National Science Review
00: 1–21, 2017
doi: 10.1093/nsr/nwx018
Advance access publication 24 February 2017

INFORMATION SCIENCE

Angel: a new large-scale machine learning system

Jie Jiang[1,2,†], Lele Yu[1,†], Jiawei Jiang[1], Yuhong Liu[2] and Bin Cui[1,*]

ABSTRACT

Machine Learning (ML) techniques now are ubiquitous tools to extract structural information from data collections. With the increasing volume of data, large-scale ML applications require an efficient implementation to accelerate the performance. Existing systems parallelize algorithms through either data parallelism or model parallelism. But data parallelism cannot obtain good statistical efficiency due to the conflicting updates to parameters while the performance is damaged by global barriers in model parallel methods. In this paper, we propose a new system, named Angel, to facilitate the development of large-scale ML applications in production environment. By allowing concurrent updates to model across different groups and scheduling the updates in each group, Angel can achieve a good balance between hardware efficiency and statistical efficiency. Besides, Angel reduces the network latency by overlapping the parameter pulling and update computing and also utilizes the sparseness of data to avoid the pulling of unnecessary parameters. We also enhance the usability of Angel by providing a set of efficient tools to integrate with application pipelines and provisioning efficient fault tolerance mechanisms. We conduct extensive experiments to demonstrate the superiority of Angel.

Keywords: machine learning, distributed system, algorithms, big data analytics

[1]Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University, Beijing 100871, China and [2]Data Platform, Tencent Inc., Shenzhen 518057, China

[†]Equally contributed to this work.

[*]**Corresponding author**. E-mail: bin.cui@pku.edu.cn

Received 19 September 2016; **Revised** 20 December 2016; **Accepted** 23 February 2017

INTRODUCTION

Machine Learning (ML) has became an indispensable workload to support in enterprise environment—by supporting it, the users can make predictions for various applications, such as commercial recommendation and online advertisement. In this paper, we focus on building a system to support distributed ML for industrial users.

(Use Case) Tencent: our motivation draws from the real production environment of Tencent, one of the largest Internet companies in China [1,2]. The cluster that we have access to at Tencent contains thousands of machines, and the analytical team needs to run a range of ML tasks, from simple models such as Logistic Regression (LR), to sophisticated models such as Latent Dirichlet Allocation (LDA), over datasets that are often hundreds of gigabytes or even terabytes. The existing ecosystem other than the ML stack is mainly based on Java.

This environment imposes challenges of directly applying existing distributed ML systems such as Spark [3,4] and Petuum [5], which we have tried at first.

(i) Integration with existing ecosystems: To integrate easily with existing ecosystems, ideally, our users wanted a system that is written in Java and can be deployed easily with HDFS and Yarn. In fact, in our experience, it is difficult to deploy systems such as Petuum in their environment. However, this imposes a question about performance—can we achieve reasonable speed with a Java-based distributed ML system instead of C/C++-based systems?
(ii) One-system-fits-all: existing ML systems either support data parallelism [3,6,7] or model parallelism [8,9,10]. Because of the diverse range of applications that we need to support, our users wanted a 'single' system to support all these different types of parallelisms. This requires us to carefully design the system architecture to accommodate both type of parallelisms in an effective and efficient way.

Motivated by these challenges, we build Angel, a distributed ML systems based on Java. Angel is used in the production environment of Tencent to run models including LR, Support Vector Machine (SVM), LDA, GBDT, KMeans, Matrix

Factorization (MF) for commercial recommendation, user profiling and various applications. We have successfully scaled Angel up to hundreds of machines and processing hundreds of gigabytes of data with billions of model parameters. Compared with existing systems such as Petuum and Spark, Angel is at least 2×, and sometimes up to 10× faster.

Premier: data parallelism, model parallelism and synchronization. We provide a brief overview of existing distributed ML systems to set the context for the system design and technical contribution of Angel.

One technical challenge in distributed ML is to achieve parallelism. In existing systems, parallelism is achieved by either data parallelism [3,6,7] or model parallelism [8,9,10]. By partitioning training instances and assigning them to different workers, data parallel methods can calculate the updates to the model parameters in parallel. But as the model has to be replicated and synchronized among all workers, data parallel methods incur significant network and memory overheads. Besides, data parallelism damages the statistical efficiency due to the conflicting updates on parameters [8,11], which means it needs more steps until convergence to a give tolerance. Different from data parallel methods, it is the model but not the data that is partitioned in model parallel methods. Model parallelism allows the deployment of the models with billions or trillion parameters that cannot be placed in one worker node. Good statistical efficiency can also be obtained by carefully scheduling the updates of parameters. But the overall performance of ML applications may still be harmed by the computation barriers in the model parallel methods.

The existence of model replicas brings in the demand to synchronize model parameters between different workers. Existing systems [5,6] proposed to employ parameter servers and delayed synchronization protocols to exchange the updates to the parameters. In practice, we observe that the convergence rate of ML applications is heavily related to the frequency of model synchronization. Typically, an ML algorithm converges faster with more frequent model synchronization. However, synchronization operations, including the pulling of parameters and the pushing of updates, introduce extra network overheads. Hence, an efficient implementation of synchronization operations is needed to reduce the incurred overheads.

System design of Angel. To address the aforementioned problems, we propose a new distributed system, named Angel, which is deployed in Tencent Inc. for ultra large-scale ML applications.

Similar to data parallel methods, Angel partitions training data into different groups and replicates the model in each group. Further, to improve the statistical efficiency, Angel partitions the model in each group and schedules the updates to the model. Compared with model parallelism, the hybrid parallel method in Angel allows concurrent updates to the same model parameters. The updates made in different groups then are merged to obtain the updated version of the model. By allowing a larger degree of parallelism, Angel can significantly improve the performance of ML applications.

Because the parameters in most ML applications such as LDA and MF can be modeled as matrices, Angel abstracts them as matrices to facilitate the development of ML applications. The matrix abstraction allows users to describe operations with linear algebra. In ML applications such as LR and SVM, the model parameters are matrices with only one row.

The parameter matrix is partitioned through a 2D method to generate a moderate size for each partition and balance the load of each server. The pulling of parameters and the pushing of updates are fully optimized in Angel to reduce network latency and bandwidth consumption. We perform the pulling of parameters and the computing of updates one row after another. By overlapping the pulling and the computing of different rows, we can reduce the performance degradation due to network latency. We also utilize data sparseness to reduce the amount of parameters to pull.

To efficiently integrate Angel with the whole pipeline of online ML applications, we provide a data pre-process module that is tailored for ML tasks to convert the raw data to ready-to-run linear algebra objects. In order to reduce the memory consumption caused by training data, Angel provides three storage levels for users to choose according to the resource usage of the cluster.

Angel now is deployed in a production cluster with thousands of nodes that is shared by many other applications. To ensure resource isolation, we utilize Yarn for resource management. We also provision efficient mechanisms to provide fault tolerated execution at scale.

We implement many common ML algorithms with Angel and conduct empirical experiments to study the performance of Angel. The experimental results demonstrate that Angel can significantly improve the performance of most ML application, exhibiting much better performance than state-of-the-art systems.

Summary of technical contributions. In summary, this paper makes the following contributions.

(i) We design and implement a system Angel for large-scale ML applications in production environment.

(ii) We propose a hybrid parallel method to allow the scheduling of updates while avoiding global barriers.
(iii) We carefully optimize the implementation of model synchronization. By deploying pipelined execution and utilizing data sparseness, we can significantly reduce the overheads due to model synchronization.
(iv) We enhance the usability of Angel by providing a set of built-in modules and provisioning efficient fault-tolerance mechanisms.
(v) We validate the effectiveness of Angel by conducting comprehensive experiments on the implementation of Angel.

Overview. The rest of this paper is organized as follows. We first review the literature in the 'Related work' section. The overview of Angel system, the hybrid parallelism, the parameter synchronization architecture and the details of system implementation are presented in the following four sections. The 'Evaluation' section presents the comparison between Angel and state-of-the-art ML systems. Finally, we conclude our work in the last section.

RELATED WORK

ML systems. Distributed systems were proposed in the past few years to help deal with large-scale ML problems. Data-flow systems [3,12], such as Hadoop and Spark, are widely adopted to conduct big data analysis in many companies. They provide easy programming interfaces for general distributed data processing and can conduct data processing through efficient data-flow operators, such as map, join and shuffle. However, as these systems lack globally shared states, they are unsuitable for ML applications where the model is shared in the computing of updates. Besides, without explicit interfaces for communication and storage, users cannot optimize the performance with customized scheduling and synchronization.

Graph processing systems provide graph-parallel programming models where the execution is organized as a graph and users can describe their applications by defining the behaviors of vertices [13,14,15]. Though the graph-parallel model can provide good abstraction for most ML algorithms, the fine-grained dependencies required in the execution limited the scalability of graph processing systems.

Systems which adopt parameter servers [5,6,16,17] partition parameters across multiple servers to perform the parameter exchanging among different workers. By performing update aggregating and allowing delayed synchronization, parameter servers can achieve acceptable performance for big models with billions or trillions of parameters. Petuum is a state-of-the-art distributed ML system, which provides two different modules, Bösen and Strads, to parallelize ML algorithms. Bösen models the parameters as a shared table and partitions the table horizontally across different servers. Different synchronization protocols are supported in Bösen to reduce the unnecessary network overheads. Strads employs a centralized coordinator to schedule the updates to the model. After the end of each iteration, a global barrier is invoked to perform the aggregation operations and select the parameters to update in the next iteration.

TensorFlow [18] is a distributed ML system developed by Google Inc. It provides the data-flow model to simplify the programming of deep learning algorithms and utilize shared variables to support efficient parallelization of ML algorithms. In the distributed environment, it enables synchronous and asynchronous protocols to accelerate the training of large datasets. But TensorFlow is designed to utilize multiple GPUs to accelerate the computation-intensive tasks, especially for deep learning applications. It cannot handle computation graphs with billions of nodes, like large-scale Bayesian graphs. Moreover, TensorFlow lacks the mechanism of delay-bounded synchronization, which can significantly improve the performance of distributed training.

Singa [19] is a distributed deep learning platform with layer abstraction-based programming model. Singa integrates both synchronous and asynchronous protocols to enable users to train with various distributed training frameworks including Sandblaster, AllReduce, Downpour and Distributed Hogwild!. However, its layer-based programming model cannot be employed for general ML algorithms. Although hybrid parallelism is also supported in Singa, it is utilized to deal with the big model problem in large deep neural networks training. Angel can still obtain speedup even when training with small models through hybrid parallelism.

Parallelization of learning algorithms. The training of ML algorithms is essentially sequential processes where the outputs of the current iteration are fed as input to the next iteration. There exist trade-offs between hardware efficiency and statistical efficiency in parallelizing optimization algorithms such as Stochastic Gradient Descent (SGD), ADMM and Coordinate Descent (CD) [20]. Various parallelized versions [21,22,23] of these optimization algorithms have been proposed to accelerate the training. The parallelization strategies

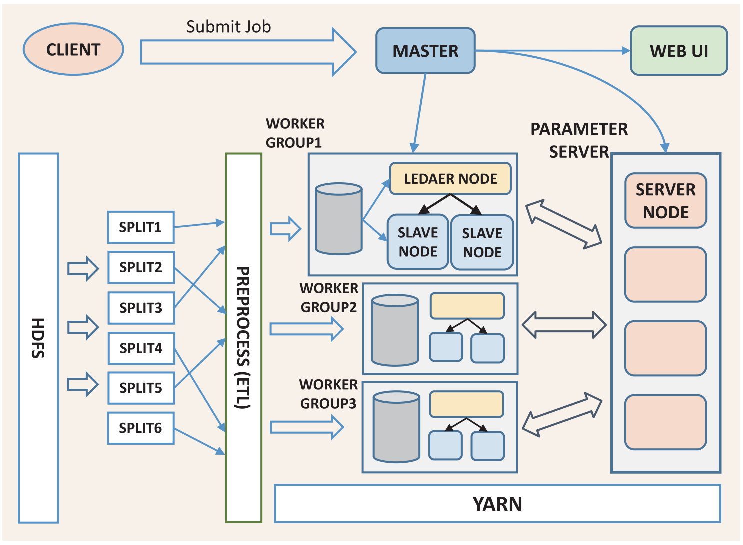


Figure 1. System architecture of Angel.

can be categorized into two types: data parallelism and model parallelism. The data parallel methods partition the training data to parallelize the ML algorithms while the model is partitioned in model parallel methods. Data parallel implementations can obtain better hardware efficiency since there is no interference between different workers during training. Model parallel implementations however are good at statistical efficiency because all the running tasks are coordinated to avoid concurrent updates to the same parameter. The study based on a single NUMA machine [11] has been conducted to find good trade-off point to balance the two aspects.

Synchronization protocols. Parallelizing ML algorithms brings in the requirement of model synchronization among different model replicas and partitions. Various synchronization protocols have been proposed and are applied to coordinate the running states of different workers. Bulk Synchronization Parallel (BSP) [24] protocol forces all running workers to synchronize their states through a global barrier at the end of each epoch. This protocol guarantees the correctness of distributed ML algorithms [21], and many existing systems [3,25,26] adopt it as their synchronization protocol. Delayed synchronization protocol [7,27,28] has been proposed to reduce the waiting time wasted in the BSP protocol. Recently, Asynchronous Synchronization Parallel (ASP) protocol is employed by some systems [16,6] and algorithms [29,30,31] to speed up distributed ML. Due to the error tolerance property of ML algorithms, these delayed synchronization protocols perform well and usually can obtain better performance for the problems with large training data collections.

DESIGN AND OVERVIEW OF ANGEL

Distribute ML has now been widely adopted to solve problems with Big Data and Big Model in current Internet companies. A general distributed ML system is necessary to help users develop various ML applications. Usually, the production environment imposes several constraints on the implementation and execution of an ML system, such as the limited memory and shared network resources. Therefore, we design and implement Angel to facilitate the development of ML applications facing practical problems. Figure 1 presents an overview of Angel's system architecture. In the following part of this section, we introduce the modules of Angel, and how Angel executes ML applications.

Components of angel

There are four types of roles in an application of Angel, including Client, Master, Worker and Server.

(i) Client is the entry point for an Angel application. It loads the configurations of the runtime environment and submits the application to Yarn. Then it will wait until the application's execution completes. Users can define the parameter matrices by specifying the block size and the dimension numbers at Client.
(ii) Master is launched by Yarn to manage the life cycle of each application, including requesting resources, launching containers and killing workers or servers. Besides that, it is also utilized to realize different synchronization protocols between workers and provide web pages that present the running states of workers and servers to users.
(iii) Worker acts as the computation node to execute the code written by users. Each worker reads the data from HDFS and accesses the model parameters on servers to exchange the updates with each other.
(iv) Server acts as a distributed storage for parameters. Each server stores one or more partitions of model parameters and provides efficient mechanism in response to get the update requests from workers.

Angel execution

An Angel application starts when users submit their compiled code which contains the logics of their task through Client. Then Yarn will launch Master to take over the running process. It will request resources from Yarn's resource manager to allocate containers for servers and workers. The model is partitioned into different partitions and the meta information of all partitions is stored at the master. After the initialization stage, each server will fetch the meta information of the partitions from the master and allocate memory storage for these partitions.

The training data are stored in HDFS. Master will partition the input data into different splits with balanced size and send the meta information of splits to workers. Before the running of ML algorithms, Worker conducts the preprocessing phase to convert the raw data into ready-to-run linear algebra objects. The output objects will be stored inside each worker. We will introduce the detailed implementation of pre-processing and data storage in the 'system implementation' section.

Before training, Master first organizes the workers into a set of groups. Then, the training data are partitioned into different splits, and the model is replicated at each group. Inside each group, a leader is selected to schedule the updates to model and the others will be categorized as slaves. The updates from different groups are aggregated to obtain a global model with the updated version. Hybrid parallelism well generalizes existing parallelization strategies. Users can execute distributed ML through data parallelism by only launching one worker inside each group. In such cases, the execution is exactly the same as those of data parallel methods. If the number of groups is set to 1, then the execution resembles model parallelism. Users should provide scheduling and computing functions for leaders and slaves, respectively.

Server acts as a distributed storage for parameters. It provides *get* and *increment* interfaces for workers to exchange parameters. During the training, each worker first pulls the parameters it needs using the *get* method. After computing the updates to the model, the updates are pushed back to the servers. Angel carefully optimizes the pulling operations, deploying various methods for different ML algorithms. For the algorithms like LDA or MF, whose models have a large amount of rows, the pulling operation is overlapped with the computation operation to reduce the network latency. The sparseness of training data is also utilized to reduce the amount of parameters to pull.

During the execution, the running states of Worker and Server are maintained at Master and presented to users through web pages. Besides, Server writes its snapshot into HDFS periodically. Once a server comes across a failure, the crash will be detected by Master through the heartbeat information or the report from workers. Then Master will allocate a new container to launch a new server and recover the state of the crashed one from its latest snapshot. At the end of each Angel application, the parameter values stored at servers will be written into the HDFS or pushed directly to the online advertising systems.

In summary, Angel is developed to solve large-scale ML problems in production environment. It provides the hybrid parallel method to accelerate the convergence speed of distributed ML algorithms. Besides, efficient parameter synchronization operations are implemented to reduce the network overheads and improve the overall performance. By automatically conducting the data pre-processing operations, Angel can be easily integrated with the whole pipeline of online applications. Angel also provisions efficient mechanisms to achieve fault tolerated execution at scale.

HYBRID PARALLELISM

For ML applications involving terabytes of data and billions of parameters, training on a single machine often takes intolerable times. Parallelizing ML programs across multiple machines is a practicable method adopted by many studies [21,23,29,32,33].

Existing systems adopt either data parallelism or model parallelism to parallelize ML algorithms. Figure 2a and b describe the architecture of the two parallel methods. In data parallel methods, the data are partitioned and the partitions are assigned to workers. Each worker owns a replica of the model. After a worker completes its computation, the updates made by the worker are pushed into the global shared model. When other workers pull parameters from the global shared model, these updates will be reflected in the computation. Since the computation performed at different workers are independent, asynchronous synchronization protocols are widely adopted to reduce the network overheads.

Different from data parallel methods, model parallel methods do not partition the training data. Instead, the model is partitioned and each worker is responsible for the updates of a portion of parameters. Since the parameters do not obey the i.i.d. properties, a coordinator is needed in model parallel methods to select those parameters that are independent from each other. By carefully scheduling the updates to the model, model parallel methods can converge with fewer epochs. But the scheduling of updates also enforces global barriers in the execution. Each worker cannot proceed to the next iteration when it completes the computation of the current iteration. It must wait for the completion of other workers as well as the notification from the coordinator which informs the parameters to update in the next iteration. Since the number of independent parameters is limited, model parallel methods also suffer from the small degree of parallelism.

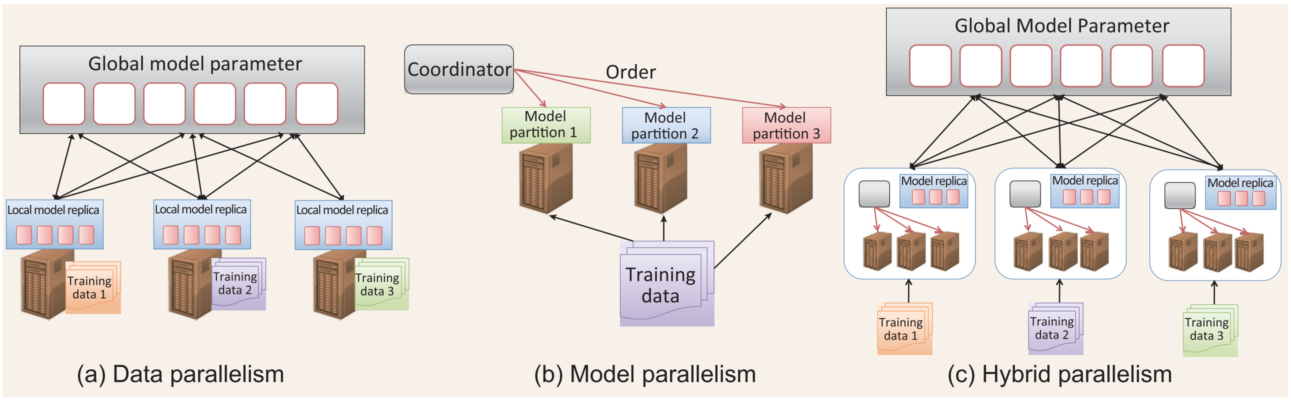


Figure 2. Various parallelization strategies for distributed ML: (a) data parallelism, (b) model parallelism and (c) hybrid parallelism.

Angel employs hybrid parallelism to avoid the problems in data parallelism and model parallelism. The architecture of hybrid parallelism is presented in Fig. 2c. Angel first divides workers into multiple groups. The training data are partitioned and are assigned to the groups. Inside each group, one worker is selected as the leader randomly and other workers are categorized as slaves. The model is replicated at different worker groups. The leader schedules the updates of the model replica inside this group. The updates generated by different groups are merged through the global parameter servers.

Data partitioning

We recognize the training data as a 2D matrix with each row as a data instance. Because the methods deployed by different ML algorithms to access data vary a lot, Angel provides three different methods to partition the data matrix inside a worker group and they are listed as follows.

(i) Replicated: the data matrix is replicated at each slave since the update of each parameter requires the whole data matrix. For example, to update each parameter in Lasso using CD, the whole data matrix is required.

(ii) Horizontal: in ML algorithms such as LDA and MF, all fields of an instance are needed in the computation. Hence, the training data are horizontally partitioned in these applications so that each slave owns a portion of the data instances.

(iii) Vertical: using this partitioning method, each training instance is partitioned and each slave owns a vertical partition of all data instances. Linear models, such as LR, SVM, can utilize this partition method to conduct model parallelism.

Users can choose the partitioning method through the configuration of the Angel application. During the read phase, each worker in a group will sweep all the training instances in this group and only store the partition which belongs to itself.

Worker execution

Inside each worker, multiple user tasks will be launched to execute the user-provided function in different threads. Each user thread reads its data partition and conducts the training operations. The user thread can access the parameters at servers through a shared module inside each worker, which is presented in Fig. 3. To eliminate duplicated parameter pulling, a lock table is maintained at the worker. Each request first tries to acquire the lock with its matrix ID and row index. If the lock has already been obtained by other thread, then the request thread will be blocked. The row locks allow us to avoid duplicated requests.

The pulling for one row will be splitted into multiple requests and each request fetches one partition from the server. The pulling requests are executed by multiple pulling threads. The pulling threads locate the server location from the lookup table inside worker. Once a partition is fetched, it will be inserted into a result buffer. An input merging thread will merge multiple partitions into a complete row once all partitions have been fetched. Then, all user threads which are waiting for this row will be waked up to perform the computation.

The updates generated by user threads will be inserted into an update buffer. An output merging thread is running to merge the updates before pushing. Once the buffer size exceeds a pre-defined threshold, the updates will be pushed to servers by the pushing threads.

Leader scheduling

The detailed model parallel implementations of different ML algorithms can vary a lot due to their

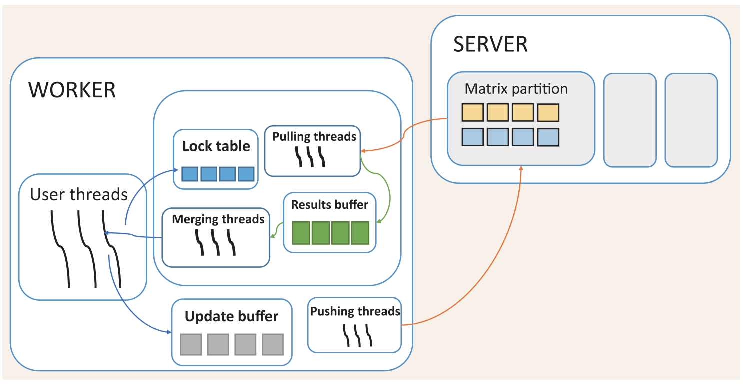


Figure 3. Execution inside worker.

```
// leader function
leader_run(slave_list):
  for (int i = 1; i < max_iteration; i ++)
    // Partition model
    for (int j = 1; j < slave.size; j ++)
      params_id_list = partition_model(j)
      for slave in slave_list
        send(UPDATE,slave.id,params_id_list(slave.id))
      wait ACKNOWLEDGE
    call CLOCK

// slave function: training one round
slave_run(leader, params_id_list):
  parameter = PS.get(params_id_list)
  foreach sample in this.data
    update = train(sample, parameter)
    PS.inc(update)
  return ACKNOWLEDGE;
```

Figure 4. The pseudo code for scheduling.

distinct scenarios. Angel provides users a set of flexible interfaces. Users can develop their applications by defining the scheduling and updating functions. Here, we present an example in Fig. 4 that arranges parameter updating in a round robin style. A leader function is running to schedule the computation of slaves. It controls how many iterations the computation will last. For each iteration, it schedules the parameters to each slave rotationally. This rotation-style scheduling guarantees that slaves will never touch the same parameter concurrently while each slave can access the whole parameters at one iteration. For each scheduling round, leader sends the UPDATE command to each slave with its corresponding parameter indices. Each slave receives the command, pulls the parameters with the indices and computes updates with its own data partition. The updates will be pushed into the servers through the increment interface. At the end of each round, each slave returns an ACKNOWLEDGE message to the leader. The leader proceeds to next computation round once all the ACKNOWLEDGE messages are received.

The master maintains the iteration states for all workers. Inside one worker group, the execution is coordinated with the BSP protocol. Therefore, the iteration number of all slaves is the same as the leader's. Each time the leader calls the CLOCK operation, the iteration number of its slaves will increase by one.

The execution of different groups however does not need to be synchronous. Various synchronization protocols can be deployed among different groups, including BSP, SSP and ASP. Before conducting the pull operation, each worker will request the minimum iteration number of all workers from the master. Once this gap of iterations has exceeded a pre-defined value, i.e. staleness, the slave will wait until the slowest worker catching up.

PARAMETER SYNCHRONIZATION

The replication of model parameters requires parameter synchronization among different workers. The overall performance is heavily affected by the frequency of model synchronization. Typically, more frequent synchronization can help improve the statistical efficiency. However, each synchronization operation, like pulling or pushing, incurs extra overheads. Angel optimizes the performance of these two operations through a 2D partitioning method, merging the local updates and designing different pulling methods for different ML algorithms.

Partition of parameters

Since ML algorithms typically treat parameters as linear algebra objects, Angel abstracts parameters as a matrix. The model parameters of most ML algorithms can be described in matrices, such as the weight vector in LR or the word-topic count matrix in LDA [34]. Angel partitions the parameter matrix into rectangle blocks through a 2D partitioning method.

Each rectangle block is the communication unit between workers and servers. Users can control the number of model partitions through setting each block's row and column size. With more model partitions, larger degree of parallelism can be achieved and the access load on each server is more balanced. However, a larger partition number will bring in more maintaining overheads when pushing updates to servers. Typically, the partition number should be larger than the number of servers.

Parameter pulling

During the training phase, workers would pull parameters from servers to calculate the gradients. For the models with billions of parameters, the pulling operations will result in massive communication cost. Moreover, as the communication is synchronous, the computation will be blocked owing to the network latency. We carefully study the structures of model parameters of different ML algorithms and group them into two categories. The first type of algorithms owns a model matrix with lots of rows, such as MF and LDA, while the other one employs a weight vector with billions of elements, like LR and SVM. We propose different pulling strategies for both types of parameters, respectively.

Pipelining pulling

To provide an efficient parameter pulling mechanism, we first analyze the model access method through two ML algorithms, LDA and MF.

Latent Dirichlet allocation. Griffiths and Steyvers [35] proposed using Gibbs sampling to perform the inference for LDA model. The posterior distribution $P(Z|W)$ can then be estimated using a collapsed Gibbs sampling algorithm, which, in each iteration, updates each topic assignment $z_{d,i} \in Z$ by sampling the full conditional posterior distribution:

$$p(z_{d,i} = k|w_{d,i} = v) \propto \frac{n_{v,k} + \beta}{\sum_{v'} n_{v',k} + V\beta} \times (n_{d,k} + \alpha), \tag{1}$$

where $k \in [1, K]$ is a topic, $v \in [1, V]$ is a word in the vocabulary, $w_{d,i}$ denotes the ith word in document d and $z_{d,i}$ the topic assignment to $w_{d,i}$. Matrix $n_{d,k}$ and $n_{v,k}$ are two-parameter models which are updated by the sample operations. When performing distributed Gibbs sampling, both the training data and the Matrix $n_{d,k}$ are partitioned to workers by document. Matrix $n_{d,k}$ is stored among different workers while matrix $n_{v,k}$ is shared. To perform each sample operation for token $w_{d,i}$, we just need to pull one row of $n_{v,k}$ where $v = w_{d,i}$. In other words, the parameters are pulled row by row in LDA.

Matrix factorization. Similar results can also be concluded for the MF algorithms. To solve the MF optimization problem, we sweep over all known ratings and update the parameters through the following equation for each rating:

$$\begin{aligned} x_u &\leftarrow x_u + \eta\left(\left(r_{ui} - x_u^T y_i\right) y_i - \lambda x_u\right), \\ y_i &\leftarrow y_i + \eta\left(\left(r_{ui} - x_u^T y_i\right) x_u - \lambda y_i\right), \end{aligned} \tag{2}$$

where x_u and y_i are the feature vectors for user u and item i, r_{ui} is the rating value of item i given by user u. η is the learning rate while λ is the regularization parameter.

Note that the user feature matrix is partitioned by user and the item feature matrix is stored at the servers. Therefore, for each observed rating (u, i, r), the optimization algorithm needs to pull y_i from the server.

From the discussion above, we can see that the computation unit, such as sampling a token or passing a rating, requires a portion of disjoint parameters as input. Therefore, we overlap the parameter pulling operation and the computation to reduce the performance degradation due to network latency. Naive parameter fetching method, Algorithm 1, will perform poorly due to the latency of remote procedure call. Instead, Angel allows each worker to request parameters for many computation units and perform the training function once a parameter is fetched, which is presented in Algorithm 2.

Angel provides an interface to pull multiple rows from servers. A list of row indices to be pulled is passed and a blocked result queue is returned immediately. The worker will randomly split the row indices into multiple disjointed sets where each one owns a mini batch of indices. The worker sends the pulling requests with one batch at a time and inserts the fetched rows into the result queue. The user program takes parameter rows from the result queue and conduct the computation. Since multiple tasks will be launched in each worker and different tasks would request the same parameter row at one iteration. The worker will eliminate duplicated requests from different tasks to reduce the network overheads.

Algorithm 1: Naive Parameter Pulling

```
initialization;
while not done do
    Get next token $w_{(d,i)} = v$ from iterator;
    Acquire row $v$ from server;
    Sampling token $w_{(d,i)}$;
    Update $n_{(d,k)}$ and $n_{(v,k)}$;
```

Algorithm 2: Pipeline Parameter Pulling

```
initialization;
Send requests for rows to servers;
while not done do
    if Pipeline has ready parameters $p$ then
        Sampling tokens with $p$;
        Update $n_{(d,k)}$ and $n_{(v,k)}$;
    else
        Wait on the Pipeline
```

Pruning pulling

For ML algorithms whose model consists of one vector with billions of elements, the parameter pulling

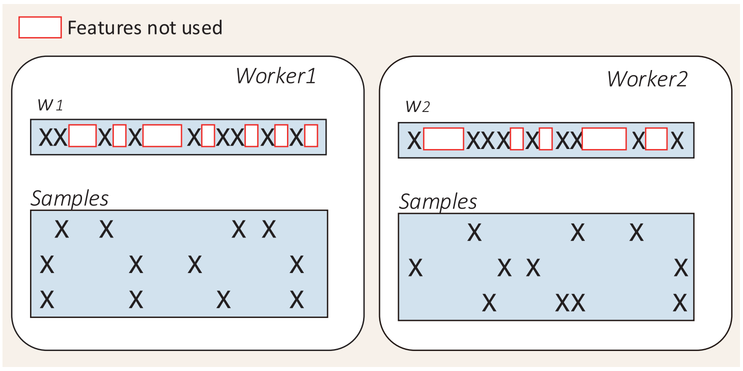


Figure 5. Sparse replicas at each worker.

message consists of a list of (key,value) pairs for each partition block, where the key is the index of non-zero elements. Since data are sparse and each training sample contains many zero elements. The corresponding parameters computed with these zero elements will not be reflected in the results. Hence, each worker can request a portion of parameters according to its data partition instead of the complete model replica.

As what is presented in Fig. 5, each worker only needs to pull and push the dimensions which are non-zero in its training data matrix. In consideration of the large size of model and scarce network bandwidth, messages compression is desirable.

Meanwhile, training data often remains unchanged between iterations. A worker might request parameters with the same set of keys multiple times at different iterations. Hence, it is natural for the server to cache the key lists. Later, the worker only sends a tag that can distinguish it from others to server rather than the whole key lists.

Pushing updates

In Angel, we asynchronously push the local updates to servers. Each worker maintains an update buffer for all the tasks running on it and a separate thread merges the generated updates. The push operation is conducted automatically once the buffer size exceeds a pre-defined threshold.

Frequent pushing operations are deserved since we want to propagate the updates generated by each worker to others as soon as possible. Since each pushing operation invokes network overheads, local updates are merged to reduce the data in transition. A matrix is used to merge the updates from different workers. Given the sparseness of the updates, the matrix is implemented by row-based linked lists instead of 2D arrays to reduce the memory consumption. For each row in the list, different data are employed according to the ML algorithms. In the LDA algorithm, each sampling operation only modifies two elements of row $n_{(v,k)}$, while the MF algorithm will modify the whole y_i feature vector for each rating. Hence, HashMap is used to merge the updates for each word in LDA while a dense array is adopted for each item in MF. When the size of this buffer exceed a pre-defined threshold, Angel will split the buffer into multiple partitions and compress them with different storage formats according to their degrees of sparsity. Then, these updates are pushed to the servers.

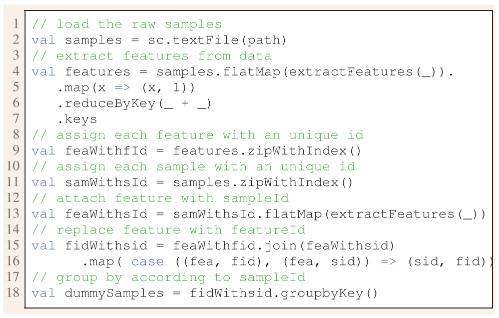
SYSTEM IMPLEMENTATION

Angel is now deployed in Tencent Inc. and applied to solve problems including online advertising and user analysis. The complex running environment and the various demands from applications rise challenges for the deployment of Angel. In the following parts of this section, we present how angel deals with these practical problems.

Data partitioning

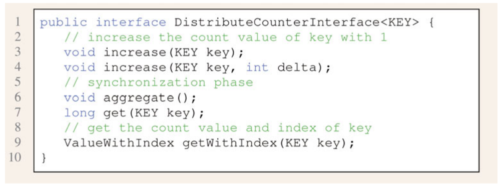
To avoid straggler workers, we should divide the data into balanced partitions before training. Generally, it requires an extra partitioning phase over the whole training data in order to balance the computation load for each worker. However, this will result in massive communication cost, especially when facing large datasets.

Instead of incurring a shuffle phase to generate a balanced partitioning results, Angel utilizes the location and meta information of data to achieve the goal. We observe that the computation complexity is proportional to the size of input data for a majority of ML algorithms. For example, the sampling times equal the number of tokens in LDA, while the number of non-zero items in one example in LR means the number of math calculation for the gradient computation. Therefore, we can balance the computation work of each worker through balancing the size of data read by each worker. Usually, in a production cluster environment, training data are stored as multiple files on HDFS, and each file is splitted into multiple blocks with each block replicated on two or three storage nodes. So, at the beginning of each Angel application, the master first collects the information of all data blocks from the NameNode of HDFS, including the length of each block and the node locations. Then, it places each data block to the worker that minimizes the



```
// load the raw samples
val samples = sc.textFile(path)
// extract features from data
val features = samples.flatMap(extractFeatures(_)).
   .map(x => (x, 1))
   .reduceByKey(_ + _)
   .keys
// assign each feature with an unique id
val feaWithfId = features.zipWithIndex()
// assign each sample with an unique id
val samWithsId = samples.zipWithIndex()
// attach feature with sampleId
val feaWithsId = samWithsId.flatMap(extractFeatures(_))
// replace feature with featureId
val fidWithsid = feaWithfid.join(feaWithsid)
      .map( case ((fea, fid), (fea, sid)) => (sid, fid))
// group by according to sampleId
val dummySamples = fidWithsid.groupbyKey()
```

Figure 6. Dummy processing using Spark.



```
public interface DistributeCounterInterface<KEY> {
   // increase the count value of key with 1
   void increase(KEY key);
   void increase(KEY key, int delta);
   // synchronization phase
   void aggregate();
   long get(KEY key);
   // get the count value and index of key
   ValueWithIndex getWithIndex(KEY key);
}
```

Figure 7. Distributed counter interface.

skew of computation load and maximizes the data locality.

Specifically, the master first sorts the blocks by the length in a descending order and assign each block to workers one by one. At each step, it finds out the worker with the most blocks that belongs to the same node with current block. If the data size of this worker is lower than the upper bound, the master assigns this block to this worker. Otherwise, it assigns this block to another one with the least computation load.

Data pre-processing

In practice, the data collected from online applications cannot be directly used for ML algorithm. Complex ETL operations must be performed to clean the raw data and convert it into ready-to-run vectors. The most common used operation is to construct dummy variables [36]. The construction phase of dummy variables is non-trivial and the pseudo code is listed in Fig. 6 with Spark RDD interface.

The *extractFeatures* function is defined by users to construct various features from original logs. Extracted features include simple features which are transformed from a single field in the original logs, as well as cross features generated by multiple fields. Then, we enumerate all distinct features according to the samples and assign a unique index to each feature. There are two shortcomings of the above Spark program. One is that there is a power-law distribution of the feature appearance times. This distribution results in unbalanced load in the join operation in Fig. 6. Besides, loading the training data between Spark and Angel is time consuming because we must materialize the data into HDFS and read it again. This problem becomes much more serious with the increasing size of the dataset.

To address this problem, we integrate a built-in module to complete the pre-process phase and abstract a distributed counter interface which is showed in Fig. 7 in Angel. User can register multiple counters in Angel and there are three types of state for each counter, including *WRITE*, *READ* and *CLEAN*. *WRITE* is the initial state of each counter. User can write to a counter with the *increase* interface once it is *WRITE* state. A counter changes to *READ* after user calls *aggregate* and allows workers to get count value and index through *getWithIndex* interface. Finally, the counter will be cleaned to release the memory. We implement the counter based on the servers. Each KEY is hashed and stored at the servers. Workers will merge the values locally before pushing to the server. The aggregate operation invokes a barrier, and the master will assign the index for each KEY.

In the implementation of Spark, the flow of RDD *feaWithsid* results in massive communication cost which is the same order of magnitude as the training dataset. However, the results of RDD *dummySamples* are still grouped by the sample id. Hence, the flow of training data is redundant and can be avoided with better implementation. Broadcasting the feature-index map to each partition can avoid the flow of training data, but it is also impractical when there exist tremendous features. Using the distributed counters, Angel avoids the flow of training data. Moreover, the unbalanced problem introduced by the join operation is also eliminated.

Data storage strategy

During the training phase of distributed ML, the training data stay immutable and is iteratively accessed by the optimization algorithm in a random order. After the initial partitioning phase, there is no demand for communication of training data among workers. Putting the training data inside each worker is suitable for the iterative access pattern of ML algorithms. But it also brings about huge memory consumption. Since Angel is run at a productive cluster, allocation requests with large memory are not allowed by Yarn. In consideration of the access pattern of ML algorithms, Angel stores the training data in workers over three strategies, including Memory, Disk and Memory_Disk.

Within each worker, user program accesses the training data through a *reader* with the following interface.

```
public interface Reader<KEY, VALUE> {
    boolean nextKeyValue();
    KEY getCurrentKey();
    VALUE getCurrentValue();
}
```

This reader provides the ability of scanning the training data with a random order multiple times without concerning the implementation details of data storage. There are three kinds of storage level behind the reader which can be configured by users and are listed as follows.

Memory All the training data samples will be put into an in-memory array and the reader traverses the array sequentially. With all the data samples residing at memory, the fastest data access speed would be gained.

Disk Under this storage level, all samples are serialized as binary bytes and written into multiple local files at worker. This storage level only keeps current key-value pair at memory, thus most of memory space can be left for other usage, such as parameters and model updates.

Memory_Disk The size of memory used to store training data is limited by a configured threshold, and it serves as a buffer for the whole training data. This hierarchical storage level can both obtain fast data access speed and low memory pressure for workers.

Angel enables launching multiple tasks inside one worker. Each task will get a reader to traverse its data partition. Angel will serialize the data into multiple files. These files will be located at different local disks to fully utilize the disk I/O bandwidth with concurrent.

Usually, the raw data read from HDFS, which is stored as libsvm format or protobuf bytes, cannot be directly used for ML algorithms which abstract data as matrices. Directly storing data samples into memory or disks in their raw format can introduce extra overheads by format converting whenever they are read. Users can personalize the *KEY* and *VALUE* type in the training data reader. Angel converts each sample into the ready-to-run objects only once, thus eliminating the duplicated data format converting operations.

Fault tolerance

Running a distributed job in a practical environment, failures are common and may happen due to various reasons, such as hardware errors or interfered by other tasks. Angel provides the ability to recover when servers and workers come across failures.

Servers store the values of model parameters which are frequently modified by workers. Hence, we choose to write the snapshot of each parameter partition periodically into persistent storage instead of writing update logs. Besides the parameter values, other systems [5,6] also maintain the clock information on servers. In these systems, inconsistent clock view would exist among different servers. Although better performance can be obtained for training, it is hard to recover from a consistent state when job encounters failures. It requires to take snapshot at each time when servers receive an update request to keep consistent on the snapshot view, which introduces big pressure on server. Another problem is introduced when a worker recovers from failures. It needs to ask for all servers to obtain the current clock value it should reside on due to the inconsistent view of clock values.

To avoid these problems, Angel manages a consistent view of clock information at the master side. The servers only act as storage service without maintaining any state information. At the initialization stage, a thread will be launched to dump the value of parameters into the HDFS periodically. User can configure the backup interval of the snapshot operation. When server is crashed, the master node can be informed through the broken heartbeat connection. A new server will be allocated and initialized with the snapshot on the HDFS. A small number of updates might be lost, but we think it is acceptable because of the error tolerance property of ML algorithm. When a worker is down, the master will pull up another worker with the clock state of the crashed one. The new worker will read the training data from HDFS, fetch the latest parameters from servers and start training from the current clock.

EVALUATION

In this section, we present the experimental evaluation of the proposed distributed ML system, Angel. We first demonstrate the benefits and costs of Angel by comparing Angel with Spark and Petuum using various ML algorithms. We also compare Angel with TensorFlow using the LR algorithm. Then, we evaluate the performance of hybrid parallelism, the effectiveness of hierarchical data management and the impact of synchronization strategies. We also study the behaviors of Angel when facing failures.

Experimental setup

The experiments are conducted on a cluster with 100 physical nodes, where each node has a 2.2 GHz

Table 1. Dataset statistics.

Model	Dataset	#row	#col	#nnz	Sparse size
LR	kdd2010	19 m	30 m	585 m	8.4 GB
	CTR1	35 m	1.5 m	1.1 b	11 G
	CTR2	256 m	4 m	9 b	85 G
	CTR3	410 m	4.7 m	14.2 b	130 G
	CTR4	528 m	101 m	22 b	167 G
MF	netflix [37]	18k	480k	100 m	1 GB
	wxjd	21.8 m	253k	2.5 b	15.7 G
LDA	pubmed	8 m	140 k	737 m	4 GB
	corpus1	159 m	3.7 m	19 b	90 G
	corpus2	507 m	5.6 m	60 b	270 G
KMeans	kdd2010	19 m	30 m	585 m	8.4 GB
	pubmed	8 m	140k	737 m	4 GB

CPU with 12 cores, 64 GB memory, 12 × 2 TB SATA hard disks and connected by 1 Gbps network. The version of Yarn used in resource management is 2.2.0. Unless otherwise stated, the maximum size of JVM heap is 10 GB. To further demonstrate the scalability of Angel when dealing with large datasets, we conduct evaluations of Angel on a shared inner-company cluster with around 5000 physical nodes.

ML algorithms. Currently, Angel supports efficient implementations for a wide range of ML algorithms to satisfy the requirement of various applications, especially aiming at large data and large model. For classification tasks, Angel provides three algorithms, including LR, SVM and LassoLR, while regression tasks can be solved with Linear Regression and Lasso. Besides that, users can utilize tree ensembles methods, like GBDT, to handle classification and regression tasks. Kmeans and LDA are supported in Angel to handle clustering problems. Moreover, MF is also integrated in Angel to deal with recommendation tasks.

Here, we adopt four ML algorithms to compare Angel with other systems in the following experiments, namely LR, LDA, MF and Kmeans, because they have different parameters access patterns. LR is a linear algorithm whose model has only one shared row, while Kmeans has one shared matrix with multiple rows. Both the models of LDA and MF are composed of multiple matrices. An LDA's model contains two shared matrices and one partitioned matrix, while MF uses one shared matrix and one partitioned matrix.

Dataset. Besides the public datasets, including *netflix*, *kdd 2010* and *pubmed*, we also use the logs of Tencent applications to conduct the experiments. *kdd2010* dataset is employed to evaluate the performance of different systems on LR. The datasets *CTR 1-4* record the click through rates of users, and they are used to evaluate the performance of scalability, data pre-processing and fault tolerance on Angel. The user behavior dataset *wxjd*, as well as *netflix*, is adopted in the MF experiments. Document corpus datasets, *pubmed*, *corpus1* and *corpus2*, are used to evaluate the performance of LDA, where *corpus1* and *corpus2* are set of web pages crawled by crawlers in Tencent. The statistics of the datasets are listed in Table 1. The sparse size of dataset is the size of text format where only non-zero elements are stored. The #nnz value counts the number of non-zero elements in the data matrix.

End-to-end comparison

We first demonstrate the efficiency of Angel by comparing it with Spark, Petuum and TensorFlow. Spark follows the paradigm of MapReduce, but is specially optimized for iterative tasks. Now Spark is one of the most popular general-purpose big data processing systems in industrial companies. Due to its feasible abstraction of distributed datasets and cache mechanism, Spark is widely adopted to processing ML applications. Here, we use the algorithm implementations from Mllib, the ML library built beyond Spark, to conduct the comparison. Petuum is an academic prototype system which provides Bösen and Strads to support data parallelism and model parallelism, respectively. Both Spark and Petuum provide the implementations of those four ML algorithms adopted in the experiments. TensorFlow is an open-sourced distributed system designed for deep learning. Due to its generalization of programming interface, TensorFlow can also support linear models, such as LR. Therefore, we conduct the comparison with TensorFlow using LR. We use the latest released version of

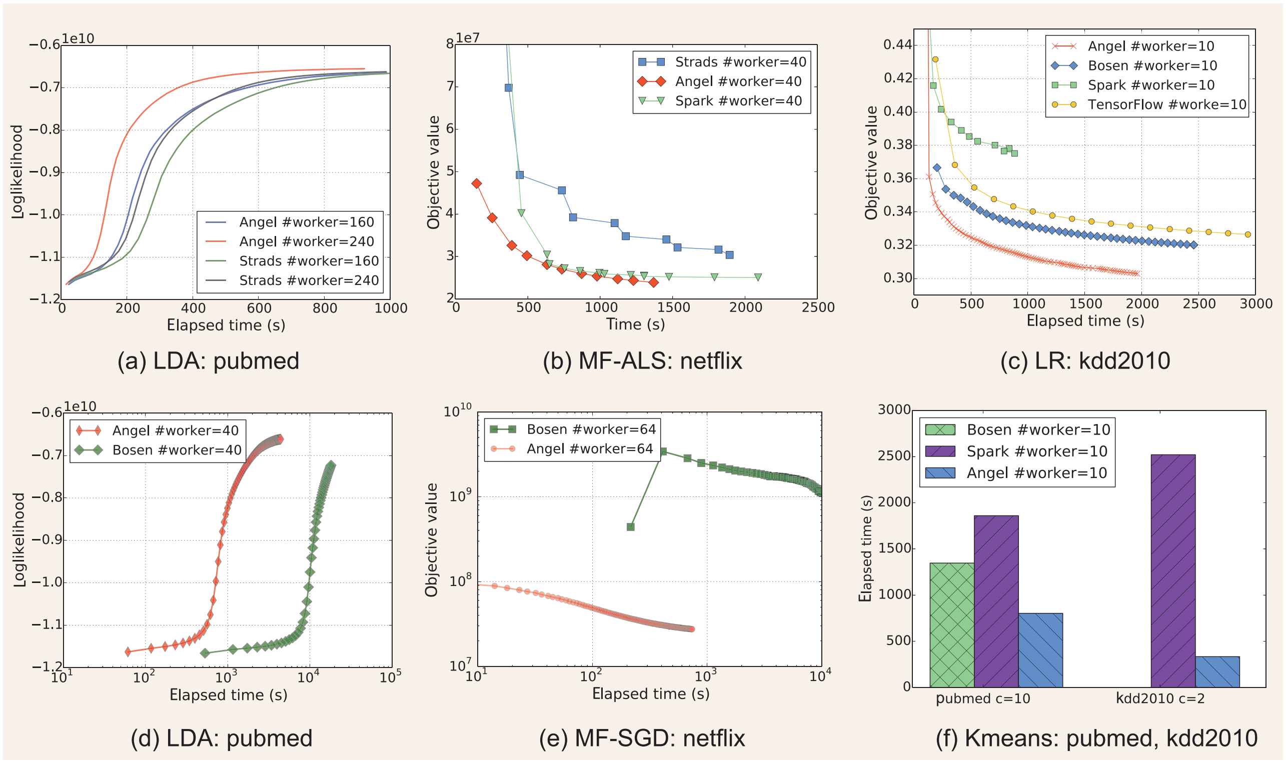


Figure 8. Performance comparison between Petuum, Spark, TensorFlow and Angel.

Petuum and TensorFlow in the experiments and the version of Spark is 2.0.2.

Effects on LDA

Comparison with strads. We first compare Angel with Strads using LDA workloads. The topic number is set to 1024 in the experiments.

Strads deploys model parallelism for sparse Gibbs sampling while Angel implements LDA with hybrid parallelism. We vary the number of machines for both Strads and Angel to evaluate their performance under different environment configurations. In Angel's implementation, each worker group is assigned with eight workers to conduct model parallel sampling. We monitor the loglikelihood value during the execution and record the time when the loglikelihood value reaches -7.8×10^9.

The results are illustrated in Fig. 8a. We can see that Angel exhibits better performance than Strads when the worker number is 160. It costs 430 s for Strads with 160 workers to reach the required value, while the time needed for Angel is only 338 s.

When more workers are involved in the execution, the running time of both Strads and Angel reduces, but Angel benefits more from the additional resources. Though Angel needs four more epoches when running with 240 workers, the running time of each epoch is reduced as well. The model parallel method enables Strads to converge with less epochs. But the achieved speedup is damaged by the global barriers. Finally, the speedup of Angel is 1.46 when the worker number increases from 160 to 240, while it is only 1.25 in Strads. As a consequence, Angel can obtain a 1.6× performance improvement against Strads. There are two reasons that enable Angel achieve faster speed than Strads. The first one is that Angel employs hybrid parallelism, while the other one is that in Angel the get operations are overlapped with the computation operations to reduce the network latency.

Comparison with Bösen. We then compare Angel with Bösen using the same workloads. Since the execution of Bösen always fails when using large number of machines, we use much less workers in these experiments. We use only 40 workers in these experiments and configure the staleness in Bösen to three clocks.

With the update scheduling in each group, Angel needs only 22 epochs to reach the required value, while Bösen needs 33 epochs. With the carefully optimized model synchronization, the performance of Angel is further optimized. From Fig. 8d, we can see that Angel can obtain a 10× performance gain over Bösen.

Comparison with Spark. We also conduct the experiment of LDA algorithms using Spark on the *pubmed* dataset. However, the performance of Spark is too slow. It will take more than 1 h to finish one iteration. Spark abstracts the computation of LDA as graph and uses GraphX to perform the computation. In the implementation of GraphX, each vertex will send the topic distribution to its neighbors to perform the inference, which will generate massive communication cost, ≈300 GB shuffle data, in each iteration. On the contrary, only the word-topic matrix needs to be shared and transferred through network in the implementation of Angel, which can significantly reduce the communication cost

Effects on MF

SGD and Alternating least squares (ALS) are two popular optimization algorithms for MF. SGD is a gradient-based method that iteratively updates the two feature matrices while ALS alternatively calculates the value for one of them. Due to the different update patterns, Strads and Spark adopt ALS as its optimization method for MF while Bösen adopts SGD. To eliminate the effects of different optimization methods, we implement both SGD and ALS on Angel to compare the performance of MF against Petuum and Spark. Specifically, when comparing Angel with Strads and Spark, we set ALS as the optimization method for Angel while SGD is employed when comparing with Bösen.

Comparison with Strads and Spark. In the implementation of ALS on Angel, we store both the user feature matrix and the item feature matrix on servers. Each iteration is divided into two phases, where workers fetch the item matrix and compute the value of user matrix at the first phase, while at the second phase workers fetch the user matrix and calculate the item matrix. At each phase, different worker calculates different parts of the item matrix or user matrix in order to parallelize the computation.

The experiment's result is showed in Fig. 8b. All three systems adopt 40 workers to conduct the evaluation. Each worker group contains only one worker in Angel. The rank of the feature matrices is set to 200 and the dataset used here is *netflix*. To reach the objective value of 3×10^7, the consumed cost for Strads, Spark and Angel is 1893, 633 and 494 s, respectively. Therefore, Angel is about four times faster than Strads and 1.3× faster than Spark.

Comparison with Bösen. Here, we give the result of comparing Angel and Bösen. When using SGD as the optimization method, we store the item feature matrix on the servers to share it among workers, while the user feature matrix is partitioned and stored on different workers. At each iteration, each worker fetches the item matrix and updates both the user matrix and item matrix for each rating.

The evaluation result is presented in Fig. 8e. The number of workers is 64, and there are eight worker groups for Angel with each group containing eight workers. In the implementation of Bösen, it adds the updates from all workers to the global parameters instead of adding the average value of updates. Therefore, the objective value might increase after the first synchronization operation. We have tried out best to adjust the learning rate for Bösen, it still takes more than 10 000 s to achieve the specified objective value. Hence, Angel is at least 10× faster than Bösen.

Effects on LR

Here, we present the comparison on LR for Angel, Spark, Petuum and TensorFlow. The experimental results are given in Fig. 8c. The number of workers is all set to 10 for all systems, and each worker group in Angel has only one worker. The dataset employed here is *kdd2010*. SGD is adopted by all systems to solve LR. We use the loss value on the training dataset to evaluate their convergence rate. The learning rate are well tuned for each system. For example, Spark requires larger learning rate since it will average the gradient value for a batch of samples, while TensorFlow requires a smaller value of learning rate since it directly adds the summation of gradients generated by samples.

Comparison with Spark. For Spark, we use the latest code of MLlib, version 2.0.2, to run this experiment. There are three different implementations on MLlib. We try all these three methods and find that only one of them can successfully complete the running with *kdd2010* dataset. The other two methods will fail on the first synchronization phase due to *OOM* error or *Requested array size exceeds VM limit* error.

We can see that Angel is consistently faster than Spark. The reason is that Spark needs to broadcast the model and collect the gradients from workers for each update operation, meaning that each update operation requires one to two network operations, which much communication costs; while Angel adopts PSGD algorithm, which enables each worker make a copy of the model parameters and synchronize them at the end of each iteration. Therefore, Spark needs a long time to reach a low training loss value while Angel can quickly reach it.

Comparison with Bösen. From Fig. 8c, we can see that Angel is about five times faster than Bösen. Angel needs about 10 s to finish one iteration while Bösen requires ≈50s. There are three reasons that result in the performance improvement of Angel. The first one is that Bösen lacks the ability to partition data. It requires users to manually copy the data to each physical node. Moreover, each worker of Bösen will read all the data into memory and partition it at the worker side, which incurs more overheads since each worker reads more data. Another reason is that the Bösen needs to synchronize the parameters more times inside one epoch. Otherwise, the training loss will diverse and the algorithm will not converge at last. Extra synchronization operations incur more network overheads. The third reason why Angel is faster is that Angel can utilize the sparsity of data to avoid fetching useless parameters while Bösen cannot.

Comparison with TensorFlow. Since the default implementation for LR in TensorFlow is designed to dense data, it requires users to convert each sparse sample to dense representation. However, it is a bad way to process dataset with high dimensions, such as *kdd2010*. Therefore, we follow the code from https://github.com/chenghuige/tensorflow-example and realize a distributed version for LR algorithm. We manually partition the input data into multiple parts since TensorFlow does not support data partitioning.

From Fig. 8c, we can see that TensorFlow is much slower than Angel. It takes more than 150 s for TensorFlow to complete one epoch which Angel only requires ≈10 s. The reason that training data must be splitted into mini-batches in TensorFlow, where each batch is one computation unit that incurs one round of gradient computation and parameter update. The problem is that larger batch size will cause errors since TensorFlow does not allow the size of one tensor to exceed 2 GB. Thus, there are about 17 batches inside one epoch, indicating that there are 17 network communication operations. We carefully tune the batch size for TensorFlow to guarantee both the success of running and the best performance under the limitation of tensor size.

Effects on KMeans

We continue presenting the performance comparison using KMeans algorithm. The number of workers is all set to 10 for Bösen, Spark and Angel and there are only one worker in each group for Angel. We use two different datasets, including *pubmed* and *kdd2010*, and different number of centers to conduct this comparison. To do the comparison, we evaluate the running time of these three system to reach the same square errors. To guarantee that each system can achieve its best performance, we carefully tune the running parameters for all of them. For example, the performance of Bösen is sensitive to the batch size. Larger batch size results in more computation overhead while small batch size incurs more communication overheads.

Figure 8f presents the performance of these three systems when using these two datasets. For *pubmed* dataset, we set the number of centers to 10 and calculate the running time until the square error is reduced to 115 464. We can see from the results that Angel is faster than Bösen and about two times faster than Spark. When using a dataset with high-dimensional data, for example *kdd2010*, Angel is ≈8× faster compared with Spark because Spark lacks efficient mechanism to gather the data-generated workers. When running with *kdd2010* dataset, Bösen cannot finish the first iteration since it will be blocked forever at the fetching operation of parameters. In the implementation of Bösen, it can only partition parameter matrix by row. Therefore, it lacks the ability to deal with matrix whose row size is larger than 1 million. It requires users to manually partition one row to multiple parts. However, the open-source implementation provided by Bösen for KMeans does not handle this problem.

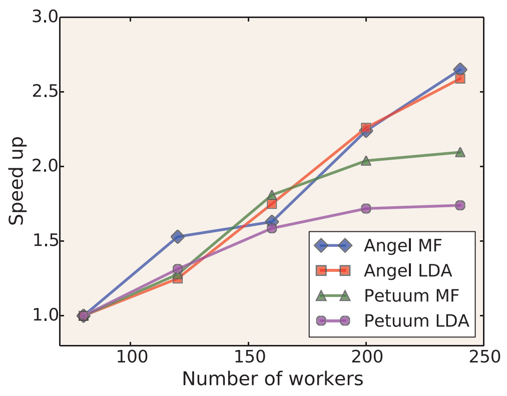


Figure 9. Scalability.

Scalability

Figure 9 presents the scalability of Angel and Strads. We vary the number of execution tasks and calculate their speedups in LDA and MF by the reduction in the running time. The datasets used in LDA and MF programs are *pubmed* and *wxjd*, respectively.

Angel behaves with better scalability than Strads. With the task number increasing from 80 to 240, the speedups achieved by Angel in MF and LDA are 2.65 and 2.59, respectively. Meanwhile, the numbers are 2.09 and 1.74 in Strads. The reason is that Angel adopts hybrid parallelism that allows concurrent updates to the model while each step of Strads must be synchronized with the global barrier. With the growth in the degree of parallelism, Angel hence can obtain better speedup.

Parallelization methods

We continue presenting the effects of hybrid parallelism in this subsection. We vary the number of worker group number and the number of workers in one group to realize data parallelism and model parallelism on Angel. Specifically, when the number of workers in one group is set to 1, the implementation of Angel is exactly data parallelism while if all workers belong to the same group, they are coordinated to train through model parallelism. To demonstrate the effectiveness of hybrid parallel method, we utilize both LR and LDA to obtain the experiment results.

Figure 10a gives the convergence rate over epoches on LDA, while Fig. 10b presents the convergence rate over running time. We employ *pubmed* dataset for this comparison. The total number of workers are set to 240 and the implementation of hybrid parallelism employs eight workers at one group. We can see from Fig. 10a that model parallel method requires the least number of epoches to obtain the same loglikelihood value, while data parallel method needs much more epoches than model parallelism and hybrid parallelism. The reason is that data parallel method incurs too much conflicting updates, while model parallel method carefully schedules the parallel computing of model to obtain the best convergence rate over epoches. However, since model parallel method requires to schedule the updates among all workers to avoid conflicts and issue global barriers to coordinate workers, it needs more time to finish one epoch. Figure 10b demonstrates that hybrid parallelism can obtain the best performance compared with the other two counterparts. It needs the least time to reach the convergence point.

Similar results can be derived from Fig. 10c and d, which use LR algorithm to evaluate three parallel methods on the *kdd2010* dataset. The overall number of workers is 10 for all of them, and there are two workers in one group for hybrid method. Figure 10c shows that model parallelism can reach a lower training loss value with the same number of epoches compared with hybrid parallelism and data parallelism. However, similar to the results of LDA, model parallelism requires more time for each epoch, which can be seen from Fig. 10d. Compared with the other two parallel methods, hybrid parallelism can achieve both good convergence rate per epoch and reduce the time cost for one iteration, reaching the given objective value with the minimum time.

Data management

In this subsection, we present the efficiency of the data management module in Angel. We first examine the impact of storage level to the performance. Then, we compare the integrated data pre-processing module with Spark.

Impact of storage level

There are three different memory storage levels for training data that reside at the workers during the training phase. Figure 11a presents the effects of these three levels to the convergence time of LR algorithm on *CTR2* dataset with 20 workers under different memory budgets. When the memory budget is limited at 4 GB, the worker task using only memory storage level fails because of the OutOfMemory exception while tasks can still complete their execution once setting the level to disk or memory_disk.

For each memory budget value, the tasks using the disk level need more running time than their counterparts due to the more I/O cost. The memory_disk level can obtain comparable performance compared to the memory level while putting no enforcement on the memory budget. By maintaining a memory buffer on each worker node when using the memory_disk level, lower accessing latency and lower memory consumption can be obtained at the same time.

Impact of pre-processing

Here, we continue evaluating the running time of the dummy processing phase. Because Petuum does not have the ability to pre-process data, we only compare Angel and Spark in the experiments. In the implementation of Spark, we broadcast features with high frequency to workers instead of join operation, which can eliminate skewed tasks. As showed in Fig. 11b, Angel outperforms Spark over an order of magnitude on each dataset. With more training data to process, Spark exhibits inferior performance than Angel since it transfers more data through shuffle operations. Thus, more disk I/O and network overheads are encountered. The inefficiency in the Spark's pre-processing significantly degrades the overall performance of ML applications. Angel can however complete the pre-processing phase with almost ignorable cost compared with the succeeding training phase.

Synchronization

We continue presenting the effects of different parameter pulling methods.

Impact of pipelining pulling

The pulling of parameters and the computing of updates are performed sequentially in the naive implementation. Network requests are issued whenever a new row is needed in the computation. Since each network operation has extra overheads and there are more than ten thousands of parameter rows in

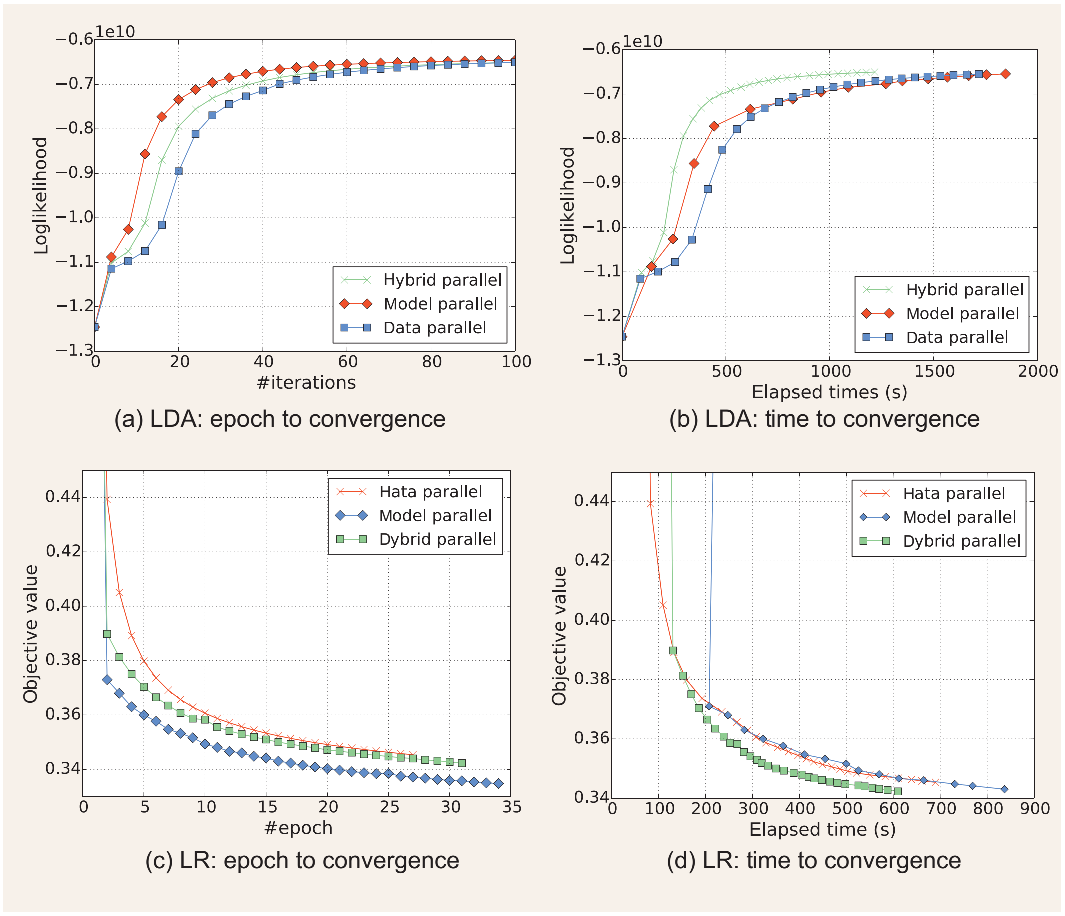


Figure 10. Performance comparison between data parallel, model parallel and hybrid parallel.

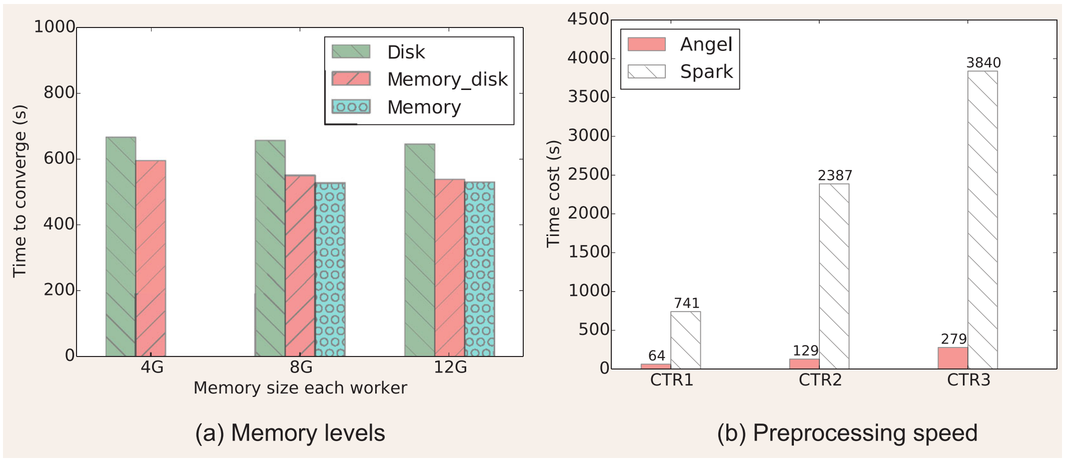


Figure 11. Effective of data management.

the model, the naive implementation incurs heavy network overheads. The pipelined execution method invokes network requests through a batch method and overlaps the parameter pulling and the computing phases. By reducing the unnecessary waiting time, the performance degraded by network operations reduces as well.

We conduct experiments to evaluate the effects of pipelined pulling with the MF programs running over the Netfix dataset. We measure the running time per iteration with different pulling methods. Since a larger value of rank will lead to more data transferred through network, we also vary the rank values in the MF programs.

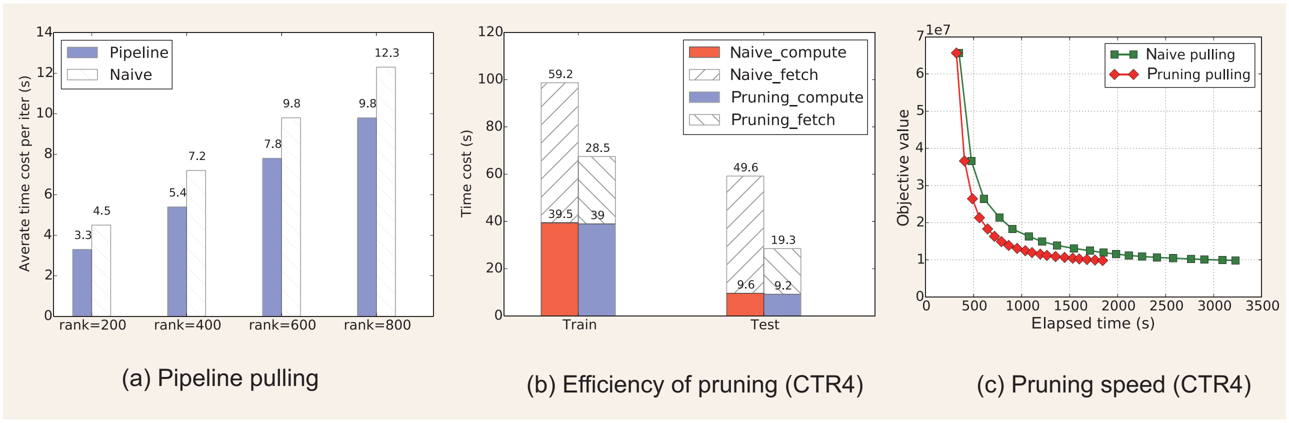


Figure 12. Effectiveness of parameter synchronization.

The experimental results presented in Fig. 12a are identical with our expectation. Because the pipelined method can facilitate the utilization of the bandwidth, it helps to improve the performance of the MF programs. With the increasing of model size, the programs using pipelined pulling obtain more performance gains than the naive ones.

Impact of pruning pulling

By pulling only those parameters needed in the computing, the pruning pulling method can reduce the number of pulled parameters. We run the LR algorithm against *CTR4* to evaluate the effect of pruning pulling. In the experiments, the number of parameters exceeds 100 million, and the programs are executed over 100 workers.

Figure 12b discriminates the time used to pull parameters and compute updates during the training phase and test phase. Comparing to the naive method which pulls all parameters to workers, the pruning method can avoid the pulling of those parameters that are not needed in the calculation of gradients. Hence, the pruning method can efficiently reduce the time needed in pulling operations. For each iteration, the pruning method only costs 28.5 s for the pulling operation while it takes 59.2 s for the naive method at the training phase. The reduction in the time of pulling operations can help accelerate the convergence rate.

Figure 12c shows the objective values over time with different pulling methods. By avoiding pulling redundant parameters, the pruning method takes only 1844 s to complete 20 epochs while the naive one requires 3227 s.

Impact of synchronization frequency

Here, we show the impact of different synchronization frequencies to the convergence speed of LDA algorithm. Three different synchronization methods are adopted in the experiments. The *auto* method which is described in Section 'Pushing updates' employs the automatic updates pushing method provided by Angel. The *10spe* method propagates local updates to servers 10 times every epoch, while the *1spe* method only pushes the updates at the end of each epoch. From Fig. 13a we can see that the *auto* method can obtain comparable statistical efficiency compared with *10spe*. They both outperform the convergent speed of *1spe* in terms of epoch numbers since they propagate their updates to other workers more frequently. But when coming to the running time, the *auto* method performs the best. This is because each update pushing operation invokes extra overheads, such as updates merging, splitting and the network cost. With more frequent synchronization, more overheads are introduced and the benefits brought by frequent synchronization become marginal. Because the automatic sync method utilizes a best-effort strategy to perform pushing operations, it can well balance the frequency and the cost of model synchronization, avoiding the performance degradation due to unnecessary synchronization operations.

Fault tolerance

In this subsection, we evaluate the recovery performance of Angel. We conducted LR algorithm on both *CTR1* and *CTR2* datasets. Figure 14a shows the recovery latency when an Angel application comes across failures. We randomly kill a server node at iteration 1 and iteration 11. Compared to other iterations without failures, it costs about more 6 s to complete the two iterations where failures show up. That is, the master node takes only 6 s to detect the crash of server node, reallocate a container for a new node and recover the state of failed server from its snapshot.

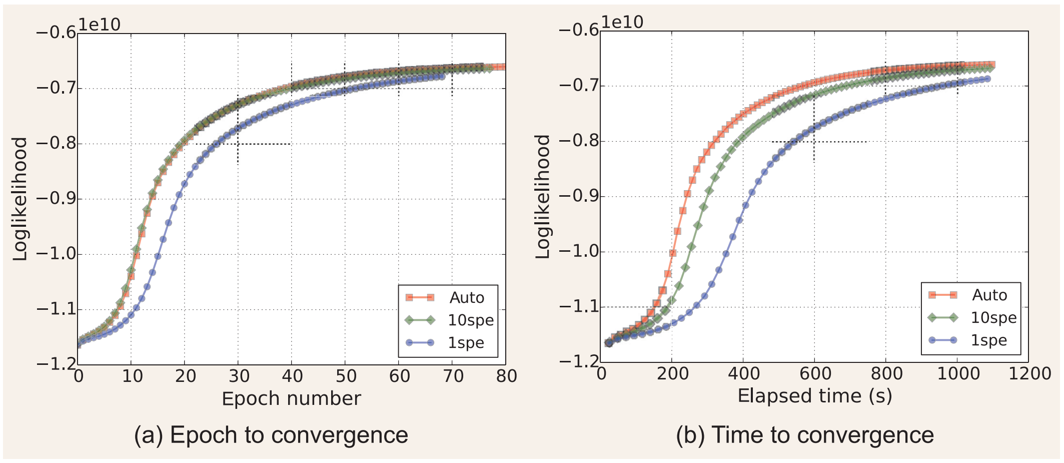


Figure 13. Impact of synchronization frequency.

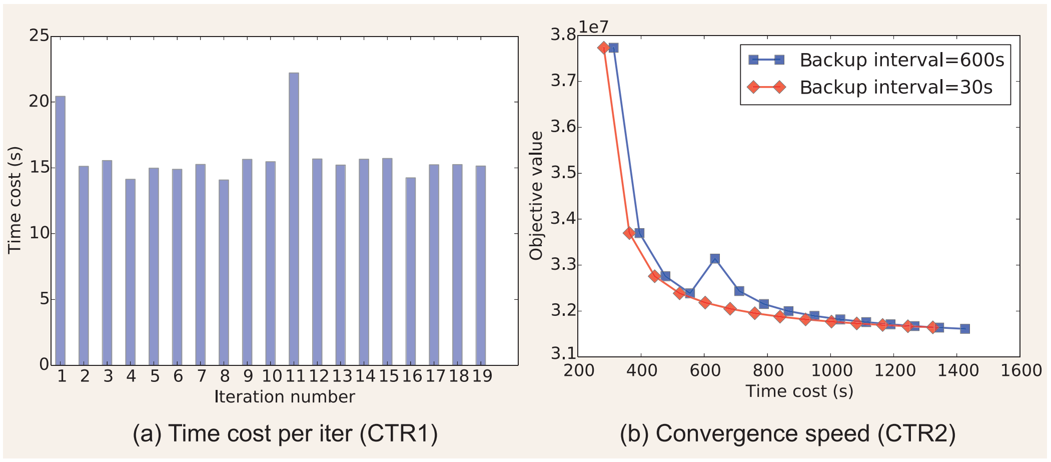


Figure 14. Impact of failover.

Angel enables users to set the backup interval for servers, and we compare the effect of different backup intervals on the convergence speed in the execution with failures. The failure is set to happen at 500 s after the application starts. If the interval is set to 600 s, no snapshot had been written to HDFS when the failure happens. Then, the convergence progress is affected by the server failure. However, since only a part of parameters are lost, we do not need to start the entire computation from scratch. Only those servers affected by the failure will be restarted. When the backup interval is set to 30 s, we can see that the execution is merely affected by the failure. Note that each iteration here takes ≈80 s to complete the training phase and test phase, the number of updates not reflected in the latest snapshot is very small. The new launched server can restore the state with few missing updates. Therefore, the execution can recover in a short time and the convergence rate is not damaged.

Scalability results

Finally, we use large datasets and more workers to demonstrate the scalability of Angel. We use LDA algorithm to conduct the evaluation since it is more complicated and its model size is much larger than others. Two datasets, including *corpus1* and *corpus2*, are employed to perform the evaluation. The experiments are running in a shared inner-company cluster with 5000 physical machines that are equipped with the same hardware configurations with the previous ones. Researchers and engineers in Tencent submit around 1.2 million applications to this cluster every data, with at most 6000 jobs running at the same time.

The experiments' results are presented in Fig. 15. The number of topics is set to 8000 for these two datasets. We vary the number of overall workers but fix the number of workers in each group at four. For *corpus1*, Angel can achieve the convergence point within 7500 s using 400 workers, while it can be

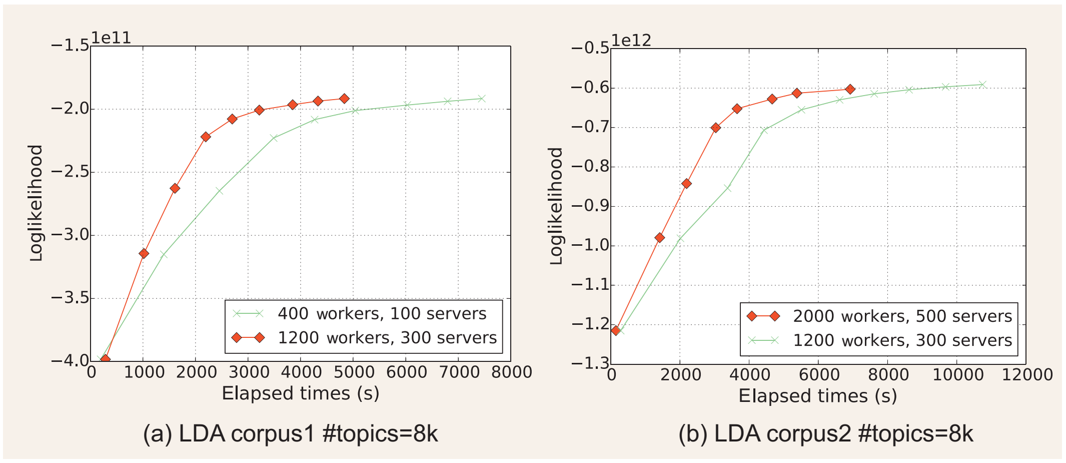


Figure 15. Scale results with more workers and large datasets: (a) LDA corpus1 #topics = 8k and (b) LDA corpus2 #topics = 8k.

decreased to 5000 s when the number of workers is increased to 1200. For *corpus2*, Angel can finish the training within about 3 h using 1200 workers, while it can achieve convergence in ≈7000 s when using 2000 workers. There are ≈30 billions and 40 billions of parameters stored on the servers when training dataset *corpus1* and *corpus2*, respectively. These two experiments prove that Angel can run in a real production environment with thousands of workers to handle ML applications that require billions of parameters.

We also try to run this experiment with Petuum and Spark. However, Spark will always fail since it lacks efficient mechanism to process large models, while Petuum lacks the ability to automatically partition the training dataset. It requires a Network File System to share the input dataset for all workers. Otherwise, user should manually copy the input dataset to every worker, which is very time consuming for large dataset and impossible for a production cluster managed by Yarn. Another problem for Strads is that the worker will fail for large data since it will read all the data into memory. In summary, Petuum cannot be deployed in production environment for industry-scale applications.

CONCLUSION

In this paper, we proposed a new general-purpose distributed ML system, named Angel, which aimed at solving large-scale ML problems faced by big data analytic applications. Angel employed hybrid parallelism to accelerate the performance of ML algorithms. The pulling of parameters and the pushing of updates were fully optimized in Angel to reduce the network overheads. Angel was deployed in a production cluster and provisioned efficient mechanisms to achieve fault-tolerated execution at scale. The comprehensive experiment results demonstrated the superiority of Angel compared to Spark and Petuum.

FUNDING

This work was supported by the National Natural Science Foundation of China (61572039), the National Basic Research Program of China (2014CB340405), Shenzhen Government Research Project (JCYJ20151014093505032) and Tecent Research Grant (PKU).

Conflict of interest statement. None declared.

REFERENCES

1. Huang Y, Cui B and Zhang W *et al.* Tencentrec: Real-time stream recommendation in practice. In: Sellis TK, Davidson SB and Ives ZG (eds). *Proceedings of SIGMOD Conference 2015.* Melbourne, Victoria, Australia: ACM 2015, 227–38.
2. Huang Y, Cui B and Jiang J *et al.* Real-time video recommendation exploration. In: Özcan F, Koutrika G and Madden S (eds). *Proceedings of, SIGMOD Conference 2016.* San Francisco, CA, USA: ACM 2016, 35–46.
3. Zaharia M, Chowdhury M and Franklin MJ *et al.* Spark: cluster computing with working sets. In: Nahum EM and Xu D (eds). *Proceedings of HotCloud 2010.* Boston, MA, USA: USENIX Association 2010.
4. Zaharia M, Chowdhury M and Das T *et al.* Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Gribble SD and Katabi D (eds). *Proceedings NSDI Conference 2012.* San Jose, CA, USA: USENIX Association 2012, 15–28.
5. Xing EP, Ho Q and Dai W *et al.* Petuum: a new platform for distributed machine learning on big data. *IEEE Trans Big Data* 2015; **1**: 49–67.
6. Li M, Andersen DG and Park JW *et al.* Scaling distributed machine learning with the parameter server. In: Flinn J and Levy H (eds). *Proceedings of OSDI Conference 2014.* Broomfield, CO, USA: USENIX Association 2014, 583–98.

7. Ho Q, Cipar J and Cui H *et al.* More effective distributed ml via a stale synchronous parallel parameter server. In: Burges CJC, Bottou L and Ghahramani Z *et al.* (eds). *Proceedings of NIPS Conference 2013*. Lake Tahoe, Nevada, United States, 2013, 1223–31.
8. Kim JK, Ho Q and Lee S *et al.* Strads: a distributed framework for scheduled model parallel machine learning. In: Cadar C, Pietzuch PR and Keeton K *et al.* (eds). *Proceedings of EuroSys Conference 2016*. London, United Kingdom: ACM 2016, 5:1–5:16.
9. Lee S, Kim JK and Zheng X *et al.* On model parallelization and scheduling strategies for distributed machine learning. In: Ghahramani Z, Welling M and Cortes C *et al.* (eds). *Proceedings of NIPS Conference 2014*. Montreal, Quebec, Canada, 2014, 2834–42.
10. Wang Y, Zhao X and Sun Z *et al.* Peacock: learning long-tail topic features for industrial applications. *ACM Trans Intell Syst Tech* 2015; **6**: 47.
11. Zhang C and Ré C. Dimmwitted: a study of main-memory statistical analytics. *Proc VLDB Conf* 2014; **7**: 1283–94.
12. Yu Y, Isard M and Fetterly D *et al.* Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In: Draves R and van Renesse R (eds). *Proceedings of OSDI Conference 2008*. San Diego, California, USA: USENIX Association 2008, 1–14.
13. Fan W and Hu C Big graph analyses: from queries to dependencies and association rules. *Data Sci Eng* 2017; **2**: 1–20.
14. Shao Y, Cui B and Ma L. Page: a partition aware engine for parallel graph computation. *IEEE Trans Data Knowl Eng* 2015; **27**: 518–30.
15. Shi X, Cui B and Shao Y *et al.* Tornado: a system for real-time iterative analysis over evolving data. In: Özcan F, Koutrika G and Madden S (eds). *Proceedings of SIGMOD Conference 2016*. San Francisco, CA, USA: ACM 2016, 417–30.
16. Dean J, Corrado G and Monga R *et al.* Large scale distributed deep networks. In: Bartlett PL, Pereira FCN and Burges CJC *et al.* (eds). *Proceedings of NIPS Conference 2012*. Lake Tahoe, Nevada, United States, 2012, 1232–40.
17. Jiang J, Cui B and Zhang C *et al. Heterogeneity-aware distributed parameter servers*. In: *Proceedings of SIGMOD Conference*. 2017.
18. Abadi M, Barham P and Chen J *et al.* TensorFlow: A system for large-scale machine learning. In: Keeton K, Roscoe T (eds). *Proceedings of OSDI Conference 2016*. Savannah, GA, USA: USENIX Association 2016, 265–83.
19. Ooi BC, Tan KL and Wang S *et al.* Singa: a distributed deep learning platform. In: Zhou X, Smeaton AF and Tian Q *et al.* (eds). *Proceedings of Multimedia Conference*. Brisbane, Australia: ACM 2015, 685–8.
20. Wu TT and Lange K. Coordinate descent algorithms for lasso penalized regression. *Ann Appl Stat* 2008; **2**: 224–44.
21. Zinkevich M, Weimer M and Li L *et al.* Parallelized stochastic gradient descent. In: Lafferty JD, Williams CKI and Shawe-Taylor J *et al.* (eds). *Proceedings of NIPS Conference 2010*. Vancouver, British Columbia, Canada: Curran Associates, Inc., 2595–603.
22. Bradley JK, Kyrola A and Bickson D *et al.* Parallel coordinate descent for l1-regularized loss minimization. In: Getoor L and Scheffer T (eds). *Proceedings of ICML Conference 2011*. Bellevue, Washington, USA: Omnipress 2011, 321–8.
23. Newman D, Smyth P and Welling M *et al.* Distributed inference for latent dirichlet allocation. In: Platt JC, Koller D and Singer Y *et al.* (eds). *Proceedings of NIPS Conference 2007*. Vancouver, British Columbia, Canada: Curran Associates, Inc., 1081–8.
24. Valiant LG A bridging model for parallel computation. *Comm ACM* 1990; **33**: 103–11.
25. Dean J and Ghemawat S Mapreduce: simplified data processing on large clusters. *Comm. ACM* 2008; **51**: 107–13.
26. Malewicz G, Austern MH and Bik AJ *et al.* Pregel: a system for large-scale graph processing. In: Elmagarmid AK and Agrawal D (eds). *Proceedings of SIGMOD Conference 2010*, Indianapolis, Indiana, USA: ACM 2010, 135–46.
27. Zinkevich M, Langford J and Smola AJ. Slow learners are fast. In: Bengio Y, Schuurmans D and Lafferty JD *et al.* (eds). *Proceedings of NIPS Conference 2009*. Vancouver, British Columbia, Canada: Curran Associates, Inc. 2009, 2331–9.
28. Agarwal A and Duchi JC. Distributed delayed stochastic optimization. In: Shawe-Taylor J, Zemel RS and Bartlett PL *et al.* (eds). *Proceedings of NIPS Conference 2011*. Granada, Spain, 2011, 873–81.
29. Recht B, Re C and Wright S *et al.* Hogwild: a lock-free approach to parallelizing stochastic gradient descent. In: Shawe-Taylor J, Zemel RS and Bartlett PL *et al.* (eds). *Proceedings of NIPS Conference 2011*. Granada, Spain, 2011, 693–701.
30. De Sa, C Olukotun K and Ré C. Ensuring rapid mixing and low bias for asynchronous gibbs sampling. In: Balcan MF and Weinberger KQ (eds). *Proceedings of ICML Conference 2016*. ACM, 2016, 1567–76.
31. Liu J, Wright SJ and Ré C *et al.* An asynchronous parallel stochastic coordinate descent algorithm. *J Mach Learn Res* 2015; **16**: 285–322.
32. Ahmed A, Aly M and Gonzalez J *et al.* Scalable inference in latent variable models. In: Adar E, Teevan J and Agichtein E *et al.* (eds). *Proceedings of WSDM Conference 2012*. Seattle, WA, USA: ACM 2012, 123–32.
33. Gemulla R, Nijkamp E and Haas PJ *et al.* Large-scale matrix factorization with distributed stochastic gradient descent. In: Apté C, Ghosh J and Smyth P (eds). *Proceedings SIGKDD Conference 2011*. San Diego, CA, USA: ACM 2011, 69–77.
34. Blei DM, Ng AY and Jordan MI. Latent dirichlet allocation. *J Mach Learn Res* 2003; **3**: 993–1022.
35. Griffiths TL and Steyvers M. Finding scientific topics. *PNAS* 2004; **101** (suppl 1): 5228–35.
36. Suits DB. Use of dummy variables in regression equations. *J Am Stat Assoc* 1957; **52**: 548–51.
37. Bennett J and Lanning S. *The netflix prize*. In: *Proceedings of KDD cup and workshop 2007*. San Jose, California, USA: ACM 2007, p. 35.