FlexPS: A flexible and scalable machine learning system

Author: Zhanhao Liu, Supervisor: Prof. James Cheng

The Chinese University of Hong Kong

Abstract

FlexPS[1] is a general and scalable Parameter Server based system for distributed machine learning. FlexPS provides a novel multi-stage abstraction to support flexible parallelism control, which breaks down a machine learning task into different stages so that task can run in different locations with different amounts of computing resources. Furthermore, we proposed a new barrier synchronization model on FlexPS, called Sparse SSP, optimized for dealing with sparse datasets based on the Stale Synchronous Parallel (SSP) model[2]. We demonstrate how such a design in light of dynamic parallelism as well as loose synchronization control, achieves significant speedups and resource saving compared with well-known implementations of several ML programs on the state-of-the-art PS systems such as Petuum[3] and Multiverso[4].

Keywords: Parameter Server, Distributed System, Large Scale Machine Learning

1. Introduction

Machine learning (ML) has been widely applied in the industry today to make sense of the massive data available in this big data era. The scale of machine learning problems today is increasing in terms of both data volume and model dimensionality, and distributed methods have been widely employed for large-scale machine learning problems.

1.1. Iterative Convergent Algorithm

There are many machine learning algorithms, one major subset of them is the iterative convergent algorithms, which begin with a guess of model parameters and proceed through multiple iterations over the input data to optimize the solution iteratively. Such algorithms typically have an objective function to qualify the quality of the current model, and in each iteration consider each input datum individually to refine the current model parameters towards the optimal value of the objective function. Examples of iterative Convergent Algorithms include Stochastic Gradient Descent (SGD), Stochastic Average Gradient (SAG), stochastic variance reduced gradient (SVRG), Stochastic Coordinate Descent (SCD), Alternating Least Squares (ALS). Also, the corresponding machine learning models that these algorithms applied are Logistic Regression, k-means clustering, linear Support Vector Machines (SVMs), neural network, matrix factorization and so on.

1.2. Parameter Server

While the nature of the iterative convergent algorithm requires frequent access of the parameters, this may bring lots of network bandwidth in the distributed implementation of such kind of algorithm. Also, the training process of the iterative convergent algorithm is sequential, thus the synchronization between different workers is hard to scale. Li et al. proposed parameter server to solve the above problems in 2014.[9] Distributing the data and workloads over worker nodes while maintain globally shared parameters in server nodes, the parameter server framework is specialized to the nature of data parallel iterative machine learning problems.

2. Background

2.1. Related works

Since its introduction, the parameter server framework has been widely adopted in existing machine learning systems to scale distributed machine learning in both academia and industry.

ps-lite. ps-lite[5] is a light and efficient implementation of the parameter server framework that introduces a number of optimizations for efficiency and scalability. It provides an asynchronous and zero-copy key-value pair communications between worker and server nodes. Also, it supports server-side programming, which allows the user to define server-side handling functions.

Bösen. Bösen[6] is a communication-efficient parameter server and a module in Petuum[3] developed by SAILING Lab from CMU. It presents a data-parallel concept and adopts the Stale Synchronous Parallel (SSP) protocol, which can efficiently reduce the network synchronization costs while maintaining a bounded convergence guarantees. Also, Bösen introduce a table-like client API, which is natural and efficient for parameter access.

Multiverso. Multiverso[4] is a parameter server-based framework which is a core module of the Distributed Machine Learning Toolkit (DMTK) [7] from Microsoft. Multiverso abstracts the distributed model storage and operation, inter-process and inter-thread communication, multi-threading management, provides a series of friendly programming interfaces for the user to train machine learning models on big data with numbers of machines. Some popular deep learning frameworks like theano[8] and torch[9] are also supported in Multiverso.

However, there are some common limitations in these existing parameter server frameworks:

1. **Inflexible parallelism control**: Existing PS implementations do not provide flexible control over parallelism; however, we have found that various machine learning algorithms, from traditional stochastic algorithms to newly invented fast convergent methods, have large improvement space with fine-grained parallelism control.

2. **Limited generality**: Even though the PS abstraction is general enough to capture various single-machine computational frameworks including the *sequential* (*Seq*) framework (using only 1 worker), lock-free framework (disabling consistency control), and *single-process multiple-thread* (*SPMT*) framework (placing all workers in the same machine), there is a lack of specialized optimization for these frameworks.
3. **Lacking support for multi-tasking and resource sharing**: Existing PS implementations are primarily designed to run a single algorithm or train a single model, thus missing new opportunities brought by multi-tasking and resource sharing.

2.2. Sparse Data

The data dimensionality is growing larger and larger nowadays, which may not be able to reside in the main memory of a machine for statistic analysis. In most cases of the high dimension dataset, the data is actually sparse (with many records are missing). To save memory and reduce computation cost, people tend to only store the non-zero entries which maintaining the key-value pairs of parameters in the storage system. As a result, sparse machine learning problem has recently emerged as a powerful tool to obtain models of high-dimensional parameters with a lower computational cost.

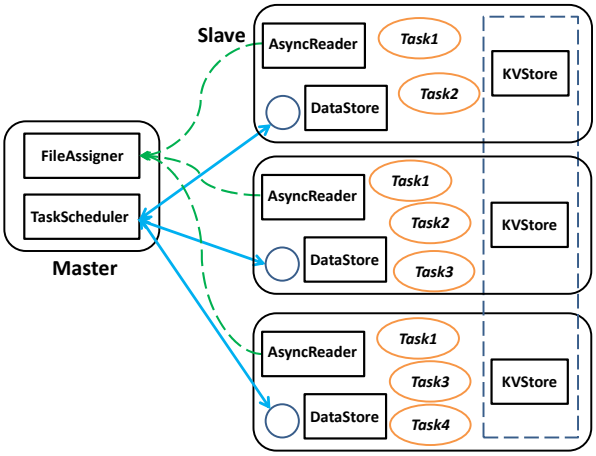


Figure 1: FlexPS System Architecture

3. System Architecture

3.1. Overview

Flexps follows the typical master-slave model in the distributed system area, as shown in Figure 1. In master process, there are *TaskScheduler* and *FileAssigner*. TaskScheduler distributes workloads to slave machines according to user-defined scheduling and slave availability, as well as tracking their progress. While the FileAssigner assigns data blocks to slaves when their corresponding tasks need to load data from the HDFS.

Each slave machine runs an event loop to poll events from the master. When a new task comes, it generates new threads to run the corresponding user-defined task. The slave machine also monitors its own tasks' process and notify the master once task finishes. The slave also have *AsyncReader* and a *DataStore* modules which control data loading and data storing. All the slave machines together maintain a shared global KV-store (maybe on one machine or on multiple servers) for parameter storing and accessing.

There are two kinds of workers in the slave machine, *ml-worker* as well as *kv-worker*. While *ml-worker* is in charge of the computation of the user-defined tasks, the *kv-worker* issues non-blocking *get()* and *put()* functions to communicate with the parameter server. We implement the worker as a thread but not process, so as to let the workers in the same process can share same loaded data in the *Datastore* of a process. Also, we implement a process cache strategy to maintain a most recently used parameter pool in each process to avoid workers repetitively put/get the model parameters.

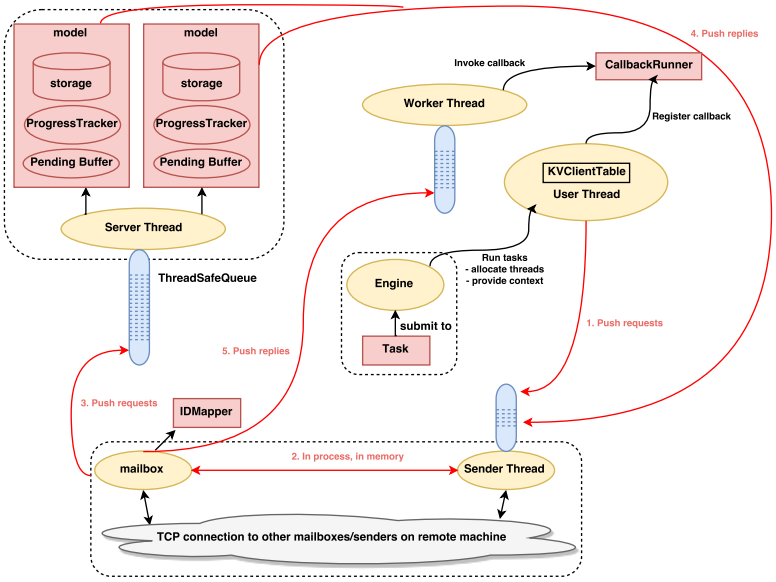


Figure 2: A typical work-flow of FlexPS

3.2. Workflow

The whole Flexps system can be generally divided into 4 modules, the server module, worker module, communication module (*mailbox*) and a driver module (user-defined machine learning tasks and scheduling, we call it *engine* in Flexps). A typical work-flow of a complete machine learning task in Flexps is as Figure 2.

1. User submits their tasks to the Engine (driver), then Engine will run the tasks and allocate the corresponding resources. The *KVClientTable* will register a *CallbackRunner* to handle the subsequent requests. After extracting the needed parameter for current iteration from the data, the user thread would push a request to the sender thread of the mailbox to ask for the corresponding parameters.

2. The sender thread process the request in a FIFO manner, using an IDMapper to find out the actual receiver's address in the cluster.
3. Mailbox send the *get*/*add* requests to the server to retrieve/update the corresponding parameters.
4. The server handles the requests and push replies to the sender thread of the mailbox.
5. Mailbox handles the replies and pushes replies to the worker thread, and worker thread invokes the callback runner to continue the user thread's tasks.

3.3. Task Scheduler

TaskScheduler decides which tasks to run according to task scheduling algorithms, which will consider task specications as well as task assignment history for scheduling. Task specication is provided by users to specify how much computing resource one stage of a task needs and where the computing resource should be allocated. Task assignment history is kept by TaskScheduler and can be useful to guide task scheduling.

A task scheduling algorithm needs to implement the *thread_nish* and *assign_task* callback functions. To notify TaskScheduler, thread nish will be called when a task nishes. TaskScheduler calls *assign_tasks* to apply the scheduling algorithm to assign new tasks to slaves when one of the following two conditions is satised: (1) the number of available threads exceeds a threshold; or (2) the time since the last scheduling exceeds a threshold. The two thresholds are user-congurable.

FlexPS provides three built-in task scheduling algorithms: (1) sequential scheduling, (2) greedy scheduling, and (3) prioritized scheduling. Sequential scheduling, which serves as a baseline, assigns tasks to slaves one by one. Greedy scheduling tries to assign as many tasks as possible to slaves. Once the *assign_task* function is called, greedy scheduling iterates through the *pending tasks list* and assign tasks to slaves in a greedy manner if their requirements are satised. Note that the optimal task scheduling algorithm can be obtained by modeling the problem as a *Multiple Knapsack Problem*[10]. But we do not consider the optimal algorithm since greedy scheduling works well in practice. In fact, even the optimal algorithm does not consider task waiting time, and may potentially starve tasks that require a lot of resources since they are much harder to be scheduled. Prioritized scheduling xes this problem by increasing the priority of tasks as task waiting time increases. When a task's priority exceeds a preset bound, it will be put into a *starvation list*. Each time when *assign_task* is called, tasks in the starvation list will be considered rst and lock the required resources to prevent starvation. Advanced users can also customize their own task scheduling algorithms by implementing *thread_nish* and assign tasks.

3.4. Data Loader

FlexPS supports reading les from Distributed File System (DFS). In a common DFS, e.g., HDFS, a le is split into one or more blocks and these blocks may have several replicas and are stored in multiple nodes. FileAssigner in the master keeps all the block information of each le, and all the loading threads ask FileAssigner for data blocks. FileAssigner assigns blocks to threads according to locality, such that local blocks will be read before remote blocks (in order to avoid reading a remote replica) for every loading thread.

FlexPS provides two ways for data loading. The rst way is to submit a specic loading task to the system and load the data before training tasks, i.e., we separate data loading and model training. We provide *DataStore* to store the loaded data in memory. Each thread is associated with a bucket in DataStore and it writes to its own bucket during loading and can access other chunks when training.

On the off chance that the dataset is vast, it needs quite a while to load the entire dataset. In this way, the second way that FlexPS used to load data from DFS is to load data on-the-y while preparing the model, which is finished by the *AsyncReader* module with a typical producer-consumer paradigm. Specifically, we use a reader thread to load data from DFS and then store the loaded data into a buffer. Then, we use several consumer threads to consume the data from the buffer and use them to train the model. This design overlaps the computation and network communication time perfectly.

3.5. Programming Model

FlexPS adopts a unified KV-Store API, shown below, the same way as other existing PS frameworks.

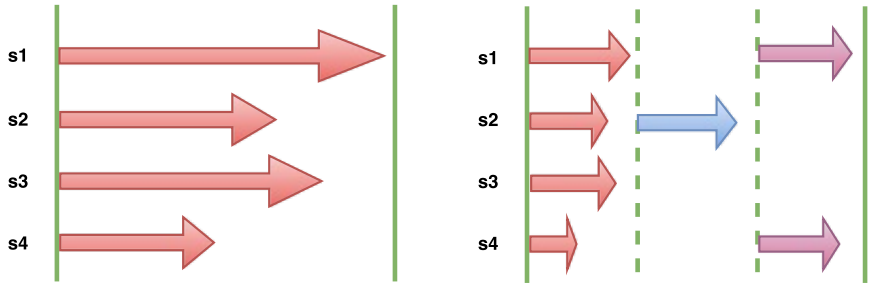
- *Get(keys)*
- *Add(keys, vals)*
- *GetChunks(keys, vals)*
- *AddChunks(keys, vals)*

All the *Add* and *Get* operations are non-blocking, which allows the worker thread to issue *asynchronous Add* and *Get* requests. We also provide the *chunk-based* API for more natural and efficient parameter accessing in some machine learning algorithms. Here we use the Logistic Regression example to illustrate how to use FlexPS to implement a machine learning task.

4. Multi-stage Design

Most existing PS frameworks keep a constant parallelism degree in the whole execution procedure of a training task and don't bolster changing the parallelism degree at runtime. This limit the performance of some machine learning applications since a large class of machine learning models have dynamic workloads in different stages of execution.

To resolve the limitation of current PS systems, FlexPS provides a flexible control over parallelism with a novel multi-stage design, which breaks down a machine learning task into different stages so that task can be assigned to run in different locations with different amounts of computing resources in different stages. Under the multi-stage design, a machine learning task is the composition of several stages. Each stage runs a user-defined function on particular computation resources with a different number of slaves and corresponding locations. (e.g., 10 slaves on node 1 and 5 slaves on node2). The traditional PS framework



(a) one stage (b) multi-stage

Figure 3: Workflow comparison

can be seen as a special case of the multi-stage PS in which a task has only one stage. The comparisons of a machine learning task execution work-flow in traditional PS system and multi-stage PS system is shown in Figure 3.

Beside the flexible parallelism control, the multi-stage design additionally brings another two advantages. First, it can facilitate data locality. Imaging if different stages of a machine learning tasks are training using different parts of the dataset, we can assign the task of each stage to the nodes containing the required data. Second, it supports fine-grained scheduling of multiple tasks, as we can run several stages of different training tasks in the cluster at the same time.

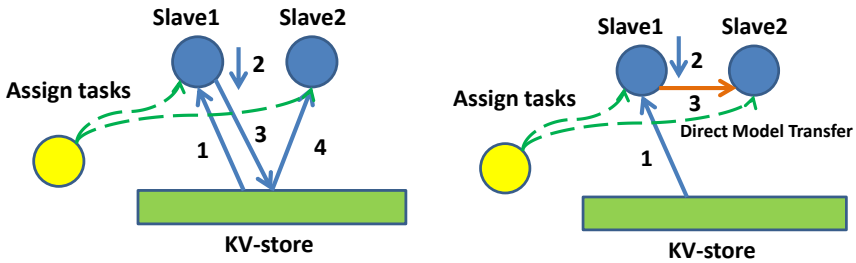
4.1. Direct Model Transfer

To avoid the overhead of loading/writing the model from/to the globally shared KV-store when we transform from one stage to another (shown as Steps 3 and 4 in Figure 4(a)), FlexPS provides a Direct Model Transfer feature to avoid these two processes. By Direct Model Transfer, the system can send the model to the node in which that the next stage will run and thus bypasses the KV-store (i.e., Step 3 in Figure 4(b)). Since a task does not know its execution position at the next stage (the decision is made by the TaskScheduler), the model will be kept in the current worker thread until the next stage of this task is scheduled. When the next stage is executed, the previous worker thread will be informed and send the model to the next worker.

5. Task Parallelism & Resource Sharing

We design a centralized task scheduler as well as different scheduling strategies to enable task parallelism control in our system, which facilitates the multi-stage design by scheduling tasks with the desired parallelism according to task locality, task preference, task scheduling history, and cluster utilization.

In FlexPS, all running tasks can share the same loaded data, so that the training data do not need to be loaded each time for each task. This is useful since it is common to use different machine learning strategies (e.g., different models, different algorithms, or different learning rates) to extract information from the same dataset. Multiple tasks are also sharing



(a) Changing stages w/o DMT (b) Changing stages w/ DMT

Figure 4: Workflow comparison

the same key/value-store module, and thus a testing task submitted after all the training tasks can retrieve the models of the training tasks. This could be useful for ensemble learning.

6. Local Consistency Control

To make the Seq and SPMT computational frameworks efcient on the PS architecture, we should reduce the network communications between the workers and servers as much as possible.

One of the important optimizations in FlexPS specically designed for Seq and SPMT is the local consistency control. We implement a *local model store* beside the global model stored in the KV-store. The local model store in FlexPS is different from the process-level cache in other PS-based systems in that it allows us to move the *consistency controller* from the server side to the local process, which signicantly reduces the synchronization cost over the network. All updates of the model and the consistency controls are performed inside the local worker process in order to avoid network communication with the globally shared KV-store.

6.1. Local model store

The local model accessing guidelines are as follows: (1) the working thread checks with the consistency controller whether it can access the model, if not, it will be blocked; (2) if the consistency requirement is met, this thread is granted to access the model; (3) after accessing the model, the consistency controller is informed and notifies all the blocked threads.

The local consistency controller maintains the *progress* of each worker and the *min progress* for stale synchronous parallel (SSP), where the min progress records the progress of the slowest worker. Multiple threads access and update the worker progress and min progress in the critical section. A worker thread will be blocked if it is faster than the min progress by a threshold specified by SSP. When the min progress is updated, threads blocked w.r.t. the old min progress will be notified.

6.2. Concurrency control

Since various threads may access the shared local model at the same time, concurrency control is implemented on the local model. We sort out the local model in chunks and attach a mutex to each chunk. It is a tradeoff between having a mutex for every single parameter (best concurrency control for accessing the model but not practical due to a large number of parameters) and having a lock for the whole local model. Besides, the chunk-based configuration also provides constant indexing time to a particular parameter (reading/writing a parameter, and checking whether a chunk is loaded). Experiments show that the locking process takes only 1-2% of the total time. This is because contention for the same mutex is not severe since we divide the model into chunks, and each read or write to the chunks is fast.

7. Repeated Pull Avoidance

In Distributed PS, multiple worker threads in one process may require the same key/value pairs from KV-servers. The extra *pull* cost can be saved by sharing the *pulled* key/value pairs within a process.We implement process cache together with a simple strategy to avoid these repeated requests.

Under SSP, each *Get* or *GetChunks* request should be returned only if the current versions of the key/value pairs in the process cache meets the user-defined staleness threshold. In the process cache module, we attach a linked-list to each chunk to keep the requests for it, ordered by versions. When a newer chunk is in need, the worker will check the version of the chunk and update the linked-list (fetch the chunk from global KV-store if the chunk's version does not meet the staleness threshold). The linked-lists are protected by mutexes to prevent duplicated fetch operations.

8. SparseSSP

Stale Synchronous Parallel (SSP)[2] is a relaxed version of the Bulk Synchronous Parallel model. It reduces the workers' waiting time for pulling parameter from the global KV-store by allowing distributed workers to read older, stale version of the parameter from a local process cache. SSP is part of the Petuum[3] system.

However, SSP fails to exploit the parameter accessing pattern when the *add*/*get* operations are sparse. We propose a new communication model specially optimized for dealing with sparse datasets, called *SparseSSP*. SparseSSP extends SSP by making use of the sparse access/update pattern of each worker. It has the following two features: First, it keeps the semantics of the SSP model, meaning that it shares the same theoretical correctness as SSP. Second, it allows fast workers to be more than s steps ahead of the slowest workers if there are no parameter access conflicts. These features make workers under SparseSSP spend even less time on waiting for remote parameter accessing compared with SSP when dealing with the sparse workload in the heterogeneous environment while preserving the similar convergence guarantees with SSP.

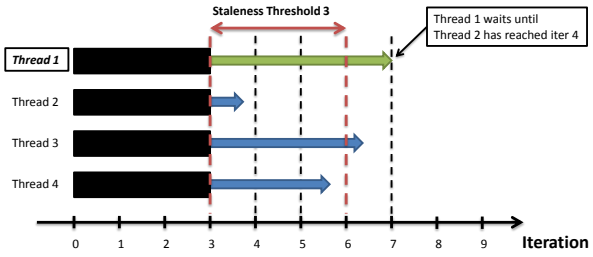


Figure 5: Bounded Staleness under the SSP Model[2]

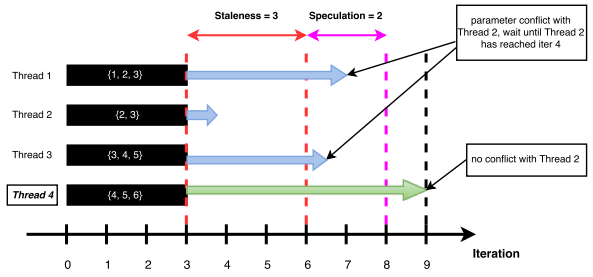


Figure 6: Bounded Speculation under the SparseSSP Model

8.1. Algorithm Analysis

SSP The PS systems that support SSP model use an integer-valued clock to keep track with the current iteration of each worker. For each machine learning task, the user can define a parameter called *staleness*. SSP allows different workers working on different iterations simultaneously, but the slowest and fastest workers must be $\leq$ *staleness* clocks apart. Otherwise, the fastest worker will be blocked until the slowest worker finished its current iteration. (see Figure 5 for a graphical illustration):

In Petuum's implementation, each worker thread caches local (stale) versions of the parameters, to achieve the within-staleness iterations parameter accessing to reduce network syncing.

SparseSSP We notice that even the fastest worker is faster than others by several iterations, but the parameters that it needs to access in the next iteration are normally not used by other workers in the case of the sparse dataset. In this situation, it's safe to let the fastest worker move on to next iteration since the parameters it gets from the server is still the updated (because no other workers would update these parameters in current iteration).

We introduces a new term called *speculation* in SparseSSP model. On the base of SSP, when the clock (current iteration) difference between the slowest and fastest workers reaches *staleness*, we check the parameters that these two workers are manipulating. If the parameters that there are using have no intersections, then we let the fastest worker move on for at most *speculation* iterations to further reduce the worker's waiting time.

The graphical illustration is shown in Figure 6 and the pseudo-algorithm of SparseSSP is as below:

Algorithm 1 SparseSSP Algorithm to check if worker i can move on to next iteration

Require: w_i: worker i c_i: the clock value w_i s: staleness sp: speculation P_i: the set of parameter that w_i is manipulating

initialize flag = True
for every worker w_j from all worker where $i \neq j$ **do**
    if $c_i - c_j \leq s$ **then**
        continue;
    else if $s < c_i - c_j <= s + sp$ **then**
        if $P_i \cap P_j = \emptyset$ **then**
            continue;
        end if
    end if
    flag = False;
    break;
end for
if flag = True **then**
    w_i can move on
else
    w_i waits until the slowest worker finish its current iteration
end if

Probability of conflicts Next, we show that in the sparse dataset, the probability of parameter conflict between the fastest and slowest worker is lower enough for the fastest worker to move on in most cases.

Consider the case of Stochastic gradient descent (SGD) algorithm. Suppose each record of a sparse dataset has n parameters, the number of parameter (non-zero parameter) that each worker needs for a iteration's training is m, and the set of parameters that the fastest/slowest worker manipulating is P_{fast}/P_{slow} Probability that P_{fast} and P_{slow} have no intersection:

$$P(no\_conflict) = \frac{\binom{n-m}{m}}{\binom{n}{m}}$$

Thus, the probability that the fastest and slowest workers have conflicts is:

$$P(conflict) = 1 - \frac{\binom{n-m}{m}}{\binom{n}{m}}$$

Here are some information frequent-used sparse datasets in machine learning and their corresponding probability of conflicts (where Sparsity is the ratio between the average number of non-zero features and the number of features of the dataset):

Table 1: Sparse datasets

Dataset	kdd12[11]	avazu[12]	criteo[13]	url[**?**]
# of features	$5.5x10^7$	10^6	10^6	$3.2x10^6$
# of samples	$1.5x10^8$	$4.0x10^7$	$4.6x10^7$	$2.4x10^6$
Sparsity	$2.0x10^{-7}$	$1.5x10^{-5}$	$3.9x10^{-5}$	$3.1x10^{-5}$
Pr of conflicts	$2.2x10^{-6}$	$2.2x10^{-4}$	$1.5x10^{-3}$	$3.0x10^{-3}$

As shown in the Table 1, in these frequent-used sparse datasets in machine learning, the probability that two workers are accessing the same parameter in quite small. It means that the fastest worker in the SparseSSP model can cross the *staleness + speculation* bounder in most cases. This gives us evidence that the SparseSSP model can greatly reduce the worker waiting time and increase the task execution efficiency in the case of sparse datasets.

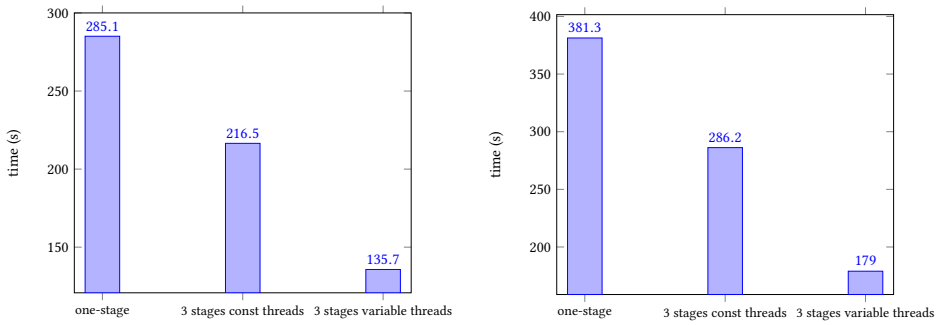
9. Evaluation

We evaluated the efficiency of the multi-stage design as well as the SparseSSP algorithm of FlexPS on our CSE Department's cluster. The cluster contains 20 nodes, each with two 2.0GHz E5-2620 Intel(R) Xeon(R) CPU, 48GB RAM, a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), connected via 1 Gbps Ethernet.

9.1. Multi-stage Abstraction

We mainly use two datasets, mnist8m[14] and svhn, to evaluate the performance of the Multi-stage Abstraction.

Multi-stage Abstraction We first demonstrate the flexibility of parallelism brought by FlexPS to adjust the parallelism degree across different stages of computation, and how it can speed up convergence and save computing resources. We used a mini-batch gradient descent algorithm to test the k-means task under three kinds of configurations of FlexPS: one-stage (160 workers for mnist8m and 80 workers for svhn), three-stage with constant parallelism (using 160 workers consistently in all stages for mnist8m and 80 workers for svhn), and three-stage with workload-adaptive parallelism (the number of workers changes according to the workload in each stage, where the numbers for the three stages are 20, 80, and 160 workers for mnist8m, and 20, 40, and 80 workers for svhn). The one-stage tasks train with mini-batch sizes of 8,100 for mnist8m and 700 for svhn. The three-stage tasks train with mini-batch sizes of 810, 8,100 and 81,000 for the three stages for webspam, and 70, 700 and 7,000 for svhn.

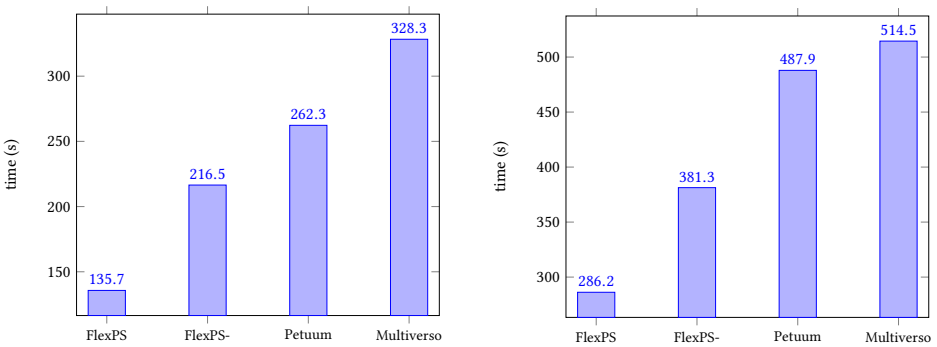


(a) mnist8m (b) svhn

Figure 7: Resource utilizaion of FlexPS

Figures 7(a), 7(b) show that the total execution time used by all worker threads with multi-stage parallelism use only half of the times of the traditional PS that uses constant computation resources in one stage for both mnist8m and svhn.

Comparison with other PS systems In this experiment, we compared FlexPS with Petuum and Multiverso on a mini-batch gradient descent k-means task with dynamic workloads. We denoted the one-stage FlexPS in above experiment by FlexPS-, in the comparison. FlexPS- simulates the way that Petuum and Multiverso do, adopting a single stage and a constant parallelism degree for a task. The number of workers used in is 160 in FlexPS-, Petuum, and Multiverso for mnist8m, and 80 in FlexPS-, Petuum, and Multiverso for svhn. The number of workers used in the multi-stage FlexPS is the same as the above experiment.



(a) mnist8m (b) svhn

Figure 8: Comparison

Figures 8(a), 8(b) show that the execution times for FlexPS are nearly half of the times used by Petuum and Multiverso for both mnist8m and svhn datasets. Even when disabling the exible parallelism control, FlexPS- still use less time than these two systems. This demonstrates the advantage of the flexibility of parallelism control of FlexPS over the existing systems.

Table 2: Experiment setup

workers per node	10
servers per node	1
number of iterations	1000
number of dimentions	10000000

Table 3: Experiment parameter

# of non-zero	10	100	1000	10000
Sparsity	10^{-6}	10^{-5}	10^{-4}	110^{-3}
Pr of conflicts	10^{-5}	10^{-4}	$9.5x10^{-2}$	0.9955

9.2. SparseSSP

As we want to explore the performance of SparseSSP in a heterogeneous cluster environment and verify its effectiveness to cope with the straggler problem, we fictitiously inject some slow machines in the cluster using the methods in [15] and compare the execution time of SparseSSP with BSP, ASP, SSP.

We list some of the parameters that we used in the experiments it Table 2, 3:

Figure 9 shows the comparison results of the execution times of different parallelism protocols in sparse data. From both Table 3 and Figure 9, we can see that when the sparsity is low ($10^{-6} - 10^{-4}$), the SparseSSP model can effectively reduce the total execution time of the worker. When the sparsity is extremely low ($< 10^{-6}$), the time that workers used to finish the same task is nearly the same as the optimal time (The time of the SSP model where staleness $= 5$ is the optimal time for our SparseSSP model, since it allows the fastest worker to surpass the slowest worker for 5 iterations without any condition).

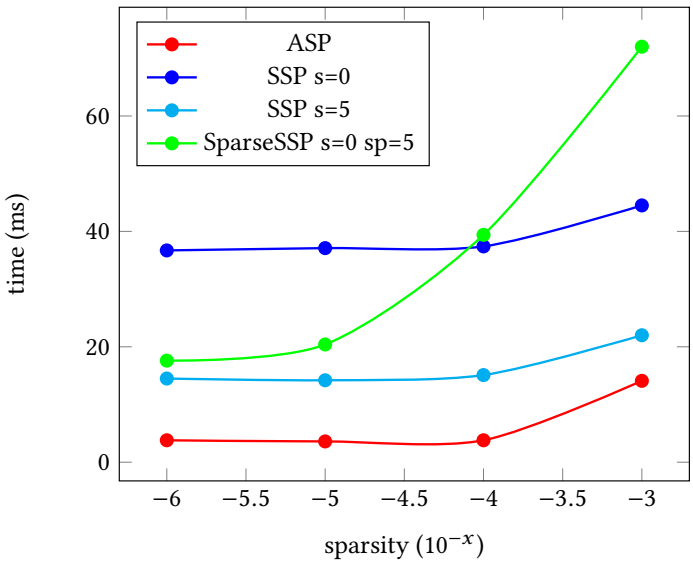


Figure 9: Comparison of different parallelism protocol in sparse data

However, when the sparsity is high ($> 10^{-4}$), we can know from Table 3 that the probability of parameter conflicts between two workers is high (nearly 1 when sparsity is 10^{-3}). Thus it's nearly impossible for the SparseSSP model to let the fastest worker move on to next iteration if it's highly likely to have parameter conflicts with other workers. We even waste some time to check for parameter conflicts and resulting in the higher execution time than the SSP model when the sparsity is high as shown in Figure 9. This experiment's results show that the SparseSSP model performs very well in the highly sparse datasets, but may not be that useful in those datasets with higher sparsity.

10. Conclusion

We proposed FlexPS, which is a distributed machine learning system that provides a flexible and general design of the traditional PS architecture. Several new system designs and optimizations designs such as the multi-stage abstraction, task parallelism, as well as SparseSSP parallelism protocol, enable FlexPS to provide a unified framework and a more efficient implementation for distributed parameter server, thus supporting a wide range of machine learning applications. We demonstrated that each of the designs and optimizations in FlexPS brings effective performance improvements in different aspects, allowing FlexPS to address the limitations of other existing parameter server systems on both the flexibility of parallelism control and the performance when dealing with straggler problems when training on highly sparse datasets.

11. Acknowledgement

The author would like to express his special thanks to Yuzhen Huang, Shuo Tu, Ruoyu Wu, Tatiana Jin, and other members of Husky Team for their kind guidance and generous assistance. This work is done under the leadership of Yuzhen Huang, in cooperation with Shuo Tu, Ruoyu Wu, Tatiana Jin, and other members of Husky Team. The author would also like to thank Prof. James Cheng for offering him such a great opportunity to explore the beauty of distributed system and machine learning.

References

[1] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, J. Cheng, Flexps: Flexible parallelism control in parameter server architecture, PVLDB 11 (2018) 566–579.

[2] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, E. P. Xing, More effective distributed ml via a stale synchronous parallel parameter server, in: In NIPS, 2013.

[3] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, Y. Yu, Petuum: A new platform for distributed machine learning on big data, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '15, ACM, New York, NY, USA, 2015, pp. 1335–1344. doi:10.1145/2783258.2783323.
URL `http://doi.acm.org/10.1145/2783258.2783323`

[4] Microsoft, Multiverso, `https://github.com/Microsoft/Multiverso` (2013).

[5] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, B.-Y. Su, Scaling distributed machine learning with the parameter server, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 583–598.
URL http://dl.acm.org/citation.cfm?id=2685048.2685095

[6] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, E. P. Xing, Managed communication and consistency for fast data-parallel iterative analytics, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, ACM, New York, NY, USA, 2015, pp. 381–394. doi:10.1145/2806777.2806778.
URL http://doi.acm.org/10.1145/2806777.2806778

[7] Microsoft, Dmtk, https://github.com/Microsoft/DMTK (2016).

[8] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, Theano: A cpu and gpu math compiler in python, in: S. van der Walt, J. Millman (Eds.), Proceedings of the 9th Python in Science Conference, 2010, pp. 3 – 10.

[9] R. Collobert, K. Kavukcuoglu, C. Farabet, Torch7: A matlab-like environment for machine learning, in: BigLearn, NIPS Workshop, 2011.

[10] T. Yamada, T. Takeoka, An exact algorithm for the fixed-charge multiple knapsack problem, European Journal of Operational Research 192 (2) (2009) 700–705.
URL https://ideas.repec.org/a/eee/ejores/v192y2009i2p700-705.html

[11] kaggle, kdd12, https://www.kaggle.com/c/kddcup2012-track1 (2012).

[12] kaggle, avazu, https://www.kaggle.com/c/avazu-ctr-prediction/data (2014).

[13] C. Labs, criteo, http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/ (2014).

[14] G. Loosli, S. Canu, L. Bottou, Training invariant support vector machines using selective sampling, in: L. Bottou, O. Chapelle, D. DeCoste, J. Weston (Eds.), Large Scale Kernel Machines, MIT Press, Cambridge, MA., 2007, pp. 301–320.
URL http://leon.bottou.org/papers/loosli-canu-bottou-2006

[15] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, E. P. Xing, Addressing the straggler problem for iterative convergent parallel ml, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, ACM, New York, NY, USA, 2016, pp. 98–111. doi:10.1145/2987550.2987554.
URL http://doi.acm.org/10.1145/2987550.2987554