one epoch. Note that an epoch on a smaller cluster took longer than on a larger cluster: roughly, an epoch on 16 nodes took 8x longer than an epoch on 2 nodes, see Figure 5.10. Also note that these measurements should be viewed with a degree of caution, as AdaPS communicates as frequently as possible. For example, a larger number of replicas can reduce the synchronization frequency, so that a small number of frequently synchronized replicas can use the same network bandwidth as a larger number of replicas that are synchronized less frequently.

A few observations about these measurements stand out. First, both the number of messages and the amount of exchanged data increased roughly linearly (MF) or sub-linearly (KGE) or remained roughly constant (WE) with increasing cluster size, but never increased super-linearly. Second, there was no consistent pattern across the tasks: the relationship between communication and cluster size was different for the different tasks. Third, the number of messages and the amount of exchanged data seemed to be connected: they increased and decreased together.



(a) KGE: # of messages (b) WE: # of messages (c) MF: # of messages

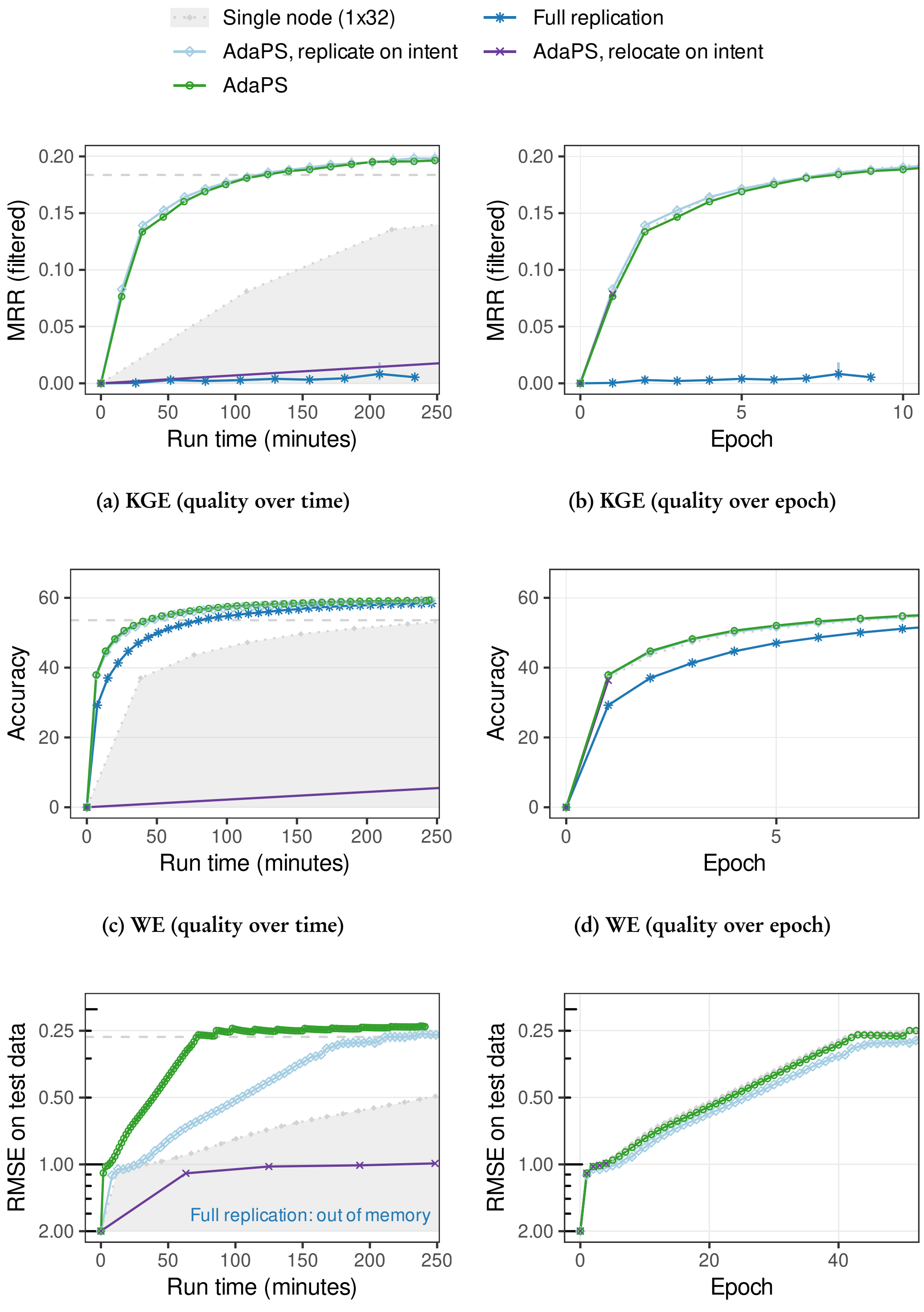
(d) KGE: data exchanged (e) WE: data exchanged (f) MF: data exchanged

Figure 5.11: Network communication during one epoch: total amount of data exchanged and total number of messages sent among all nodes.

We leave a detailed analysis of the communication overhead scaling for future work and merely speculate on a few key factors here. We speculate that a key cause for the differences among the tasks is how AdaPS manages each task. In Section 5.4.6, we will see that AdaPS heavily employs replication for WE, less, but still a significant amount of replication for KGE, and almost no replication for MF. We expect the following effects on communication overhead. First, replication combined with linear scaling and constant synchronization frequency leads to communication constant in the number of nodes. In more detail: imagine a hotspot parameter for which there is a replica on all nodes at all times. Whether AdaPS synchronizes 2 replicas for t minutes or $2 \cdot 4$ replicas for $t/4$ minutes results in roughly the same communication overhead (assuming a constant time-based synchronization frequency). Second, for a parameter that is managed by relocation, communication increases with increasing number of nodes, because the chance that the parameter is already allocated at the accessing node decreases with increasing cluster size. For example, in a cluster of 2 nodes, when one of the two nodes signals intent for a parameter, there is a 50% probability that this parameter is already allocated at this node (so that no network communication is required). This probability decreases to 25% with 4 nodes, to 12.5% with 8 nodes, and to 6.25% with 16 nodes. Note that the differences between these steps get smaller the larger the cluster. Third, communication overhead increases when replication is used instead of relocation (see also the following Section 5.4.4). And the probability that AdaPS employs replication increases with an increasing number of nodes, because the number of workers that read and write parameters in parallel increases. Fourth, local sampling (used in KGE and WE) contributes to a communication overhead that is constant in the number of nodes because it draws all samples from the local parameter partition, such that it causes no network communication, regardless of cluster size.

5.4.4 Efficiency of Techniques

The performance of AdaPS is the result of several components. To investigate the contribution of individual management techniques, we ran AdaPS with different techniques. By default, AdaPS employs relocation and replication, and chooses automatically which to employ. We built one ablation variant that restricts AdaPS to replication, i.e., that replicates a parameter whenever there is active intent (*replicate-on-intent*), and one that restricts AdaPS to relocation, i.e., that relocates parameters when there is active intent (*relocate-on-intent*). Additionally, we ran AdaPS with static full replication



(a) KGE (quality over time)

(b) KGE (quality over epoch)

(c) WE (quality over time)

(d) WE (quality over epoch)

(e) MF (quality over time)

(f) MF (quality over epoch)

Figure 5.12: Performance of AdaPS and ablation variants on 8 nodes.

(by signaling intent for all parameters on all nodes throughout training). Figure 5.12 depicts the results. **In summary, relocate-on-intent and full-replication were inefficient. Replicate-on-intent was efficient for some tasks. Combining relocation and replication was most efficient.**

Static Full Replication

Static full replication provided good (but worse than AdaPS) performance for WE because the model in WE is smaller than the ones in KGE and MF. It provided poor model quality in KGE because synchronizing replicas of the entire model on all nodes caused synchronization frequency to drop (as network bandwidth is fixed). It ran out of memory for MF because the model is large (and memory is also required for storing synchronization deltas, training data, and message buffers).

Relocate on Intent

Restricting AdaPS to relocation provided poor performance for all tasks because relocation is inefficient for hot spot parameters, as observed previously in Section 4.4.2.

Replicate on Intent

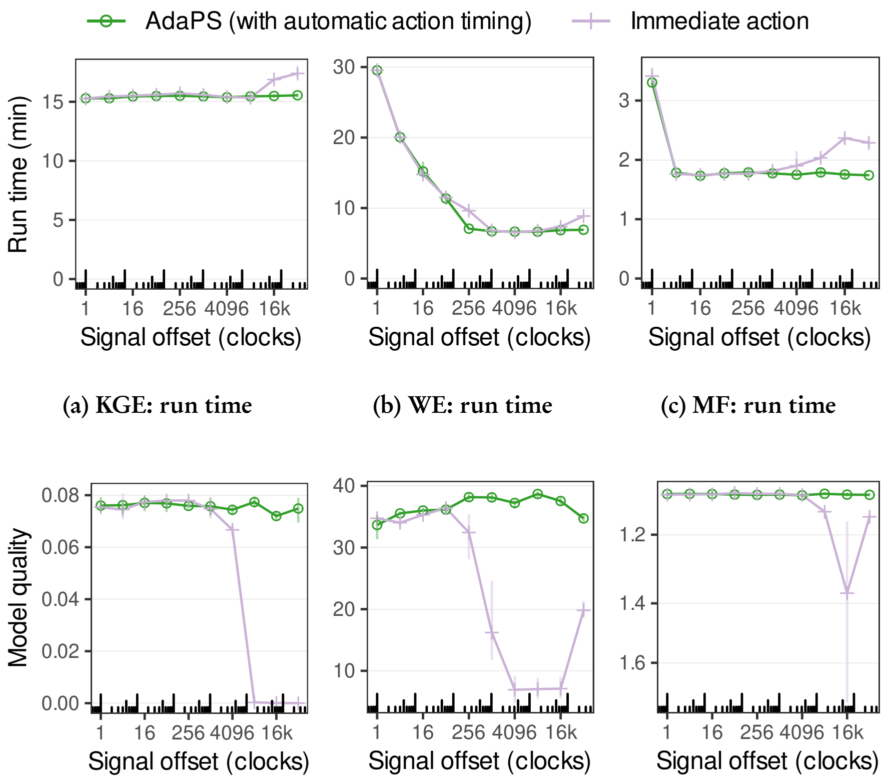
In contrast, restricting AdaPS to replication provided good run times for KGE and WE. For MF, it was 3.0x slower than AdaPS because the MF task exhibits locality (due to row-partitioning, each row parameter is accessed by only one node) and replication is inefficient for managing locality.

AdaPS

Even for tasks without explicit locality (KGE and WE), combining relocation and replication (as AdaPS does) was more communication-efficient than relying exclusively on replication: replicate-on-intent sent 40% more data over the network for one epoch of KGE, and 46% more for one epoch of WE. AdaPS was more communication-efficient because relocation is more efficient than replication if two nodes access a parameter after each other: relocation can send a parameter directly from the first to the second node, whereas replication synchronizes via the allocation node.

5.4.5 Effect of Action Timing

To investigate the effect of action timing, we compared AdaPS to an ablation variant that acts immediately after the intent signal. We ran both variants on



(a) KGE: run time

(b) WE: run time

(c) MF: run time

(d) KGE: quality

(e) WE: quality

(f) MF: quality

Figure 5.13: The effect of automatic action timing on epoch run time (a-c) and model quality after one epoch (d-f). Automatic action timing makes AdaPS efficient for early signals.

workloads with varying signal offsets. Figure 5.13 depicts the results. **With automatic action timing, AdaPS was efficient for any sufficiently large signal offset.**

Early Signals

With automatic action timing, AdaPS 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 (or even collapsed). The reason for this was that the immediate action variant maintained replicas for longer than necessary, and created replicas in situations in which (the more efficient) relocation was possible. Specifically, quality decreased because the number of replicas increased drastically, such that synchronization frequency dropped.

Late Signals

Smaller relocation offsets improved performance for the immediate action variant, but did not further improve performance of AdaPS. For both variants, epoch run time was poor when intent was signaled so late that the PS had insufficient time for setting up replicas or relocating parameters (and the workers had to access the parameters remotely). Thus, with immediate action, there was a task-specific optimum value for the signal offset. In contrast, with automatic action timing, large signal offsets provided good performance across all tasks.

5.4.6 AdaPS in Action

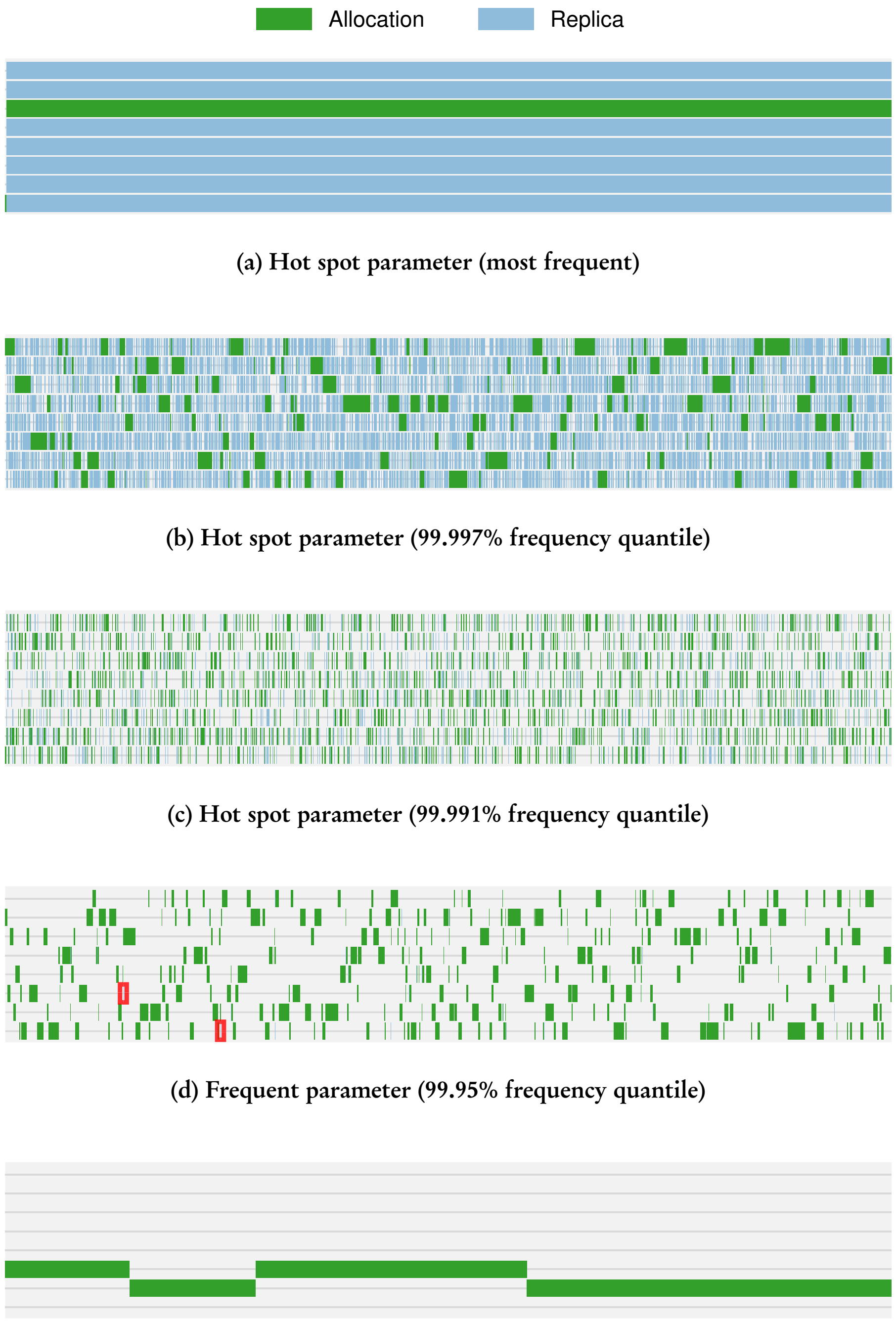
AdaPS decides dynamically and automatically between relocation and replication. To explore how AdaPS actually manages parameters, we traced the parameter management of AdaPS. Figure 5.14 depicts parameter management for selected parameters during the first half of the first epoch of KGE training on 8 nodes. **AdaPS managed extreme hot spots and extreme cold spots the same way a multi-technique PS does, but used more efficient approaches for parameters between the extremes.**

The Extremes

NuPS decides statically how to manage a parameter. The NuPS heuristic would use static full replication for the first three depicted parameters (Figures 5.14a, and 5.14b, and 5.14c) and relocation for the others. AdaPS ended up managing the two extremes—i.e., the extreme hot spot (Figure 5.14a) and the rarely accessed parameter (Figure 5.14e)—as NuPS would.

Between the Extremes

For the parameters between these two extremes, AdaPS took more fine-grained approaches than NuPS. For example, for the parameter in Figure 5.14b, AdaPS maintained replicas exactly while they were needed. Concretely, the gray areas in Figures 5.14b and 5.14c indicate periods in which AdaPS communicates less than NuPS: in contrast to AdaPS, NuPS would maintain replicas during these periods. For the parameter in Figure 5.14d, AdaPS created (short-lived) replicas whenever multiple nodes accessed this parameter concurrently. These short-lived replicas are barely visible in the figure, so we highlighted two of them with red boxes. The short-term replicas prevented workers from having to access the parameter remotely. In contrast, NuPS would manage



(a) Hot spot parameter (most frequent)

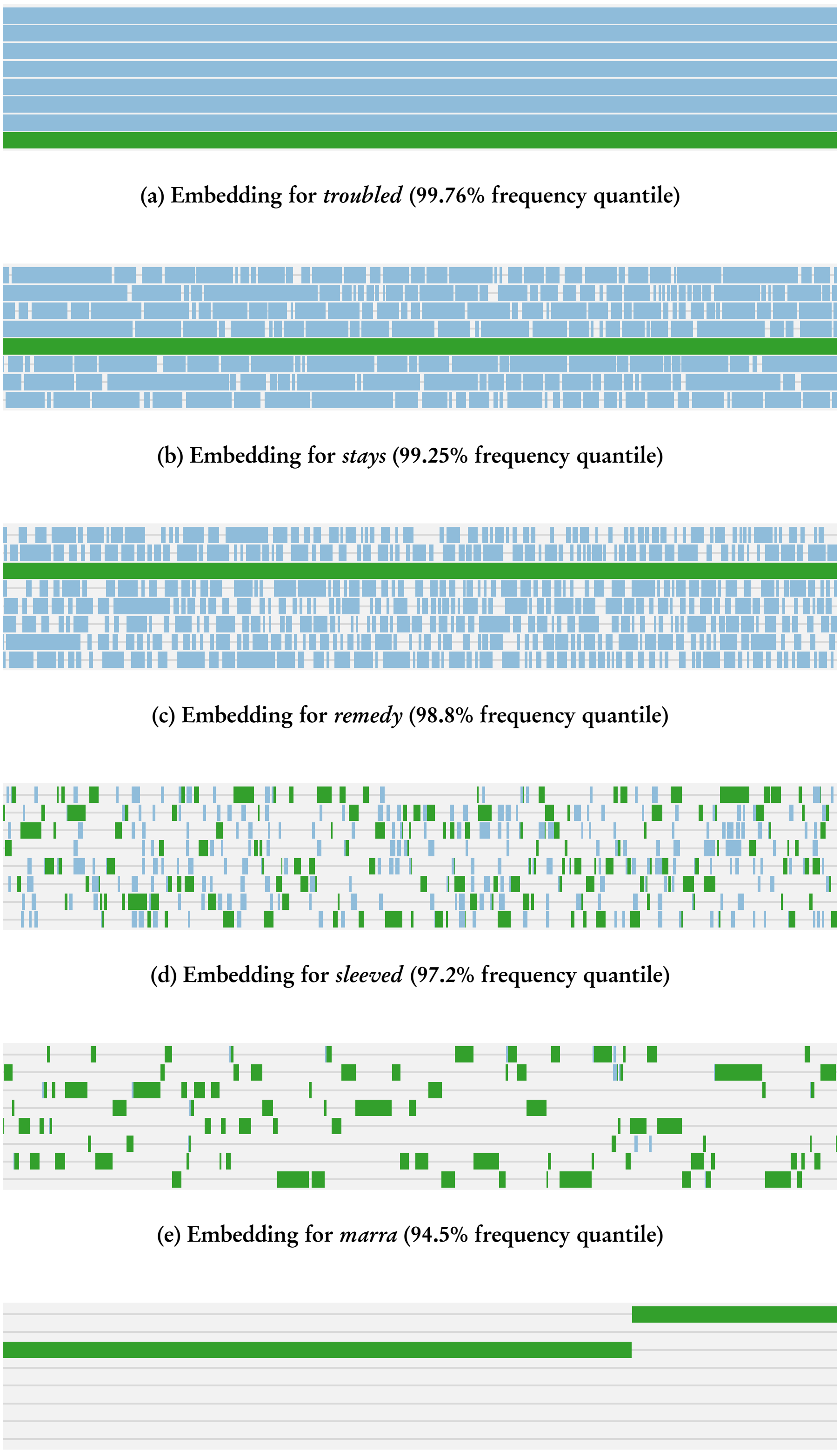
(b) Hot spot parameter (99.997% frequency quantile)

(c) Hot spot parameter (99.991% frequency quantile)

(d) Frequent parameter (99.95% frequency quantile)

(e) Typical parameter (median frequency)

Figure 5.14: Parameter management for selected parameters in the KGE task. Each row corresponds to one of 8 nodes. The red boxes indicate two of the many hardly visible short-lived replicas.



(a) Embedding for *troubled* (99.76% frequency quantile)

(b) Embedding for *stays* (99.25% frequency quantile)

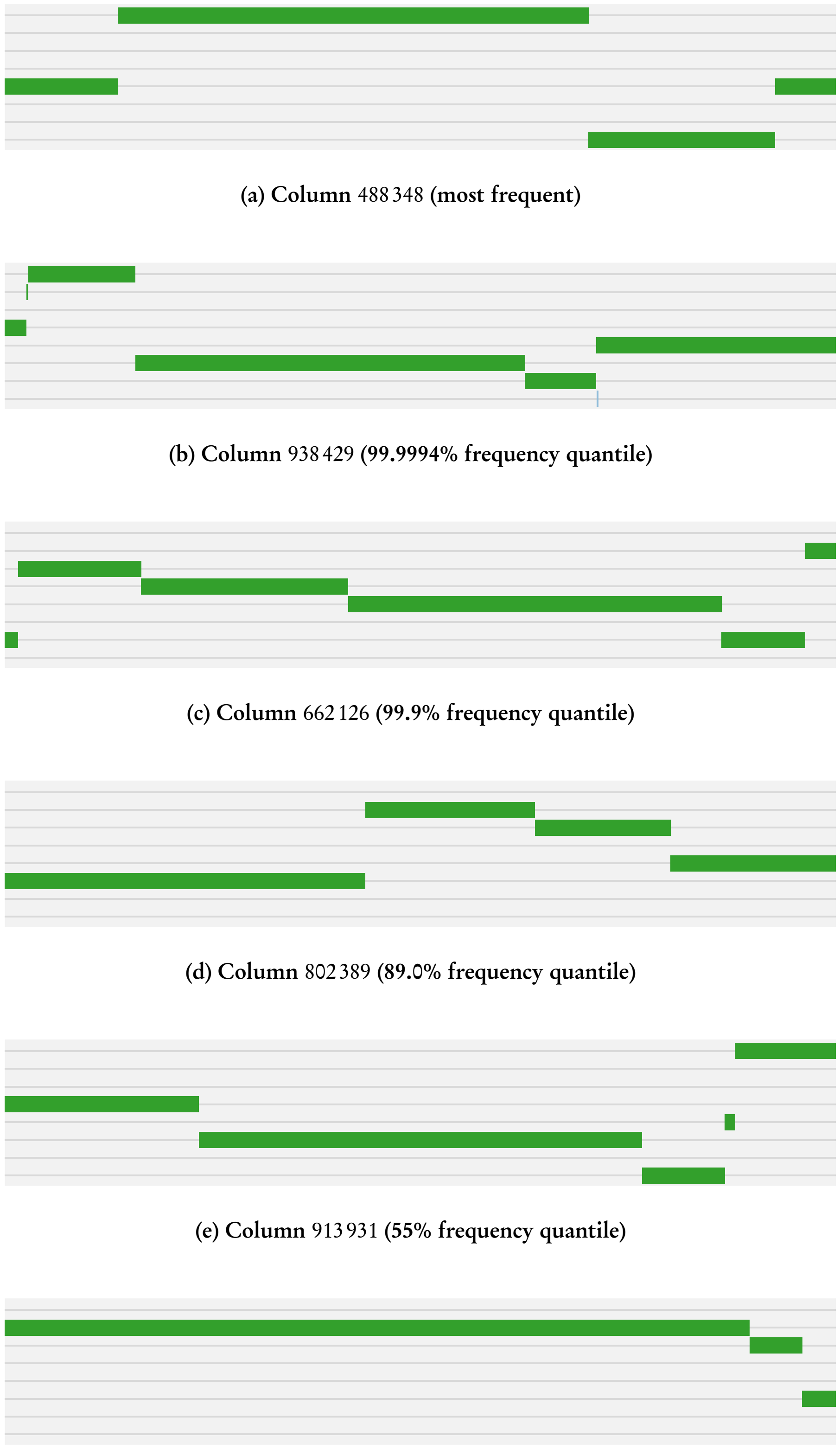
(c) Embedding for *remedy* (98.8% frequency quantile)

(d) Embedding for *sleeved* (97.2% frequency quantile)

(e) Embedding for *marra* (94.5% frequency quantile)

(f) Embedding for *lockwoods* (55% frequency quantile)

Figure 5.15: Parameter management for selected parameters in the WE task.



(a) Column 488 348 (most frequent)

(b) Column 938 429 (99.9994% frequency quantile)

(c) Column 662 126 (99.9% frequency quantile)

(d) Column 802 389 (89.0% frequency quantile)

(e) Column 913 931 (55% frequency quantile)

(f) Column 255 321 (8% frequency quantile)

Figure 5.16: Parameter management for selected parameters in the MF task.

this parameter exclusively by relocation, such that workers are slowed down by remote parameter accesses.

Differences Among ML Tasks

Figures 5.15 and 5.16 depict AdaPS's parameter management for selected parameters of the WE and MF tasks, respectively. Again, the figures depict parameter management during the first half of the first epoch on 8 nodes. In WE, AdaPS employed more replication than in KGE, i.e., it employed replication for a larger share of the parameters: AdaPS ended up fully replicating all WE parameters above the 99.76% frequency quantile. In contrast, in KGE, AdaPS ended up fully replicating only parameters above the 99.999% frequency quantile. Also, AdaPS made heavy use of replication for parameters down to the 98% frequency quantile, e.g., for the embedding of the word *remedy*, see Figure 5.15c. The reason for this is that the access frequency distribution in the WE task is less skewed (i.e., broader) than the one in the KGE task (see Figure 4.2 on page 66). This matches our findings in Section 4.4.6, where we found that it is beneficial for NuPS to replicate a larger share of parameters in the WE task than in the KGE and MF tasks.

In contrast, AdaPS almost never employed replication for MF. The reason for this was the locality in the MF task: the training algorithm visited the data points ordered by their column. This means that the algorithm picks a column and then processes all the data points that belong to this column, before picking another column. Only rarely did multiple nodes access the same column at the same time. Thus, AdaPS employed relocation most of the time and only rarely replication. For example, one of these rare occurrences is that node 5 and node 8 concurrently accessed column 938 429 (Figure 5.16b) for a brief period, which caused AdaPS to maintain a replica on node 8 for a brief period. In general, over an entire epoch (of which only half is visible), each parameter was relocated 8 times (once to each node).[11] At each node, frequent parameters were accessed many times (e.g., around 1.1 million times at each node for the most frequent parameter, column 488 348), and infrequent parameters only a few times (e.g., around 31 times at each node for column 913 931 at the 55% frequency quantile).

[11] Of course, if one training data partition contained no data point of a specific column, then the corresponding parameters were not transferred to this node. Thus, the corresponding parameters were relocated fewer than 8 times in an epoch.

5.5 Summary

In this chapter, we explored whether PSs can adapt to the underlying ML task automatically, without prior tuning. To this end, we proposed intent signaling, a novel mechanism to enable automatic adaptation. And we described AdaPS, a PS that automatically adapts to the underlying workload based solely on information that this mechanism provides. In our experimental study, AdaPS was efficient for multiple ML tasks out of the box, without requiring any tuning, and matched or even outperformed state-of-the-art—more complex to use—PSs.

Chapter 6

Conclusions

In this thesis, we studied the efficiency of PSs for ML tasks with sparse parameter access. We observed that existing PSs are inefficient for such tasks: they barely outperform efficient single node implementations. Starting from this observation, we investigated whether and to what extent PSs can become more efficient by *adapting to the underlying ML task*. Step by step, we made the PS more adaptive. We explored adapting parameter allocation to exploit access locality. We explored adapting management techniques to tailor the PS to the access patterns of individual parameters. And we explored accounting for the type of parameter access to efficiently support sampling access.

These adaptations can make the PS more efficient, but they also made the PS more and more complex to use, as the application needs to control adaptation manually. To reduce usage complexity, we presented an approach that allows for automatic adaptation, without the application's manual control. This approach consists of a mechanism to pass relevant information from the application to the PS, and a PS that adapts automatically based on this information. Our experiments indicate that our work enables efficient distributed training for a range of ML tasks with sparse parameter access.

Such efficient distributed training allows researchers and practitioners (i) to train larger models and (ii) to train models faster. Thus, our contributions towards efficient PSs enable researchers and practitioners to develop better ML-based solutions for challenging problems. And our work contributes to squandering fewer of our planet's precious resources, by using them—in particular, everything that is necessary to produce computers, network hardware, and data center infrastructure, and the energy necessary to operate them—more efficiently.

Future Work

There are several interesting research directions based on the work in this thesis. We list some of these in the following.

Integration into Common ML Systems

ML systems like PyTorch (Paszke et al. 2019) or TensorFlow (Abadi et al. 2016) provide convenient abstractions to define and run ML tasks. An interesting direction for research is the integration of adaptive parameter management into these common ML systems. Key research questions in this direction are (i) whether and how the data loaders of these systems can automatically generate intent signals from model definitions and (ii) how fine-grained parameter management can be realized behind coarse-grained tensor parameter definitions. A further aspect in this direction is whether PSs can better integrate with the optimizer abstractions of these ML systems.

Integration of Hardware Accelerators

The PSs presented in this thesis store model parameters in main memory. However, especially for deeper models, it is common to use hardware accelerators such as GPUs and TPUs for the gradient computation. The local memory available on these accelerators is more limited than main memory, and transferring model parameters to accelerator memory can induce significant latency. An interesting direction for future work is to investigate how adaptive PSs can better integrate such accelerators. Research questions in this direction include (i) how a PS could incorporate accelerator memory (Adnan et al. 2021) and (ii) how a PS can leverage (heterogeneous) communication links among the accelerators (e.g., NVLink across (some) of the GPUs on one node).

Other ML tasks

Throughout this thesis, we evaluated adaptive PSs mostly on shallow embedding models. It would be interesting to apply adaptive PSs to deeper models, such as deep learning recommender models (H.-T. Cheng et al. 2016; Guo et al. 2017; R. Wang et al. 2017; Zhou et al. 2018), graph neural networks (Schlichtkrull et al. 2018; Shang et al. 2019; Vashishth et al. 2020), and natural language processing (Peters et al. 2018; Devlin et al. 2019). In such models, parts of the model are accessed sparsely (typically the first and sometimes the last layer), and other parts are accessed densely (e.g., fully connected hidden layers). Another class of models that is potentially interesting are models with conditional routing (Shazeer et al. 2017; Riquelme et al. 2021; Lepikhin et al. 2021; Gururangan et al. 2022; Fedus et al. 2022; Margaret Li et al. 2022). In such models, parts of the model (and, thus, the corresponding parameters) are activated sparsely. A router component determines which part(s) of the model are accessed for a specific training example. The key question for these models is whether intent signaling can be used (e.g., by executing the routing component during batch preparation) and whether that could make distributed training more efficient.

Other Intent Signaling PSs

AdaPS is one of many conceivable PSs based on intent signaling. An interesting direction for future work is to further explore the design space of intent signaling PSs, either by developing fundamentally different systems or by exploring potential improvements to individual components of AdaPS. Within AdaPS, it would be interesting to explore parameter management that makes different decisions for different combinations of intent types. Further, one could explore what other synchronization and communication protocols are possible and whether they can further improve communication efficiency. Or one could develop and incorporate additional, potentially highly tailored, management techniques or novel sampling schemes.

Other Directions for Improving Efficiency

In this thesis, we explored how a PS can be efficient for a given (i.e., fixed) workload. Another direction is to try to improve efficiency by manipulating the workload. One direction in this field is whether explicitly ordering update steps could improve efficiency, e.g., by arranging update functions in a way that increases access locality. Another interesting direction is to explore how adaptive PSs can be combined with work on compression (see Section 2.3.1).

Bibliography

Abadi, Martín, 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*. OSDI '16. USENIX Association, pp. 265–283 (cit. on pp. 2, 9, 13, 18, 25, 108, 144).

Adnan, Muhammad, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair (2021). "Accelerating Recommendation System Training by Leveraging Popular Choices". In: *PVLDB* 15.1, pp. 127–140 (cit. on pp. 23, 144).

Agarwal, Saurabh, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos (2022). "On the Utility of Gradient Compression in Distributed Training Systems". In: *Proceedings of Machine Learning and Systems*. Vol. 4, pp. 652–672 (cit. on p. 22).

Ahmed, Amr, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander Smola (2012). "Scalable Inference in Latent Variable Models". In: *Proceedings of the 5th ACM International Conference on Web Search and Data Mining*. WSDM '12. Association for Computing Machinery, pp. 123–132 (cit. on pp. 2, 14, 15, 28, 68).

Aji, Alham Fikri and Kenneth Heafield (2017). "Sparse Communication for Distributed Gradient Descent". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. EMNLP '17. Association for Computational Linguistics, pp. 440–445 (cit. on p. 22).

Alistarh, Dan, Demjan Grubic, Jerry Z. Li, Ryota Tomioka, and Milan Vojnovic (2017). "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding". In: *Advances in Neural Information Processing Systems*. NeurIPS '17. Curran Associates, pp. 1707–1718 (cit. on p. 22).

Assran, Mahmoud, Nicolas Loizou, Nicolas Ballas, and Mike Rabbat (2019). "Stochastic Gradient Push for Distributed Deep Learning". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. ICML '19. PMLR, pp. 344–353 (cit. on p. 24).

Auer, Sören, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives (2007). "DBpedia: A nucleus for a web of open data". In: *The Semantic Web*. ISWC '07. Springer, pp. 722–735 (cit. on p. 49).

Awan, Ammar Ahmad, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda (2017). "S-Caffe: Co-Designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '17. Association for Computing Machinery, pp. 193–205 (cit. on p. 23).

Bakshy, Eytan, Lili Dworkin, Brian Karrer, Konstantin Kashin, Benjamin Letham, Ashwin Murthy, and Shaun Singh (2018). "AE: A domain-agnostic platform for adaptive experimentation". In: *Workshop on Systems for ML and Open Source Software at NeurIPS 2018* (cit. on p. 125).

Balazevic, Ivana, 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, pp. 5185–5194 (cit. on pp. 2, 9).

Bamler, Robert and Stephan Mandt (2020). "Extreme Classification via Adversarial Softmax Approximation". In: *Proceedings of the 8th International Conference on Learning Representations*. ICLR '20 (cit. on pp. 64, 67).

Battiti, Roberto (1989). "Accelerated Backpropagation Learning: Two Optimization Methods". In: *Complex systems* 3.4, pp. 331–342 (cit. on p. 89).

Bengio, Yoshua, Réjean Ducharme, and Pascal Vincent (2000). "A Neural Probabilistic Language Model". In: *Advances in Neural Information Processing Systems*. Vol. 13. MIT Press (cit. on pp. 1, 11).

Bernstein, Jeremy, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar (2018). "signSGD: Compressed Optimisation for Non-Convex Problems". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. ICML '2018. PMLR, pp. 560–569 (cit. on p. 22).

Beutel, Alex, Partha Pratim Talukdar, Abhimanu Kumar, Christos Faloutsos, Evangelos Papalexakis, and Eric Xing (2014). "FlexiFaCT: Scalable Flexible Factorization of Coupled Tensors on Hadoop". In: *Proceedings of the 2014 SIAM International Conference on Data Mining*. SDM '14, pp. 109–117 (cit. on pp. 27, 30).

Boehm, Matthias, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede (2020). "SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle". In: *10th Conference on Innovative Data Systems Research*. CIDR' 20 (cit. on p. 25).

Boehm, Matthias, Michael Dusenberry, Deron Eriksson, Alexandre Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda (2016). "SystemML: Declarative Machine Learning on Spark". In: *PVLDB* 9.13, pp. 1425–1436 (cit. on p. 25).

Bordes, Antoine, 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 (cit. on pp. 2, 9, 49).

Bottou, Léon, Frank E. Curtis, and Jorge Nocedal (2016). "Optimization Methods for Large-Scale Machine Learning". In: *CoRR* abs/1606.04838 (cit. on p. 8).

Broscheit, Samuel, Daniel Ruffinelli, Adrian Kochsiek, Patrick Betz, and Rainer Gemulla (2020). "LibKGE - A Knowledge Graph Embedding Library for Reproducible research". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, pp. 165–174 (cit. on pp. 76, 77, 84).

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei (2020). "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. NeurIPS '20. Curran Associates, pp. 1877–1901 (cit. on p. 1).

Carbone, Paris, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas (2015). "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38.4, pp. 28–38 (cit. on p. 25).

Chechik, Gal, Varun Sharma, Uri Shalit, and Samy Bengio (2010). "Large Scale Online Learning of Image Similarity Through Ranking". In: *Journal of Machine Learning Research* 11.36, pp. 1109–1135 (cit. on pp. 64, 67).

Chelba, Ciprian, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn (2013). "One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling". In: *CoRR* abs/1312.3005 (cit. on pp. 50, 84).

Chen, Jianmin, Rajat Monga, Samy Bengio, and Rafal Józefowicz (2016). "Revisiting Distributed Synchronous SGD". In: *CoRR* abs/1604.00981 (cit. on pp. 10, 11).

Chen, Po-Lung, 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*. Vol. 18. Proceedings of Machine Learning Research. PMLR, pp. 21–60 (cit. on pp. 1, 2, 9).

Chen, Rong, Jiaxin Shi, Yanzhe Chen, and Haibo Chen (2015). "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs". In: *Proceedings of the 10th European Conference on Computer Systems*. EuroSys '15. Association for Computing Machinery (cit. on p. 28).

Chen, Tianqi, 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". In: *CoRR* abs/1512.01274 (cit. on pp. 2, 9, 13, 18, 25, 108).

Cheng, Heng-Tze, 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*. DLRS '16. Association for Computing Machinery, pp. 7–10 (cit. on pp. 2, 9, 144).

Cheng, Wenliang, Chengyu Wang, Bing Xiao, Weining Qian, and Aoying Zhou (2016). “On Statistical Characteristics of Real-Life Knowledge Graphs”. In: *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, pp. 37–49 (cit. on pp. 63, 65).

Chilimbi, Trishul, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman (2014). “Project Adam: Building an Efficient and Scalable Deep Learning Training System”. In: *Proceedings of the 11th Conference on Operating Systems Design and Implementation*. OSDI ’14. USENIX Association, pp. 571–582 (cit. on pp. 2, 14).

Ciciani, Bruno, Daniel Dias, and Philip Yu (1990). “Analysis of Replication in Distributed Database Systems”. In: *IEEE Transactions on Knowledge & Data Engineering* 2.02, pp. 247–261 (cit. on p. 70).

Clauset, Aaron, Cosma Rohilla Shalizi, and M. E. J. Newman (2009). “Power-Law Distributions in Empirical Data”. In: *SIAM Review* 51.4, pp. 661–703 (cit. on pp. 63, 65).

Cui, Henggang, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory Ganger, Phillip Gibbons, Garth Gibson, and Eric Xing (2014). “Exploiting Iterative-ness for Parallel ML Computations”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC ’14. Association for Computing Machinery (cit. on pp. 18, 31, 69).

Dai, Wei, 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. AAAI Press, pp. 79–87 (cit. on pp. 2, 16, 18–21, 25, 31, 43, 58, 64, 69, 70, 77, 101).

Das, Abhinandan S., Mayur Datar, Ashutosh Garg, and Shyam Rajaram (2007). “Google News Personalization: Scalable Online Collaborative Filtering”. In: *Proceedings of the 16th International Conference on World Wide Web*. WWW ’07. Association for Computing Machinery, pp. 271–280 (cit. on p. 1).

Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc Le, Mark Mao, Marc’Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Ng (2012). “Large Scale Distributed Deep Networks”. In: *Advances in Neural Information Processing Systems*. NeurIPS ’12. Curran Associates, pp. 1223–1231 (cit. on pp. 11, 25).

Demers, Alan, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry (1987). “Epidemic Algorithms for Replicated Database Maintenance”. In: *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*.

PODC '87. Association for Computing Machinery, pp. 1–12 (cit. on pp. 24, 46).

Devlin, Jacob, 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, pp. 4171–4186 (cit. on pp. 1, 9, 10, 66, 144).

Dowdy, Lawrence W. and Derrell V. Foster (1982). "Comparative Models of the File Assignment Problem". In: *ACM Computing Surveys* 14.2, pp. 287–313 (cit. on p. 70).

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12, pp. 2121–2159 (cit. on pp. 49, 83, 125).

El Abbadi, Amr (1991). "Adaptive protocols for managing replicated distributed databases". In: *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pp. 36–43 (cit. on p. 70).

Faloutsos, Michalis, Petros Faloutsos, and Christos Faloutsos (1999). "On Power-Law Relationships of the Internet Topology". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '99. Association for Computing Machinery, pp. 251–262 (cit. on pp. 63, 65).

Fan, Shiqing, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin (2021). "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Association for Computing Machinery, pp. 431–445 (cit. on p. 23).

Fedus, William, Barret Zoph, and Noam Shazeer (2022). "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *Journal of Machine Learning Research* 23.120, pp. 1–39 (cit. on p. 145).

Gemulla, Rainer, Erik Nijkamp, Peter Haas, and Yannis Sismanis (2011). "Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent". In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '11. Association for Computing Machinery, pp. 69–77 (cit. on pp. 27, 29, 30, 46, 48, 87).

Ghoting, Amol, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan (2011). “SystemML: Declarative Machine Learning on MapReduce”. In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. ICDE ’11. IEEE Computer Society, pp. 231–242 (cit. on p. 25).

Gonzalez, Joseph, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin (2012). “PowerGraph: Distributed Graph-parallel Computation on Natural Graphs”. In: *Proceedings of the 10th Conference on Operating Systems Design and Implementation*. OSDI ’12. USENIX Association, pp. 17–30 (cit. on pp. 27, 28, 63, 65).

Goyal, Priya, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He (2017). “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *CoRR* abs/1706.02677 (cit. on p. 22).

Grover, Aditya and Jure Leskovec (2016). “Node2vec: Scalable Feature Learning for Networks”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. Association for Computing Machinery, pp. 855–864 (cit. on pp. 64, 65, 67).

Guo, Huifeng, 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*. IJCAI’17. AAAI Press, pp. 1725–1731 (cit. on pp. 2, 9, 144).

Gururangan, Suchin, Mike Lewis, Ari Holtzman, Noah A. Smith, and Luke Zettlemoyer (2022). “DEMix Layers: Disentangling Domains for Modular Language Modeling”. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL ’22. Association for Computational Linguistics, pp. 5557–5576 (cit. on p. 145).

Ho, Qirong, 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*. NeurIPS ’13. Curran Associates, pp. 1223–1231 (cit. on pp. 2, 14, 16, 18, 19, 25, 43, 47, 58, 64, 69, 73, 77, 101, 103).

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8, pp. 1735–1780 (cit. on p. 1).

Howard, Jeremy and Sebastian Ruder (2018). “Universal Language Model Fine-tuning for Text Classification”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. ACL ’18. Association for Computational Linguistics, pp. 328–339 (cit. on pp. 2, 9, 10, 66).

Hu, Yifan, Yehuda Koren, and Chris Volinsky (2008). "Collaborative Filtering for Implicit Feedback Datasets". In: *8th IEEE International Conference on Data Mining*. ICDM '08. IEEE Computer Society, pp. 263–272 (cit. on pp. 2, 9).

Huang, Yanping, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen (2019). "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems*. Vol. 32. NeurIPS '19. Curran Associates (cit. on p. 23).

Huang, Yuzhen, 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". In: *PVLDB* 11.5, pp. 566–579 (cit. on pp. 2, 14, 18, 25, 29, 60, 69, 77).

Huang, Yuzhen, Xiaohan Wei, Xing Wang, Jiyan Yang, Bor-Yiing Su, Shivam Bharuka, Dhruv Choudhary, Zewei Jiang, Hai Zheng, and Jack Langman (2021). "Hierarchical Training: Scaling Deep Recommendation Models on Large CPU Clusters". In: *Proceedings of the 27th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '21. Association for Computing Machinery, pp. 3050–3058 (cit. on p. 23).

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio (2017). "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *Journal of Machine Learning Research* 18.1, pp. 6869–6898 (cit. on p. 22).

Huo, Zhouyuan, Bin Gu, and Heng Huang (2021). "Large Batch Optimization for Deep Learning Using New Complete Layer-Wise Adaptive Rate Scaling". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI '21 35.9, pp. 7883–7890 (cit. on p. 23).

Hutto, Phillip and Mustaque Ahamad (1990). "Slow memory: weakening consistency to enhance concurrency in distributed shared memories". In: *Proceedings of the 10th International Conference on Distributed Computing Systems*. ICDCS '90, pp. 302–309 (cit. on p. 43).

Jagerman, Rolf, Carsten Eickhoff, and Maarten de Rijke (2017). "Computing Web-scale Topic Models Using an Asynchronous Parameter Server". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '17. Association for Computing Machinery, pp. 1337–1340 (cit. on pp. 2, 14, 25, 29).

Ji, S., N. Satish, S. Li, and P. K. Dubey (2019). "Parallelizing Word2Vec in Shared and Distributed Memory". In: *IEEE Transactions on Parallel and Distributed Systems* 30.9, pp. 2090–2100 (cit. on pp. 64, 73, 76, 77).

Jiang, Jiawei, Bin Cui, Ce Zhang, and Lele Yu (2017). "Heterogeneity-Aware Distributed Parameter Servers". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Association for Computing Machinery, pp. 463–478 (cit. on pp. 2, 14, 18, 24, 25, 29, 69).

Jiang, Yimin, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo (2020). "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *Proceedings of the 14th Conference on Operating Systems Design and Implementation*. OSDI '20. USENIX Association, pp. 463–479 (cit. on pp. 2, 14, 17, 23–25).

Kalia, Anuj, Michael Kaminsky, and David G. Andersen (2016). "Design Guidelines for High Performance RDMA Systems". In: *Proceedings of the 2016 USENIX Annual Technical Conference*. USENIX '16. USENIX Association, pp. 437–450 (cit. on p. 24).

Kazemi, Seyed Mehran and David Poole (2018). "SimplE Embedding for Link Prediction in Knowledge Graphs". In: *Advances in Neural Information Processing Systems*. NeurIPS '18. Curran Associates (cit. on pp. 2, 9).

Keskar, Nitish Shirish, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang (2017). "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *Proceedings of the 5th International Conference on Learning Representations*. ICLR '17 (cit. on p. 22).

Kim, Jin Kyu, Abutalib Aghayev, Garth Gibson, and Eric Xing (2019). "STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model". In: *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX '19. USENIX Association, pp. 207–222 (cit. on p. 14).

Kim, Jin Kyu, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth Gibson, and Eric Xing (2016). "STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning". In: *Proceedings of the 11th European Conference on Computer Systems*. EuroSys '16. Association for Computing Machinery (cit. on pp. 2, 13).

Kim, Soojeong, 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*. EuroSys '19. Association for Computing Machinery (cit. on pp. 70, 106, 107).

Kochsiek, Adrian and Rainer Gemulla (2021). "Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques". In: *PVLDB* 15.3, pp. 633–645 (cit. on pp. 76, 84).

Koren, Yehuda, Robert Bell, and Chris Volinsky (2009). "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42.8, pp. 30–37 (cit. on pp. 1, 2, 9, 48, 63, 65).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. NeurIPS '12. Curran Associates (cit. on p. 1).

Lamport, Leslie (1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* 28.9, pp. 690–691 (cit. on pp. 16, 43).

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551 (cit. on pp. 1, 10, 17, 66).

Lepikhin, Dmitry, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen (2021). "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding". In: *Proceedings of the 9th International Conference on Learning Representations*. ICLR '21 (cit. on p. 145).

Lerer, Adam, 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*. Vol. 1. MLSys '19, pp. 120–131 (cit. on pp. 2, 13, 27, 30, 64, 73, 76, 77, 99).

Li, Ang, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker (2020). "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect". In: *IEEE Transactions on Parallel and Distributed Systems* 31.1, pp. 94–110 (cit. on p. 24).

Li, Margaret, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A. Smith, and Luke Zettlemoyer (2022). "Branch-Train-Merge: Embarrassingly Parallel Training of Expert Language Models". In: *CoRR* abs/2208.03306 (cit. on p. 145).

Li, Mu, David Andersen, Jun Woo Park, Alexander Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene Shekita, and Bor-Yiing Su (2014a). "Scaling Distributed Machine Learning with the Parameter Server". In: *Proceedings of the 11th Conference on Operating Systems Design and Imple-*

mentation. OSDI '14. USENIX Association, pp. 583–598 (cit. on pp. 2, 14, 15, 25, 29, 34, 35, 43, 47, 51, 68, 85, 99, 125, 126).

Li, Mu, David Andersen, Alexander Smola, and Kai Yu (2014b). "Communication Efficient Distributed Machine Learning with the Parameter Server". In: *Advances in Neural Information Processing Systems*. NeurIPS '14. MIT Press, pp. 19–27 (cit. on pp. 22, 25).

Li, Shen, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala (2020). "PyTorch Distributed: Experiences on Accelerating Data Parallel Training". In: *PVLDB* 13.12, pp. 3005–3018 (cit. on pp. 17, 18, 25).

Lian, Xiangru, Yijun Huang, Yuncheng Li, and Ji Liu (2015). "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization". In: *Advances in Neural Information Processing Systems*. NeurIPS '15. MIT Press, pp. 2737–2745 (cit. on p. 73).

Lian, Xiangru, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu (2017). "Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. Vol. 30. NeurIPS '17. Curran Associates (cit. on pp. 24, 25).

Lin, Yujun, Song Han, Huizi Mao, Yu Wang, and Bill Dally (2018). "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training". In: *Proceesings of the 6th International Conference on Learning Representations*. ICLR '18 (cit. on p. 22).

Lipton, Richard J and Jonathan S Sandberg (1988). *PRAM: A scalable shared memory*. Tech. rep. Princeton University, Department of Computer Science (cit. on p. 43).

Liu, Hanxiao, Yuexin Wu, and Yiming Yang (2017). "Analogical Inference for Multi-relational Embeddings". In: *Proceedings of the 34th International Conference on Machine Learning*. ICML '17. PMLR, pp. 2168–2178 (cit. on pp. 49, 56, 64, 67, 83, 99).

Liu, Xiaodong, Yelong Shen, Kevin Duh, and Jianfeng Gao (2018). "Stochastic Answer Networks for Machine Reading Comprehension". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. ACL '18. Association for Computational Linguistics, pp. 1694–1704 (cit. on p. 50).

Low, Yucheng, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph Hellerstein (2012). "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *PVLDB* 5.8, pp. 716–727 (cit. on pp. 25, 27, 28, 70).

Makari, Faraz, Christina Teflioudi, Rainer Gemulla, Peter Haas, and Yannis Sismanis (2015). “Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion”. In: *Knowledge and Information Systems* 42.3, pp. 493–523 (cit. on pp. 48, 84, 89).

Malewicz, Grzegorz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski (2010). “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Association for Computing Machinery, pp. 135–146 (cit. on p. 25).

Masters, Dominic and Carlo Luschi (2018). “Revisiting Small Batch Training for Deep Neural Networks”. In: *CoRR* abs/1804.07612 (cit. on p. 22).

Meka, Raghu, Prateek Jain, and Inderjit Dhillon (2009). “Matrix Completion from Power-Law Distributed Samples”. In: *Advances in Neural Information Processing Systems*. NeurIPS ’09. Curran Associates, pp. 1258–1266 (cit. on pp. 48, 63, 65).

Miao, Xupeng, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui (2021). “Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce”. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Association for Computing Machinery, pp. 2262–2270 (cit. on p. 24).

Miao, Xupeng, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui (2022). “HET-GMP: A Graph-Based System Approach to Scaling Large Embedding Model Training”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. Association for Computing Machinery, pp. 470–480 (cit. on p. 23).

Mikolov, Tomas, 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 (cit. on pp. 1, 2, 9, 12, 50, 51, 56, 63–65, 67, 68, 74, 77, 84, 99).

Min, Seung Won, 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”. In: *PVLDB* 14.11, pp. 2087–2100 (cit. on p. 23).

Mitchell, Tom M. (1997). *Machine Learning*. McGraw-Hill (cit. on p. 7).

Mohamed, Aisha, Shameem Parambath, Zoi Kaoudi, and Ashraf Aboulnaga (2020). “Popularity Agnostic Evaluation of Knowledge Graph Embeddings”. In: *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence*. UAI ’20. PMLR, pp. 1059–1068 (cit. on p. 65).

Moreno-Sánchez, Isabel, Francesc Font-Clos, and Álvaro Corral (2016). "Large-Scale Analysis of Zipf's Law in English Texts". eng. In: *PloS one* 11.1 (cit. on pp. 63, 65).

Nakandala, Supun, Yuhao Zhang, and Arun Kumar (2019). "Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems". In: *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*. DEEM '19. Association for Computing Machinery, 6:1–6:4 (cit. on pp. 27, 30).

Narayanan, Deepak, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia (2019). "PipeDream: Generalized Pipeline Parallelism for DNN Training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Association for Computing Machinery, pp. 1–15 (cit. on p. 23).

Németh, Gábor, Dániel Géhberger, and Péter Mátray (2017). "DAL: A Locality-Optimizing Distributed Shared Memory System". In: *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*. HotCloud '17. USENIX Association (cit. on p. 60).

Nickel, Maximilian, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich (2016a). "A Review of Relational Machine Learning for Knowledge Graphs". In: *Proceedings of the IEEE* 104.1, pp. 11–33 (cit. on p. 49).

Nickel, Maximilian, Lorenzo Rosasco, and Tomaso Poggio (2016b). "Holographic Embeddings of Knowledge Graphs". In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI '16. AAAI Press, pp. 1955–1961 (cit. on p. 49).

Nickel, Maximilian, 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*. ICML '11. Omnipress, pp. 809–816 (cit. on pp. 2, 9, 49, 63, 65).

Niu, Feng, Benjamin Recht, Christopher Re, and Stephen Wright (2011). "HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. NeurIPS '11. Curran Associates, pp. 693–701 (cit. on pp. 11, 12, 40).

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the Difficulty of Training Recurrent Neural Networks". In: *Proceedings of the 30th International Conference on Machine Learning*. ICML '13. JMLR.org, pp. 1310–1318 (cit. on p. 85).

Paszke, Adam, 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 (cit. on pp. 2, 9, 25, 108, 144).

Patterson, David A. (2004). "Latency Lags Bandwith". In: *Communications of the ACM* 47.10, pp. 71–75 (cit. on p. 24).

Peng, Bo, Bingjing Zhang, Langshi Chen, Mihai Avram, Robert Henschel, Craig Stewart, Shaojuan Zhu, Emily Mccallum, Lisa Smith, Tom Zahniser, et al. (2017). "HarpLDA+: Optimizing latent dirichlet allocation for parallel efficiency". In: *2017 IEEE International Conference on Big Data*. BigData '17. IEEE Computer Society, pp. 243–252 (cit. on pp. 27, 30).

Peng, Jingshu, 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". In: *PVLDB* 15.9, pp. 1937–1950 (cit. on p. 23).

Pennington, Jeffrey, 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, pp. 1532–1543 (cit. on pp. 2, 9, 50).

Peters, Matthew, 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, pp. 2227–2237 (cit. on pp. 2, 9, 10, 50, 66, 67, 144).

Raman, Parameswaran, Sriram Srinivasan, Shin Matsushima, Xinhua Zhang, Hyokun Yun, and S.V.N. Vishwanathan (2019). "Scaling Multinomial Logistic Regression via Hybrid Parallelism". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '19. Association for Computing Machinery, pp. 1460–1470 (cit. on pp. 27, 30).

Ramesh, Aditya, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever (2021). "Zero-Shot Text-to-Image Generation". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, pp. 8821–8831 (cit. on pp. 1, 10).

Ratnasamy, Sylvia, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker (2001). "A Scalable Content-addressable Network". In: *Proceed-*

ings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. SIGCOMM '01. Association for Computing Machinery, pp. 161–172 (cit. on p. 46).

Rawat, Ankit Singh, Aditya Krishna Menon, Wittawat Jitkrittum, Sadeep Jayasumana, Felix X. Yu, Sashank J. Reddi, and Sanjiv Kumar (2021). "Disentangling Sampling and Labeling Bias for Learning in Large-Output Spaces". In: *CoRR* abs/2105.05736 (cit. on p. 67).

Řehůřek, Radim and Petr Sojka (2010). "Software Framework for Topic Modelling with Large Corpora". English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. LREC '10. ELRA, pp. 45–50 (cit. on pp. 50, 84, 99).

Rendle, Steffen, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme (2009). "BPR: Bayesian Personalized Ranking from Implicit Feedback". In: *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*. UAI '09. AUAI Press, pp. 452–461 (cit. on pp. 64, 67).

Renz-Wieland, Alexander, Tobias Drobisch, Zoi Kaoudi, Rainer Gemulla, and Volker Markl (2021). "Just Move It! Dynamic Parameter Allocation in Action". In: *PVLDB* 14.12, pp. 2707–2710 (cit. on p. 4).

Renz-Wieland, Alexander, Rainer Gemulla, Zoi Kaoudi, and Volker Markl (2022a). "NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access". In: *Proceedings of the 2022 ACM International Conference on Management of Data*. SIGMOD '22. Association for Computing Machinery (cit. on p. 4).

Renz-Wieland, Alexander, Rainer Gemulla, Steffen Zeuch, and Volker Markl (2020). "Dynamic Parameter Allocation in Parameter Servers". In: *PVLDB* 13.12, pp. 1877–1890 (cit. on p. 4).

Renz-Wieland, Alexander, Andreas Kieslinger, Robert Gericke, Rainer Gemulla, Zoi Kaoudi, and Volker Markl (2022b). "Good Intentions: Adaptive Parameter Servers via Intent Signaling". In: *CoRR* abs/2206.00470 (cit. on p. 4).

Riquelme, Carlos, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby (2021). "Scaling Vision with Sparse Mixture of Experts". In: *Advances in Neural Information Processing Systems*. NeurIPS '21. Curran Associates, pp. 8583–8595 (cit. on pp. 1, 145).

Rowstron, Antony and Peter Druschel (2001). "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Proceedings of the IFIP/ACM International Conference on Distributed*

Systems Platforms Heidelberg. Middleware '01. Springer, pp. 329–350 (cit. on p. 46).

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (cit. on p. 7).

Ruffinelli, Daniel, Samuel Broscheit, and Rainer Gemulla (2020). "You CAN Teach an Old Dog New Tricks! On Training Knowledge Graph Embeddings". In: *Proceedings of the 8th International Conference on Learning Representations*. ICLR '20 (cit. on pp. 49, 56, 64, 67, 74, 83, 84).

Schlichtkrull, Michael, 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, pp. 593–607 (cit. on pp. 2, 9, 144).

Schroff, F., D. Kalenichenko, and J. Philbin (2015). "FaceNet: A unified embedding for face recognition and clustering". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '15. IEEE Computer Society, pp. 815–823 (cit. on pp. 64, 67).

Seide, Frank, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu (2014). "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs". In: *Proceedings of the 15th Annual Conference of the International Speech Communication Association*. Interspeech '14, pp. 1058–1062 (cit. on p. 22).

Sergeev, Alexander and Mike Del Balso (2018). "Horovod: fast and easy distributed deep learning in TensorFlow". In: *CoRR* abs/1802.05799 (cit. on pp. 2, 17, 23).

Shallue, Christopher J., Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl (2019). "Measuring the Effects of Data Parallelism on Neural Network Training". In: *Journal of Machine Learning Research* 20.112, pp. 1–49 (cit. on p. 22).

Shang, Chao, 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. AAAI Press, pp. 3060–3067 (cit. on pp. 2, 9, 144).

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean (2017). "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer". In: *Proceedings of the 5th International Conference on Learning Representations*. ICLR '17 (cit. on p. 145).

Shi, Baoxu and Tim Weninger (2018). "Open-World Knowledge Graph Completion". In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*. AAAI '18. AAAI Press (cit. on p. 49).

Simonyan, Karen and Andrew Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *Proceedings of the 3rd International Conference on Learning Representations*. ICLR '15 (cit. on p. 1).

Smith, Alan (1982). "Cache Memories". In: *ACM Computing Surveys* 14.3, pp. 473–530 (cit. on p. 31).

Smola, Alexander and Shravan Narayanamurthy (2010). "An Architecture for Parallel Topic Models". In: *PVLDB* 3.1-2, pp. 703–710 (cit. on pp. 2, 14, 15, 25, 28, 68).

Socher, Richard, John Bauer, Christopher Manning, and Andrew Ng (2013). "Parsing with Compositional Vector Grammars". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*. ACL '13. Association for Computational Linguistics, pp. 455–465 (cit. on p. 50).

Steen, Maarten van and Andrew Tanenbaum (2017). *Distributed Systems*. 3rd (cit. on pp. 31, 35, 46, 121).

Stergiou, Stergios, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis (2017). "Distributed Negative Sampling for Word Embeddings". In: *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI '17. AAAI Press, pp. 2569–2575 (cit. on pp. 64, 73, 76).

Stoica, Ion, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan (2001). "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. Association for Computing Machinery, pp. 149–160 (cit. on p. 46).

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 27. NeurIPS '14. Curran Associates (cit. on p. 1).

Tang, Hanlin, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu (2018). "D^2: Decentralized Training over Decentralized Data". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. ICML '18. PMLR, pp. 4848–4856 (cit. on p. 24).

Teflioudi, Christina, Faraz Makari, and Rainer Gemulla (2012). "Distributed Matrix Completion". In: *12th IEEE International Conference on Data*

Mining. ICDM '12. IEEE Computer Society, pp. 655–664 (cit. on pp. 27, 30, 31, 46, 84, 98).

Thakur, Rajeev and William D. Gropp (2003). "Improving the Performance of Collective Operations in MPICH". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. EuroPVM/MPI '03. Springer, pp. 257–267 (cit. on p. 73).

Träff, Jesper Larsson (2010). "Transparent Neutral Element Elimination in MPI Reduction Operations". In: *Recent Advances in the Message Passing Interface*. Springer, pp. 275–284 (cit. on p. 72).

Trouillon, Théo, 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*. ICML '16. JMLR.org, pp. 2071–2080 (cit. on pp. 2, 9, 49, 83).

Vashishth, Shikhar, 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 (cit. on pp. 2, 9, 144).

Wang, Guanhua, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica (2020). "Blink: Fast and Generic Collectives for Distributed ML". In: *Proceedings of Machine Learning and Systems*. Vol. 2. MLSys '20, pp. 172–186 (cit. on p. 24).

Wang, Hongyi, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright (2018). "ATOMO: Communication-efficient Learning via Atomic Sparsification". In: *Advances in Neural Information Processing Systems*. Vol. 31. NeurIPS '18. Curran Associates (cit. on p. 22).

Wang, Ruoxi, Bin Fu, Gang Fu, and Mingliang Wang (2017). "Deep & Cross Network for Ad Click Predictions". In: *Proceedings of the ADKDD'17*. ADKDD'17. Association for Computing Machinery (cit. on pp. 2, 9, 144).

Wang, Shuai, Dan Li, Jinkun Geng, Yue Gu, and Yang Cheng (2019). "Impact of Network Topology on the Performance of DML: Theoretical Analysis and Practical Factors". In: *Proceedings of the 2019 IEEE International Conference on Computer Communications*. INFOCOM '19. IEEE Computer Society, pp. 1729–1737 (cit. on p. 23).

Wang, Xiaozhi, Tianyu Gao, Zhaocheng Zhu, Zhiyuan Liu, Juanzi Li, and Jian Tang (2019). "KEPLER: A Unified Model for Knowledge Embedding and Pre-trained Language Representation". In: *CoRR* abs/1911.06136 (cit. on p. 83).

Wen, Wei, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li (2017). "TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning". In: *Advances in Neural Information Processing Systems*. Vol. 30. NeurIPS '17. Curran Associates (cit. on p. 22).

Wolfson, Ouri and Sushil Jajodia (1992). "Distributed Algorithms for Dynamic Replication of Data". In: *Proceedings of the 11fh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '92. Association for Computing Machinery, pp. 149–163 (cit. on p. 70).

Xing, Eric, 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*. KDD '15. Association for Computing Machinery, pp. 1335–1344 (cit. on pp. 29, 40, 43, 51, 85, 103, 108, 122).

Yang, Bishan, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng (2015). "Embedding Entities and Relations for Learning and Inference in Knowledge Bases". In: *Proceedings of the 3rd International Conference on Learning Representations*. ICLR '15 (cit. on p. 49).

Yang, Bowen, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa (2021). "PipeMare: Asynchronous Pipeline Parallel DNN Training". In: *Proceedings of Machine Learning and Systems*. Vol. 3. MLSys '21, pp. 269–296 (cit. on p. 23).

Yang, Fan, Jinfeng Li, and James Cheng (2016). "Husky: Towards a More Efficient and Expressive Distributed Computing Framework". In: *PVLDB* 9.5, pp. 420–431 (cit. on p. 60).

Yang, Zhen, Ming Ding, Chang Zhou, Hongxia Yang, Jingren Zhou, and Jie Tang (2020). "Understanding Negative Sampling in Graph Representation Learning". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '20. Association for Computing Machinery, pp. 1666–1676 (cit. on p. 67).

You, Yang, Igor Gitman, and Boris Ginsburg (2017a). "Scaling SGD Batch Size to 32K for ImageNet Training". In: *CoRR* abs/1708.03888 (cit. on p. 22).

You, Yang, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh (2020). "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes". In: *Proceedings of the 8th International Conference on Learning Representations*. ICLR '20 (cit. on p. 22).

You, Yang, Zhao Zhang, Cho-Jui Hsieh, and James Demmel (2017b). "100-epoch ImageNet Training with AlexNet in 24 Minutes". In: *CoRR* abs/1709.05011 (cit. on p. 22).

Yu, Hsiang-Fu, Cho-Jui Hsieh, Hyokun Yun, S.V.N. Vishwanathan, and Inderjit Dhillon (2015). "A Scalable Asynchronous Distributed Algorithm for Topic Modeling". In: *Proceedings of the 24th International Conference on World Wide Web*. WWW '15. International World Wide Web Conferences Steering Committee, pp. 1340–1350 (cit. on pp. 27, 30).

Yun, Hyokun, Hsiang-Fu Yu, Cho-Jui Hsieh, S.V.N. Vishwanathan, and Inderjit Dhillon (2014). "NOMAD: Non-locking, Stochastic Multi-machine Algorithm for Asynchronous and Decentralized Matrix Completion". In: *PVLDB* 7.11, pp. 975–986 (cit. on pp. 27, 29, 30, 46).

Zaharia, Matei, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica (2010). "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd Conference on Hot Topics in Cloud Computing*. HotCloud '10. USENIX Association, pp. 10–10 (cit. on p. 25).

Zhang, Dalong, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi (2020). "AGL: A Scalable System for Industrial-Purpose Graph Machine Learning". In: *PVLDB* 13.12, pp. 3125–3137 (cit. on p. 23).

Zhang, Hantian, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang (2017). "ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. ICML '17. PMLR, pp. 4035–4043 (cit. on p. 22).

Zhang, Xin, Jia Liu, and Zhengyuan Zhu (2018). "Taming Convergence for Asynchronous Stochastic Gradient Descent with Unbounded Delay in Non-Convex Learning". In: *CoRR* abs/1805.09470 (cit. on p. 73).

Zhang, Zhipeng, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, and Xupeng Miao (2019). "PS2: Parameter Server on Spark". In: *Proceedings of the 2019 ACM International Conference on Management of Data*. SIGMOD '19. Association for Computing Machinery, pp. 376–388 (cit. on pp. 2, 14, 25).

Zhao, Ben, Ling Huang, Jeremy Stribling, Sean Rhea, Anthony Joseph, and John Kubiatowicz (2004). "Tapestry: A resilient global-scale overlay for service deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1, pp. 41–53 (cit. on p. 46).

Zhao, Weijie, 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*. Vol. 2. MLSys '20, pp. 412–428 (cit. on p. 23).

Zheng, Da, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis (2020a). "DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs". In: *CoRR* abs/2010.05337 (cit. on p. 23).

Zheng, Da, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis (2020b). "DGL-KE: Training Knowledge Graph Embeddings at Scale". In: *CoRR* abs/2004.08532 (cit. on pp. 64, 73, 76, 77, 94).

Zheng, Qiming, 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 Computer Society, pp. 609–618 (cit. on pp. 70, 106, 114).

Zhou, Guorui, 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*. KDD '18. Association for Computing Machinery, pp. 1059–1068 (cit. on pp. 2, 9, 144).

Zhu, Rong, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou (2019). "AliGraph: A Comprehensive Graph Neural Network Platform". In: *PVLDB* 12.12, pp. 2094–2105 (cit. on p. 23).

Ziegler, Tobias, Carsten Binnig, and Viktor Leis (2022). "ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Association for Computing Machinery, pp. 685–699 (cit. on p. 24).

Ziegler, Tobias, Viktor Leis, and Carsten Binnig (2020). "RDMA Communciation Patterns". In: *Datenbank-Spektrum* 20, pp. 199–210 (cit. on p. 24).

Zinkevich, Martin, Markus Weimer, Lihong Li, and Alex Smola (2010). "Parallelized Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. Vol. 23. NeurIPS '10. Curran Associates (cit. on p. 10).

List of Figures

List of Tables

List of Algorithms