Just Move It! Dynamic Parameter Allocation in Action

Alexander Renz-Wieland
Tobias Drobisch
Zoi Kaoudi
Technische Universität Berlin
firstname.lastname@tu-berlin.de

Rainer Gemulla
Universität Mannheim
rgemulla@uni-mannheim.de

Volker Markl
Technische Universität Berlin
German Research Center for Artificial Intelligence (DFKI)
volker.markl@tu-berlin.de

ABSTRACT

Parameter servers (PSs) ease the implementation of distributed machine learning systems, but their performance can fall behind that of single machine baselines due to communication overhead. We demonstrate Lapse, an open source PS with *dynamic parameter allocation*. Previous work has shown that dynamic parameter allocation can improve PS performance by up to two orders of magnitude and lead to near-linear speed-ups over single machine baselines. This demonstration illustrates how Lapse is used and why it can provide order-of-magnitude speed-ups over other PSs. To do so, this demonstration interactively analyzes and visualizes how dynamic parameter allocation looks like in action.

PVLDB Reference Format:
Alexander Renz-Wieland, Tobias Drobisch, Zoi Kaoudi, Rainer Gemulla, and Volker Markl. Just Move It! Dynamic Parameter Allocation in Action. PVLDB, 14(12): 2707 - 2710, 2021.
doi:10.14778/3476311.3476325

1 INTRODUCTION

Distributed training has become a necessity for large machine learning (ML) tasks to keep up with increasing dataset size and model complexity. In distributed ML, both training data and model parameters are partitioned across a compute cluster. Each node usually accesses only its local part of the training data, but reads and/or updates most of the model parameters. Applications either manage model parameters manually using low-level distributed programming 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 [1, 4, 5, 14–16], and there exist multiple standalone PSs [10–13, 17, 25].

In prior work, we have argued that traditional PSs provide limited scalability and their performance can even fall behind that of single machine baselines [21]. The key problem is that the majority of parameter accesses in traditional PSs involves network communication. *Dynamic parameter allocation* can reduce this communication overhead drastically, and thus provide up to linear speed-ups over single machine baselines and outperform prior state-of-the-art PSs by up to one order of magnitude [21]. Dynamic allocation allocates and re-allocates parameters where they are accessed, while providing location transparency and PS consistency guarantees, i.e., sequential consistency. Dynamic allocation reduces communication overhead because it allows for exploiting common techniques that increase *parameter access locality* [3, 8, 9, 16, 18–20, 23, 24]. Intuitively, these techniques ensure that most parameter accesses do not require (synchronous) communication; example techniques include exploiting natural clustering of data, parameter blocking, and latency hiding.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476325

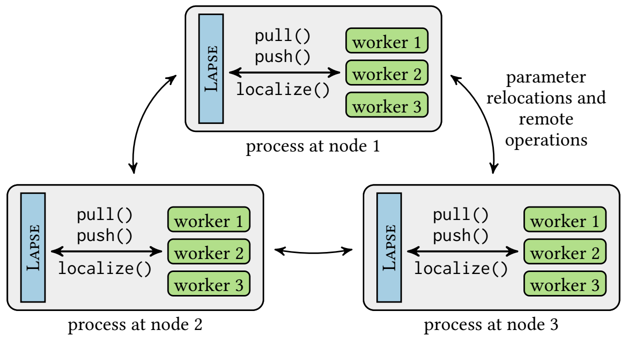


Figure 1: Architecture of Lapse. Workers interact with Lapse via the `pull`, `push`, and `localize` primitives. Lapse relocates parameters dynamically (and transparently) among nodes to reduce communication overhead.

We demonstrate Lapse, the first PS that supports dynamic parameter allocation. The demonstration is available as an interactive web application at **https://alexrenz.github.io/dpa-in-action**. Lapse itself is open source and also available online.[1] The demonstration illustrates parameter access locality techniques and dynamic allocation, how Lapse can achieve the observed order-of-magnitude speed-ups, and how one can use Lapse. To do so, the demonstration visualizes what dynamic allocation looks like “in action”. It allows for interactively inspecting static and dynamic parameter allocation in several ML training tasks and comparing them against each other. Similarly, it allows for inspecting different parameter access locality techniques and comparing their effects against each other. The demonstration provides analysis tools to (1) replay parameter allocation, (2) map allocation over time, and (3) calculate parameter affinity to different nodes of the cluster. These analyses can be applied on provided traces of three distributed ML tasks, namely knowledge graph embeddings, matrix factorization, and word vectors. Using these tasks as examples, the demonstration explains how different parameter access locality techniques can

[1]The source code of Lapse is available at https://github.com/alexrenz/lapse-ps.

be implemented in Lapse. Advanced users can additionally upload traces of their own distributed ML tasks and use the analysis tools to identify patterns and investigate bottlenecks. As an entry point for the majority of users, we provide an interactive tour through the most interesting analyses of the demonstration.

2 PARAMETER SERVERS

PSs [2, 7, 10, 17, 22] partition the parameters of an ML model 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, a PS assigns to each parameter a unique *key*. The PS provides `pull` and `push` primitives for parameter reads and writes, respectively. Both operations can be performed synchronously or asynchronously. Although servers and workers can in theory reside on different machines, they are often co-located for efficiency reasons. Some PS architectures [12, 13, 17] run one server process and one or more worker processes on each machine, others [10, 11], including Lapse [21], combine both server and worker threads into the same process to reduce inter-process communication.

Parameter allocation. The *classic PS* architecture allocates parameters to servers *statically* (e.g., via a range partitioning of the parameter keys) and employs no replication [2, 17, 22]. Thus, exactly one server holds the current value of a parameter, and this server processes all pull and push operations for this parameter. Classic PSs typically guarantee per-key sequential consistency [21]. 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. The *stale PS* architecture also allocates each parameter statically to one server, but the PS may replicate a subset of parameters to additional servers to reduce communication overhead (by tolerating some amount of staleness in the replicas) [6, 10, 11, 13]. The *dynamic allocation PS* architecture allocates and re-allocates parameters dynamically to different servers.

3 LAPSE

In this section, we give an overview of the architecture of Lapse and discuss three parameter access locality techniques that Lapse exploits to improve performance by orders of magnitude.

3.1 Architecture Overview

Lapse [21] is the first PS that supports dynamic parameter allocation. It can re-allocate parameters among cluster nodes during run time. It maintains the semantics of the PS API: `pull` and `push` operations can be issued for any key on any node at any time. The operations provide correct results regardless of a parameter's current allocation and whether or not the parameter is currently being relocated. Lapse provides an additional `localize` primitive that allows workers to initiate parameter relocations to their node. Upon invocation, Lapse transparently relocates the requested parameters to the worker's node, such that future accesses by the worker require no further network communication. These relocations take place in the background and can be initiated *before* a parameter is actually accessed. Lapse ensures that access to local parameters is fast by co-locating worker and server threads in one process per node and using shared memory to access local parameters. Lapse provides sequential consistency, the same consistency guarantee as classic PSs [21]. Figure 1 provides an overview of the architecture.

3.2 Parameter Access Locality Techniques

We describe three *parameter access locality* (PAL) techniques that Lapse utilizes to improve performance. PAL techniques are often used in distributed ML algorithms to reduce communication overhead by increasing locality in parameter accesses. Lapse can exploit these PAL techniques because it allocates parameters dynamically. This results in up to two orders of magnitude better performance for Lapse compared to other PSs, which allocate parameters statically and thus cannot exploit PAL [21].

3.2.1 *Data Clustering.*

A common PAL technique is to exploit structure in training data [2, 9, 18, 22]. 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. Distributed ML applications can exploit this fact through *data clustering*. For example, if a separate worker is used for the documents of each language, different workers access mostly separate parameters.

To exploit locality from data clustering, Lapse allocates each parameter to the machine that accesses the parameter most frequently. Figure 2a depicts an example of how Lapse allocates parameters when data clustering is used. The figure shows parameter allocation over time. Each row corresponds to one parameter, the x-axis depicts time, and colors indicate the current allocation of a parameter. Initially, parameters are range-partitioned by key (the default parameter allocation); then, parameters are relocated according to the clustering once, such that each parameter is allocated at the node where it is accessed most frequently. Each parameter then remains allocated at this node throughout the task.

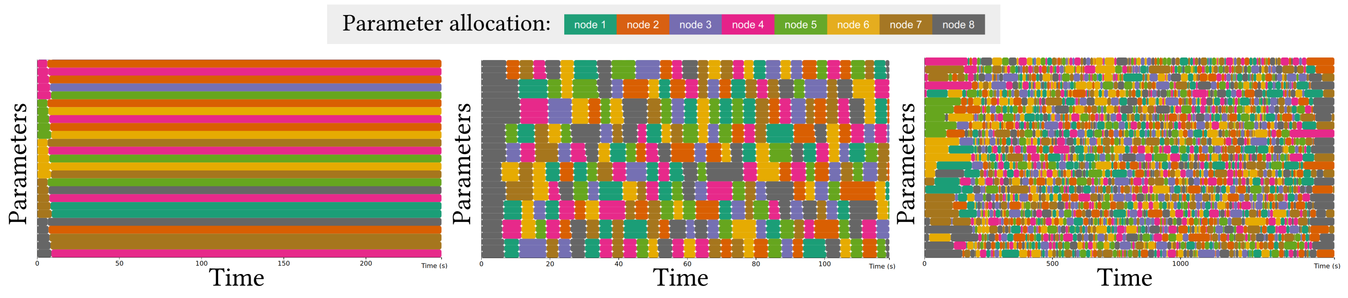
3.2.2 *Parameter Blocking.*

Another common PAL technique 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. Such *parameter blocking* approaches have been developed for a variety of ML algorithms [3, 8, 16, 19, 20, 23, 24].

To exploit locality from parameter blocking, Lapse allocates each parameter to the node where it is currently accessed. This eliminates network communication for individual parameter accesses. Communication is required only for parameter relocations between subepochs. Figure 2b depicts an example of how Lapse allocates parameters when parameter blocking is used. The parameters are relocated periodically, between subepochs. At the end of a subepoch, the each parameter block is relocated to the worker that accesses it in the next subepoch.

3.2.3 *Latency Hiding.*

In distributed ML, *latency hiding* can reduce communication overhead (but not communication itself) by ensuring that a parameter value is already present at a worker at the time



(a) Data clustering

(b) Parameter blocking

(c) Latency hiding

Figure 2: Three techniques that increase *parameter access locality*, as visible in the *allocations over time* tool of this demonstration. Each row corresponds to one parameter, the x-axis depicts time, and colors indicate the current allocation of a parameter. (a) The training data are clustered such that each worker accesses mostly a separate subset of parameters. (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. (c) Asynchronously prelocalizing parameters, such that they can be accessed locally, hides access latency.

it is accessed [6, 23]. Such an approach is beneficial when parameter access is sparse, i.e., each worker accesses few parameters at a time.

To hide latency, Lapse *pre*localizes a parameter before access, i.e., it reallocates each parameter from its current node to the node where it will be accessed and keeps it there afterwards (until some other worker accesses it). In contrast to parameter prefetching, prelocalization does not replicate parameters. Consequently, parameter updates by other workers are immediately visible. Moreover, there is no need to write local updates back to a remote location as the parameter is now stored locally. Figure 2c depicts an example of how Lapse allocates parameters when latency hiding is used: parameters relocate frequently, with no visible allocation patterns. Nevertheless, latency hiding has provided near-linear speed-ups in prior experiments [21].

4 DEMONSTRATION

This demonstration includes multiple analysis tools, combined in one interactive web application. To provide effective learning experiences for users with different levels of prior topic knowledge, we designed the demonstration to provide three different user scenarios: free exploration, guided exploration, and advanced usage. In the following, we go through each of these scenarios.

4.1 Free Exploration

Step ❶: Task selection. The user selects which ML task they want to investigate. They can choose from a set of provided traces or upload one of their own traces (see Section 4.3). The demonstration provides traces from several ML tasks (training knowledge graph embeddings, word vectors, and matrix factorization), each with either static or dynamic parameter allocation. Different tasks employ different PAL techniques.

Step ❷: Task-specific information. The user familiarizes themselves with the chosen task by reading a brief summary of information about the chosen task: e.g., the type of parameter allocation, the used PAL techniques, the dataset, and the number of nodes and workers. They can also inspect performance results of static and dynamic allocation for this task.

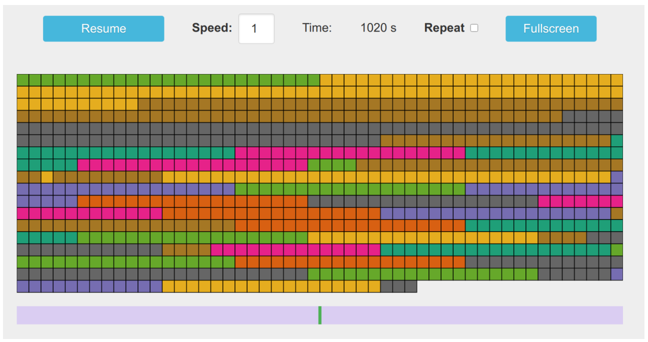


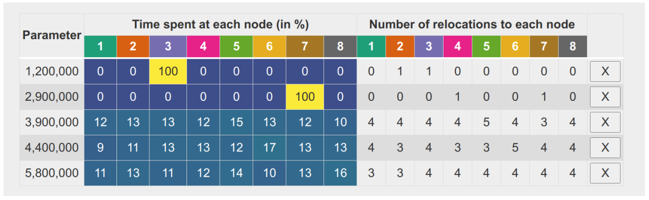
Figure 3: Example screenshot of the *allocation replay* analysis tool. Each square corresponds to one parameter. The square's color indicates at which node the parameter is currently allocated. The user can adjust replay speed and jump to any point in time, using the progress bar at the bottom.

Step ❸: Allocation replay. The user replays the parameter allocations. They can pause the replay to inspect the allocation at a specific point in time. They can also jump to any point in time (using the time bar below the allocation plot) and specify the replay speed. Figure 3 shows an example screenshot of this analysis tool.

Step ❹: Allocation over time. The user plots an overview of parameter allocation over time. By default, the plot is generated for a subset of the selected parameters. The user can choose to plot allocations for all parameters. Figure 2 shows example screenshots of this analysis tool.

Step ❺: Parameter affinity. The user investigates the affinity of parameters to nodes. To do so, they select any parameter in any of the two previous analysis tools. This tool then shows this parameter's affinity: how long the parameter resided at and how often it was relocated to each node. Multiple parameters can be added (and removed) to the tool to compare affinity of different parameters. Figure 4 shows an example screenshot of this tool.

Step ❻: Static vs. dynamic allocation. Both static and dynamic allocation runs are provided for every ML task, such that the



Parameter	Time spent at each node (in %)								Number of relocations to each node								
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	
1,200,000	0	0	100	0	0	0	0	0	0	1	1	0	0	0	0	0	X
2,900,000	0	0	0	0	0	0	100	0	0	0	0	1	0	0	1	0	X
3,900,000	12	13	13	12	15	13	12	10	4	4	4	4	5	4	3	4	X
4,400,000	9	11	13	13	12	17	13	13	4	3	4	3	3	5	4	4	X
5,800,000	11	13	11	12	14	10	13	16	3	3	4	4	4	4	4	4	X

Figure 4: Example screenshot of the *parameter affinity* analysis tool. It displays the time spent at each node and the number of relocations to each node for interactively selected parameters.

user can compare allocation for these strategies. For example, they can compare a trace of training knowledge graph embeddings with dynamic allocation to one with static allocation.

4.2 Guided Exploration

We provide a guided tour that takes the user through a series of steps to learn about static and dynamic parameter allocation, Lapse, and PAL techniques.

Step ❶: Background. This step introduces distributed ML training and the use of PSs.

Step ❷: Static parameter allocation. This step introduces static parameter allocation and the communication overhead that is associated with it. The user is introduced to the *allocation replay* and *allocation over time* analysis tools and inspects a static allocation job in both tools.

Step ❸: Dynamic parameter allocation and Lapse. This step introduces dynamic parameter allocation, features of Lapse, and how Lapse is used. The user then inspects an example of dynamic parameter allocation in the *allocation over time* analysis tool.

Step ❹: Matrix factorization (clustering and parameter blocking). This step introduces matrix factorization, which PAL techniques are used for which parameters in the provided task, and how the `localize` primitive is used to exploit these in Lapse. It presents performance results of static and dynamic parameter allocation on the provided matrix factorization task. The user inspects the dynamic allocation run in the *allocation over time* analysis tool and is then introduced to the *parameter affinity* analysis tool.

Step ❺: Knowledge graph embeddings (clustering and latency hiding). This step introduces knowledge graph embeddings, which PAL techniques are used for which parameters in the provided task, and how these are implemented in Lapse. It presents performance results of static and dynamic parameter allocation on the provided knowledge graph embeddings task. The user inspects the dynamic allocation run in the different analysis tools.

4.3 Advanced Usage

Step ❶: User-provided tasks. An advanced user may upload traces of their own ML jobs. They can collect a trace for any task in Lapse by enabling a compiler switch in Lapse (the demonstration tool provides specific information on how to do this).

Step ❷: Parameter focus. An advanced user can focus the analysis on specific parameter subgroups. This is particularly useful for large traces that include millions of parameters. The user includes specific parameters by specifying a list of individual keys and/or key ranges. By default, all parameters of a trace are included in the subsequent analyses.

Steps ❸ to ❺: Task analysis. After upload, the user can use the same analysis tools as for the provided tasks (Steps ❸ to ❺ in Section 4.1). The analyses are run on the selected parameter subset of the uploaded task.

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, et al. TensorFlow: A system for large-scale machine learning. OSDI '16, pp. 265–283.
[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, A. Smola. Scalable inference in latent variable models. WSDM '12, pp. 123–132.
[3] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. Papalexakis, E. Xing. FlexiFaCT: Scalable flexible factorization of coupled tensors on Hadoop. SDM '14, pp. 109–117.
[4] T. Chen, M. Li, Y. Li, et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.
[5] T. Chilimbi, Y. Suzue, J. Apacible, K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. OSDI '14, p. 571–582.
[6] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, E. P. Xing. High-performance distributed ML at scale through parameter server consistency models. AAAI '15.
[7] J. Dean, G. Corrado, R. Monga, et al. Large scale distributed deep networks. NIPS '12, pp. 1223–1231.
[8] R. Gemulla, E. Nijkamp, P. Haas, Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. KDD '11, pp. 69–77.
[9] J. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. OSDI '12, pp. 17–30.
[10] Q. Ho, J. Cipar, H. Cui, et al. More effective distributed ML via a stale synchronous parallel parameter server. NIPS '13, pp. 1223–1231.
[11] Y. Huang, T. Jin, Y. Wu, et al. FlexPS: Flexible parallelism control in parameter server architecture. *PVLDB*, 11(5):566–579, 2018.
[12] R. Jagerman, C. Eickhoff, M. de Rijke. Computing web-scale topic models using an asynchronous parameter server. SIGIR '17, pp. 1337–1340.
[13] J. Jiang, B. Cui, C. Zhang, L. Yu. Heterogeneity-aware distributed parameter servers. SIGMOD '17, p. 463–478.
[14] J. Kim, Q. Ho, S. Lee, et al. STRADS: A distributed framework for scheduled model parallel machine learning. EuroSys '16, pp. 5:1–5:16.
[15] J. K. Kim, A. Aghayev, G. Gibson, E. Xing. STRADS-AP: Simplifying distributed machine learning programming without introducing a new programming model. USENIX '19, pp. 207–222.
[16] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, A. Peysakhovich. PyTorch-BigGraph: A large-scale graph embedding system. SysML '19.
[17] M. Li, D. Andersen, J. W. Park, A. Smola, A. Ahmed, V. Josifovski, J. Long, E. Shekita, B.-Y. Su. Scaling distributed machine learning with the parameter server. OSDI '14, pp. 583–598.
[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.
[19] B. Peng, B. Zhang, L. Chen, M. Avram, R. Henschel, C. Stewart, S. Zhu, E. Mccallum, L. Smith, T. Zahniser, et al. HarpLDA+: Optimizing latent dirichlet allocation for parallel efficiency. BigData '17, pp. 243–252.
[20] P. Raman, S. Srinivasan, S. Matsushima, X. Zhang, H. Yun, S. Vishwanathan. Scaling multinomial logistic regression via hybrid parallelism. KDD '19, pp. 1460–1470.
[21] A. Renz-Wieland, R. Gemulla, S. Zeuch, V. Markl. Dynamic parameter allocation in parameter servers. *PVLDB*, 13(12):1877–1890, 2020.
[22] A. Smola, S. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1-2):703–710, 2010.
[23] C. Teflioudi, F. Makari, R. Gemulla. Distributed matrix completion. ICDM '12, pp. 655–664.
[24] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, I. Dhillon. NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.
[25] Z. Zhang, B. Cui, Y. Shao, L. Yu, J. Jiang, X. Miao. PS2: Parameter server on Spark. SIGMOD '19, pp. 376–388.