# Dynamic Parameter Allocation in Parameter Servers

Alexander Renz-Wieland [1], Rainer Gemulla [2], Steffen Zeuch [1,3], Volker Markl [1,3]

[1]Technische Universität Berlin, [2]Universität Mannheim, [3]German Research Center for Artificial Intelligence

[1]firstname.lastname@tu-berlin.de, [2]rgemulla@uni-mannheim.de, [3]firstname.lastname@dfki.de

## ABSTRACT

To keep up with increasing dataset sizes and model complexity, distributed training has become a necessity for large machine learning tasks. Parameter servers ease the implementation of distributed parameter management—a key concern in distributed training—, but can induce severe communication overhead. To reduce communication overhead, distributed machine learning algorithms use techniques to increase parameter access locality (PAL), achieving up to linear speed-ups. We found that existing parameter servers provide only limited support for PAL techniques, however, and therefore prevent efficient training. In this paper, we explore whether and to what extent PAL techniques can be supported, and whether such support is beneficial. We propose to integrate dynamic parameter allocation into parameter servers, describe an efficient implementation of such a parameter server called LAPSE, and experimentally compare its performance to existing parameter servers across a number of machine learning tasks. We found that LAPSE provides near-linear scaling and can be orders of magnitude faster than existing parameter servers.

## 1. INTRODUCTION

To keep up with increasing dataset sizes and model complexity, distributed training has become a necessity for large machine learning (ML) tasks. Distributed ML allows (1) for models and data larger than the memory of a single machine, and (2) for faster training by leveraging distributed compute. In distributed ML, both training data and model parameters are partitioned across a compute cluster. Each node in the cluster usually accesses only its local part of the training data, but reads and/or updates most of the model parameters. Parameter management is thus a key concern in distributed ML. Applications either manage model parameters manually using low-level distributed programming
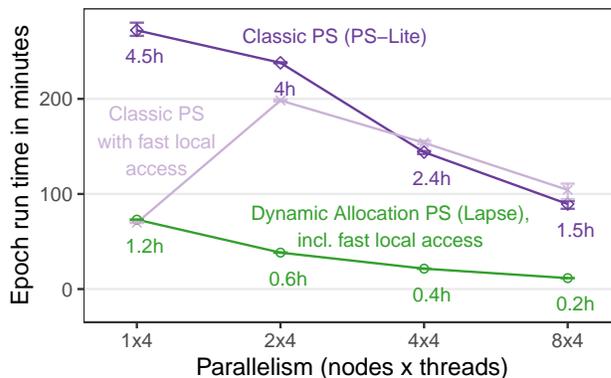
**Figure 1:** Parameter server (PS) performance for a large knowledge graph embeddings task (RESCAL, dimension 100). The performance of the classic PSs falls behind the performance of a single node due to communication overhead. In contrast, dynamic parameter allocation enables LAPSE to scale near-linearly. Details in Section 4.1.

primitives or delegate parameter management to a *parameter server* (PS). PSs provide primitives for reading and writing parameters and handle partitioning and synchronization across nodes. Many ML stacks use PSs as a component, e.g., TensorFlow [1], MXNet [8], PyTorch BigGraph [27], STRADS [24], STRADS-AP [23], or Project Adam [9], and there exist multiple standalone PSs, e.g., Petuum [17], PS-Lite [28], Angel [22], FlexPS [18], Glint [20], and PS2 [64].

As parameters are accessed by multiple nodes in the cluster and therefore need to be transferred between nodes, distributed ML algorithms may suffer from severe communication overhead when compared to single-machine implementations. Figure 1 shows exemplarily that the performance of a distributed ML algorithm may fall behind the performance of single machine algorithms when a classic PS such as PS-Lite is used. To reduce the impact of communication, distributed ML algorithms employ techniques [14, 63, 56, 4, 27, 44, 62, 41, 33, 15, 36] that increase *parameter access locality* (PAL) and can achieve linear speed-ups. Intuitively, PAL techniques ensure that most parameter accesses do not require (synchronous) communication; example techniques include exploiting natural clustering of data, parameter blocking, and latency hiding. Algorithms that use PAL techniques typically manage parameters manually using low-level distributed programming primitives.

Most existing PSs are easy to use—there is no need for low-level distributed programming—, but provide only limited support for PAL techniques. One limitation, for exam-
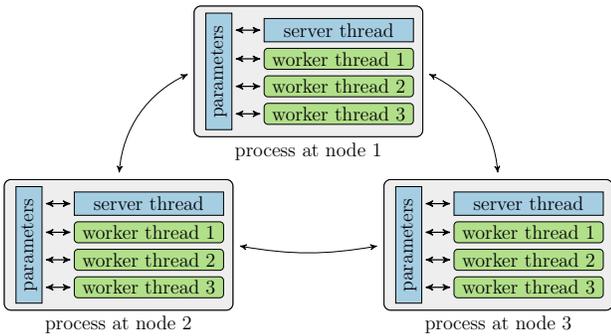
**Figure 2:** PS architecture with server and worker threads co-located in one process per node. Lapse employs this architecture.

ple, is that they allocate parameters statically. Moreover, existing approaches to reducing communication overhead in PSs provide only limited scalability compared to using PAL techniques (e.g., replication and bounded staleness [17, 11]) or are not applicable to the ML algorithms that we study (e.g., dynamically reducing cluster size [18]).

In this paper, we explore whether and to what extent PAL techniques can be supported in PSs, and whether such support is beneficial. To improve PS performance and suitability, we propose to integrate *dynamic parameter allocation* (DPA) into PSs. DPA dynamically allocates parameters where they are accessed, while providing location transparency and PS consistency guarantees, i.e., sequential consistency. By doing so, PAL techniques can be exploited directly. We discuss design options for PSs with DPA and describe an efficient implementation of such a PS called Lapse.

Figure 1 shows the performance of Lapse for the task of training knowledge graph embeddings [40] using data clustering and latency hiding PAL techniques. In contrast to classic PSs, Lapse outperformed the single-machine baseline and showed near-linear speed-ups. In our experimental study, we observed similar results for multiple other ML tasks (matrix factorization and word vectors): the classic PS approach barely outperformed the single-machine baseline, whereas Lapse scaled near-linearly, with speed-ups of up to two orders of magnitude compared to classic PSs and up to one order of magnitude compared to state-of-the-art PSs. Figure 1 further shows that—although critical to the performance of Lapse—fast local access alone does not alleviate the communication overhead of the classic PS approach.

In summary, our contributions are as follows. (i) We examine whether and to what extent existing PSs support using PAL techniques to reduce communication overhead. (ii) We propose to integrate DPA into PSs to be able to support PAL techniques directly. (iii) We describe Lapse, an efficient implementation of a PS with DPA. (iv) We experimentally investigate the efficiency of classic PSs, PSs with bounded staleness, and Lapse on a number of ML tasks.

## 2. THE CASE FOR DYNAMIC PARAMETER ALLOCATION

We start by reviewing basic PS architectures (Section 2.1). Second, we outline common PAL techniques used in distributed ML (Section 2.2). For each technique, we discuss

**Table 1:** Per-key consistency guarantees of PS architectures, using representatives for types: PS-Lite [28] for classic and Petuum [59] for stale.

| Parameter Server | Classic | | Lapse | | | Stale |
|---|---|---|---|---|---|---|
| Synchronous | sync | async | sync | async | | sync, async |
| Location caches | | | | off | on | |
| Eventual | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| PRAM[a][30] | ✓ | ✓[b] | ✓ | ✓[b] | ✗ | ✓ |
| Causal [19] | ✓ | ✓[b] | ✓ | ✓[b] | ✗ | ✗ |
| Sequential [26] | ✓ | ✓[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)

to what extent it is supported in existing PSs and identify which features would be required to enable or improve support. Finally, we introduce DPA, which enables PSs to exploit PAL techniques directly (Section 2.3).

### 2.1 Basic PS Architectures

PSs [53, 2, 12, 17, 28] partition the model parameters across a set of *servers*. The training data are usually partitioned across a set of *workers*. During training, each worker processes its local part of the training data (often multiple times) and continuously reads and updates model parameters. To coordinate parameter accesses across workers, each parameter is assigned a unique *key* and the PS provides `pull` and `push` primitives for reads and writes, respectively; cf. Table 2. Both operations can be performed synchronously or asynchronously. 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.

Although servers and workers may reside on different machines, they are often co-located for efficiency reasons (especially when PAL techniques are used). Some architectures [28, 20, 22] run one server process and one or more worker processes on each machine, others [17, 18] use a single process with one server thread and multiple worker threads to reduce inter-process communication. Figure 2 depicts such a PS architecture with one server and three worker threads per node.

In the *classic PS* architecture, parameters are statically allocated to servers (e.g., via a range partitioning of the parameter keys) and there is no replication. Thus precisely one server holds the current value of a parameter, and this server is used for all pull and push operations on this parameter. Classic PSs typically guarantee sequential consistency [26] for operations on the same key. This means that (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. Note that lost updates do not occur in PSs when updates are cumulative. Table 1 gives an overview of provided consistency guarantees for different types of PSs. PSs give no guarantees across multiple keys.

The *stale PS* architecture employs replication and tolerates some amount of staleness in the replicas [17, 18, 22, 11, 10]. In such architectures, parameters are still statically allocated to servers as in a classic PS, but the PS may replicate a subset of the parameters to additional servers to reduce communication overhead [17]. This is beneficial especially when servers and workers are co-located because
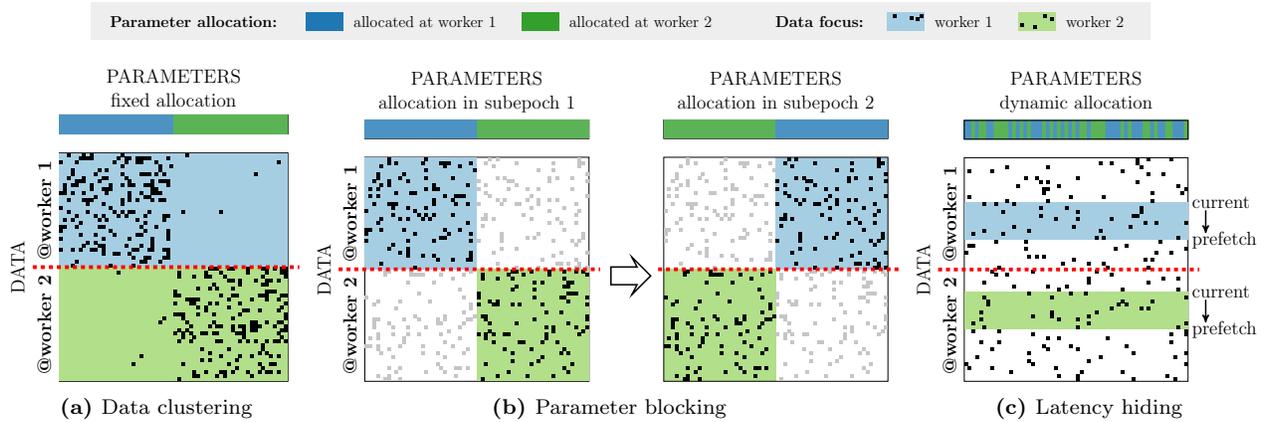
**Figure 3:** Techniques to reduce communication cost, based on parameter access locality: workers access different subsets of the model parameters over time. Rows correspond to data points, columns to parameters, black dots to parameter accesses. (a) Training data are clustered such that each worker accesses mostly a separate subset of parameters (Section 2.2.1). (b) Within each subepoch, each worker is restricted to one block of parameters. Which worker has access to which block changes from subepoch to subepoch (Section 2.2.2). (c) Asynchronously prefetching (or prelocalizing) of parameter values, such that they can be accessed locally, hides access latency (Section 2.2.3).

parameters can be replicated to the subset of servers that access them. Stale PSs often provide weaker forms of consistency than classic PSs (PRAM consistency or only eventual consistency). They commonly do support *bounded staleness* and require applications to explicitly control staleness via special primitives (e.g., an "advance the clock" operation).

## 2.2 PAL Techniques

We consider three common PAL techniques: data clustering, parameter blocking, and latency hiding.

### 2.2.1 Data Clustering

One method to reduce communication cost is to exploit structure in training data [33, 15, 7, 53, 2]. For example, consider a training data set 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 vectors). 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 3a depicts an example; here rows correspond to documents, dots to words, and each parameter is allocated to the node where it is accessed 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 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 [14, 63]. However, to the best of our knowledge, all existing PSs access parameters either through inter-process [28] or inter-thread communication [59], leading to overly high access latency.

### 2.2.2 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 [14, 63, 56], tensor factorization [4], latent dirichlet allocation [62, 41], multinomial logistic regression [44], and knowledge graph embeddings [27]. They were also proposed for efficient multi-model training [36].

Manual implementations exploit parameter blocking by allocating parameters to the node where they are currently accessed [14, 56, 63], thus eliminating network communication for individual parameter accesses. Communication is required only between subepochs. Figure 3b depicts a simplified example for a matrix factorization task [14]. 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 stale PS architectures, however. This re-

quires 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 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 parameters among nodes during run time.

### 2.2.3 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 [52] or distributed systems [58]. In distributed ML, the latency of parameter access can be reduced by ensuring that a parameter value is already present at a worker at the time it is accessed [11, 10, 56]. Such an approach is beneficial when parameter access is sparse, i.e., each worker accesses few parameters at a time.

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 (cf. Table 1). 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 [11], which proactively replicates all previously accessed parameters at a node (during its "clock" operation). This approach avoids parameter management, but does not provide sequential consistency.

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 3c. 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.

## 2.3 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 runtime.

**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 runtime. 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 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.

## 3. 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 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.

## 3.1 Overview

LAPSE co-locates worker and server threads, as illustrated in Figure 2, 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 2. 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. 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) [28]. Workers access localized parameters in the same way as non-localized parameters. This allows LAPSE to relocate parameters without affecting workers that use them.

**Table 2:** 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)` | ✓ | ✓ | Retrieve 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`. |

**Location management.** LAPSE manages parameter locations with a decentralized home node approach: 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 3.5.

**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 3.3 for details.

**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.

**Consistency.** In general, LAPSE provides the sequential consistency guarantees of classic PSs even in the presence of parameter relocations. We show in Section 3.4 that the use of 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.

## 3.2 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). 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.
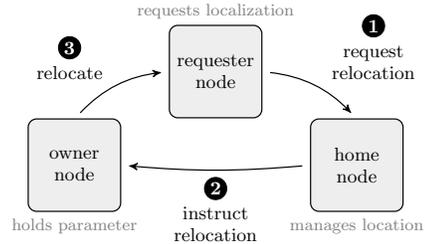


**Figure 4:** A worker requests to localize a parameter. LAPSE transparently relocates the parameter from the current owner to the requester node and informs the home node of the location change.

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 answer operations for the parameter (but instead queues operations for later processing). The two measures usually differ because the current owner of the parameter 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 4. 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 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 3.4, 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
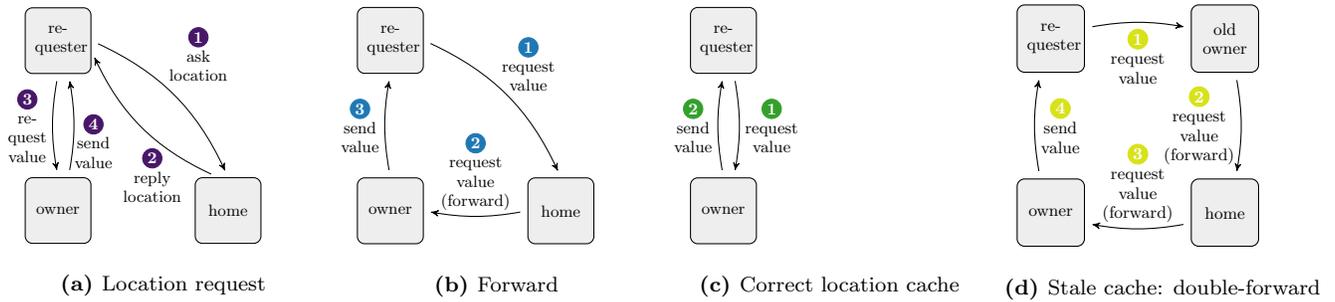
**(a)** Location request  **(b)** Forward  **(c)** Correct location cache  **(d)** Stale cache: double-forward

**Figure 5:** 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)*

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 network 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. We did not consider such an approach in LAPSE.

## 3.3 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 2) 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 [59], for example). As other PSs, LAPSE guarantees per-key atomic reads and writes; it does so using latches (i.e., locks held for the time of the operation) for local accesses (see Section 3.7).

**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 re-

quest to that owner (Figure 5a). In the *forward* strategy, the worker sends the request itself to the home node, which then forwards it to the current owner (Figure 5b). 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 the request to the owner (message **2**). In contrast, in *location request*, the requester node sends the request (message **3**) 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 **2**). In this case, the requester node may send message **3** 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 5c), 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 5d).

**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.

## 3.4 Consistency

In this section, we analyze the consistency properties of Lapse and compare them to classic PSs, i.e., to PS-Lite [28]. Table 1 shows a summary. Consistency guarantees affect the convergence of ML algorithms in the distributed setting; in particular, relaxed consistency can slow down convergence [17, 11]. The extent of this impact differs from task to task [17]. None of the existing PSs guarantee serializability, as pull and push operations of different workers can overlap arbitrarily. Neither do PSs give guarantees across multiple keys. PSs can, however, provide per-key sequential consistency. Sequential consistency provides two properties [26]: (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. Note that stale PSs do not provide sequential consistency, as discussed in Section 2.1. 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 SKETCH. In the absence of relocations, Lapse provides sequential consistency, analogously to classic PSs. 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. Lapse employs latches to guarantee a sequential execution among local threads.

**Asynchronous operations.** A classic PS such as PS-Lite provides sequential consistency for asynchronous operations.[1] 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 SKETCH. For property (1), first suppose that there is no concurrent relocation. Lapse routes the operations of

---

[1] 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:** Location management strategies. $N$ is the number of nodes, $K$ is the number of parameter keys.

| 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

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.*

PROOF SKETCH. 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 $O_2$. $O_1$ is sent to the currently cached, but outdated, owner. Then the location cache is updated (by another returning operation) and $O_2$ is sent directly to the current owner. Now, it is possible that $O_2$ is processed before $O_1$, because $O_1$ has to be double-forwarded to the current owner. This breaks sequential, causal, and PRAM consistency.

## 3.5 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 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

**Table 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 | Data set | Data points | Size | Keys/s | MB/s |
| Matrix | Latent Factors, rank 100 | 6.4 M | 640 M | 4.8 GB | 3.4m × 3m matrix | 1000 M | 31 GB | 414 k | 315 |
| Factorization | Latent Factors, rank 100 | 11.0 M | 1100 M | 8.2 GB | 10m × 1m matrix | 1000 M | 31 GB | 316 k | 241 |
| Knowledge | ComplEx, dim. 100 | 0.5 M | 98 M | 0.7 GB | DBpedia-500k | 3 M | 47 MB | 312 k | 476 |
| Graph | ComplEx, dim. 4000 | 0.5 M | 3929 M | 29.3 GB | DBpedia-500k | 3 M | 47 MB | 11 k | 643 |
| Embeddings | RESCAL, dim. 100 | 0.5 M | 110 M | 0.8 GB | DBpedia-500k | 3 M | 47 MB | 12 k | 614 |
| Word Vectors | Word2Vec, dim. 1000 | 1.1 M | 1102 M | 4.1 GB | 1b word benchmark | 375 M | 3 GB | 17 k | 65 |

that were not involved in a parameter relocation. Gossip protocols [13] could reduce this communication overhead.

**Home node.** Lapse uses a home node strategy, inspired by distributed hash tables [45, 55, 48, 65] and home-based approaches in general [58]. 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 3.3, 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 a centralized home node that knows the locations of all parameters. We discard this strategy because it limits the number of parameters to the size of one node and creates a 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.

## 3.6 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 [14, 56, 63] 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.

## 3.7 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 localizes 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. Mes-

sage grouping adds system complexity, but is very beneficial when clients access or localize sets of parameters at once.

**Local parameter store.** As other PSs [17, 28], 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.

## 4. EXPERIMENTS

We conducted an experimental study to investigate the efficiency of classic PSs (Section 4.2) and whether it is beneficial to integrate PAL techniques into PSs (Section 4.3). Further, we investigated how efficient Lapse is in comparison to a task-specific low-level implementation (Section 4.4) and stale PSs (Section 4.5), and conducted an ablation study (Section 4.6). 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 stale PS (8 nodes were 9x vs. 2.9x faster than 1 node).

## 4.1 Experimental Setup

We considered three popular ML tasks that require long training: matrix factorization, knowledge graph embeddings, and word vectors. Table 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. Appendix A of a long version of this paper [47] provides further details.

**Matrix factorization.** Low-rank matrix factorization is a common tool for analyzing and modeling dyadic data,

e.g., in collaborative filtering for recommender systems [25]. We employed a parameter blocking approach [14] 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 stale PS), and LAPSE. Further, we compared to a task-specific and tuned low-level implementation of this parameter blocking approach.[2] We used two synthetic datasets from [34], because the largest openly available dataset that we are aware of is only 7.6 GB large.

**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 [38]. A vast number of KGE models has been proposed [40, 5, 39, 60, 31], with different training techniques [49]. We studied two models as representatives: RESCAL [40] and ComplEx [57]. We employed data clustering and latency hiding to create and exploit PAL. We used the DBpedia-500k dataset [51], a real-world knowledge graph that contains 490 598 entities and 573 relations of DBpedia [3].

**Word vectors.** Word vectors are a language modeling technique in natural language processing: each word of a vocabulary is mapped to a vector of real numbers [35, 42, 43]. These vectors are useful as input for many natural language processing tasks, for example, syntactic parsing [54] or question answering [32]. In our experimental study, we used the skip-gram Word2Vec [35] model and employed latency hiding to create and exploit PAL. We used the One Billion Word Benchmark [6] dataset, with stop words of the Gensim [46] stop word list removed.

**Implementation and cluster.** We implemented LAPSE in C++, using ZeroMQ and Protocol Buffers for communication, drawing from PS-Lite [28]. Our code is open source and we provide information on how to reproduce our experiments.[3] We ran version 1.1 of Petuum[4] [59] and the version of Sep 1, 2019 of PS-Lite [28]. 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 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 4.6 for details). For all tasks but word vectors, we measured epoch run time, because epochs are identical (or near-identical). This allowed us to conduct experiments in more reasonable time. For word vectors, epochs are not identical because the chosen latency hiding approach changes the sampling distribution of negative samples (see Appendix A of the long version [47]). Thus, we measure model accuracy over time. We calculated model accuracy

---
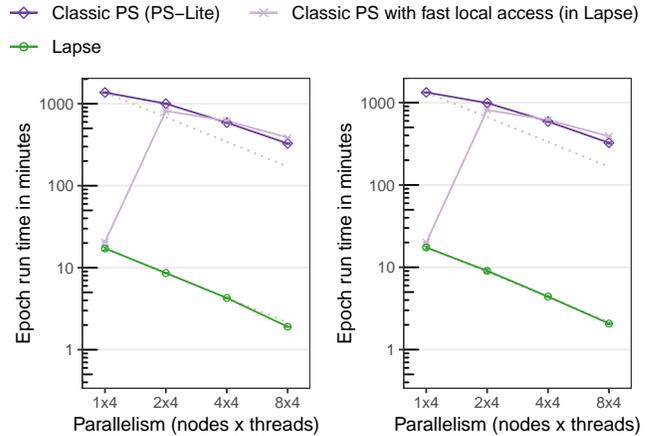


(a) 10m × 1m matrix, 1b entries  (b) 3.4m × 3m matrix, 1b entries

**Figure 6:** Performance results for matrix factorization. LAPSE scaled above-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 4.2. The gray dotted lines indicate linear scaling. Error bars depict minimum and maximum run time (hardly visible here because of low variance).

using a common analogical reasoning task of 19 544 semantic and syntactic questions [35]. 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.

## 4.2 Performance of Classic Parameter Servers

We investigated the performance of classic PSs and how it compares to the performance of efficient single-machine implementations. To this end, we measured the performance of a classic PS on 1–8 nodes for matrix factorization (Figure 6), knowledge graph embeddings (Figure 7), and word vectors (Figure 8). Besides PS-Lite, we ran LAPSE as a classic PS (with shared 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.[5] We omitted PS-Lite from the word vector task due to its 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 vectors. 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

---

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

[3]https://github.com/alexrenz/lapse-ps/

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

**(a)** ComplEx-Small (dim. 100/100)  **(b)** ComplEx-Large (dim. 4000/4000)  **(c)** RESCAL-Large (dim. 100/10 000)
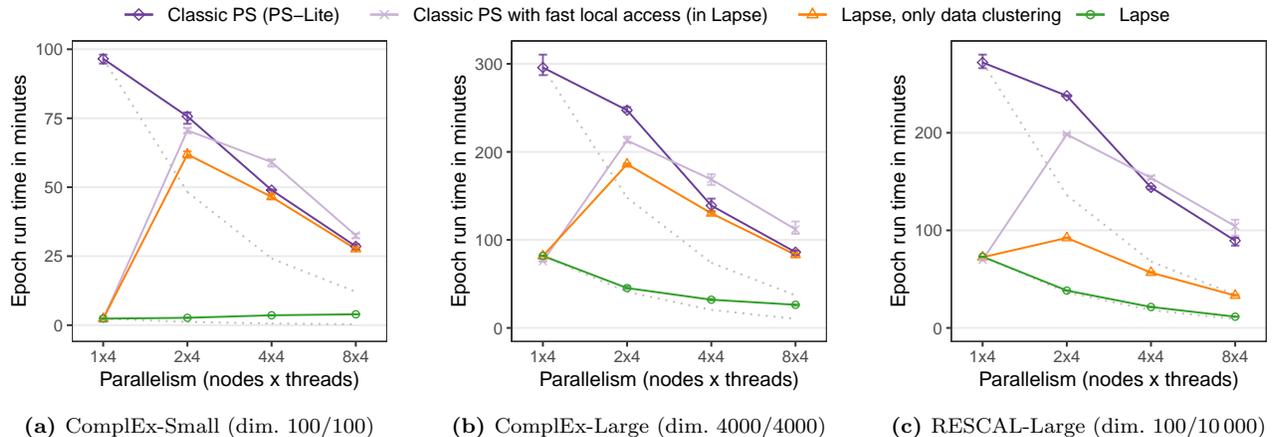
**Figure 7:** Performance results for training knowledge graph embeddings. Distributed training using the classic PS approach did not outperform a single node in any task. LAPSE scaled well for the large tasks (b and c), but not for the small task (a).

part of parameters with low latency, and thus has a performance benefit. Further performance differences stem from differences in the system implementations.

**Single-node performance.** On a single node, the run times of PS-Lite and LAPSE differed significantly (e.g., see Figure 6), 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.[6] 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 task-specific communication-to-computation ratio. The two rightmost columns of Table 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 7a) accessed the PS frequently (312k accessed keys per second) and displayed high communication overhead (8 nodes were 14x slower than 1 node). ComplEx-Large (Figure 7b) accessed the PS less frequently (11k accesses keys per second) and displayed lower communication overhead (8 nodes were 1.4x slower than 1 node).

### 4.3 Effect of Dynamic Parameter Allocation

We compared the performance of LAPSE to a classic PS approach for matrix factorization (Figure 6), KGE (Figure 7), and word vectors (Figure 8). 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

---

[6] 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 5:** Parameter reads, relocations, and relocation times in ComplEx-Large. In this task, each key holds a vector of 8000 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

speed-ups over the single node (see Figure 6). 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 7). It scaled well for the large tasks (ComplEx-Large and RESCAL-Large) despite 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 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 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 7c) more than for ComplEx (Figures 7a and 7b), 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 4000 in ComplEx-Large).

**Word vectors.** For word vectors, LAPSE executed an epoch 44x faster than a classic PS (Figure 8). Further, 8 nodes reached, for example, 39% error 3.9x faster than a single node. The speed-up for word vectors is lower than for knowledge graph embeddings, because word vector training
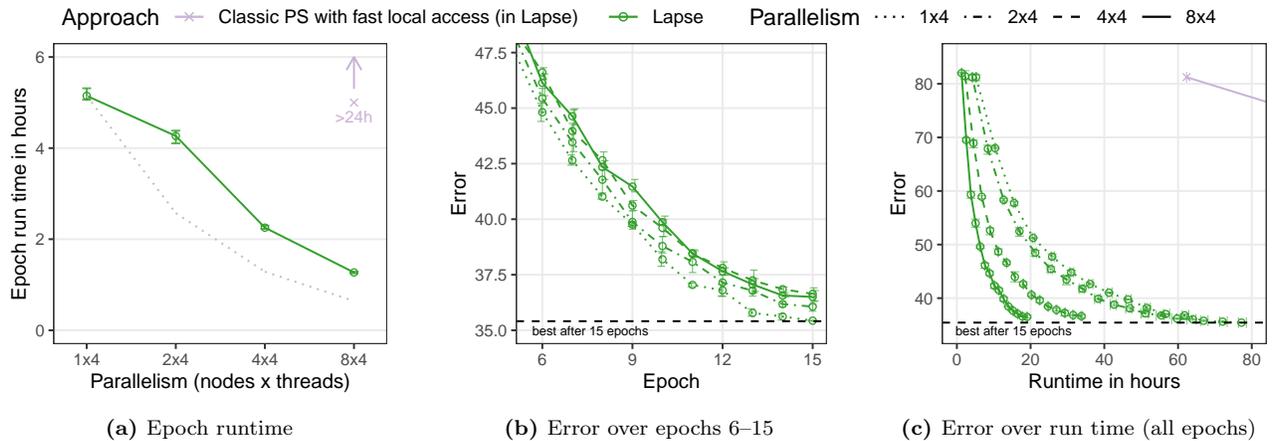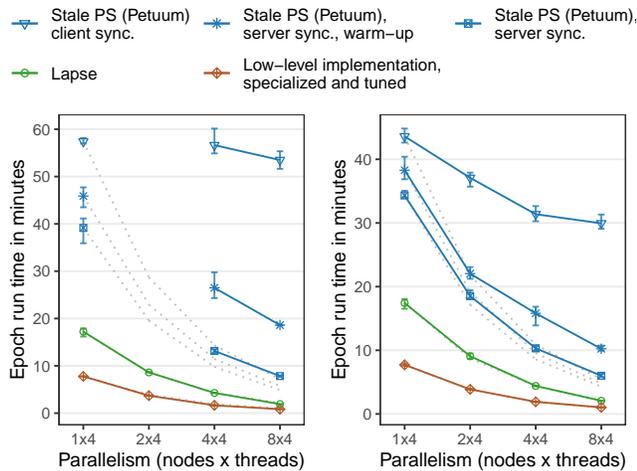
**(a)** Epoch runtime      **(b)** Error over epochs 6–15      **(c)** Error over run time (all epochs)

**Figure 8:** Performance results for training word vectors. 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.



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

**Figure 9:** Performance comparison to manual parameter management and to Petuum, a state-of-the-art stale 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.

exhibits strongly skewed access to parameters: few parameters are accessed frequently [35]. This lead to more frequent localization conflicts in the latency hiding approach than in knowledge graph embeddings, where negative samples are sampled uniformly [49, 31].

## 4.4 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 9). This low-level implementation cannot be used for other ML tasks. Both the low-level implementation and LAPSE scaled linearly (or slightly above-linearly[7]).

---

[7]The reason for the above-linear scaling for the 10m × 1m matrix is that CPU caches work better the more workers are

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 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 allocation required 100s of lines of MPI code. In contrast, in LAPSE, the same allocation required only 4 lines of additional code.

## 4.5 Comparison to Stale PSs

We compared LAPSE to Petuum, a popular stale PS, for matrix factorization (see Figure 9). We found that the stale 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 2.2.2, this can support synchronous parameter blocking algorithms, such as the one we test for matrix factorization. We compared separately to the two synchronization strategies that Petuum provides: client-based and server-based. On the 10m × 1m dataset, Petuum crashed on two nodes with a network error.

**Client-based synchronization.** Client-based synchronization (*SSP* consistency model in Petuum) 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.

---

used, because each worker then focuses on a smaller part of the dataset and the model [56].

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).

**Server-based synchronization.** Server-based synchronization (*SSPPush* consistency model in Petuum) 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 server-based synchronization eagerly synchronizes to a node all parameters that this node accessed previously. On the one hand, this eliminated the network latency overhead of client-based 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 9.

### 4.6 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 6 and 7. 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 7), 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 3.3). 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 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).

### 5. RELATED WORK

We discuss related work on reducing PS communication and dynamic allocation in general-purpose key–value stores.

**Dynamic parallelism**. FlexPS [18] 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 paper. 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.

**PS communication reduction.** We discuss replication and bounded staleness in Section 2. Another approach to reducing communication overhead in PSs is to combine reads and updates of multiple workers locally before sending them to the remote server [9, 8, 17, 18]. However, this technique may break sequential consistency and reduces communication only if different workers access the same parameters at the same time. Application-side communication reduction approaches include sending lower-precision updates [50], prioritizing important parts of updates [21], applying filters to updates [29], and increasing mini-batch size [16]. Many of these approaches can be combined with DPA, which makes for interesting areas of future work. To reduce the load on the PS, STRADS [24] allows for parameter management through task-specific ring-based parameter passing schemes. These schemes can support a subset of parameter blocking approaches, but not data clustering or latency hiding. Developers manually implement such schemes as task-specific additions outside the PS. In contrast, LAPSE integrates PAL support into the PS framework.

**Dynamic allocation in key–value stores.** DAL [37], 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 [61] 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. 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.

### 6. CONCLUSION

We explored whether and to what extent PAL techniques—i.e., techniques that distributed ML algorithms employ to reduce communication overhead—can be supported in PSs, and whether such support is beneficial. To this end, we introduced DPA to PSs and described LAPSE, an implementation of a PS with DPA. We found that DPA can reduce the communication overhead of PSs significantly, achieving up to linear scaling.

With these results, a next step is to facilitate the use of PAL methods in PSs. For example, a possible direction is to research how PSs can automatically use latency hiding through pre-localization. Another area for future work is to improve the scalability of the latency hiding technique. The main bottleneck for this technique were localization conflicts on frequently accessed parameters. Alternative management approaches might be more suitable for these parameters.

# 7. REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. OSDI '16.

[2] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, Alexander Smola. Scalable inference in latent variable models. WSDM '12.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, Zachary Ives. DBpedia: A nucleus for a web of open data. ISWC '07.

[4] Alex Beutel, Partha Pratim Talukdar, Abhimanu Kumar, Christos Faloutsos, Evangelos Papalexakis, Eric Xing. FlexiFaCT: Scalable flexible factorization of coupled tensors on Hadoop. SDM '14.

[5] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. NIPS '13.

[6] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. EuroSys '15.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[9] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, Karthik Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. OSDI '14.

[10] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory Ganger, Phillip Gibbons, Garth Gibson, Eric Xing. Exploiting iterative-ness for parallel ML computations. SOCC '14.

[11] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, Eric P Xing. High-performance distributed ML at scale through parameter server consistency models. AAAI '15.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc Le, Mark Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Ng. Large scale distributed deep networks. NIPS '12.

[13] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry. Epidemic algorithms for replicated database maintenance. PODC '87.

[14] Rainer Gemulla, Erik Nijkamp, Peter Haas, Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. KDD '11.

[15] Joseph Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. OSDI '12.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[17] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip Gibbons, Garth Gibson, Gregory Ganger, Eric Xing. More effective distributed ML via a stale synchronous parallel parameter server. NIPS '13.

[18] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, James Cheng. FlexPS: Flexible parallelism control in parameter server architecture. *PVLDB*, 11(5):566–579, 2018.

[19] Phillip Hutto, Mustaque Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. ICDCS '90.

[20] Rolf Jagerman, Carsten Eickhoff, Maarten de Rijke. Computing web-scale topic models using an asynchronous parameter server. SIGIR '17.

[21] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. *CoRR*, abs/1905.03960, 2019.

[22] Jiawei Jiang, Bin Cui, Ce Zhang, Lele Yu. Heterogeneity-aware distributed parameter servers. SIGMOD '17.

[23] Jin Kyu Kim, Abutalib Aghayev, Garth Gibson, Eric Xing. STRADS-AP: Simplifying distributed machine learning programming without introducing a new programming model. USENIX '19.

[24] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth Gibson, Eric Xing. STRADS: A distributed framework for scheduled model parallel machine learning. EuroSys '16.

[25] Yehuda Koren, Robert Bell, Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):3037, 2009.

[26] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[27] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, Alex Peysakhovich. PyTorch-BigGraph: A large-scale graph embedding system. SysML '19.

[28] Mu Li, David Andersen, Jun Woo Park, Alexander Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene Shekita, Bor-Yiing Su. Scaling distributed machine learning with the parameter server. OSDI '14.

[29] Mu Li, David Andersen, Alexander Smola, Kai Yu. Communication efficient distributed machine learning with the parameter server. NIPS '14.

[30] Richard J Lipton, Jonathan S Sandberg. PRAM: A scalable shared memory. Technical report, Princeton University, Department of Computer Science, 1988.

[31] Hanxiao Liu, Yuexin Wu, Yiming Yang. Analogical inference for multi-relational embeddings. ICML '17.

[32] Xiaodong Liu, Yelong Shen, Kevin Duh, Jianfeng Gao. Stochastic answer networks for machine reading comprehension. ACL '18.

[33] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, Joseph Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[34] Faraz Makari, Christina Teflioudi, Rainer Gemulla, Peter Haas, Yannis Sismanis. Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems*, 42(3):493–523, 2015.

[35] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. Efficient estimation of word representations in vector space. ICLR '13.

[36] Supun Nakandala, Yuhao Zhang, Arun Kumar. Cerebro: Efficient and reproducible model selection on deep learning systems. DEEM '19.

[37] Gábor Németh, Dániel Géhberger, Péter Mátray. DAL: A locality-optimizing distributed shared memory system. HotCloud '17.

[38] Maximilian Nickel, Kevin Murphy, Volker Tresp, Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[39] Maximilian Nickel, Lorenzo Rosasco, Tomaso Poggio. Holographic embeddings of knowledge graphs. AAAI '16.

[40] Maximilian Nickel, Volker Tresp, Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. ICML '11.

[41] Bo Peng, Bingjing Zhang, Langshi Chen, Mihai Avram, Robert Henschel, Craig Stewart, Shaojuan Zhu, Emily Mccallum, Lisa Smith, Tom Zahniser, et al. HarpLDA+: Optimizing latent dirichlet allocation for parallel efficiency. BigData '17.

[42] Jeffrey Pennington, Richard Socher, Christopher Manning. GloVe: Global vectors for word representation. ACL '14.

[43] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer. Deep contextualized word representations. NAACL '18.

[44] Parameswaran Raman, Sriram Srinivasan, Shin Matsushima, Xinhua Zhang, Hyokun Yun, S.V.N. Vishwanathan. Scaling multinomial logistic regression via hybrid parallelism. KDD '19.

[45] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. A scalable content-addressable network. SIGCOMM '01.

[46] Radim Řehůřek, Petr Sojka. Software framework for topic modelling with large corpora. LREC '10.

[47] Alexander Renz-Wieland, Rainer Gemulla, Steffen Zeuch, Volker Markl. Dynamic parameter allocation in parameter servers. *CoRR*, abs/2002.00655, 2020.

[48] Antony Rowstron, Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Middleware '01.

[49] Daniel Ruffinelli, Samuel Broscheit, Rainer Gemulla. You can teach an old dog new tricks! On training

[50] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs. Interspeech '14.

[51] Baoxu Shi, Tim Weninger. Open-world knowledge graph completion. AAAI '18.

[52] Alan Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[53] Alexander Smola, Shravan Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1-2):703–710, 2010.

[54] Richard Socher, John Bauer, Christopher Manning, Andrew Ng. Parsing with compositional vector grammars. ACL '13.

[55] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM '01.

[56] Christina Teflioudi, Faraz Makari, Rainer Gemulla. Distributed matrix completion. ICDM '12.

[57] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, Guillaume Bouchard. Complex embeddings for simple link prediction. ICML '16.

[58] Maarten van Steen, Andrew Tanenbaum. *Distributed Systems*. 3rd edition, 2017.

[59] Eric Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. KDD '15.

[60] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, Li Deng. Embedding entities and relations for learning and inference in knowledge bases. ICLR '15.

[61] Fan Yang, Jinfeng Li, James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[62] Hsiang-Fu Yu, Cho-Jui Hsieh, Hyokun Yun, S.V.N. Vishwanathan, Inderjit Dhillon. A scalable asynchronous distributed algorithm for topic modeling. WWW '15.

[63] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S.V.N. Vishwanathan, Inderjit Dhillon. NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.

[64] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, Xupeng Miao. PS2: Parameter server on Spark. SIGMOD '19.

[65] Ben Zhao, Ling Huang, Jeremy Stribling, Sean Rhea, Anthony Joseph, John Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

knowledge graph embeddings. ICLR '20.