FlexPS: Flexible Parallelism Control in Parameter Server Architecture

Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, James Cheng
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{yzhuang, tjin, ydwu, zkcai, xyan, fyang, jfli, yyguo, jcheng}@cse.cuhk.edu.hk

ABSTRACT

As a general abstraction for coordinating the distributed storage and access of model parameters, the parameter server (PS) architecture enables distributed machine learning to handle large datasets and high dimensional models. Many systems, such as Parameter Server and Petuum, have been developed based on the PS architecture and widely used in practice. However, none of these systems supports changing parallelism during runtime, which is crucial for the efficient execution of machine learning tasks with dynamic workloads. We propose a new system, called FlexPS, which introduces a novel multi-stage abstraction to support flexible parallelism control. With the multi-stage abstraction, a machine learning task can be mapped to a series of stages and the parallelism for a stage can be set according to its workload. Optimizations such as stage scheduler, stage-aware consistency controller, and direct model transfer are proposed for the efficiency of multi-stage machine learning in FlexPS. As a general and complete PS systems, FlexPS also incorporates many optimizations that are not limited to multi-stage machine learning. We conduct extensive experiments using a variety of machine learning workloads, showing that FlexPS achieves significant speedups and resource saving compared with the state-of-the-art PS systems such as Petuum and Multiverso.

PVLDB Reference Format:
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 - 679, 2018.
DOI: https://doi.org/10.1145/3177732.3177734

1. INTRODUCTION

Machine learning nowadays often needs to handle large datasets and high dimensional models that exceed the storage and processing capability of a single machine. Distributed machine learning offers a solution by utilizing the aggregate capability of a cluster of machines to finish large-scale tasks under a reasonable time budget.

Our work focuses on a class of widely-used machine learning models that adopt an *iterative-convergent* solution, that is, the model is defined by a set of model parameters which is refined iteratively until convergence in the training process. Representatives of these machine learning models include logistic regression, SVM, matrix factorization, latent Dirichlet allocation, k-means, and neural networks. Algorithms such as stochastic gradient descent (SGD), stochastic coordinate descent (SCD), alternating least squares (ALS) [24], Gibbs sampling [15], and recently proposed variance-reduced stochastic gradient descent methods [21, 4, 3] are widely used to train these models.

Among the distributed machine learning frameworks proposed in recent years, the *parameter server* (*PS*) abstraction [38, 12, 10, 26, 30, 43, 45] is a natural fit for iterative tasks and has been regarded as the de facto solution for distributed machine learning. PS separates the working units into *workers* and *servers*. The servers serve as a distributed storage of the model parameters, and the workers update the model in parallel with their partition of the training data. A simple key-value-store (KV-store) interface is provided for users, while details about the underlying system such as synchronization, and consistency control are abstracted away from users.

Significant efforts have been made to improve the efficiency and usability of the PS architecture from various aspects such as consistency control [17, 26], network communication [43], fault tolerance [26], and straggler problem [16]. Many PS-based systems, including Petuum [43, 45], Parameter Server [26], Multiverso [30], etc., have been developed and widely used in industry and academia. We provide a brief introduction to some representative PS systems and their key design concerns and ideas in Section 2. However, all these systems overlooked an important feature, namely *flexible parallelism control*, which is crucial for the efficient execution of iterative machine learning algorithms with dynamic workloads.

Many important machine learning algorithms, including stochastic gradient descent (with increasing batch size) [7, 8, 14] and the more efficient variance-reduced stochastic gradient descent methods (e.g., SVRG [21], SVRG++ [4], Katyusha [3], etc.) have varying workloads during the process of their execution. For distributed machine learning with the PS architecture, it is crucial to set the right parallelism degree that balances between the computation time and the communication overhead for efficiency. As the workload is an important factor in determining the parallelism degree, the varying workloads of these algorithms suggest that we need to change the parallelism degree dynamically in the execution process. However, existing PS systems only adopt a constant parallelism degree (for a task) and do not support changing the parallelism degree during runtime. As we will show in Section 3, the lack of flexible parallelism can hinder the performance of distributed machine learning significantly, resulting in not only long running time but also waste of resources.

To enable flexible parallelism control, we propose a novel **multi-stage abstraction** for iterative machine learning and design a new PS system named **FlexPS** based on it. In the multi-stage abstraction, a machine learning task is viewed as the composition of a series of stages and the stages can have distinct parallelism degrees. A machine learning task with dynamic workloads can be mapped to multiple stages, and each stage adopts its individual parallelism degree according to its workload. For the efficient execution of multi-stage machine learning tasks, we introduce system designs including stage scheduler, stage-aware consistency controller and data store. We find that small parallelism degree offers good performance for the types of algorithms like SGD and SVRG when the batch size is small and the worker threads can be placed on the same machine. For such small parallelism cases, we provide optimizations including local consistency controller, flexible model preparation, and direct model transfer for efficiency. To eliminate the burden of setting the parallelism degree of each stage explicitly, we provide a module in FlexPS that can adjust the parallelism degree automatically during runtime.

In addition to supporting flexible parallelism, FlexPS is a general and complete PS system that also supports machine learning tasks with static workloads as existing PS systems were designed for. Therefore, optimizations that are not limited to the support of flexible parallelism, such as customizable parameter partition, customizable KV-server, and repetitive *get* avoidance, are also incorporated into the system design.

We summarize our main contributions as follows.

- We identify the need for *flexible parallelism control* of algorithms with dynamic workloads (Section 3). These algorithms include SGD (which is one of the most commonly used algorithms in machine learning) and the more efficient variance-reduced algorithms.

- We propose a novel *multi-stage abstraction* (Section 4), which not only provides inherent support for flexible parallelism control but also brings benefits including data locality and fine-grained scheduling.

- We develop a new PS system (Sections 5), *FlexPS*, and devise tailored system designs, such as stage scheduler, data store and direct model transfer, for the efficient execution of the multi-stage abstraction. As a complete system, FlexPS also incorporates many optimizations (Sections 6) for general machine learning.

- We conduct extensive experiments (Section 7) to evaluate the performance of FlexPS and compare it with the state-of-the-art PS systems such as Petuum and Multiverso on various machine learning tasks. The results show that FlexPS yields significant performance improvement.

2. BACKGROUND

In this section, we give a brief introduction to the PS architecture and some representative PS systems.

2.1 Parameter Server Architecture

In the PS architecture, there are two types of entities, i.e., *workers* and *servers*. The model parameters are stored distributedly in servers, while the training data are partitioned among workers. Servers provide a KV-store interface for workers to access the model parameters. Workers use this interface to read (part of) the model from servers, perform computation such as calculating stochastic gradient using (local) training data, and then write the updates back to servers. Servers aggregate the local updates from workers to update the global model, for example, servers can average the stochastic gradients from workers to conduct full gradient descent.

A key issue in the PS architecture is how to manage the synchronization of the model replicas in workers. There are three typical consistency protocols: Bulk Synchronous Parallel (BSP) [41], Stale Synchronous Parallel (SSP) [17, 26], and Asynchronous Parallel (ASP). BSP enforces a global barrier after each iteration, and thus guarantees that all updates from the previous iteration can be seen by all workers in the current iteration. Under BSP, distributed machine learning follows the same execution logic as sequential algorithms on a single machine, making the proof of algorithm correctness simple. As BSP is prone to the straggler problem, SSP was proposed by relaxing BSP to allow the fastest workers to be s iterations ahead of the slowest workers. ASP completely removes the synchronization requirement and can be adopted for algorithms such as SGD [32], SCD [28], and Gibbs sampling [34]. BSP and ASP can be viewed as special cases of SSP with $s = 0$ and $s = \infty$, respectively.

2.2 Parameter Server Systems

As a natural abstraction for distributed machine learning, the PS architecture is the basis of many systems. We call these systems *PS systems* and briefly introduce some representative ones.

Application-specific systems. Initially, the PS architecture is used for scaling specific machine learning applications. YahooLDA [38] stores the latent factors distributedly and enables workers to update them in an ad hoc manner. DistBelief [12] and Project Adam [10] partition the neural network weights among the machines and each worker can read and send updates to a specific part of the weights. These systems do not target general machine learning tasks and are often optimized for specific applications.

Parameter Server. Parameter Server [26] is a PS based general distributed machine learning system that introduces a number of optimizations for efficiency and scalability. In Parameter Server, the tasks are conducted in an asynchronous manner to hide communication delay by overlapping communication with computation. A scheduler tracks the progresses of workers to support various consistency protocols. It also enables elastic scalability and continuous fault tolerance with range-based vector clock, two-phase worker/server management and model replication.

Multiverso. Multiverso [30] is a PS system which serves as a core module of the Distributed Machine Learning Toolkit (DMTK) [11] from Microsoft. Multiverso adopts the actor model and separates workers and servers in different processes for clear programming logic. It enables automatic pipelining to overlap data loading and training. As a PS framework, it also provides support for distributed deep learning systems such as Torch [40] and Theano [2]. Applications built on top of Multiverso include LightLDA [50] and Distribtued Word Embedding (DWE). However, Multiverso does not support the SSP protocol.

Bösen. Bösen [43] is a data-parallel PS system and a module in Petuum [45]. It adopts the SSP protocol and provides a table-like client API. A row in the table is the smallest unit for parameter access. In Bösen, an ML program is linked with the client library to read/update the model concurrently. The client library contains a group of background threads to synchronize the local models/updates with servers, while the user threads execute the application logic. For communication, Bösen conducts synchronization aggressively under the bandwidth budget following the leaky bucket algorithm [39].

Other PS systems. Besides the aforementioned systems, there are many other PS systems with diverse characteristics. STRADS [23] extends the PS architecture to support model-parallel ML by scheduling the parameter updates via dependency checking and prioritization. Poseidon [52] is designed for data-parallel deep learning on distributed GPUs. FlexRR [16] addresses the straggler problem using a dynamic peer-to-peer reassignment strategy which enables the fast workers to help the slow workers. Petuum [45] is a versatile system that contains Bösen, STRADS and Poseidon as its modules. Angel [5], developed by Tencent, employs hybrid parallelism which combines data parallelism with model parallelism. There are also attempts to improve the convergence rate of the SSP protocol by assigning weights to the model updates (generated by different machines) according to their lags with respect to the current model [20].

The PS architecture is also used in deep learning systems including MXNet [9] and TensorFlow [1] as the underlying distributed parameter management module. However, the focuses of PS systems and deep learning systems are different. PS systems focus on low-level worker-server communication, while deep learning systems provide a high-level abstraction for users to construct a dataflow graph and focus on optimizing the graph execution.

In conclusion, existing PS systems have attempted to improve the usability and efficiency of the PS architecture from various perspectives including scalability, fault tolerance, support of various consistency protocols, addressing the straggler problem, and good programming interface. However, in the next section, we will show that an important feature, flexible parallelism control, which has not been studied in existing work, can be exploited to achieve significant performance improvement in terms of both efficiency and resource utilization.

3. FLEXIBLE PARALLELISM CONTROL

A fundamental trade-off in PS systems is the computation time and the communication time. Intuitively, for a fixed workload, increasing the parallelism degree, i.e., using a larger number of workers, will reduce the computation time. However, larger parallelism degree also means more workers need to pull/push the model from/to servers, resulting in more network communication and longer communication time. As the *per-iteration delay* of iterative machine learning is the sum of computation time T_p and communication time T_c [1], most PS systems allow users to set a parallelism degree that balances between the computation time and the communication time in order to minimize the per-iteration delay. However, existing PS systems maintain a *constant parallelism degree* in the entire execution process of a task and do not support changing the parallelism degree at runtime. This limitation should be addressed because a large class of machine learning algorithms have dynamic workloads, and changing the parallelism degree correspondingly is crucial for their efficient execution.

Algorithm 1 SVRG

```
Input: Number of epochs $S$, learning rate $\eta$, batch size $b$.
Initialize: $\widetilde{x}^0$.
1: for $s = 1, 2, \ldots, S$ do
2:     $\widetilde{\mu} = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\widetilde{x}^{s-1})$;
3:     for $k = 1, 2, \ldots, m$ do
4:         Pick $b$ samples uniformly at random;
5:         $\widetilde{\nabla} f(x_{k-1}^s) = \frac{1}{b}\sum_{i=1}^{b}\left[\nabla f_i(x_{k-1}^s) - \nabla f_i(\widetilde{x}^{s-1}) + \widetilde{\mu}\right]$;
6:         $x_k^s = x_{k-1}^s - \eta\widetilde{\nabla} f(x_{k-1}^s)$;
7:     end for
8:     $x_0^{s+1} = x_m^s$;
9: end for
```

One important algorithm with dynamic workloads is SGD with growing batch size. SGD is widely used in training a variety of machine learning models including logistic regression, support vector machine, neural networks, etc., but it suffers from oscillation around the optimum. A widely used fix is to increase the batch size in the learning process [7, 8, 14] and a larger batch size means higher workload. As a result, the workload varies during the learning process.

Another example is the variance-reduced stochastic gradient descent (VR-SGD) algorithms such as SVRG [21], SVRG++ [4], and Katyusha [3]. These algorithms came as a major breakthrough in machine learning as they need only $\log 1/\epsilon$ iterations to obtain an ϵ-accurate solution[2] for smooth and strongly convex problems, while SGD needs $1/\epsilon$ iterations. Due to their fast convergence, they are widely used in machine learning applications [13, 27, 44, 37, 33, 36]. One common characteristic of these algorithms is that their workloads vary significantly in the process of execution. We illustrate this fact by SVRG in Algorithm 1. The algorithm needs to compute the full gradient at the start of each epoch (i.e., Line 2) by scanning the entire training dataset. In contrast, only a small number of samples are used for the following stochastic update steps (i.e., Lines 3-7).

Figure 1 plots the per-iteration delay (i.e., Total), computation time and communication time of the training of a logistic regression model with SGD against parallelism

[1]Strictly speaking, this statement is not accurate in a design that can overlap communication with computation. As a fix, we consider the time that workers stand idle waiting for pull/push from/to servers as the communication time.

[2]Denote the cost function as $F(x)$ and the minimum of $F(x)$ as $F(x^\star)$, an ϵ-accurate solution x satisfies $F(x) - F(x^\star) \leq \epsilon$.

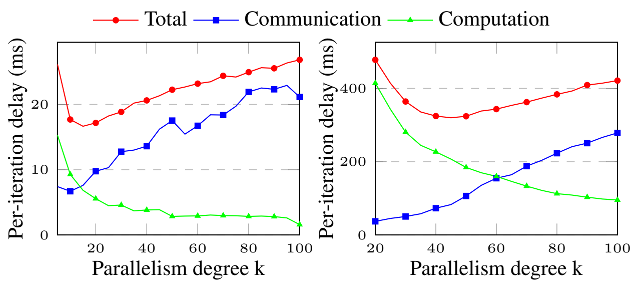


Figure 1: Per-iteration delay versus parallelism degree (best viewed in color). The solid lines and dotted lines represent the actual value and fitted value, respectively.

degrees for batch sizes of 100 and 10000[3]. The results verify our analysis, that is, the computation time decreases with the parallelism degree k, while the communication time increases with k. Moreover, the optimal parallelism degrees for different workloads (batch size) vary greatly. In our case, the optimal parallelism degrees for batch sizes of 100 and 10000 are 15 and 45, respectively. The per-iteration delay curves in Figure 1 are more flat at high parallelism degree, suggesting that setting a high parallelism degree will not degrade the per-iteration delay significantly. However, a considerable amount of resource will be wasted using more workers. For example, if we set the parallelism degree as 90 for a batch size of 10000, we will end up with a per-iteration delay 30% longer than the optimum and 2.6x resource consumption.

Limitations of existing PS systems. Existing PS systems are inefficient in supporting flexible parallelism control. For these systems, a straightforward way to change the parallelism degree is to dump the model parameters to disk, then start a new job with the desired parallelism degree and reload the model parameters. However, the cost of disk I/O and task initialization may outweigh the benefits of flexible parallelism control. We tested the parallelism degree adjustment delay of Multiverso and Bösen by model dumping, and found that the delay is typically more than 60 seconds [4]. Changing the parallelism degree at runtime can avoid expensive I/O but requires substantial engineering effort. For Bösen, the consistency controller needs to be revised to allow the worker threads to register and log out dynamically. Moreover, many components including communication bus, background worker, and client/server table need to be notified about the changes in parallelism degree. For process-based systems such as Multiverso and Parameter Server, adjusting the parallelism degree will involve the dynamic starting/killing of process, and the context initialization (e.g., data loading, parameter pulling, consistency controller notification, etc.) for processes can be expensive.

4. MULTI-STAGE ABSTRACTION & API

In this section, we first introduce our multi-stage abstraction, and then present the programming model of FlexPS with an example.

[3]For details of the experiment that produces Figure 1, please refer to Appendix A in our technical report [18].

[4]For details of the experiment of measuring the parallelism degree adjustment delay, please refer to Appendix C in [18].

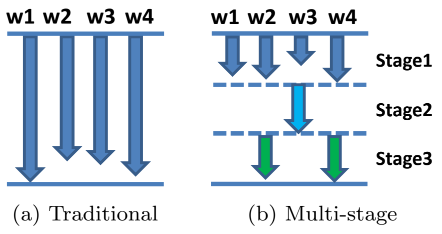


Figure 2: Single-stage vs. multi-stage

4.1 Multi-Stage Abstraction

In the multi-stage abstraction, a machine learning task is the composition of a series of stages. A *stage* runs a user-defined function (UDF) on a specific set of computing resources characterized by the number of workers and the location of these workers (e.g., 10 workers on Node 1 and 5 workers on Node 2). Stage is the basic unit for scheduling, and users can specify the resource allocation (including the parallelism degree) for each stage. In fact, the traditional PS architecture can be viewed as a special case of the multi-stage abstraction in which a task has only one stage. The execution of a typical task in existing PS systems and in FlexPS is depicted in Figure 2, which shows that the multi-stage design offers inherent support for changing the parallelism degree between stages.

To show how the multi-stage abstraction supports flexible parallelism control, we illustrate by a machine learning task that trains a logistic regression model using SVRG in Algorithm 1. SVRG is a representative of algorithms with dynamic workloads as its full-gradient step has much higher workload than the stochastic steps. We can naturally map this task into two kinds of stages, i.e., the full-gradient stage and the stochastic stage. The full-gradient stage calculates the full gradient by scanning the entire training dataset, and a large number of workers can be assigned to handle the heavy workload. The stochastic stage updates the model parameters using gradient estimated on a small batch of data samples, and thus only a small number of workers are needed. The logic in Algorithm 1 can be realized by executing the full-gradient stage and (multiple) stochastic stages alternatively.

The multi-stage abstraction provides inherent support for flexible parallelism control. As shown in Section 3, many machine learning algorithms have varying workloads and call for dynamic parallelism degree. With the multi-stage abstraction, we can decompose these algorithms into stages according to the workloads and set each stage with the proper parallelism degree.

Apart from flexible parallelism, the multi-stage abstraction also brings two additional benefits. First, it enables us to better utilize data locality. If different stages are manipulating different parts of the dataset, they can be assigned to the machines where the required data reside to avoid transferring the training data across the network. This is especially useful for algorithms such as SGD and SVRG, as their stochastic update steps should be conducted on different partitions of the training dataset to ensure convergence[5]. Second, it facilitates *fine-grained* scheduling of mul-

[5]Although the stochastic update steps are homogeneous in workload, we can map them to different stages and assign the stages to different machines to enjoy data locality.

tiple tasks, as a stage of a task is much lighter than the task and the stages of different machine learning tasks can run simultaneously on a cluster.

Challenges of multi-stage implementation. Although the multi-stage abstraction has many benefits, implementing it on top of existing PS systems is either difficult or inefficient. One option is to map each stage to a task, but handling the transition between the tasks (e.g., ensuring consecutive tasks can see the same model parameters, resetting the consistency controller, reusing the loaded data, and so on) can be difficult. Another option is killing/starting worker threads/processes dynamically. However, it requires revising many system components and may not be efficient as explained in Section 3.

More importantly, existing systems lack tailored optimizations for the multi-stage abstraction. Many important questions in system design need to be addressed in order to make multi-stage machine learning efficient. These questions include *(1) how to schedule the stages in an efficient way*, *(2) how to facilitate the efficient transition between stages*, and *(3) how to reduce the communication overhead*. We present the corresponding optimizations in FlexPS to solve these questions in Section 5.

4.2 Programming Model

FlexPS adopts a KV-store API similar to existing PS systems. Two functions, *Get(keys)* and *Put(keys, vals)*, are provided for workers to read and update the model parameters stored in servers. System details such as consistency control, communication between workers and servers, and scheduling are hidden from users. We use SVRG to illustrate how to use FlexPS to implement a multi-stage task.

Typically, defining a multi-stage task consists of three steps as shown in the code snippet below. In Step 1, we need to define the stages, i.e., what the workers need to do in each stage. For SVRG, we define two UDFs, *fgd_lambda* and *sgd_lambda*, for the full-gradient stage and stochastic-gradient stage, respectively. In *fgd_lambda*, each worker reads the model parameters and calculates the gradient using part of the training data, and then updates the full gradient stored in the servers. In *sgd_lambda*, each worker reads the model parameters and full gradient, and conducts the variance-reduced update according to Line 5 in Algorithm 1. In Step 2, we specify the execution order of the stages and set the parallelism degree. In this example, the parallelism degrees of the *fgd_lambda* and *sgd_lambda* stages are set to be 100 and 10, respectively. Finally, Step 3 submits the task to the FlexPS engine for execution.

```
/* Step 1: define the stages */
auto fgd_lambda = [](Info info) {
  // Get model parameter w from the KV-store
  // Calculate gradient
  // Update full gradient u in the KV-store
};
auto sgd_lambda = [](Info info) {
  // Get w and u from the KV-store
  // Calculate gradient
  // Perform variance-reduced update
  // Update w to the KV-store
};

/* Step 2: set the parallelism degree */
MultiStageTask task;
task.SetStages({{fgd_lambda, 100},
                {sgd_lambda, 10},
               });

/* Step 3: submit the task */
engine.SubmitAndWait(task);
```

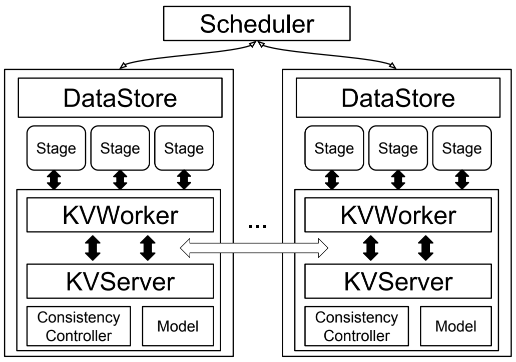


Figure 3: System Architecture

Inside each stage, users can use the KV-store API to access the model parameters in a way similar to other PS systems. An example is shown below, which uses *Get()* to read the required model parameters, performs local computations, and then updates the model parameters using *Put()*.

```
auto stage_lambda = [](Info info) {
  auto& worker = info.GetWorker();
  auto w = worker.Get(/* key vector */);
  // Conduct local computation
  ...
  worker.Put(/* key vector */, /* delta vector */);
};
```

For the efficient execution of multi-stage machine learning tasks on FlexPS, users should set a good parallelism degree for each stage. There is no closed-form formula for the optimal parallelism degree as it depends on a number of factors such as the dataset, the learning task and the setting of the cluster. However, setting the parallelism degree is not a difficult task as most existing PS systems also require users to set the parallelism degree and our experience shows that one can choose a fairly good parallelism degree with some experience. To remove the burden of users in setting the parallelism degree, we provide a method in FlexPS that can adjust the parallelism degree automatically during runtime, which will be presented in Section 5.4.

5. ARCHITECTURE AND OPTIMIZATION

In this section, we introduce the system architecture of FlexPS and present the optimizations that lead to an efficient implementation of the multi-stage abstraction.

5.1 System Architecture

FlexPS adopts a master-slave architecture, as shown in Figure 3. A *scheduler* lies at the core of the master, which is in charge of assigning the stages (of tasks) to slaves by considering various factors including stage specifications, slave availability, parallelism degree, and data locality. The scheduler also tracks the progresses of the stages and the available resources on the slaves. Scheduling is conducted either when the number of available worker threads has exceeded a threshold, or when a certain period of time has passed since the last scheduling. To utilize the resources in the cluster efficiently, the scheduler also supports multiple tasks to run simultaneously.

Each slave machine runs an event loop to poll events from the master. When new stages are assigned, it spawns new threads to run the stages according to the specifications given by the master. It also tracks the progresses of its stages, and notifies the master when a stage is finished. A *data store* module is in each slave to serve the data for all stages. The slave machines also serve as the parameter servers. Each slave has a *KV-store* module containing KV-worker threads and KV-server threads, which are implemented in actor model to allow asynchronous operations. The functions defined by users link with the KV-worker threads to issue non-blocking *get()* and *put()*. The KV-server threads are in charge of the parameter management including consistency control, answering the pull requests and aggregating the updates.

We implemented the workers as threads rather than processes (as in Multiverso) for two reasons. One is that multiple worker threads on the same machine can share the loaded data and avoid the repetitive push/pull of model parameters. We use a process cache and a simple version control strategy to avoid the repetitive pull of the same model parameters. The other reason is that the dynamic starting/killing of threads is cheaper than that of processes. To ensure that a stage can see the model parameters from its preceding stage, we enable the KV-store to support multiple tables, and index each table by a unique id. Different stages of a task can access the same table using table id.

5.2 Optimizations for Multi-stage

Stage scheduler. Different from existing PS systems, the basic scheduling unit in FlexPS is stage rather than task. The scheduler decides which stages to run and where the stages are to be run according to the stage specifications, scheduling history, and data locality. A stage specification contains the number of workers in this stage and (optionally) the placement of these workers. The scheduling history records where each stage was placed, and is kept by the task scheduler to guide scheduling. If users provide a complete stage specification, the scheduler will follow the specification strictly. In case that users do not specify the location of a stage, the scheduler will decide where to place it mainly according to two rules. First, the worker threads of a stage will be allocated on the same slave machine if possible to reduce the communication overhead, since workers on the same machine can avoid issuing repetitive push/pull requests to servers. Second, a stage will be assigned to the slave that holds (part of) the training data but has the longest interval since the last scheduling of the task. This is because the type of algorithms like SGD require a uniform sampling of the training data, and this rule ensures that different parts of the training dataset have approximately the same probability of being processed while enjoying data locality. If user does not provide the parallelism degree in the stage specification, FlexPS uses an automatic parallelism degree setting functionality that will be introduced in Section 5.4 to adjust the parallelism during runtime.

To improve resource utilization and overlap communication with computation, we enable the scheduler to support the concurrent execution of multiple tasks. The concept of task dependency is introduced, with which users can specify the precedent tasks of a task. One example of using task dependency is that a testing task which depends on a set of training tasks can perform ensemble learning on the trained models. Stages from tasks with no dependency can be run simultaneously.

The "fat" stages, which require a lot of resources such as the full-gradient stage in SVRG, may suffer from starvation due to insufficient resources, resulting in long completion time of a task. To tackle this problem, we introduce the concept of priority. Each stage is associated with a priority which increases with its waiting time to be executed. Once the priority exceeds a threshold, the stage will be put into a starvation list. The scheduler considers the stages in the starvation list first and may lock the required resources to prevent starvation. Users can also specify the priority of stages. For a delay-sensitive task, users can give all its stages a high priority to minimize the waiting time.

Stage-aware consistency controller. The consistency controller (in each KV-server) uses an array to record the progress of the worker threads for consistency control. Specifically, when the server receives a request, it will check the array to determine if the request should be blocked according to the consistency protocol. For example, under SSP, the consistency controller will block the *Get* request of a worker if its progress is too fast. The blocked requests are buffered and will be replied with the required data when the slowest worker catches up. In the multi-stage design, during the transition between stages, the consistency controller needs to be reset to clear worker progresses in the previous stage and notified about the worker threads that participate in the current stage. For this purpose, we make the consistency controller stage-aware. At the beginning of a stage, each worker thread will send the *InitConsistencyController* signal with the stage specification (indicating the number of workers) to all consistency controllers (each KV-server has a consistency controller). The consistency controllers will store the stage specification and reset the worker progress when receiving this signal. We do not assign the scheduler to send the InitConsistencyController signal as the cost of repetitive message sending is low due to the small message size and it is more natural to let workers communicate with servers in the PS architecture.

Data store. Each slave machine has an in-memory data store to hold the training data across stages. The data store is simply implemented as a two-dimensional static vector. When loading the training data, each local worker only writes to a specific row and every worker can access the whole local data store for training. A typical workload will first submit a loading task to the scheduler before the training tasks. The loading task normally uses a large parallelism degree to speed up the loading process. Multiple tasks can share the same loaded data and the data are only removed from the data store when all tasks depending on the data are finished.

5.3 Optimizations for Small Parallelism

With the multi-stage abstraction, machine learning tasks are divided into stages with varying workloads and optimal parallelism degrees. The full-gradient step in SVRG and large batch size steps (stages) in SGD requires a large parallelism degree. On the contrary, the initial steps of SGD and the stochastic step of SVRG-type algorithms typically use a small batch size and need a small parallelism degree. Existing PS systems have proposed optimizations techniques such as local process cache and asynchronous communication for large parallelism. We not only incorporate these techniques in FlexPS, but also optimize for the small parallelism case.

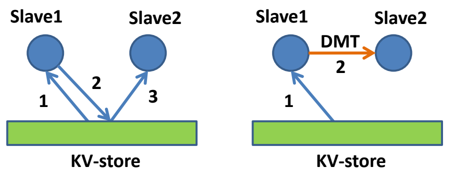
When the optimal parallelism degree is small, we found that placing all the worker threads on the machine where the data reside and transferring the model parameters to the worker threads can reduce the communication overhead compared with remote parameter push/pull from servers. Moreover, we can map the stochastic steps of SGD and SVRG-type algorithms to multiple stages, and allocate the stages to different machines to ensure a uniform sampling of the training data and enjoy data locality. We call this learning/scheduling pattern *single-machine mode* and propose optimizations for it as follows.

Local model store and consistency controller. For single-machine mode, we provide a local model store and a local consistency controller. Our design is different from the process-level cache in other PS systems in that we also move the consistency controller from the server side to the local process. As all worker threads are within the same process, the local consistency controller has enough knowledge about the progress of the task for consistency management. With the local model store and consistency controller, all the updates and consistency controls are performed within the local process, and thus expensive network communication with the global KV-store is avoided.

Chunk-based race control. Since multiple threads may be approved by the local consistency controller and race for the shared local model, a *chunk-based race control* strategy is adopted. We organize the local model in chunks and each chunk is associated with a mutex. The chunk-based mutex is a balanced granularity between having a mutex for every single parameter (best concurrency for model access but high overhead) and having a lock for the whole local model. The parameters in the chunk-based organization are indexed so that reading/writing a parameter requires constant time.

Flexible model preparation. We provide two options for preparing the local model, *integral model* and *on-demand model*. For integral model preparation, all the model parameters are pulled from servers at the beginning of a stage. Although it consumes more space and takes some time to prepare, the advantage of integral model is that there will be no communication between the worker threads and the KV-store during training. If the model is very large or the available memory is limited, users may choose to use the on-demand model, which loads only relevant chunks of the model when needed. In cases many chunks need to be loaded but memory is limited, we provide different *chunk replacement policies* to swap some chunks to disks, which are similar to the standard cache replacement policies.

Direct model transfer (DMT). To avoid the overhead of loading/dumping the model from/to the global KV-store in the transition from one small parallelism stage to another (shown as Steps 2 and 3 in Figure 4a), FlexPS provides the DMT option. With DMT, the system directly sends the model to the machine that will be scheduled for the next stage, and thus bypasses the global KV-store (i.e., Step 2 in



(a) Without DMT (b) With DMT

Figure 4: Illustration of direct model transfer

Figure 4b). Since a task may not know where it will be run at the next stage (before the scheduler makes the decision), the model will be temporarily kept in the current slave until the next stage of this task is scheduled. When the next stage is scheduled, the DMT module will be notified and send the model to the target machine.

Unified programming model. With the optimizations for single-machine mode, FlexPS also supports the efficient execution of single-machine programming models, thus providing a unified programming model for both single-machine learning and distributed learning. Representative single-machine learning programming models include *sequential*, *single process multiple threads* (*SPMT*), and *Hogwild!* [32]. Sequential uses a single thread to update the model parameters and many theoretical researches focus on sequential algorithms for the ease of analysis. SPMT and Hogwild! use multiple threads to update the model parameters and may not follow the same logic as sequential algorithms due to outdated model parameters and lost updates. However, algorithms such as SGD and SVRG perform quite well on SPMT and Hogwild! in practice.

These single-machine programming models can be easily realized in FlexPS by constraining the worker threads on one machine and adopting the right consistency protocol (e.g., ASP for Hogwild!) on the local consistency controller. Although distributed learning is commonly used to handle big datasets and large models, single-machine learning is still important for correctness verification and algorithm calibration. With a unified programming model for single-machine learning and distributed learning, FlexPS can significantly reduce the effort of transforming the single machine validation implementation of an algorithm into distributed production deployment.

5.4 Automatic Parallelism Adjustment

For the efficient execution of a multi-stage machine learning task in FlexPS, users should set a good parallelism degree for each stage according to the workload. To remove the burden of manual parallelism degree setting, we provide an automatic parallelism adjustment method.

To achieve automatic parallelism adjustment, a stage is further divided into smaller sub-stages by the scheduler. A smaller sub-stage here is defined in terms of the number of iterations; for example, a stage that conducts SGD with a certain batch size for 100 iterations can be divided into sub-stages with the same batch size but with only 5 iterations. The scheduler schedules the sub-stages with different parallelism degrees and monitors the per-iteration delay to search for the optimal parallelism degree. Once the optimal parallelism degree is identified, the remaining iterations in the stage are executed with the optimal degree. Note that the searching sub-stages also conduct parts of the work from its corresponding stage and thus no computation is wasted. We set the number of iterations, m, in each sub-stage to 10 to balance between an accurate estimation of the per-iteration delay (more accurate with larger m) and the cost of executing the sub-stages with poor parallelism degrees (higher cost with larger m). We run the searching sub-stages with increasing parallelism degrees and terminate the search when a sub-stage has a longer per-iteration delay than its preceding sub-stage (note that the per-iteration delay curve is unimodal, as shown in Figure 1).

Our experiments in Section 7 show that the automatic parallelism adjustment method achieves satisfactory performance. In addition to supporting flexible parallelism, automatic parallelism adjustment can also be used to set the parallelism degree for single-stage tasks, which makes FlexPS easier to use than existing PS systems as they often require the user to specify the parallelism degree explicitly. Note that the automatic parallelism adjustment method is enabled by both the multi-stage abstraction and the optimizations for flexible parallelism. First, the multi-stage abstraction allows us to divide a stage into sub-stages to conduct the searches. Then, the optimizations for flexible parallelism give low stage transition delay (in the order of 10ms as reported in Section 7.4), which helps reduce the overhead of automatic parallelism adjustment. We also remark that the method may not be effective for stages with a small number of iterations; however, typically machine learning tasks require a large number of iterations.

6. SYSTEM IMPLEMENTATION

In this section, we present some important implementation aspects of FlexPS including fault tolerance, load balancing, data preparation, and the KV-store.

6.1 Fault Tolerance and Load Balancing

Fault tolerance. In FlexPS, a simple checkpointing technique is used for fault tolerance. Users can specify the checkpoint interval in terms of the number of stages or iterations. At each checkpoint, the KV-store dumps the model parameters to persistent storage such as network file system (NFS) or distributed file system (DFS). The scheduler also saves the current progresses of tasks to disk. When tasks or machines fail, FlexPS stops the current tasks and restarts them by loading the model parameters (for the KV-store) and task progresses (for the scheduler) from the last checkpoint. This checkpointing technique is similar to the one used in Bösen. We did not adopt the replication-based method in Parameter Server due to its high overhead.

Load balancing. FlexPS supports user-customized model parameter partition for load balancing. A *range manager* keeps track of the model partitioning information and provides an interface for users to register the customized partitioning. The KV-workers issue requests according to the information stored in the range manager. This could be useful when knowledge about the distribution of model access pattern is available (e.g., from logs) so that frequently accessed parameters can be distributed to different servers to balance the workload. Following certain model partitioning strategy [35], the convergence for problems such as matrix factorization can be accelerated. FlexPS also supports partitioning the model to only a subset of the KV-servers, which can reduce the latency for low-dimensional models (note that for low-dimensional models, often the whole model is loaded in each iteration, and thus the latency is high if the model is loaded from many servers). We will show in the experiments that user-customized model partition can also be utilized to enjoy data locality. Compared with FlexPS, existing systems such as Parameter Server, Bösen and Multiverso do not support user-customized partitioning for load balancing.

6.2 The Data Preparation Module

FlexPS supports reading training data from DFS. In a common DFS, e.g., HDFS, a file is partitioned into blocks and each block may have a few replicas distributed to multiple machines. FlexPS has a global *file assigner* to keep the block information of all files. The loading threads ask the file assigner for data blocks, and the file assigner assigns blocks to the threads according to data locality. In contrast, other PS systems like Bösen and Parameter Server do not have a file assigner and the data could be loaded from remote machines.

FlexPS provides two ways for preparing the data for training. The first way is to run a loading task and store the data in the data store before all the training tasks. When the dataset is large, loading the entire training dataset can be time consuming. Thus, the second way is to load the data on-the-fly while training the model, which is done by an *async reader* module with the classical *producer-consumer* paradigm. Specifically, the reader threads load the data from DFS and store the data to a pre-allocated buffer as long as the buffer is not full. The worker threads consume the data in the buffer and use the data to train the model. This design overlaps computation (training) and I/O (loading) to reduce the task completion time.

6.3 The KV-Store Module

The KV-store module of FlexPS adopts several novel implementation techniques for efficiency.

Local zero-copy communication. In FlexPS, the local KV-workers and KV-servers are organized in the same process, so that we can optimize the *Get/Put* requests issued to the local KV-servers using the *zero-copy* functionality, i.e., passing the pointer instead of the data, such that the pushed or pulled data need not be serialized and de-serialized. Our experiments show that the zero-copy communication boosts the local *Get/Put* performance by 3 times on various workloads.

Repetitive *Get* avoidance. In the PS architecture, multiple worker threads in one process may require the same parameter chunks from the KV-servers. A process cache is usually used to save the cost of repetitive remote *Get*. However, SSP makes the management of the process cache tricky because different worker threads may have different progresses (in different iterations) and require different versions of chunks. In FlexPS, we design a simple linked-list based strategy for process cache management. In the process cache, each chunk (of parameters) is associated with a linked-list maintaining the pending *Get* requests for this chunk, ordered by the requested versions. If a worker needs a chunk, it first checks the version of this chunk in the process cache. If the version is new enough (i.e., satisfying the staleness requirement), it directly uses the chunk. Otherwise, it inserts the request into the linked-list. Whenever a reply for a chunk is received, its linked-list is updated by erasing all requests that require no newer version than the

Table 1: Datasets

Dataset	# of samples	# of features	Sparsity
webspam	350,000	16,609,143	2.24×10^{-4}
kdd	149,639,105	54,686,452	2.01×10^{-7}

received version (these requests are also unblocked). A new remote *Get* request for a chunk will be issued if its linked-list is not empty. This simple strategy works well in practice, while Bösen manages the process cache with a sophisticated client library using a bunch of background threads.

Customizable KV-server. The KV-store supports consistency protocols including BSP, SSP, and ASP. The KV-servers control the consistency and block the fast workers by not responding to their requests until the consistency requirement is satisfied. Users can customize the data structure used in the KV-servers to store the model parameters, and FlexPS has built-in support for data structures including dense array, tree map, and hash map. The behavior of the KV-servers on the updates/model parameters can also be customized. For example, a user can instruct the KV-servers to apply addition (to the global model) for each update to calculate the full gradient or conduct scaling on the model parameters for regularization.

7. EVALUATION

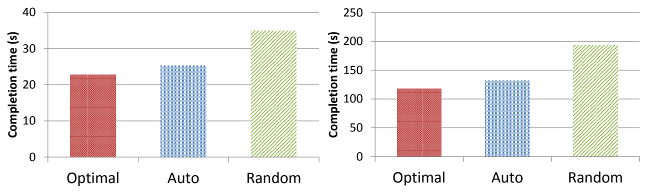
We evaluated FlexPS on a cluster with 20 machines connected via 1 Gbps Ethernet. Each machine is equipped with two 2.0GHz E5-2620 Intel(R) Xeon(R) CPU (12 physical cores in total), 48GB RAM, a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), and running on 64-bit CentOS release 7.2. We mainly used two datasets: webspam[6] and kdd[7] which were also used in [22] and [42]. Some statistics of the two datasets are listed in Table 1, where sparsity is the average potion of non-zero features in a sample.

We compared FlexPS with two state-of-the-art PS systems, Multiverso and Petuum (or Bösen, as Bösen is the module for data-parallel distributed machine learning in Petuum). As we are mainly interested in the performance from the system perspective, we adopt the same execution plan for all the systems to be compared; that is, the number of phases, the batch size and number of iterations in each phase, and the learning rate are the same for all the systems. The primary performance indicator we use is task completion time, which is the time taken to finish a task (not including data loading time). We also measure the total worker time (the sum of the working time of all the worker threads) of the systems as an indicator of resource consumption. Although FlexPS can support the concurrent execution of multiple ML tasks, in our experiments we used the cluster exclusively for running only one task for each system to exclude the influence of concurrent tasks.

7.1 Automatic Parallelism Adjustment

To verify the effectiveness of the automatic parallelism adjustment method in Section 5.4, we trained SVM using SGD with constant batch size on FlexPS. The batch sizes were set as 0.1% and 1% of the dataset size for kdd and webspam, respectively, while the number of iterations were obtained by dividing the dataset size by batch size. As

[6]www.cc.gatech.edu/projects/doi/WebbSpamCorpus.html
[7]www.kddcup2012.org/c/kddcup2012-track2/data



(a) webspam (b) kdd

Figure 5: Validation of automatic parallelism adjustment

Table 2: Total worker time (sec) of SVM with SGD

	Optimal	Auto	Random
webspam	912	999	7,609
kdd	9,448	10,384	36,903

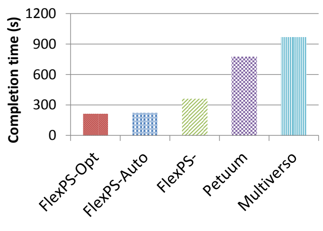
the workload was static, we mapped it to a single stage in FlexPS. We report the task completion time of *Optimal*, *Auto* and *Random* in Figure 5. For *Optimal*, the parallelism degree was obtained by an offline search over multiples of 20 to minimize the per-iteration delay (the search time is not included in the task completion time). *Auto* corresponds to automatic parallelism adjustment method. *Random* is the average task completion time of 20 randomly selected parallelism degrees in the range of [20, 400], where 20 is the number of machines in the cluster and 400 is the number of cores used. *Random* simulates the choice of an inexperienced user.

The results show that automatic parallelism adjustment significantly outperforms *Random* and the loss compared with *Optimal* is marginal. To be more specific, the task completion time of *Auto* is only 12% and 11% longer than that of *Optimal* for kdd and webspam, respectively. Compared with *Random*, *Auto* achieves a reduction of over 45% in task completion time. We also report the total worker time of the three parallelism degree setting schemes in Table 2. The results show that *Auto* and *Optimal* consume similar amounts of total worker time, while the total worker time of *Random* is significantly longer. The results can be explained as follows. As shown in Figure 1, the task completion time curve is relatively flat for large parallelism degrees; thus, using an excessively large parallelism degree will not degrade task completion time severely, but this will result in much longer total worker time and waste a lot of resources.

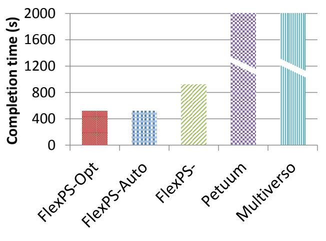
7.2 Comparison on Dynamic Workloads

In this experiment, we compared FlexPS with Petuum and Multiverso on tasks with dynamic workloads. For the machine learning models, we tested logistic regression and support vector machine (SVM). For the learning algorithms, SGD with growing batch size and SVRG were used. We included a variant of FlexPS by disabling flexible parallelism control, denoted by *FlexPS-*, in the comparison. Similar to Petuum and Multiverso, FlexPS- only adopts a single stage and a constant parallelism degree for a task. We optimized the parallelism degree to minimize task completion time for Petuum, Multiverso and FlexPS- with an offline search over multiples of 10.

To demonstrate the benefit of flexible parallelism, we also included *FlexPS-Opt* and *FlexPS-Auto*, which support multiple stages but adopt different parallelism degree setting schemes as follows. The parallelism degree of FlexPS-Opt



(a) webspam



(b) kdd

Figure 6: Completion time of logistic regression with SVRG

Table 3: Total worker time (sec) of logistic regression with SVRG

	FlexPS-Opt	FlexPS-Auto	FlexPS-	Petuum	Multiverso
webspam	8855	8862	43020	61840	38680
kdd	25761	25838	111000	-	-

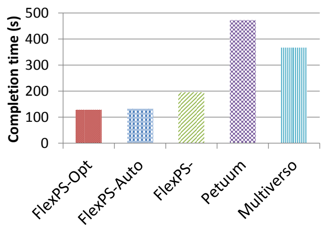
was obtained by an offline search over multiples of 10 for each stage, while the parallelism degrees of FlexPS-Auto were adjusted during runtime by the automatic parallelism adjustment method. FlexPS-Opt represents the maximum performance benefit one can obtain with flexible parallelism, while FlexPS-Auto demonstrates the performance benefit users can enjoy without tuning the parallelism degree.

Due to space limitation, we give the detailed parallelism degrees of each system in Appendix B in [18]. Note that different systems may have varying optimal parallelism degrees because of the differences in their system designs. Overall, Multiverso favors small parallelism degree as it uses processes as workers and the communication cost grows rapidly with parallelism degree due to the lack of optimizations such as repetitive *get* avoidance. Petuum and FlexPS- usually adopt larger optimal parallelism degree, which can be explained by their process cache design.

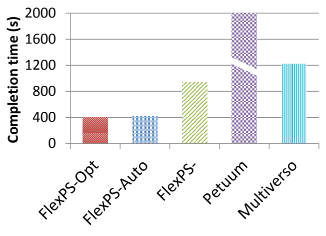
Logistic regression with SVRG. For SVRG, we set the batch size b of the stochastic update step in Algorithm 1 as 0.1% and 0.001% of the dataset size for webspam and kdd, respectively. As the SVRG paper [21] recommends to scan the dataset twice in one epoch, the number of stochastic steps in an epoch is set as 2000 and 200000 for the two datasets, respectively. One epoch was mapped to a full gradient stage and a stochastic update stage in both FlexPS-Opt and FlexPS-Auto.

We ran SVRG for 10 epochs, and report the average completion time of an epoch in Figure 6. Petuum and Multiverso cannot finish in 10,000 seconds for kdd and they are reported as fractured bars. Compared with Petuum and Multiverso, FlexPS- achieves a minimum speedup of 2 times in task completion time, which proves that FlexPS is a more efficient PS system even without flexible parallelism. Multiverso is inefficient because it uses individual processes as workers, and thus multiple workers cannot share the same communication module and have high overhead when increasing the parallelism. Petuum's poor performance is due to its process cache design, which uses a row as the minimum communication unit, making sparse access to the model inefficient. We also note that the process cache is a fundamental design in Petuum and cannot be disabled. FlexPS, on the other hand, can also support sparse access to model efficiently.

Compared with FlexPS-, FlexPS-Opt reduces the task completion time by 44% and 41% for kdd and webspam, respectively, because of its support of flexible parallelism.



(a) webspam



(b) kdd

Figure 7: Completion time of SVM with SGD

Impressively, FlexPS-Auto achieves a task completion time similar to FlexPS-Opt without any manual tuning of parallelism degree. To better demonstrate the benefits of flexible parallelism, we report the total worker time of the systems in Table 3. The results show that FlexPS- spends more than 4 times of total worker time compared with FlexPS-Opt and FlexPS-Auto, which shows that flexible parallelism can significantly reduce resource consumption.

SVM with SGD. We trained SVM using the SGD algorithm in [32], which exploits the update pattern of sparse datasets for speedup. We partitioned the SGD algorithm into three phases. For webspam (kdd), the batch sizes are 0.1%, 1%, 10% (0.01%, 0.1%, 1%) of the dataset size. In each phase, we scanned the dataset once, and thus the number of iterations in each stage was obtained by dividing the dataset size by the batch size.

The task completion time of the systems is reported in Figure 7. Due to the row design mentioned earlier, Petuum performs the worst among the systems. For kdd, its task completion time exceeds 10,000 seconds. Compared with Multiverso and Petuum, FlexPS- achieves more than 47% of reduction in task completion time for webspam due to the optimizations in Section 6. Thanks to flexible parallelism, FlexPS-Opt provides another 57% and 34% of reduction in task completion time over FlexPS- for the two datasets, respectively. Similar to the case with SVRG, FlexPS-Opt and FlexPS-Auto have almost the same task completion time. We also report the total worker time of the systems in Table 4, which shows that FlexPS-Opt and FlexPS-Auto only spend less than 50% of the total worker time compared with FlexPS-, and only 10% in the best case.

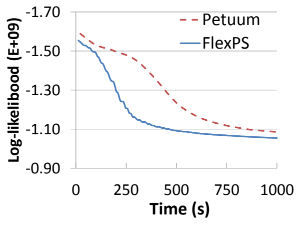
An interesting phenomenon is that FlexPS-Auto spends less total worker time than FlexPS-Opt, which is because we searched the parallelism degree to minimize per-iteration delay rather than the total worker time for FlexPS-Opt. Moreover, Multiverso spends considerably less total worker time than FlexPS- as it uses much smaller parallelism as shown in Table 6 of Appendix B in [18][8]. Note that FlexPS-Opt and FlexPS-Auto outperforms Multiverso in both task completion time and total worker time.

The experimental results demonstrate that flexible parallelism achieves significant performance improvements in terms of both task completion time and total worker time. In addition, the results also show that FlexPS is a more effi-

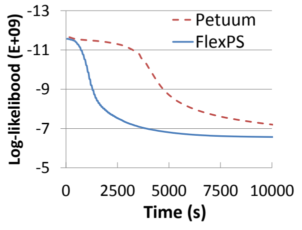
[8]Using smaller parallelism degree helps in reducing total worker time as the amount of communication is reduced. Intuitively, the optimal total worker time is achieved at a parallelism degree of 1 which spends no time on communication, but may result in significantly longer task completion time.

Table 4: Total worker time (sec) of SVM with SGD

	FlexPS-Opt	FlexPS-Auto	FlexPS-	Petuum	Multiverso
webspam	3877	3449	33376	94540	14676
kdd	31852	31275	70770	-	36540



(a) NYTimes



(b) Pubmed

Figure 8: Convergence speed comparison on LDA

cient PS system than Petuum and Multiverso even without flexible parallelism.

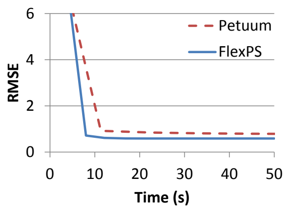
7.3 Comparison on Other ML Tasks

Apart from the tasks with dynamic workloads, there are also many machine learning tasks with almost constant workloads. In this experiment, we compared the performance of FlexPS with Petuum on two representatives of such tasks, in order to show that FlexPS is also a very efficient PS system for other types of machine learning workloads. For these tasks, FlexPS used only one stage and did not adopt flexible parallelism control and automatic parallelism adjustment. Multiverso was not compared as it does not adopt process cache, which is crucial for the performance of the dense row access in these applications.

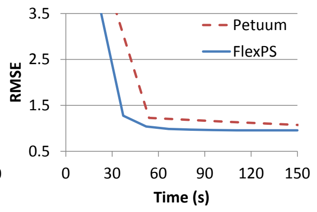
Latent Dirichlet allocation. LDA is an unsupervised topic modeling method, which discovers hidden topics for words in the vocabulary and identifies the topic distribution for each document in the corpus. We used the Gibbs sampler and stored the word-topic table, doc-topic table, and the topic summary table in the KV-store. For both Petunm and FlexPS, we set the storage data structure of the model to be dense row. Two datasets were used: NYTimes contains 300,000 documents and approximately 100,000,000 tokens; while Pubmed is relatively larger with 8,200,000 documents and about 730,000,000 tokens. The dirichlet prior parameters α and β were both set as 0.1, and the number of topics and the staleness parameters were set as 1000 and 1, respectively.

We plotted the log-likelihood (the larger the better) versus training time for both systems in Figure 8. The results show that FlexPS can achieve the same log-likelihood as Petuum using approximately 1/3 of the time. This performance gain is mainly due to the low overhead process cache management strategy and efficient KV-store communication in FlexPS.

Matrix factorization. Matrix factorization (MF) is widely used for recommendation. Given a user-item rating matrix $R \in \mathbb{R}^{M\times N}$, it attempts to find a user latent matrix $U \in \mathbb{R}^{M\times K}$ and item latent matrix $I \in \mathbb{R}^{M\times K}$ that satisfies $R \approx UI^T$. Alternating least squares (ALS) algorithm is usually used for MF, which fixes one latent matrix to train another in an alternating manner. We used the Netflix dataset [6], which contains 480,189 users, 17,770 items and 100,480,507 ratings, and the Yahoo! Music

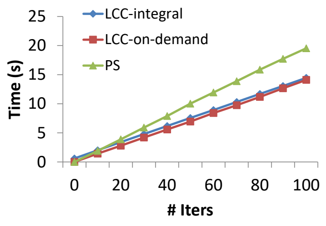


(a) Netflix

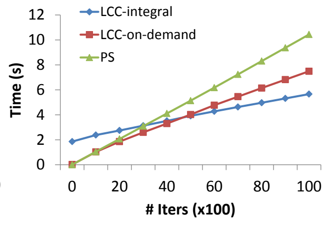


(b) Yahoo! Music

Figure 9: RMSE vs. training time on matrix factorization



(a) webspam



(b) kdd

Figure 10: Performance gain of local consistency control and flexible model preparation

dataset [46], which contains 1,823,179 users, 136,736 items and 699,640,226 ratings.

The training root-mean-square-error (RMSE) was plotted against training time in Figure 9. For both datasets, FlexPS shows faster convergence than Petuum. This is because FlexPS supports customized model partitioning and we use this functionality to partition both the U and I evenly among the servers following the method in [35].

7.4 Performance of Optimization Techniques

In this experiment, we examined the effects of the optimizations techniques presented in Section 5 on the performance of FlexPS.

Local consistency controller and flexible model preparation. FlexPS moves the consistency controller to the workers in single machine mode and provides two methods to prepare the local model, on-demand model (the default) and integral model, for user to choose from. To test the benefits of these designs under small parallelism, we used 20 worker threads on the same machine to train logistic regression with SGD. We plotted the time consumption against the number of iterations in Figure 10, where *LCC-integral* stands for local consistency controller plus integral model preparation while *LCC-on-demand* is local consistency controller with on-demand model preparation. PS is used as baseline and stands for the case that both the consistency controller and model are on the server side.

The results show that local consistency controller and flexible model preparation considerably reduces the running time for small parallelism. The performance gap widens with iteration number as PS has to pay constant communication cost in each iteration, while the local consistency controller does not need to communicate with the servers once the local model is ready. Moreover, on-demand model preparation should be used if the stage only has a small number of iterations as integral model preparation has higher cost of model initialization as illustrated in Figure 10b. The perfor-

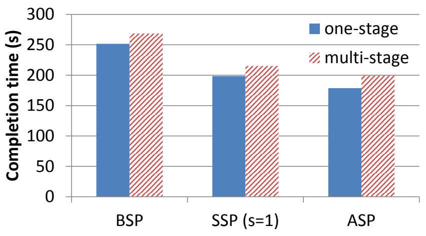


Figure 11: The overhead of stage transition

mance of integral model preparation for webspam is similar to that of on-demand mainly because its model is small and the initialization cost of integral model is marginal.

Stage transition. As FlexPS maps a machine learning task to multiple stages, the efficient transition between stages is crucial for high efficiency. To examine the stage transition cost, we trained a logistic regression model on the kdd dataset using SGD with a uniform batch size of 149639 (0.1% of the dataset size) for 100 iterations. One-stage scheme and multi-stage scheme were compared, and both of them used 200 worker threads in the entire training process. The one-stage scheme did not have stage transition; but in order to test the stage transition efficiency for the multi-stage scheme, we split the 100 iterations into 10 stages, each with 10 iterations for the multi-stage scheme. In each stage, the worker threads were uniformly assigned to the machines in the cluster.

We report the task completion time of the one-stage scheme and multi-stage scheme in Figure 11 under different consistency protocols. As the task completion time of the multi-stage scheme is similar to that of the one-stage scheme, we can conclude that the stage transition cost is marginal considering the multi-stage scheme takes 9 stage transitions. Intuitively, the stage transition cost is higher for ASP and SSP than for BSP, since in BSP a barrier is enforced at the end of a stage. However, the results show that the difference in the stage transition cost for the three consistency protocols is not significant. We also tested the cost of stage transition by measuring the time between the slowest worker thread finishing its work in the previous stage and all threads completing the initialization of the consistency controllers in the current stage (denoted as *stage transition delay*). The average stage transition delay is around 0.02 seconds for 200 workers and 0.01 seconds for 100 workers, which again shows that stage transition in FlexPS is efficient.

8. RELATED WORK

We have introduced existing PS systems in Sections 2. In this section, we discuss other related work.

Tailored ML solutions. There are many parallel solutions tailored for specific machine learning problems. GraphLab [29] uses a graph abstraction to manage the communication asynchronously and is optimized for iterative algorithms with sparse computational dependencies; however, its generality is limited by its structural constraints. Husky [48, 47, 25] introduces the notion of object interaction pattern that allows users to construct a PS framework and it also supports the efficient implementation of asynchronous ML algorithms [49], which provides an efficient alternative solution for distributed ML. NOMAD [51] is a distributed lock-free framework for matrix factorization; however, its application is limited to doubly-separable models. Hogwild! [32] is a single machine lock-free SPMT framework for sparse dataset, but it cannot scale to the distributed setting. Hogwild! is also extended to handle large model that cannot fit in the memory with an intelligent disk access scheme [31].

Deep learning systems. Deep learning systems such as TensorFlow [1], MXNet [9], Caffe [19], Theano [2], Torch [40], CNTK [11] became popular due to the recent popularity of deep neural networks and the availability of GPUs. As mentioned in Section 2, deep learning systems provide a high-level abstraction, while PS systems focus on the low-level worker-server communication. When it comes to distributed execution, deep learning systems (e.g., MXNet, TensorFlow) still adopt the PS architecture as the underlying parameter management module. As SGD is widely used to train neural networks and existing PS systems is inefficient in handling SGD with growing batch size, the design ideas in FlexPS can also benefit deep learning systems.

Theoretical ML research. On the more theoretical side, many machine learning researches focus on sequential algorithms with convergence guarantee. Recently, a major breakthrough has been made by employing the variance reduction techniques, and algorithms including SVRG [21], SVRG++ [4], and Katyusha [3] are proposed. These algorithms are proven to be faster than SGD and achieve a linear convergence rate for smooth and strongly convex problems. The dynamic workloads of these algorithms make the traditional PS systems inefficient, but they can be implemented efficiently on FlexPS with our multi-stage design. As FlexPS provides a unified programming model for single machine learning and distributed learning, machine learning theory researchers can also experiment their sequential algorithms in the distributed setting with marginal effort.

9. CONCLUSIONS

We proposed FlexPS, a PS system that provides flexible parallelism control. We demonstrated that flexible parallelism control is crucial for the efficient execution of machine learning tasks with dynamic workloads. FlexPS supports flexible parallelism control with a novel multi-stage abstraction. To support the efficient execution of the multi-stage abstraction, system designs such as stage scheduler, stage-aware consistency controller, flexible model preparation, and direct model transfer are introduced. Optimizations such as customizable parameter partition, customizable KV-server, and repetitive *get* avoidance, are also incorporated to make FlexPS a general and complete PS system. Extensive experiments on a variety of machine learning tasks show that FlexPS achieves significant speedups and resource saving compared with existing PS systems.

Acknowledgments. We thank the reviewers for their valuable comments. This work was supported in part by Grants (CUHK 14206715 & 14222816) from the Hong Kong RGC.

10. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P. L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M. Côté, M. Côté, A. C. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. J. Goodfellow, M. Graham, Ç. Gülçehre, P. Hamel, I. Harlouchet, J. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrançois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P. Manzagol, O. Mastropietro, R. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. J. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. P. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[3] Z. Allen Zhu. Katyusha: the first direct acceleration of stochastic gradient methods. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1200–1205, 2017.

[4] Z. Allen Zhu and Y. Yuan. Improved SVRG for non-strongly-convex or sum-of-non-convex objectives. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1080–1089, 2016.

[5] Angel. https://github.com/tencent/angel.

[6] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[7] Bertsekas and D. P. A new class of incremental gradient methods for least squares problems. *SIAM Journal on Optimization*, 7(4):913–926, 1997.

[8] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012.

[9] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[10] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[11] CNTK. https://github.com/microsoft/cntk.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[13] A. Defazio, F. R. Bach, and S. Lacoste-Julien. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *NIPS*, pages 1646–1654, 2014.

[14] M. P. Friedlander and M. Schmidt. Hybrid deterministic-stochastic methods for data fitting. *SIAM Journal on Scientific Computing*, 34(3):A1380–A1405, 2012.

[15] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, 1984.

[16] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In *SoCC*, pages 98–111, 2016.

[17] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[18] Y. Huang, T. Jin, Y. Wu, Y. Guo, Z. Cai, X. Yan, F. Yang, J. Li, and J. Cheng. FlexPS: Flexible parallelism control in parameter server architecture. Technical Report. *http://www.cse.cuhk.edu.hk/proj-h/pub/flexps.pdf*, 2017.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *MM*, pages 675–678, 2014.

[20] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 463–478, 2017.

[21] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *NIPS*, pages 315–323, 2013.

[22] Y. Juan, Y. Zhuang, W. Chin, and C. Lin. Field-aware factorization machines for CTR prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, pages 43–50, 2016.

[23] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. STRADS: a distributed framework for scheduled model parallel machine learning. In *EuroSys*, pages 5:1–5:16, 2016.

[24] Y. Koren, R. M. Bell, and C. Volinsky. Matrix

factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[25] J. Li, J. Cheng, Y. Zhao, F. Yang, Y. Huang, H. Chen, and R. Zhao. A comparison of general-purpose distributed systems for data processing. In *IEEE International Conference on Big Data*, pages 378–383, 2016.

[26] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[27] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 661–670, 2014.

[28] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 469–477, 2014.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[30] Multiverso. https://github.com/microsoft/multiverso.

[31] C. Qin, M. Torres, and F. Rusu. Scalable asynchronous gradient descent optimization for out-of-core models. *PVLDB*, 10(10):986–997, 2017.

[32] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[33] S. J. Reddi, A. Hefny, S. Sra, B. Póczos, and A. J. Smola. Stochastic variance reduction for nonconvex optimization. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 314–323, 2016.

[34] C. D. Sa, C. Ré, and K. Olukotun. Ensuring rapid mixing and low bias for asynchronous Gibbs sampling. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1567–1576, 2016.

[35] S. Schelter, V. Satuluri, and R. Zadeh. Factorbird - a parameter server approach to distributed matrix factorization. *CoRR*, abs/1411.0602, 2014.

[36] M. W. Schmidt, N. L. Roux, and F. R. Bach. Minimizing finite sums with the stochastic average gradient. *Math. Program.*, 162(1-2):83–112, 2017.

[37] O. Shamir. A stochastic PCA and SVD algorithm with an exponential convergence rate. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 144–152, 2015.

[38] A. J. Smola and S. M. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1):703–710, 2010.

[39] A. S. Tanenbaum. *Computer networks, 4th Edition.* Prentice Hall, 2002.

[40] Torch. https://github.com/torch/torch7.

[41] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[42] S. Webb, J. Caverlee, and C. Pu. Introducing the webb spam corpus: Using email spam to identify web spam automatically. In *CEAS 2006 - The Third Conference on Email and Anti-Spam, July 27-28, 2006, Mountain View, California, USA*, 2006.

[43] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, pages 381–394, 2015.

[44] L. Xiao and T. Zhang. A proximal stochastic gradient method with progressive variance reduction. *SIAM Journal on Optimization*, 24(4):2057–2075, 2014.

[45] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[46] Yahoo! Webscope. http://webscope.sandbox.yahoo.com/.

[47] F. Yang, Y. Huang, Y. Zhao, J. Li, G. Jiang, and J. Cheng. The best of both worlds: Big data programming with both productivity and performance. In *SIGMOD*, pages 1619–1622, 2017.

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

[49] F. Yang, F. Shang, Y. Huang, J. Cheng, J. Li, Y. Zhao, and R. Zhao. LFTF: A framework for efficient tensor analytics at scale. *PVLDB*, 10(7):745–756, 2017.

[50] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T. Liu, and W. Ma. Lightlda: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361, 2015.

[51] H. Yun, H. Yu, C. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. NOMAD: nonlocking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.

[52] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 181–193, 2017.