# Summary of Scaling Distributed Machine Learning with the Parameter Server

Runyu Zheng (runyuz), Shangquan Sun (sunsean)

## Problem and Motivation

Recently, more and more machine learning problems are at a large scale, so distributed optimization is needed to solve these problems. In existing frameworks, there are several problems still unsolved:

1. As the data is large, we can create complex models with parameters in a dimension of $10^9$ to $10^{12}$. In addition, training data could expand to 1 TB or even 1 PB (1024TB). The two problems requires gigantic memory and bandwidth.
2. Besides the limitation of memory and bandwidth, there is a problem that many machine learning problems are sequential, which cause high latency and difficulty of synchronization.
3. The third issue is fault tolerance. Because of huge computation amount, increasing data scale, and tendency to performing training on cloud platform where machine might be unreliable, fault tolerance is a key garuantee for sound computation.

Besides the issues about framework design, there are two other engineering problems: 1) Communication: how to pass and update parameters among workers, 2) Fault tolerance: how to achieve hot failover.

## Hypothesis

Previously there have been many related works, e.g. Amazon, Baidu, Facebook, Google, Microsoft, Yahoo, but none of them could solve all the problems above. For exmaple, the first generation of parameter servers could not have a high performance and flexibility.

The authors introduce a new generation of parameter server framework, a framework for solving distributed machine learning problems. It consists of multiple worker node groups responsible for training data, computing/updating parameters and one server node group responsible for aggregating all workers' updated parameters.

## Solution Overview

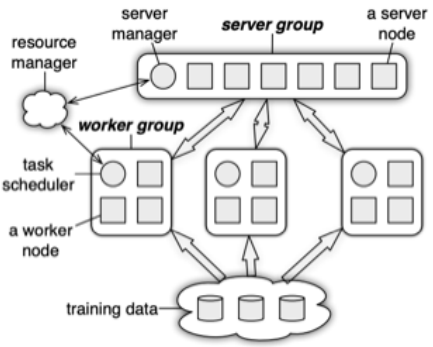
### Architecture Overview

For a proposed parameter server framework, there are one server group consisting of multiple server nodes and one server manager, as well as multiple worker groups consisting of a task scheduler and multiple worker nodes. At the beginning of excecuting a machine learning program, huge training data are divided into multiple parts and be assigned to each worker group. Then each worker group starts training, e.g. computing gradients. After iterations of training, parameters are pushed to the server group, in a form of (Key,Value) Vectors. The server group then aggregates updatings of parameters pushed from worker groups. Also worker groups need to pull back updated and aggregated parameters from server group, in order to replace and update their local old ones.

### (Key,Value) Vectors & Range Push and Pull

One key point in the architecture is that when pushing and pulling parameters only parts of parameters are passed by specifying a desired range of Key. This is done by a w. push(R, dest) or $\mathsf{w.pull(R, dest)}, where w is parameters, R is a specified range

and dest is destination. In this way, the problem of limited memory and bandwidth could be solved and performance could be increased.




### Asynchronous Tasks and Dependency & Flexible Consistency

There are many task dependencies. For instance, in the figure below, there is a task dependency between iteration 12 and iteration 11, but no such dependency exists for 11 and 10, which means 11 could be excecuted without waiting 10. This allows asynchronous processing.

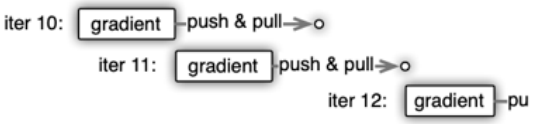



Figure 5: Iteration 12 depends on 11, while 10 and 11 are independent, thus allowing asynchronous processing.

But this is only one type of task dependencies, called 1 bounded delay. But there are also: sequential and eventual. For sequential, only synchronous processing is allowed. For eventual, fully asynchronous processing is possible. The authors define a coefficient called $\tau$, maximal delay time, which makes "a new task be blocked until all previous tasks τ times ago have been finished." By setting it 0, it becomes a sequential model, and algorithmic convergence rate increases as system efficiency decreases. By setting it infinity, it becomes an eventual model, and algorithmic convergence rate decreases as system efficiency increases. As a result, the value of $\tau$ determines a trade-off between "algorithmic convergence rate and system efficiency".

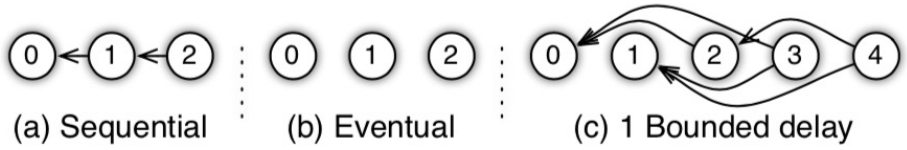



Figure 6: Directed acyclic graphs for different consistency models. The size of the DAG increases with the delay.

### User-Defined Functions on the Server & User-defined Filters

The framework allows users to define their own function except push and pull. Also users can define a method to filter those parameter entries not to be updated. In other words, based on the user-defined filters, the range R of parameter pair, (Key Value), is determined and specified.

## Limitations and Possible Improvements

There are many advantages of the proposed framework:

1. Ease of use, e.g. globally shared parameters
2. Efficiency due to asynchronous communication
3. Flexible consistency models for three kinds of models: sequential, eventual, and 1 bounded delay
4. Elastic scalability since it consists of multiple worker node groups and one server node group
5. Fault tolerance for stable long-term deployment.

But there are still som limitations and possible improvement:

1. Flexible consistency requires user to determine a maximal delay $\tau$, which leaves the systematic problem to users. Empirically, it is hard for a user to determine an appropriate maximal delay time so that algorithmatic convergence rate and systematic efficiency could be balanced well.
2. Addition and removal of nodes are discussed, but failures of the server manager or a task scheduler are not discussed.

## Summary of Class Discussion

**Q:** Is it always single server group or could it be more than one server group?

Yes, single server group.

**Q:** What value of $\tau$ should be used to keep the best balance of algorithmatic convergence rate and systematic efficiency?

If it is too high, it influence the speed of convergence. But if it is too low, the system processing is too slow. Without some exhaustive searching, Some number from 0 to infinity is really difficult to find.

**Q:** Replication happens for push and pull after aggregation is completed, what if aggregation is still happening while the server fails?

There should be acknowledgement from server to worker saying that the data is received. Replication mechanism is used for synchronized training, but not for asynchronized training.

# Summary of STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning

Runyu Zheng (runyuz), Shangquan Sun (sunsean)

## Problem and Motivation

As the development of computing power and data size, machine learning model become more complex. Complex machine learning models motivate model parallelism, which means splitting ML model parameter across machines. There are two main challenges for model parallelism:

- the model parameters are not independent
- different model parameters may take different numbers of iterations to converge. Hence, the effectiveness of a model-parallel algorithm is greatly affected by its schedule.

This paper aim to improve ML algorithm convergence speed by efficiently scheduling parameter updates, taking into account parameter dependencies and uneven convergence.

## Hypothesis

Typically, machine learning program implementations take the form of an iterative convergent procedure.

$$A^{(t)} = A^{(t-1)} + \Delta(D, A^{(t-1)})$$

where $t$ is current iteration, $A$ is model parameter, $D$ is input data, $\Delta()$ means model update.

In model parallel ML programs, parallel workers recursively update subsets of model parameters until convergence. With a scheuler, we have **scheduled model parallelism**(SchMP).

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^{P} \Delta_p(A^{(t-1)}, S_p(A^{(t-1)}))$$

where $\Delta_p$ is model update at worker $p$, the "schedule" $S_p()$ identifies a subset of parameters in A, instructing what a $p$-th parallel worker should work on.

In addition to the update rule, there are several intrinsic properties of ML algorithms:

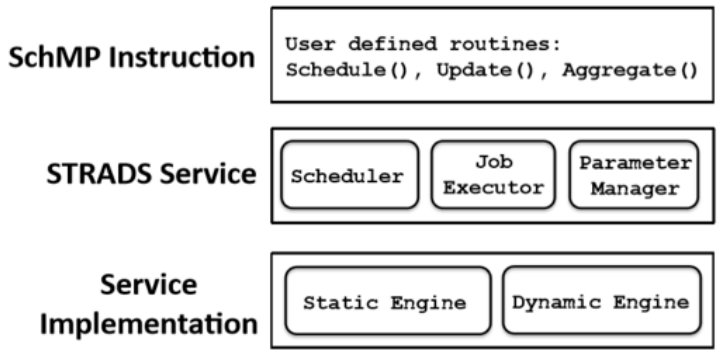
- **Model dependencies**: element in $A$ may affect each other.
- **Uneven convergence**: different model parameters converge at different rate.
- **Error-Tolerant**: limited amout of stoochastic error during calculating $\Delta(D, A^{t-1})$ does not lead to algorithm failure.

Since achieve **ideal model parallel** is expensive and still has the risk of violating model dependencies and incur errors, in this paper, the author restrict their attention to partition $M$ model parammeters across $P$ worker threads in an **approximately load-balanced manner**, with a number of partition strategies:

- **Static Partitioning**: pre-defined (highly depend on model)
- **Dynamic Partitioning**: partition via dependencies (good performance, low throughput)
- **Pipelining**: start next iteration before the previous one finishes (good)
- **Prioritization**: prefer parameters that will result in more convergence progress, need cheap ways to estimate potential progress (good)

## Solution Overview

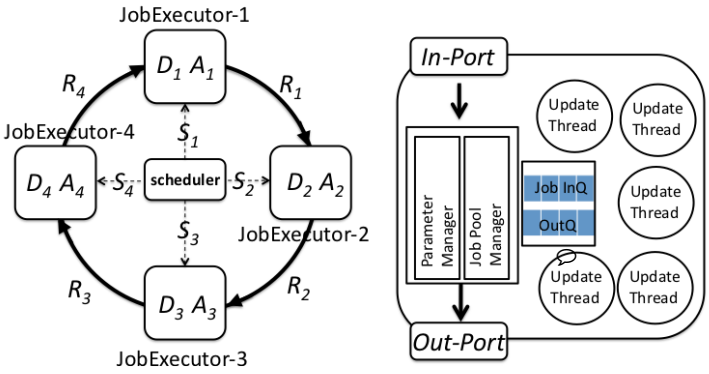
To execute SchMP programs, **STRADS** is built.




The goal is to improve ML convergence in two ways:

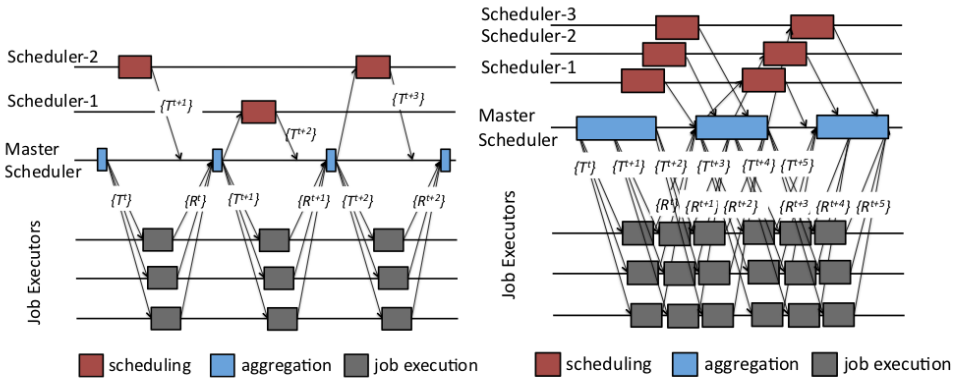
- Users can easily experiment with new model parallel schedules(created from **SchMP instruction** by users) for ML programs.
- STRADS **service for running SchMP over a cluster** and **system optimization** on different types of SchMP programs.

Because static- and dynamic-schedule SchMP algorithms have different need, so there are two engines in the third service. The following two figures are **static engine** and **Dynamic engine** for **service implementation**.




(a) Ring topology (b) Job Executor

Figure 2: **Static Engine:** (a) Parameters $A_p$ and intermediate results $R_p$ are exchanged over a ring topology. (b) **Job Executor** architecture: the **Parameter Manager** and a job pool manager receive and dispatch jobs to executor threads; results $R_p$ are immediately forwarded without waiting for other jobs to complete.




(a) Non-pipelined (b) Pipelined

Figure 3: **Dynamic Engine** pipelining: (a) Non-pipelined execution: network latency dominates; (b) Pipelining overlaps networking and computation.

## Limitations and Possible Improvements

- Users might not have a sense of what `schedule()` is good, so algorithm might be needed to create a schedule function for them.
- The system can be extended to hybrid data-parallelism.
- The use of SchMP can be applied to other platform.

## Summary of Class Discussion

**Q:** In these two papers, most of the benefits come from asynchronous update using stale data, do they give insight about when to use stale data and how much stale data can we use?

- Tradeoff:
  high staleness - not converge
  low staleness - too slow
- May get insight from dependency graph by figuring out how important a node is;
  depend on user's need(e.g., in some cases, high consistency is desired).

**Q:** Why cannot dynamic engine be used as a static one as well?

If things remain static, it would be more efficient.