# Adaptive Consistency Management for Distributed Machine Learning

by

Christoph Alt

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

TECHNISCHE UNIVERSITÄT BERLIN

April 15, 2017

Supervisors:

Prof. Dr. Odej Kao

Prof. Dr. Volker Markl

Tobias Herb

# Adaptive Consistency Management for Distributed Machine Learning

by

Christoph Alt

Submitted to the Department of Electrical Engineering and Computer Science
on April 15, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

## Abstract

In recent years, machine learning emerged as the core of many successful applications and businesses. While the rise of artificial intelligence continues, the amount of data to be processed grows even faster. Unsurprisingly a lot of research has since focused on methods to parallelize commonly used algorithms and new systems and frameworks have been published, trying to improve the performance of distributed machine learning. Even though there has been a lot of progress towards more efficient systems, many state-of-the-art systems still have limitations. Current frameworks are neither expressible nor flexible enough to allow for efficient development of distributed machine learning algorithms, making them unsuited for experimentation and quick prototyping even though this is essential for achieving optimal performance. On the other hand, most parallelization strategies exploit the algorithms inherent stochastic nature to enable parallel execution at the expense of lowered consistency among the shared state. Even though this does not necessarily affect quality of the results, an improper chosen level of consistency can severely affect algorithm performance, resulting in a non optimal convergence rate and therefore increased runtime.

This thesis presents a novel framework for distributed machine learning, which uses a state centric programming model with yield semantics. The result is a programming model that allows the user to focus on the key elements of developing distributed machine learning algorithms, namely progress communication and consistency management among distributed workers. The framework also takes care of utilizing the cluster resources in an optimal fashion. Experiments using lasso linear regression show an increased performance compared to state-of-the-art data processing systems like Apache Spark and at the same time reduce the effort of developing, parallelizing and experimenting with distributed machine learning algorithms at scale.

## Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Hiermit versichere ich an Eides statt, dass ich die vorliegende Masterarbeit ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Teile, die wörtlich oder sinngemäß einer Veröffentlichung entstammen, sind als solche kenntlich gemacht. Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Christoph Alt

# Contents

# List of Figures

# Chapter 1

# Introduction

One of the most challenging tasks in computer science and engineering resolves around improving algorithm performance. In general this has been done by making hardware faster and inventing new strategies and algorithms to parallelize work more efficiently. Since it is clear that Moore's Law will not hold anymore, a lot of effort has been spent to horizontally scale algorithm computation across multiple machines. Machine learning is no exception and efficient parallelization is a key aspect towards more intelligent systems. By now, many general purpose frameworks for large scale data processing have been developed and published. Many of those are used for running more complex machine learning algorithms at scale as well. Unfortunately, the performance often is not satisfying due to the architecture and programming model not reflecting the underlying structure of most commonly used machine learning algorithms. Common data processing tasks can be represented as an extract-transform-load (ETL) pipeline, which is often easily parallelizable. This does not hold for machine learning, where algorithms are mostly sequential in nature and the only way of enabling parallel computation is by exploiting their inherent stochastic properties. This allows to break the sequential execution in favor of parallel learning on subsets, which then needs to be combined in order to obtain a global solution to the task. While this can lead to a great speedup in terms of the amount of data processed, it can have a negative affect on the learning progress. Therefore a key part of horizontally scaling machine learning algorithms is to ensure all participating learners have a consistent view on each others progress while at the same time maintaining a trade-off between communicating progress and spending time on their own learning task.

## 1.1 MapReduce and Beyond

Many of todays successful businesses throughout fields like finance, e-commerce and health-care rely heavily on the ability to process vast amounts of user or sensorical data, collected to make services smarter and user experience better. In order to learn from the collected data, discover patterns and ultimately gain new insights, it needs to be processed by an algorithm. It is not uncommon that the input ranges somewhere between hundred gigabytes and tenth of petabytes. In the past, processing this much data required either a supercomputer, which was only available to large institutions or government entities, or some proprietary compute cluster. All this changed when Google introduced MapReduce [1] in 2004. The MapReduce framework made it possible to process data in a distributed and fault tolerant way with the help of a compute cluster formed by hundredth or thousandth of machines. Instead of using a single, expensive, special hardware supercomputer the framework provides the foundation to assemble commodity hardware machines into a compute cluster. The framework takes care of all necessary aspects to ensure a fault tolerant and parallel execution of a task submitted to the cluster. The advantage compared to previous approaches is that the framework can be run entirely on top of machines using commodity hardware, which does not require special hardware and therefore equals low cost.

MapReduce essentially led the path to a convenient and widespread use of big data processing, which found its open source implementation in the Apache Hadoop project [2]. The project quickly gained traction and has spawned many business grade platforms, which quickly gained widespread adoption and by now provides a whole ecosystem around big data processing. The software stack includes a fault tolerant distributed file-system (HDFS) a MapReduce framework and a cluster resource manager (YARN) [3]. On the other hand, MapReduce suffers from some practical limitations that lead to the development of new, more sophisticated and specialized big data frameworks. With the most widely used frameworks being Apache Spark [4], Apache Flink [5] and GraphLab [6]. The first two frameworks use at it's core a data-flow pipeline based architecture, whereas the latter uses a graph abstraction to model particular algorithms. All this works well for algorithms that can be expressed as an extract-transform-load pipeline and are often embarrassingly parallel in nature. On the other hand, machine learning algorithms often rely heavily on many, computationally light, iterations to iteratively update a shared state (model) such as logistic regression or Latent Dirichlet Allocation (LDA). These so called iterative-convergent algorithms required a change in how systems for distributed machine learning operate at its core.

## 1.2 Distributed Machine Learning

This limitation essentially sparked the development of specialized frameworks for distributed execution of iterative-convergent algorithms used in common machine learning tasks. The most widely recognized systems are Petuum [7], ParameterServer [8] and MALT [9]. Different from the MapReduce paradigm, these frameworks, instead of using a pipeline to transform an immutable dataset into another immutable dataset, operate on a fixed but mutable state, which is held by a single machine or distributed over multiple machines. This state can then be updated by workers computing an update locally and sending it to these state keepers taking updates and merging these into the state by some user defined function. While these systems can increase the performance on machine learning algorithms by an order of magnitude [7] compared to data-flow systems, most systems come with either limited usability, which makes it difficult to implement additional algorithms, are tied to a specific algorithm or are very low level frameworks. Efficiently distributing machine learning algorithms while at the same time provide the ability to concisely express machine learning algorithms remains an extremely challenging problem. A system targeting the execution of those algorithms at scale must therefore provide the ability to concisely express the underlying algebraic structure and at the same time be flexible enough to allow experimentation and fine tuning. Where consistency management is an essential part to ensure that algorithms are executed both fast and efficient but most importantly, correct.

## 1.3 Thesis Outline

This thesis introduces a novel framework for large scale distributed machine learning. It improves upon currently available systems by providing a powerful programming abstraction that can concisely express state of the art machine learning algorithms and at the same time minimizes the effort necessary to move from a single machine to a cluster. The framework design is optimized for efficient parallel asynchronous execution of iterative-convergent algorithms in a cluster and ensures the required consistency is enforced among parallel learners, depending on the algorithm properties. By allowing the user to decide how to maintaining the best trade-off between algorithm execution and progress communication the performance is improved as well. All of this can be easily customized for quick prototyping and fine tuning, which makes the system suited for developers as well as researchers. The goal of this thesis is to implement the state centric programming model and show its performance in comparison with Apache Spark on an example implementation of the CoCoA [10] framework. Chapter 2 starts off by providing a background on the architecture and inner workings of state of the art frameworks for big data processing and distributed machine learning. Additionally the most commonly used algorithms and optimization techniques are introduced. The majority of those algorithms can be classified into the group of so called iterative-convergent algorithms for which a more formal treatment is provided. Furthermore the chapter introduces the common algorithm parallelization strategies in distributed machine learning. The chapter concludes by providing an overview over the challenges and issues that need to be addressed and considered when developing a distributed machine learning system and how this is achieved by current frameworks. This will give rise to understanding why a different framework architecture and abstraction is necessary to improve the performance and expressibility of large scale distributed machine learning applications. Chapter 3 therefore introduces the state centric programming model, which treats the state as a mutable first class citizen, which can be distributed and altered by updates that result from distributed computation. This allows the system to reason about the most optimal distribution of state in the cluster, which is then scheduled with computation that can update the state. Subsequently the chapter describes the architecture of the system and how the essential components are implemented. When updating a shared state from multiple different locations, consistency must be maintained in order to ensure the algorithm progresses as expected. The system is responsible for managing a state's consistency among all of its instances across the cluster. Chapter 4 therefore describes several schemes for ensuring consistency and at the same time optimizing for bandwidth and computational resource usage. In order to show the system and its consistency management in action, Chapter 5 compares the system against Apache Spark by running the CoCoA framework on two datasets with elastic-net regularized linear regression as the chosen algorithm. Chapter 6 summarize the experiments with the lessons learned and provides suggestions on how to further improve systems for large scale distributed machine learning.

# Chapter 2

# Background

This section provides the necessary background to follow the argumentation in the following chapters regarding state centric programming model and consistency management. This includes an understanding of algorithms and optimization techniques commonly used in practice (not limited to distributed machine learning), the current state-of-the-art in data-flow systems and how data-flow is used to provide a fault tolerant and distributed framework for large scale data processing and machine learning. Furthermore the field of distributed machine learning is explained in more detail, including the current state-of-the-art frameworks used for this purpose, their limitations and challenges that arise when machine learning algorithms are parallelized in a distributed fashion among multiple physical machines.

## 2.1 Algorithms and Optimization

### 2.1.1 Iterative-convergent Algorithms

Consider a supervised learning setup with a dataset $D = \{z_1, \ldots, z_n\}$ with each example $z_i$ being represented by a pair $(x_i, y_i)$ consisting of an input $x_i$ and a scalar output $y_i$. Consider also a loss function $\ell(\hat{y}, y)$ quantifying the cost of predicting $\hat{y}$ when the true output is $y$. As a model, a family $F$ of functions $f_w(x)$ parameterized by a weight vector $w$ is chosen. The goal is to find a function $f \in F$ that minimizes the loss $Q(z, w) = \ell(f_w(x), y)$. The empirical risk

$$E_n(f_w) = \frac{1}{n} \sum_{i=0}^{n} \ell(f_w(x_i), y_i) \tag{2.1}$$

quantifies the performance on training set. In order to find an optimal solution many algorithms used in large scale machine learning such as regression, topic models, matrix factorization or neural networks employ either gradient based methods or Markov chain Monte Carlo methods. To obtain the optimal solution those algorithms try to iteratively update the weight vector $w$. At each iteration $t$ an updated weight vector $w^t$ is computed based on the vector of the previous iteration $w^{(t-1)}$ and the data $D$. The resulting model $f_{w^t}$ is again a better summary of the data $D$ under the objective $Q$. 2.2 shows the process of refining the model, with $\Delta$ being an arbitrary update function.

$$w^t = w^{(t-1)} + \Delta(D, w^{(t-1)}) \tag{2.2}$$

The update function depends on the algorithm employed and can be viewed as a procedure of obtaining a step towards a better model. At each iteration an update $\Delta w$ is computed and applied to the previous weight vector until a stopping condition is satisfied. E.g. the objective difference between two iterations or the empirical loss on a test set is monitored. If the difference is below a certain threshold the computation stops and the algorithm is said to be converged.

### 2.1.2 Convex Optimization

In order to estimate the optimal parameters $w^*$ of a function belonging to class $f_{w^*} \in F$, numerous techniques can be employed to estimate said parameters. In many cases, especially large scale machine learning, methods such as (stochastic) gradient descent and coordinate ascent are used to iteratively optimized the parameterization of the chosen function class. Both techniques represent different rules of computing the update shown in 2.2. Gradient descent updates the weights $w$ at each iteration $t$

on the basis of the gradient of $E_n(f_w)$,

$$w^t = w^{(t-1)} - \eta \frac{1}{n} \sum_{i=0}^{n} \nabla_w Q(z_i, w^{(t-1)}) \tag{2.3}$$

where $\eta$ is a chosen gain, often referred to as learning rate. While this achieves linear convergence under sufficient regularity assumptions and a sufficiently small learning rate $\eta$ [11] [12], a more simplified version called stochastic gradient descent (SGD) is commonly used in practice. Instead of computing the gradient $\nabla_w E_n(f_w)$ exactly, the gradient is estimated at each iteration $t$ based on a single randomly picked example $z_t$.

$$w^t = w^{(t-1)} - \eta_t \nabla_w Q(z_t, w^{(t-1)}) \tag{2.4}$$

The assumption is that the gradient obtained by 2.4 behaves similar to its expectation in 2.3. The convergence properties have been studied extensively and under mild conditions an almost sure convergence can be established when the learning rate satisfies the conditions $\sum_t \eta_t^2 < \infty$ and $\sum_t \eta_t = \infty$ [12]. The general structure of stochastic gradient descent is described in (1). On the other hand, coordinate

**Algorithm 1** Stochastic Gradient Descent

```
1: $t \leftarrow 0$ and initialize $w^0 \in \mathbb{R}^d$
2: **repeat**
3:     $t \leftarrow t + 1$
4:     **for** $k$ in $shuffle([1, \ldots, n])$ **do**
5:         $w^t \leftarrow w^{(t-1)} - \eta_t \nabla_w Q(z_k, w^{(t-1)})$
6: **until** termination criteria satisfied
```

descent, described in (2), iteratively tries to optimize a given objective by successively performing approximate minimization along a coordinate direction while keeping the other directions fixed. Where $e_k$ is the unit vector in dimension $k$ and $\alpha_k$ is the step size. The step size can be chosen in multiple ways, e.g. solving for the exact minimizer along a single dimension with the other dimensions fixed or a traditional line search.

**Algorithm 2** Stochastic Coordinate Descent

```
1: $t \leftarrow 0$ and initialize $w^0 \in \mathbb{R}^n$
2: repeat
3:     $t \leftarrow t + 1$
4:     for $k$ in $shuffle([1, \dots, d])$ do
5:         $w^t \leftarrow w^{(t-1)} - \alpha_k [\nabla_w Q(w^{(t-1)})]_k e_k$
6: until termination criteria satisfied
```

### 2.1.3 CoCoA

Due to their widespread application in large scale machine learning and recent advances in the field of distributed optimization the thesis focuses on linear regularized objectives. The theoretical contemplation as well as the experiments in Section 5 focus on a framework for convex optimization problems called CoCoA (Communication-efficient distributed dual Coordinate Ascent) [10] and its successors CoCoA$^+$[13] and PROXCoCoA$^+$[14]. CoCoA as described in Algorithm 3 provides a communication-efficient framework for solving convex optimization problems of the following form

$$\min_{\alpha} Q(z, \alpha) = \ell(f_\alpha(x), y) + r(\alpha) \tag{2.5}$$

in a distributed setting. Where $\alpha$ denotes the weight vector, $\ell$ is convex and smooth and $r$ is assumed to be separable, which in this context means $r(x) = \sum_{i=0}^{n} r_i(x_i)$. Commonly the term $\ell$ is an empirical loss over the data of the form $\sum_i \ell(f_w(x_i), y_i)$ and the term $r$ is a regularizer, e.g. $r(w) = \lambda \|w\|_p$ where $\lambda$ is a regularization parameter. Many algorithms in machine learning can be expressed in this form, such as logistic and linear regression, lasso and sparse logistic regression and support vector machines.

CoCoA leverages the primal-dual relation which allows for solving the problem in either the primal or dual formulation. For some application where the number of examples $n$ is much smaller than the number of features $d$, $n \ll d$, solving the problem in the dual may be easier because this problem has $n$ variables to optimize, compared to $d$ for the primal. The CoCoA framework leverages Fenchel-Rockafellar duality to quadratically approximate the global objective in 2.5. This leads to separability of the problem over the coordinates of $\alpha$ and the partitions, where the local subproblems are similar in structure to the global problem and also exploit second order information within the local data partition. Therefore the dataset $D \in \mathbb{R}^{d\times n}$ can be distributed either example-wise or feature-wise over $K$ physical machines according to the partitioning $\{P\}_{k=1}^{K}$, depending which is more efficient to solve. The size of the partition on machine $k$ is denoted by $n_k =\mid P_k \mid$. The key to efficient distributed optimization is that the local subproblems can be solved independently on each worker in parallel and only a single parameter vector $v = \nabla f(D\alpha) \in \mathbb{R}^n$ needs to be shared after each round in order to communicate the progress of each local worker on its subproblem. As the data stays local and only a single parameter vector of dimension $n$ needs to be exchanged, CoCoA is very communication efficient and as the local subproblems are very similar to the global problem, arbitrary solvers can be employed as well. The local quadratic subproblem has the following form

$$\min_{\alpha_{[k]}\in\mathbb{R}^n} \zeta_k^{\sigma'}(\Delta\alpha_{[k]}, v, \alpha_{[k]}), \tag{2.6}$$

where

$$\zeta_k^{\sigma'}(\Delta\alpha_{[k]}, v, \alpha_{[k]}) = \frac{1}{K} f(v) + w^T A_{[k]}\Delta\alpha_{[k]} + \frac{\sigma'}{2\tau}\|A_{[k]}\Delta\alpha_{[k]}\|^2 + \sum_{i\in P_k} g_i(\alpha_i + \Delta\alpha_{[k]_i}), \tag{2.7}$$

with $A_{[k]}$ referring to the the local data and $w = \nabla f(D\alpha) \in \mathbb{R}^n$. $\alpha_{[k]_i}$ denotes the local weight vector stored on machine $k$ with $\alpha_{[k]_i} = \alpha_i$ if $i \in P_k$ otherwise $\alpha_{[k]_i} = 0$. The aggregation parameter $\gamma$ decides how the updates computed from the subproblems are combined. Commonly $\gamma$ is chosen to be in the range between $\frac{1}{K}$ (averaging) and 1 (adding). $\sigma'$ is the subproblem parameter, measuring the difficulty of the data partitioning $\{P\}_{k=1}^{K}$. The parameter is commonly set to $\sigma' = \gamma K$ but could also be improved, depending on the data. In summary the framework provides a procedure to effectively combine the results from local computation without having to deal with conflicts resulting from similar updates computed on other machines.

**Algorithm 3** CoCoA Framework

```
Data: Data matrix $D$, distributed column-wise according to partition $\{P_k\}_{k=1}^K$
Input: aggregation parameter $\gamma \in (0,1]$, subproblem parameter $\sigma'$
Initialize: $t \leftarrow 0$, $\alpha^{(0)} \leftarrow 0 \in \mathbb{R}^n$, $v^{(0)} \leftarrow 0 \in \mathbb{R}^d$
1: repeat
2:     $t \leftarrow t + 1$
3:     for $k \in \{1, \dots, K\}$ do
4:         obtain an approximate solution $\Delta\alpha_{[k]}$ for the local subproblem 2.6
5:         update local weights $\alpha_{[k]}^t = \alpha_{[k]}^{(t-1)} + \gamma\Delta\alpha_{[k]}$
6:         update shared parameter vector $\Delta v_k = A_{[k]}\Delta\alpha_{[k]}$
7:     compute $v^t = v^{t-1} + \gamma \sum_{k=1}^K \Delta v_k$
8: until termination criteria satisfied
```

## 2.2 Machine Learning with Data-flow Systems

Data-flow based systems such as Apache Spark [4] and Apache Flink [5] are not only the most widely used systems for processing large amounts of data but also for distributed machine learning. Figure 2-1 shows a data-flow pipeline as it would be used for executing an iterative-convergent algorithm such as linear regression, logistic regression or support-vector machines at scale. At first, the data is loaded partition-

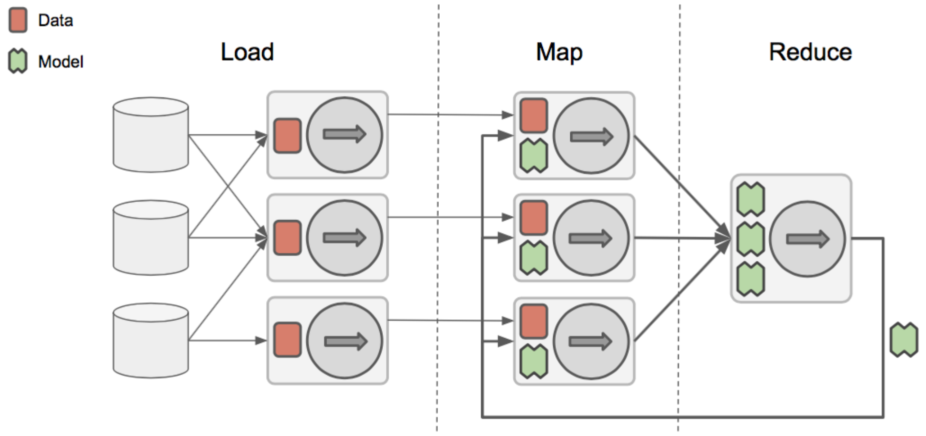

Figure 2-1: Iterative-Convergent Algorithms with Data-Flow

wise from the storage into an Resilient Distributed Dataset (RDD), which is the general representation of a dataset in Spark. As data-flow systems follow closely the paradigms of functional programming and therefore must satisfy referential transparency, each dataset is an immutable data structure that can be transformed into another dataset by applying some function to it. The immutability of a dataset eases the development of distributed algorithms considerably, as no means of synchronization such as locks are necessary when multiple machines work on the same dataset. Following the data loading, a mapping phase is scheduled that does one pass over the local partition of the input dataset in order to compute a refined model according to the employed optimization technique. In the last phase, all local models are sent to a reducer over the network, which in turn combines the received models into a single model according to a user defined function. Depending on the number of iterations configured, the flow of mapping over the partitioned data followed by reducing the local models is executed repeatedly. This procedure follows the scheme proposed in [15] and therefore has the same limitations. The weak points are the inflexibility to control the communication frequency of the refined models as well as limiting the communication to only the important changes in model parameters. Another limiting factor is the synchronization barrier between each phase, which makes the scheme susceptible to the well-studied stragglers problem [16].

## 2.3 The Challenges of Distributed Machine Learning

While the execution of iterative-convergent algorithms on a single machine is straightforward, time constraints and the ever growing amount of data to be processed require these algorithms to be executed in parallel. This posses a number of challenges which are often observed when parallelizing algorithms, such as partitioning the state used in the algorithm as well as communication and consistency management. In this context, state refers to a structure containing arbitrary data e.g. an array, tensor or list. While intra-node parallelism in multi-core and multi-processor systems can mitigate these problems, it is not satisfying in terms of cost and scalability. On the other hand, inter-node parallelism has the desired properties but can not be easily achieved while maintaining high performance. Therefore in the past decade a lot of research has focused on inventing new systems to deal with those challenges and make distributed machine learning more efficient and scalable.

### 2.3.1 State Partitioning

Depending on the algorithm and optimization technique employed to solve for the optimal solution, there exist multiple approaches to distribute the state used in the algorithm (e.g. input data, model). As depicted in Figure 2-2, an initial approach by Zinkevich et. al [15] introduced a data-parallel approach for computing stochastic gradient descent (SGD) via the MapReduce framework. In this context, data-parallelism

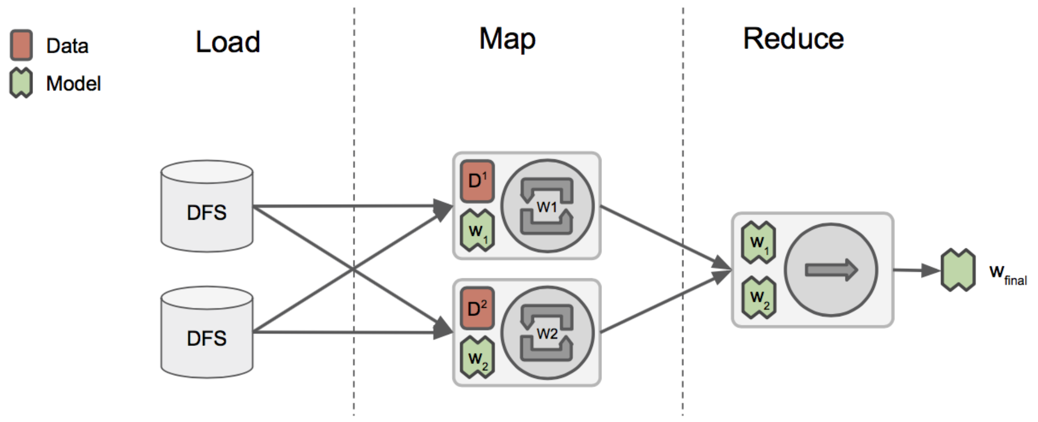

Figure 2-2: Data-Parallelism with MapReduce

means that $K$ machines work in parallel on the input data $D$, hence the data is distributed according to partition $\{P_k\}_{k=1}^{K}$ into local parts $D^k$. Each machine maintains a local model $w_k$ that is iteratively refined based on 2.2 until convergence, using only the local part of the input data $D^k$. The local update is of the form

$$w_k^t = w_k^{(t-1)} + \Delta(w_k^{(t-1)}, D^k) \tag{2.8}$$

The final model $w_{final}$ is then obtained in the reduce step by averaging over all local models $w_k$.

$$w_{final} = \frac{1}{K}\sum_{k=1}^{K} w_k \tag{2.9}$$

Even though this approach works well in practice and gives considerable good results, it suffers from three limitations. First, in certain cases the size of a local model $w_k$ exceeds the available memory on a single machine, which leads to the algorithm not work properly. This is the case e.g. for topic models at web scale or deep neural networks used in the Google Brain project [17], consisting of billions of parameters. Second, even though the scalability improved by introducing data-parallelism the lack of parameter exchange during runtime can lead to suboptimal performance [18] as the algorithm essentially resembles batch gradient descent, which is known to have suboptimal convergence properties compared to mini-batch or stochastic gradient descent [12] [14]. Third, even though state immutability is beneficial in data-flow extract-transform-load pipelines it seriously affects performance when running algorithms that simultaneously refine a state frequently in multiple consecutive iterations. In order to improve the performance, a system therefore must be able to employ mutable state, communicate more frequently and must also be able to distribute a model across multiple machines to scale with the size of the model. Following those requirements resulted in the publication of the parameter server [8], which provides a framework for inter-node parallelism of iterative-convergent algorithms.

### 2.3.2 Communication

As depicted in Figure 2-3, the parameter server is a group of an arbitrary number of machines $\{S\}_{l=1}^{L}$, e.g. $S = \{S_1, S_2\}$ where each member of the group is responsible for storing a part of the model according to the partitioning $\{P_w\}_{l=1}^{L}$ and making it accessible to the workers $\{W\}_{k=1}^{K}$, e.g. $W = \{W_1, W_2, W_3\}$, via a defined interface similar to a key-value store. The model is partitioned among the $L$ machines of the server to provide optimal throughput, fault tolerance and to mitigate the effect of a model exceeding the memory of a single machine, if necessary. Each of the workers maintains a local partition of the input data $\{D\}_{k=1}^{K}$, which is used to iteratively compute updates for the parameters $w$ according to

$$\Delta w_{k_i}^{t} = \Delta(w_{k_i}^{(t-1)}, D^k). \tag{2.10}$$

In the parameter server setup, the local state $w$ acts as a cache for global parameters in order to reduce network usage. Depending on the caching policy, $w_{k_i}$ can either

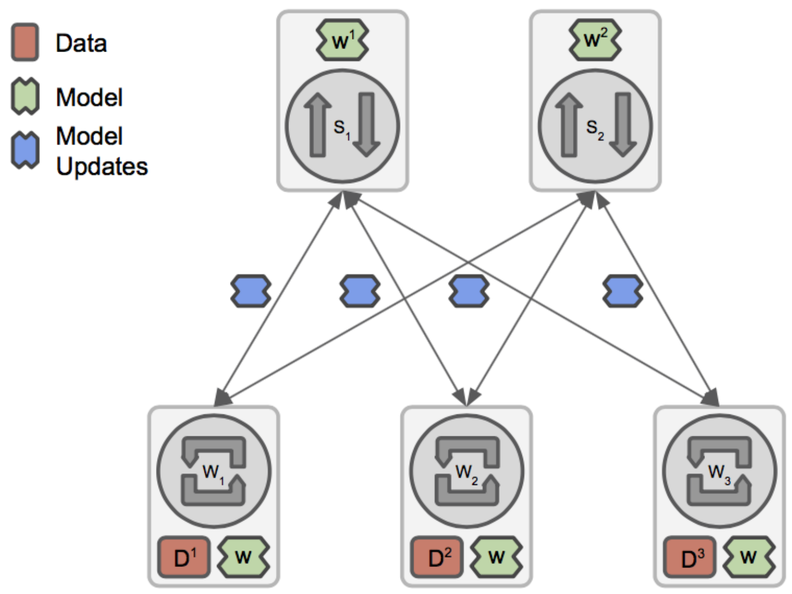

Figure 2-3: Parameter Server

be directly read from the local cache or must be retrieved from the parameter server. Additionally to applying the update to the local model $w$, the delta $\Delta w_{k_i}^t$ is published to the parameter server as well. The logic on the server takes care of applying the update to the corresponding entry $w_i$, in order to make the progress available to all other workers $W_i$, $i \neq k$. In case multiple updates for the same parameter $w_i$ arrive at the same time, a user defined function (UDF) needs to be provided to the server, which takes care of combining those updates so it can be applied to the parameter stored on the server. In contrast to other algorithms and similar to conflict-free replicated data types (CRDTs) the updates are associative and hence do not need to follow a total ordering when combined. Therefore, the procedure of retrieving, updating and publishing can be executed concurrently on all workers. This enables all workers to work in parallel on the iterative parameter refinement while asynchronously updating and retrieving the parameters necessary for computing the next update.

While this schema has been proven to work well in practice [8] [19], a couple of issues still remain when running iterative-convergent algorithms at scale using the parameter server. According to [20] this can be viewed as finding the trade-off between algorithm throughput and data throughput. In other words the challenge is to find a balance between the quality of parameter refinements and the quantity at which they are generated. As with distributed systems in general, network communication is the bottleneck in distributed machine learning as well. Even though, due to the stochastic nature of many machine learning algorithms, there exists a flexibility in the rate and amount of communication compared to the exact serial algorithm, it has been proven that fresher model parameters increase the algorithm throughput per iteration [21]. Therefore, in order to guarantee an optimal algorithm throughput, a worker should always work with the latest parameters. On the other hand, exchanging parameter updates over the network more frequently consumes more system resources, which leaves less time to run local computation and therefore essentially decreases the data throughput due to increased time spent on network management. The second issue concerns the relaxed consistency among participating workers due to reduced communication compared to the serial algorithm. Though this is what makes the parameter server concept so powerful because it increases the data throughput by relaxing the consistency requirements when updating parameters on the server. By decoupling the progress of workers it is possible to minimize the effect of stragglers and synchronization delay between workers [16]. However, as discussed in the next section, combining model parameters obtained from workers with greatly differing algorithm progress can have a detrimental effect on algorithm throughput. Finding the balance between algorithm throughput and data throughput can therefore be seen as consistency management.

### 2.3.3 Consistency

The most important part of any distributed system is the synchronization strategy used to ensure consistency of a state among multiple machines concurrently accessing and updating it. It must be emphasized that consistency not only concerns proper synchronization of simultaneous state access but also synchronization of parallel computation. In distributed machine learning the shared state is the model, which is for example stored in a parameter server and continuously refined by updates locally computed by a worker and then published to the server. There are three schemes used to synchronize computation on workers during the iterative parameter refinement. *Bulk synchronous parallelization (BSP)* leads to the best algorithm throughput (convergence achieved per number of examples processed). In this scheme, all workers are required to finish their current iteration and at the end successfully publish all updates to the parameter server. The server then computes a refined model $w^t$ according to 2.11 and each worker retrieves the updated parameters before beginning the next iteration. This synchronization scheme guarantees consistency among all nodes at all times.

$$w^t = w^{(t-1)} + \frac{1}{K}\sum_{k=1}^{K} \Delta(w_k^{(t-1)}, Dk) \tag{2.11}$$

While this synchronization strategy essentially recovers the sequential algorithm for a single machine and has the same convergence properties and guarantees, it suffers from a severe limitation when used in a distributed setup [21]. In case one of the workers is for some reason a lot slower than the others the synchronization barrier imposed by BSP forces all workers to wait for this particular worker in the group. This is well known as the straggler problem [16] and can seriously affect performance in a distributed environment, because the progress is limited by the slowest node in the cluster. BSP is commonly used in MapReduce frameworks such as Hadoop and data-flow systems like Apache Spark and Apache Flink to ensure correct program execution. The second strategy is known as *total asynchronous parallelization (TAP)*. Similar to BSP, all workers publish their locally computed parameter updates to the server after each iteration but in this case the changes are applied to the model immediately. Therefore no waiting for other workers is required, resulting in a very high data throughput. The straggler problem can be mitigated by this synchronization scheme as well, as depicted in Figure 2-4. Even though worker $W_3$ is a straggler, which would have prevented the remaining workers $\{W_2, W_3\}$ from proceeding beyond the synchronization barrier of iteration 1, the workers can continue with their next iterations without waiting for the slower worker. Although this consistency scheme seems to work quite well in practice [8], it lacks formal convergence guarantees and can even diverge [19]. One reason for this is the fact that the divergence in

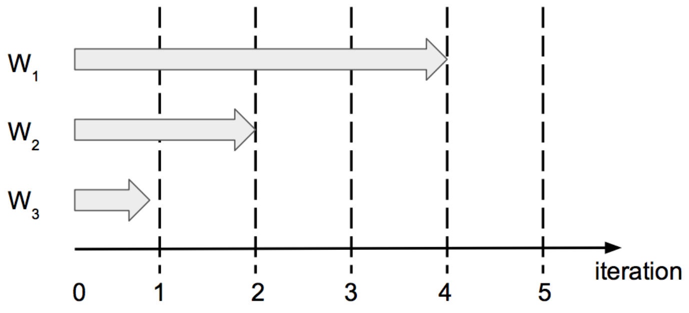

Figure 2-4: Straggler in TAP

iterations between workers is unbound, therefore no theoretical convergence bound can be established. Furthermore the lack of synchronization can lead to a situation where parameter updates computed on slower workers based on stale parameters spoil the current global solution. A middle ground between bulk synchronous paralleliza-

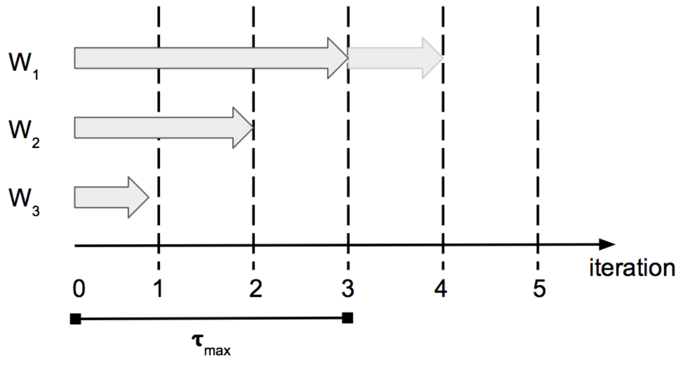

Figure 2-5: Straggler in SSP

tion and total asynchronous parallelization is *stale synchronous parallel (SSP)* [22] or *bounded staleness (BS)*. As shown in Figure 2-4, SSP introduces a fixed maximum delay, or staleness threshold, of $\tau_{max}$ between the slowest and fastest node. In the example, for $\tau_{max} = 3$ worker $W_1$ is blocked and can not proceed beyond iteration 3 as the slowest worker $W_3$ has not finished its first iteration. As soon as worker $W_3$ has completed its first iteration, $W_1$ is unblocked and can proceed as long as the difference in iteration stays below $\tau_{max}$. SSP overcomes some of the limitations of TAP by introducing a bound on divergence in number of iterations between workers. The staleness threshold resembles a bound which can be used to restore formal convergence guarantees while still maintaining the flexibility of asynchronous parallelization and limiting but not completely preventing the straggler problem [23]. In general this helps to compensate e.g. update related communication between iterations or fluctuations in worker performance, which explains why SSP works so well in practice.

# Chapter 3

# State Centric Programming Model

Most state-of-the-art frameworks for distributed machine learning like Petuum [7] and Parameter Server [8] are based on the parameter server concept introduced in the previous section. This framework essentially provides a low level API[1] for publishing and retrieving values similar to a distributed key-value store, where the key $i$ is for example the index of a weight vector $w$ stored on the server and the value is the weight $w_i$. Implementing an algorithm that relies on a parameter server requires incorporating publishing and retrieval of parameters deeply into the algorithm definition. This contrasts the general work flow of developing and testing an algorithm locally on a single machine and then transition to a distributed environment such as a cluster. Also current frameworks provide only a minor abstraction, leaving the developer with the task of distributing state, scheduling distributed computation, consistency management and managing cluster resources. A developer should be able to focus on the main goal, namely solving a domain specific problem by the help of distributed machine learning. This section introduces the state centric programming model, general architecture of the framework and its main parts based on the example of iterative-convergent algorithms. Although its application is not limited to this particular algorithm family.

As depicted in Figure 3-1 the framework consists of three major parts, designed to support the developer with the development as well as execution of distributed

[1]Application programming interface

machine learning algorithms. First, at edit-time it provides a collection of primitives that can used to describe the algorithm with the help of the state centric programming model. The programming model treats state as a first class citizen which can be distributed according to a given partitioning and altered by local and remote transformation in parallel. Depending on how a state is distributed and the type of transformation applied to it, the consistency management coordinates the distributed execution of all algorithm steps in the most efficient way that still ensures the correctness of the result. In the context of machine learning, a correct result can be obtained in different ways as discussed in Section 2.3.3. Therefore additional primitives are offered by the framework to provide the consistency management with further instructions on how to ensure a consistent distributed execution given a particular state and transformation. Secondly, at compile-time the framework provides an algorithm

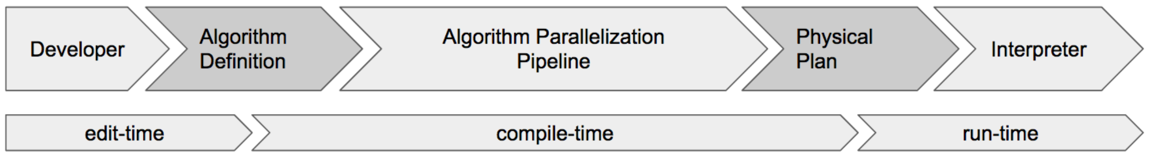

Figure 3-1: Framework Architecture

parallelization pipeline, which takes the algorithm definition as an input and compiles a physical plan by applying a sequence of enrichment steps to it. These enrichment steps, as discussed in Section 3.2, take properties of the algorithm and cluster infrastructure into account to find the optimal execution strategy. The physical plan contains a detailed description of how to execute the algorithm in a distributed manner on a specific group of machines within the cluster. As the third and final part of the framework, at run-time the interpreter is responsible for translating the physical plan into a sequence of control instructions. These instructions are used to control the distributed execution of the algorithm among the participating machines, as can be seen in Figure 3-2. The machine running the interpreter is called the driver, which can be either a dedicated machine in the cluster or the developers own machine. According to the physical plan, a sequence of steps $S_{ij}, i \in \{1, \ldots, M\}, j \in \{1, 2\}$ is executed on each machine. Each step can either be computation, or a control flow

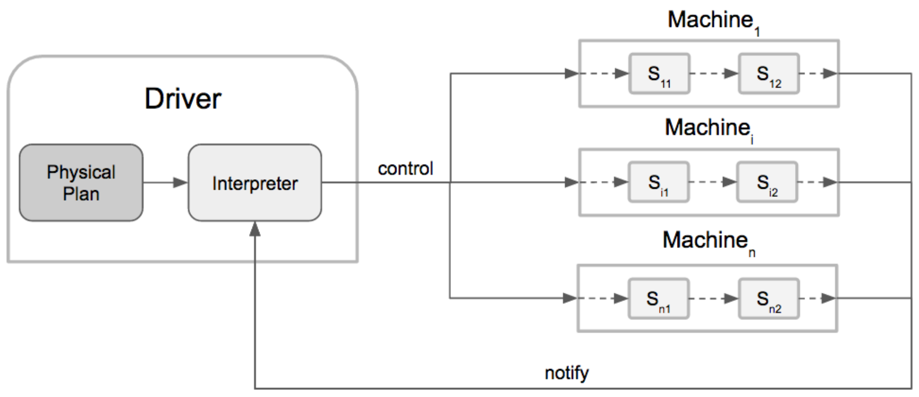

Figure 3-2: Driver for Algorithm Execution

operator such as a loop, sequence or parallel. The operators are higher order, meaning each operator possibly contains multiple operators as well.

## 3.1 Programming Model Primitives

In order to support the development process, a programming model is required that is expressive enough to conveniently model the complex dependencies when executing a machine learning algorithm in a distributed manner. For this purpose the framework provides a so called state centric programming model, which is motivated by means of the example algorithm definition in (4), describing a generic iterative-convergent algorithm as it would be implemented by a developer. The example is kept generic because the algorithm definition is very similar among members of the algorithm family. Only $f_p$ and $\Delta(\ldots)$ must be replaced by the corresponding preprocessing transformation respectively the optimization technique used to iteratively approximating the optimal solution according to Section 2.1.2. Figure 3-3 depicts the definition in (4) from a control-flow perspective, describing the algorithm in terms of a sequence of nested operators such as computation/transformation, loops and branches. The algorithm starts with loading the input data $D$ from an arbitrary storage, followed by one ore more preprocessing steps (e.g. normalization or standardization as well as splitting the data into training and test set **(A)**). A square represents a step of the algorithm, which contains an arbitrary number of input states (e.g. data, model) and some kind

**Algorithm 4** Generic Iterative-Convergent Algorithm

```
State: Data tensor $D \in \mathbb{R}^{n \times d}$, weight tensor $w \in \mathbb{R}^{m \times d}$
Input: algorithm specific hyper-parameters $\theta$, if any
Initialize: $t \leftarrow 0$, $w^{(0)} \leftarrow 0$
1: $D \leftarrow f_p(D)$                                          ▷ (A)
2: repeat                                                          ▷ (B)
3:     $t \leftarrow t + 1$
4:     $w^{(t)} \leftarrow w^{(t-1)} + \Delta(w^{(t-1)}, \theta, D)$     ▷ (C)
5: until termination criteria satisfied                            ▷ (D)
```

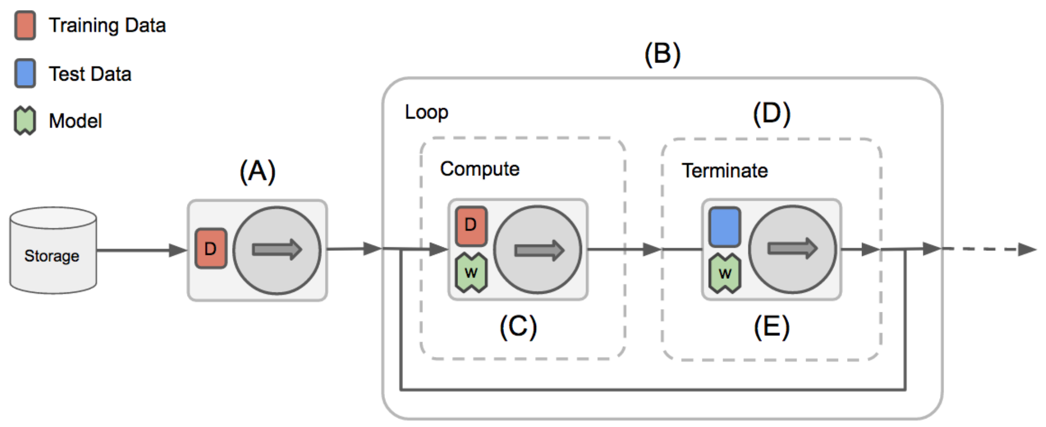

Figure 3-3: Control-Flow, Iterative-Convergent Algorithm

of transformation or computation applied to them. In the programming model, this step is called a unit and is described in more detail in Section 3.1.2. A transformation is depicted as a circle and can either be applied once (arrow) or multiple times (cyclic arrows) to the state(s) during this particular step. After applying the preprocessing the actual training process is triggered, which is contained in a loop **(B)**. A loop denotes the containing steps are executed repeatedly until some termination criterion **(D)** is satisfied, which is computed in **(E)**. For most machine learning algorithms the termination criterion can either be a fixed number of iterations, the change in objective $Q$ between iterations or the generalization performance. **(C)** is the actual training step which iteratively refines the model by updates computed from the input data according to 2.2. It can already be seen from the example that the framework must be capable of executing a complex sequence of arbitrary control flow operators and transformation steps in parallel on multiple machines. The programming model therefore introduces primitives to define the control flow as well as units and state. An algorithm expressed in this form could be executed as is on a single machine without modification because its sequential, non-parallel execution ensures the consistency of all involved states throughout the algorithm execution. Consistency in this context means that, because of its sequential execution, no conflicts can occur when altering a particular state as described in Section 2.3.3. Distributing said algorithm therefore requires additional instructions on how to ensure that conflicts can be resolved properly. These instructions are then combined with the logical representation of the algorithm and additional information regarding the distribution of state and the cluster environment to obtain a physical representation. This is the responsibility of the algorithm parallelization pipeline, described in Section 3.2, which takes the algorithm definition as input and returns a physical plan. The next sections introduce these key elements of the programming model.

### 3.1.1 State

Assuming the algorithm described in (4) should be parallelized with a degree of parallelism of two. The parallelization makes it necessary to replicate each transformation step of the sequential algorithm on $K = 2$ machines, as shown in Figure 3-4. It is

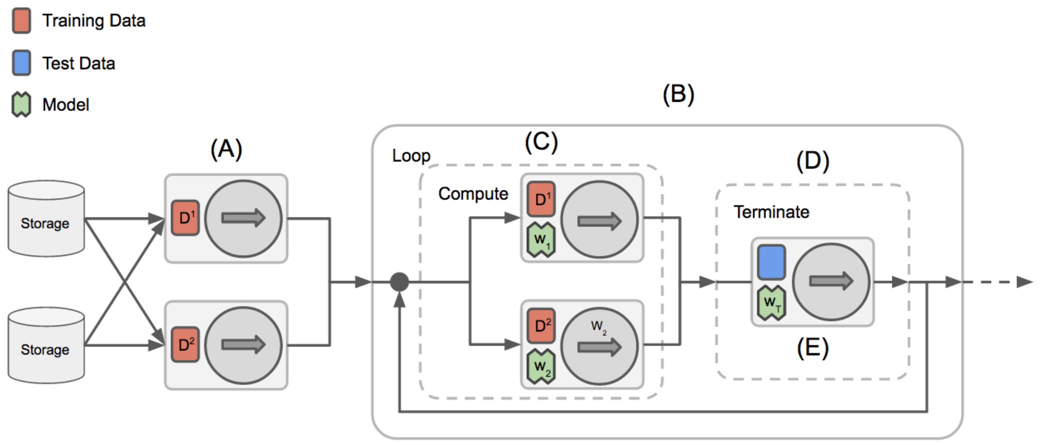

Figure 3-4: Distributed Control-Flow, Iterative-Convergent Algorithm

worth noting that the transformation logic stays the same independently of the degree of parallelism. Including a degree of parallelism of one, which is essentially a single machine. Therefore the framework instead of distributing the transformation focuses on how to distribute the state and keeping it consistent throughout the execution of each transformation step. Hence the programming model introduces the concept of state, which is essentially a container for an arbitrary, distributable data structure. A state $\gamma$ must be partitionable, meaning it can be distributed according to a partitioning $\{P_\gamma\}_{k=1}^{K}$, where $K$ is the degree of parallelism. In the area of machine learning, state is commonly represented by tensors of arbitrary type, such as floating point numbers, integers or strings. Therefore any further discussion assumes that a state is represented by a tensor of order two (matrix). For example the algorithm in (4) requires two states, namely the input data $D$ and the model $w$. In order to parallelize the algorithm, the parallelization pipeline requires a partitioning $\{P_\gamma\}_{k=1}^{K}$ for both states, which in the context of machine learning can be divided into the following cases, shown in Table 3.1.1. Where $S \subset \{1, \dots, M\}$ and $T \subset \{1, \dots, M\}$,

| $\gamma_w \setminus \gamma_D$ | partitioned | replicated |
|---|---|---|
| partitioned | $\gamma_w^S \wedge \gamma_D^T$ | $\gamma_w^S \wedge \gamma_{D_T}$ |
| replicated | $\gamma_{w_S} \wedge \gamma_D^T$ | $\times$ |

Table 3.1: State Partitioning Schemes

with $M$ being the number of available machines in the cluster and $\mid S \mid = \mid T \mid = K$. In this context, a subscript set of indices denotes the state is replicated among the set of machines, whereas a superscript set of indices indicates the state is partitioned among the set of machines. As described in Section 2.3.1, $w_S \wedge D^T$ equals data-parallelism, $w^S \wedge D_T$ equals model-parallelism and $w^S \wedge D^T$ is a hybrid approach that is often used in a parameter server setup where model and input data are both distributed among a set of machines. The example in Figure 3-4 therefore depicts a data-parallel approach because the input data $D$ is partitioned into $D^{\{1,2\}}$, whereas the model $w$ is replicated across machines $w_{\{1,2\}}$. In general the distribution of state in machine learning depends on the size of the problem and the algorithm employed to obtain an optimal solution for the given objective. E.g. in cases where the model $w$ does not fit into the memory of a single machine it is partitioned among a sufficient number of machines. The same holds for the size of the input data $D$. A special case is the partitioning $\{P_D\}_{k=1}^{K}$ of the input data, which is in general a matrix with rows consisting of examples. For iterative-convergent algorithms, depending on the optimization technique used to iteratively optimize the objective $Q$ of interest, two partitioning schemes are commonly used. If the optimization technique requires access to a complete example in order to update the model, such as it is the case with stochastic gradient descent, the input data is partitioned row-wise. On the other hand, if a coordinate-wise optimization technique is used, which only needs access to a single feature, the input data is distributed column-wise as shown in Figure 3-5 for a dop of two.

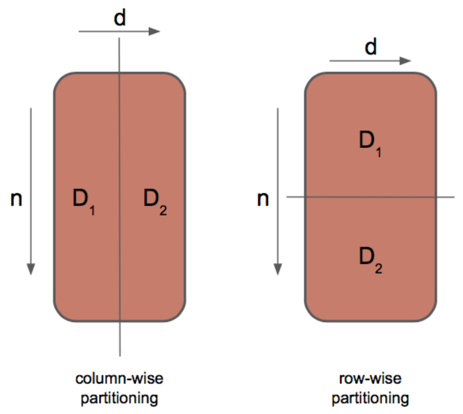

Figure 3-5: State Partitioning in Distributed Machine Learning

**Definition 1.** *A state is denoted by $\gamma$ and resembles a distributable data structure with a partitioning $\{P_\gamma\}_{k=1}^{K}$ and a degree of parallelism $K$. Each state has a placement $\{S_\gamma\}_{k=1}^{K}$ assigned to it, with $\{S_\gamma\}_k = i$ denoting partition $\{P_\gamma\}_k$ is assigned to machine $i$ and $i \in \{1, \dots, M\}$ and $M$ refers to the number of available machines. A state can be either replicated, which is denoted by $\gamma_{\{S_\gamma\}}$, or partitioned, which is denoted by $\gamma^{\{S_\gamma\}}$.*

### 3.1.2 Unit

As shown in Figure 3-4, distributing the algorithm not only requires the state to be distributed but also the transformation applied to the state must be replicated and executed in parallel across a set of machines. For this purpose the framework provides a primitive to define a unit of work. A unit $\Omega$ can be thought of as a function, taking an arbitrary number of states as input and either transforming the input state(s) or updating a state based on an update computed from the input. More specifically a unit defines an atomic step of an algorithm, described by the state centric programming model. This is necessary because each unit spans a scope for the consistency management and further instructions may be provided in order to ensure a consistent, efficient, distributed execution of the work defined in this particular unit. For this to work, the transformation or computation defined in a unit must satisfy an independence property regarding the input states, meaning that the work can be applied to a partition of the input state(s) without changes. Figure 3-6 depicts the general layout of a unit. The connectors describe an interface for receiving a

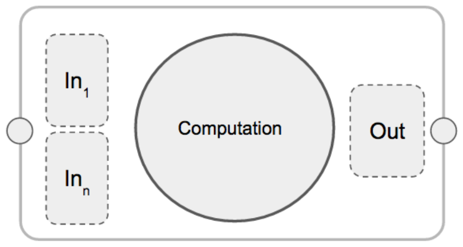

Figure 3-6: Unit Architecture

control flow signal, triggering the execution of the computation and emitting a control flow signal after the work is done. This is used by the interpreter to schedule and execute units according to the control-flow described in the physical plan. Figure 3-7 shows two units chained together, where the first unit triggers the execution of the second unit, which in turn notifies the interpreter after the work is done. This closely resembles the control flow known from imperative programming languages, the difference is that this control flow can be distributed with an arbitrary degree of parallelism on a set of machines in a cluster.

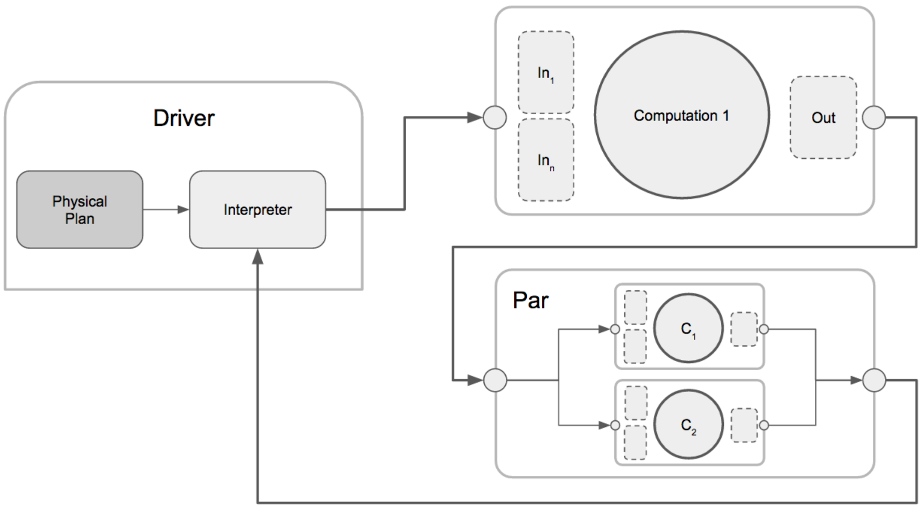

Figure 3-7: Driver controlling Units

**Definition 2.** *A unit is denoted by* $\Omega$ *and resembles a distributable computation step. Each unit has a placement* $\{S_\Omega\}_{k=1}^{K}$ *assigned to it, with* $\{S_\Omega\}_k = i$ *denoting unit* $\Omega_k$ *is assigned to machine* $i$ *and* $i \in \{1, \dots, M\}$ *and* $M$ *refers to the number of available machines. In addition, each unit* $\Omega_k$ *has a state assignment* $\{\gamma_{\Omega_k}\} \subset \{\gamma_1, \dots, \gamma_N\}$ *specifying the states that are either the input or result of the transformation applied inside the unit.*

### 3.1.3 Synchronization

As mentioned before, when distributing an algorithm, each unit scope has different requirements regarding consistency. As shown in Figure 3-4, the input state $D$ is partitioned into $K = 2$ parts, which means that also the units must be replicated $K$ times. First, in order to ensure a correct parallel execution, the consistency management must ensure that all parallel instances of a unit are properly synchronized according to the schemes discussed in Section 2.3.3. This synchronization happens on the control flow level and instructions on how to synchronize a unit or a control-flow operator such as a loop must be provided by the developer. In case of the example, the preprocessing step (A) must be finished before the next step (B) can be triggered. Therefore the parallel instances of unit (A) must be BSP, which is the default in case no further instructions are provided for unit synchronization. Things are different for step (B), which is the loop containing the training step. In order to improve performance, this loop is often executed with SSP or TAP synchronization, meaning that all instances of unit (C) can be executed in parallel without the need to synchronize on the outer loop. This increases the data throughput and can help to improve the overall performance. Secondly, as the parallel instances of a unit span a consistency scope, as can be seen in Figure 3-8, consistency management must also ensure consistency on a state level, which is discussed in the next chapter.

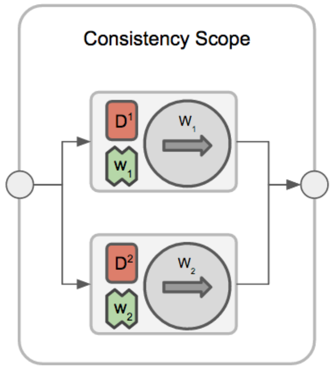

Figure 3-8: Consistency Scope of Unit Replicas

### 3.1.4 Framework in Action

This section combines the primitives introduced in the previous sections with the example machine learning algorithm definition in (4), in order to show how it can be utilized by a developer to distribute said algorithm. In addition to the original definition, a partitioning is required for each of the states $D$ and $w$, serving as instructions for the framework on how to distribute the state. Furthermore the unit scopes $\Omega_1, \Omega_2$ must be defined within the algorithm and additionally the synchronization scheme must be specified in order for the consistency management to work properly.

**Algorithm 5** State-Centric Iterative-Convergent Algorithm Definition

```
State: Data tensor $D \in \mathbb{R}^{n \times d}$, weight tensor $w \in \mathbb{R}^{m \times d}$
Partitioning: $\{P_D\}_{k=1}^{K}$, $\{P_w\}_{k=1}^{K}$
Input: algorithm specific hyper-parameters $\theta$, if any
Initialize: $t \leftarrow 0$, $w^{(0)} \leftarrow 0$
1: with BSP
2:     $\Omega_1[D \leftarrow f_p(D)]$
3: repeat
4:     with SSP
5:         $\Omega_2[t \leftarrow t + 1]$
6:         $\Omega_2[w^{(t)} \leftarrow w^{(t-1)} + \Delta(w^{(t-1)}, \theta, D)]$
7: until termination criteria satisfied
```

## 3.2 Algorithm Parallelization Pipeline

The algorithm parallelization pipeline is a core part of the framework and provides the functionality to translate an algorithm defined with the state centric programming model (5) into a representation suited for distributed execution by the driver. The sequence of steps to go from an algorithm definition to a distributed or physical plan is depicted in Figure 3-9. Where a dark arrow resembles an intermediate representation and a light arrow depicts a transformation step between those representations. At first, a logical plan is derived from the algorithm definition by the compiler, which is then passed on to the parallelizer. The logical plan is then combined with information about the cluster, such as the available resources on each machine and further instructions for the consistency management, to obtain a physical plan that is used by the driver to actually schedule the distributed execution of the algorithm in the cluster. The following sections describe each step and the resulting intermediate

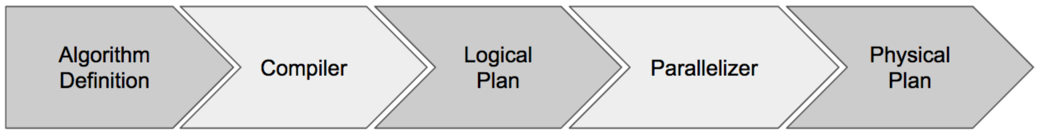

Figure 3-9: Algorithm Parallelization Pipeline

representation in more detail.

### 3.2.1 Compiler

The compiler is the first step in the algorithm parallelization pipeline and it is responsible for building a logical representation, or logical plan, of the algorithm by analyzing the algorithm definition. Figure 3-10 shows the logical plan derived from the example definition in (5). At first a tree is built, resembling the control-flow

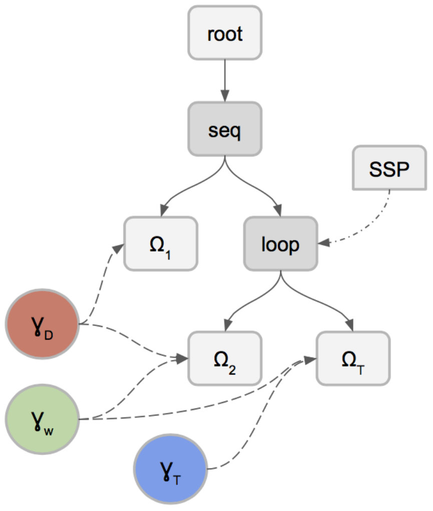

Figure 3-10: Logical Plan

of the algorithm, with intermediate nodes describing control-flow operators (sequential, parallel, loop) and the leafs consisting of units $\Omega_{ij}$ with $i \in \{1, \ldots, M\}$ and $j \in \{1, \ldots, K\}$, where $K$ is the degree of parallelism. Leafs always consist of units as these are the atomic parts of an algorithm and therefore can not be further divided. The next step connects all states $\gamma$ with the units $\Omega$ that need access to a particular state. This is then used to identify when a specific state must be created and when it can be safely disposed. For example $\gamma_D$ is required in unit $\Omega_1$ and $\Omega_2$, which is denoted by $\Omega(\gamma_D) = \{1, 2\}$. Finally the synchronization instructions for the consistency management are attached to the units and control-flow operators according to the definition. In case of the example, the loop operator should be executed using SSP. Control-flow operators and units without additional synchronization instructions default to BSP synchronization. It is worth noting that the logical plan is built using only the information derived from the algorithm definition. One can think of the logical plan as an abstract representation of the algorithm, which can be parallelized in different ways by providing further instructions such as the degree of parallelism or the architecture of the cluster. This information is then used by the parallelizer to built a specific architecture dependent physical plan.

### 3.2.2 Parallelizer

The second step in the parallelization pipeline is the parallelizer. It receives the logical plan of the compiler and combines it with additional information required to derive an executable, physical plan from it. Figure 3-11 shows the physical plan corresponding to the logical plan in Figure 3-10. As in the previous example, a dop

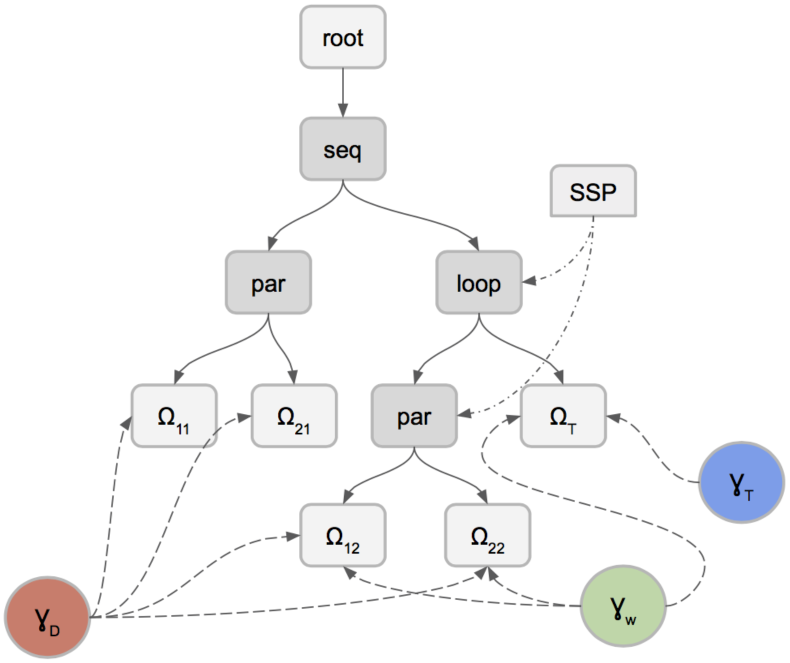

Figure 3-11: Physical Plan

of two is used and the parallelizer transforms the logical plan into a physical plan with distributed control-flow for the interpreter. The parallelizer first uses the dop $k$ to distribute each state according to the configured partitioning $\{P_\gamma\}_{k=1}^{K}$, which also implies that the corresponding units must be replicated $k$ times. Each unit $\Omega_j$ is translated into a parallel control-flow operator containing the replicas $\Omega_{ij}$ for $i \in \{1, \ldots, k\}$. Furthermore a machine is assigned to each unit so it can be scheduled accordingly by the interpreter. The parallelizer also must take care of the locality of state and schedule the units accordingly.

# Chapter 4

# Consistency Management

Consistency management in the context of distributed machine learning describes the process of ensuring an algorithm is parallelized most efficiently without loosing its correctness. Iterative-convergent algorithms are mostly sequential in nature and parallelization is commonly achieved by exploiting their inherent stochastic properties [24]. Staying with the example of the previous section, as shown in Figure 3-3, unit $\Omega_2$ of training step (C) is parallelized with a dop of $K = 2$ by partitioning the input data $D$ and replicating the model $w$ across all parallel units $\{\Omega_{2j}\}_{j=0}^{K}$. Proper consistency management targets two domains of the algorithm execution process. First, as discussed in Section 3.1.3, the control-flow must be synchronized properly according to the transformation applied during the execution of a unit $\Omega_i$ and its replicas $\{\Omega_{ij}\}_{j=0}^{K}$. In the example, the loop in (B) and its containing units $\Omega_{2j}$ are defined as SSP in order to increase the data throughput on the input data $D$. Ultimately this is supposed to result in a better overall performance due to a lower dependency on other units. Unfortunately, increasing the data throughput alone does not necessarily lead to a better overall performance, as each unit $\Omega_{2j}$ now works independently on a replica $w_j$ of the model without sharing the local progress. As discussed in Section 2.3.3, this staleness of state has a negative effect on the algorithm throughput and can be mitigated by frequently exchanging the updates applied to a state, in this case $w$, between all units $\Omega_{2j}$. Exchanging updates on the other hand has a negative effect on the data throughput as more computation time must be spent on network management. Adaptive consistency management therefore is responsible for controlling the best trade-off between communication and computation, based on the requirements for synchronization and consistency at each step of the algorithm as well as continuous feedback about cluster metrics and algorithm progress.

## 4.1 Adaptive Consistency Management

Consistency management in general can be seen as a cooperative task between the driver and the machines involved in executing the workload defined by the algorithm definition. Figure 4-1 depicts the architecture and connections between driver and machines, where the interpreter is responsible for instructing the machines according to the given program flow and also taking care of managing unit and state life cycles. The first part of the consistency management resides on the driver and is responsible

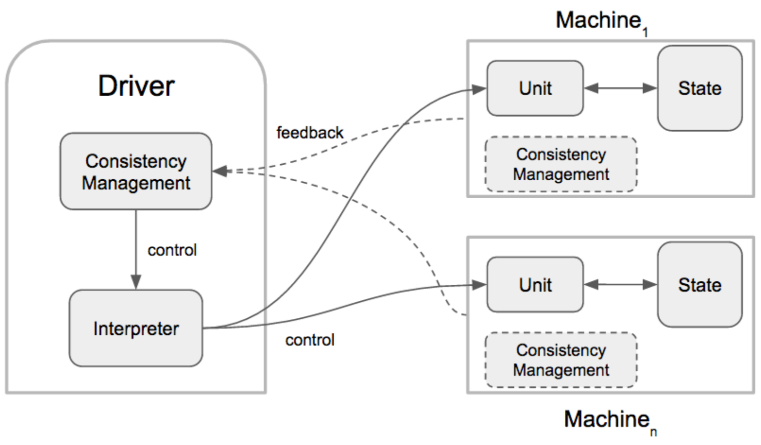

Figure 4-1: Adaptive Consistency Management Architecture

for collecting feedback from the machines. This is then used to exercise further control over the algorithm execution. Feedback can be metrics about the machines, such as resource usage, network bandwidth consumption or algorithm specific properties such as the convergence rate. Based on the collected feedback, the consistency management is able to provide instructions for the interpreter in order to influence the consistency on a unit level such as the synchronization between parallel instances of a unit in order to improve the algorithm throughput. Furthermore, the second part of the consistency management resides on each machine. This is necessary to gain control over the state level consistency as discussed in the previous section. Consistency management on a state level becomes necessary when a unit is parallelized by creating replicas on multiple machines. Parallelizing a unit results in replication or partitioning of the attached states and therefore it is necessary to control the communication frequency of model or parameter updates based on the available network bandwidth or some network quota. Also it is in general possible to control algorithm specific parameters such as the learning rate based on the progress of the other parallel units. This generic architecture allows to cover a variety of different use-cases in consistency management. While unit level consistency can be achieved solely by the interpreter via control-flow, state-level consistency requires more effort and coordination by the consistency management. In algorithm steps where a unit is parallelized, state level consistency management depends mainly on the partitioning of the attached states as well as the access pattern. As depicted in Figure 4-2, read-only states such as

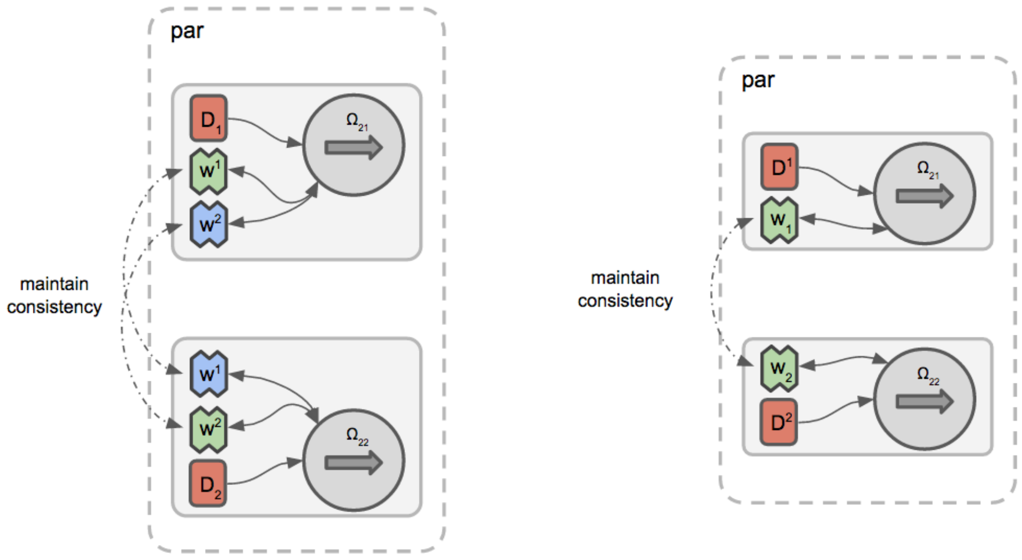

Figure 4-2: Partitioned and Replicated State

the input data $D$ are not subject to consistency management, whereas states that are altered by more than one unit in parallel must be kept consistent. This stems from the fact that during the execution of a unit in parallel a non read-only state is in general subject to local and remote updates. Depending on the partitioning the strategy to keep a state consistent varies slightly. In case a state is partitioned across machines, there exists a main partition for each part of the state, containing the source of truth. All units working in parallel on this part of the state, communicate the updates to the machine holding the main partition and also fetch the latest updates from this partition. On the other hand, if a state is replicated there is no main partition holding the latest version of the state. In this case each unit either has to broadcast all updates to all other parallel units or the units have to elect a unit that acts as the source of truth. E.g. in case of the example depicted in Figure 3-4, unit $\Omega_{2i}$ updates its model $w_i$ locally based on some update procedure $\Delta$, according to (2.8), but also receives the models $w_j, j \neq i$ of units $\Omega_{2j}, j \neq i$ as it is necessary to incorporate the progress of parallel instances in order to increase the algorithm throughput and mitigate slower convergence due to stale models. The frequency at which the progress communication takes place is controlled by the consistency management based on the collected feedback. As the second part of the consistency management must be present on each machine, the question arises which part on these machines is best suited to be responsible for controlling the consistency on a state level.

## 4.2 State-centric vs. Computation-centric

From an architectural perspective, two solutions can be considered for integrating the consistency management into each machine. First, with state-centric consistency, as depicted in Figure 4-3, the consistency management is part of the state itself. The consistency management and actual state are hidden behind a proxy, which mimics the interface of the state. Each update is forwarded to the consistency management, which takes care of updating the local state as well as communicating updates to all other replicas or partitions depending on the partitioning. The consistency management related parameters can be controlled by the driver. This has the advantage that it offers a very convenient way to interact with a state because from a unit perspective a state behaves similar to a non-parallel version. Furthermore, the design of a unit is considerably simplified as the computation is just a function that must be executed

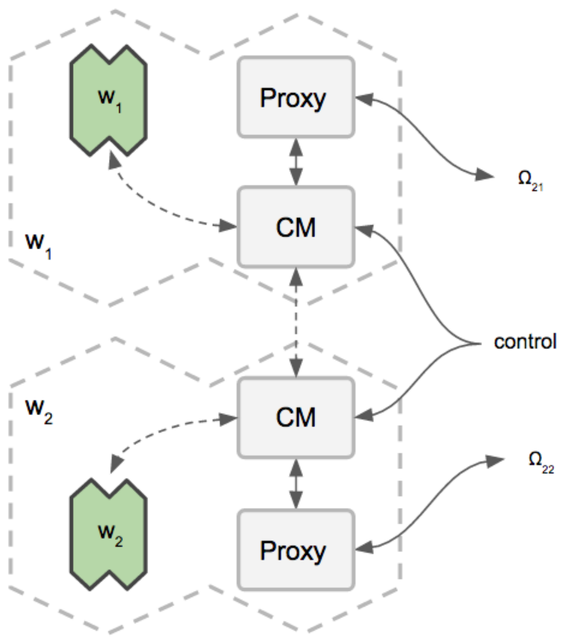

Figure 4-3: State-Centric Consistency Management

in parallel on multiple units and therefore very lightweight. On the other hand, this approach requires a deep integration of consistency management logic into the state and therefore induces a tight coupling between logic and data. This unnecessarily increases the dependency between data and framework and requires the developer to reimplement the logic in case the type of state changes.

Second, with computation-centric consistency, as can be seen in Figure 4-4, the consistency management is part of the unit. No coupling between state and framework exists, which makes it easier for a developer to reuse existing code. Furthermore it provides a better abstraction due to the separation of logic and data and also a unit is already a part of the control-flow, responsible for executing the computational part. Lastly, no proxy is required, which considerable simplifies the process of adding new types of state to the framework or reuse existing implementations such as matrix libraries. In this architecture, the consistency management is periodically invoked to communicate with the driver in order to fetch new instructions and exchange progress with parallel units.

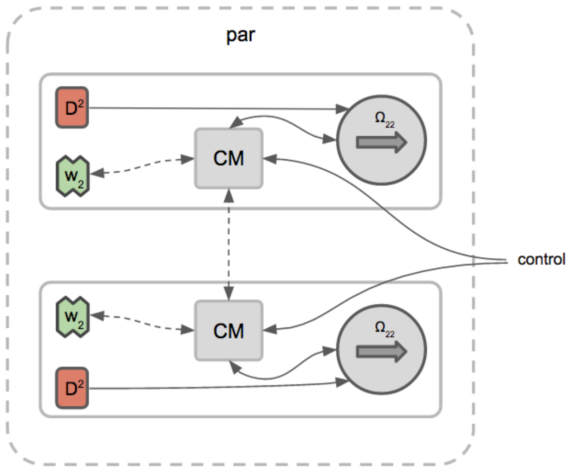

Figure 4-4: Computation-Centric Consistency Management

## 4.3 Unit-internal Control-flow

Considering both approaches, computation-centric consistency seems the best fit for the framework. Therefore, in order to achieve the greatest flexibility and fine grained control over the consistency during algorithm execution, the consistency management is part of the unit itself. Figure 4-5 depicts the unit internal control-flow that is executed when the unit execution is triggered. The control-flow contains the actual computation followed by a number of steps associated with consistency management. Previous to executing the computation enclosed in a unit, a cache is created for each state that is altered by said computation (A). The reasoning behind this strategy is the fact that updates to the state are best communicated as relative change compared to a previous state (e.g. previous iteration), whereas the actual algorithm requires the absolute change. In order to be able to compute a delta between two defined states, only the cache is altered and compared with the previous version before communicating updates. After this initialization step the actual computation $\Delta(\ldots)$ is executed according to (2.2), which results in a delta $\Delta w$ that can be used to update the state $w$. Instead of applying the delta to actual state $w$, it is applied to the cache $w_{cache}$ only (C). When running iterative-convergent algorithms this procedure of updating and caching is repeated multiple times according to the algorithm definition. In order for the consistency management to gain a fine grained control over the update

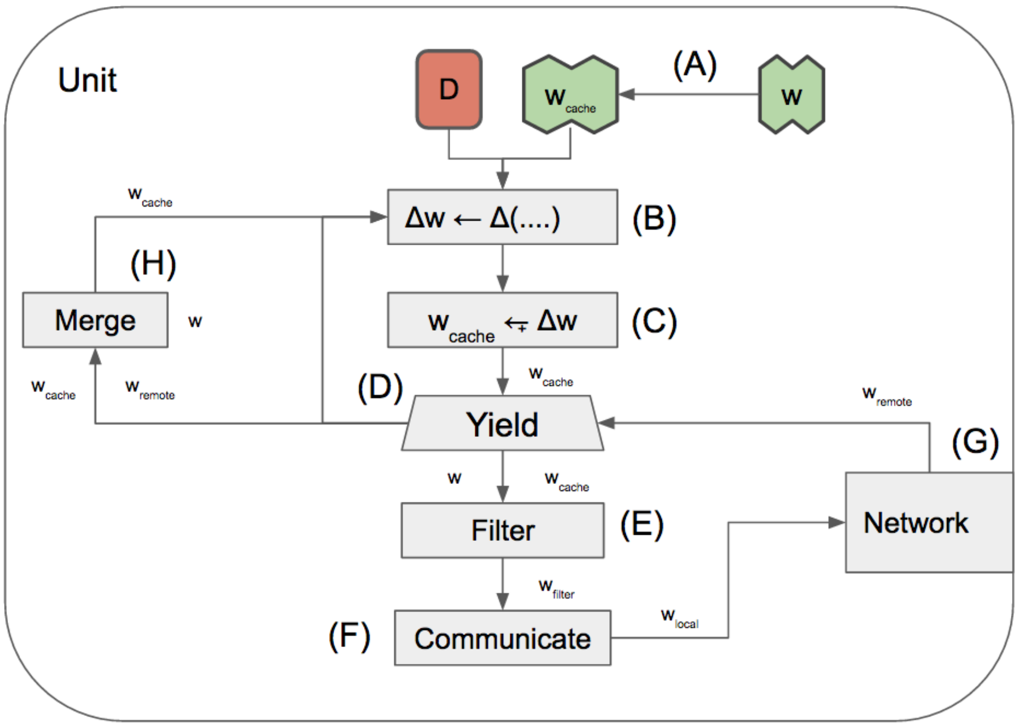

Figure 4-5: Unit-Internal Control-flow

communication frequency during the execution, step (D) in the control-flow is a yield component similar to the primitive used in coroutines. As with coroutines, the yield suspends the execution of the (update) computation in favor of executing other tasks. In case of a unit with consistency management, the yield is triggered on request of the consistency management and at a frequency that is agreed on with the other replicas of the unit. Upon a yield, the local progress is communicated to the other units $\Omega_{ij}$ in order to ensure a consistent state. For this to happen, the actual state $w$ and the cached version $w_{cache}$ are used to compute $\Delta w = w_{cache} - w$, which is then forwarded to the filter stage (E) if configured. The filter stage provides the developer with the ability to control which updates are actually forwarded to the communication stage (F) based on a defined criterion. This is necessary for example when there is a bandwidth quota for each machine/unit and only a subset of the updates can be communicated. All updates passing the filter stage are then forwarded to the communication stage and sent to the other units over the network (G). Furthermore, if remote updates $w_{remote}$ have been received over the network, these are merged into the actual state $w$ together with the locally cached updates $w_{cache}$ (H). A summary of merge strategies together with filter strategies is described in the following section.

## 4.4 Strategies

When executing an algorithm in parallel on a cluster of machines, often additional constraints are induced by the architecture. For example, multiple independent tasks could be executed on the same machine and therefore a quota on resources like network bandwidth is enforced on a process level. In such cases the consistency management needs additional means to identify the updates that are most important to the algorithm progress. Filtering or prioritizing updates allows the consistency management to identify the most important updates that need to be communicated first without exceeding the network quota.

### 4.4.1 Filter

The simplest strategy for prioritizing updates is randomly choosing updated parameters of the state for communication until the network quota is exceeded. Round-robin prioritization on the other hand repeatedly selects parameters following a fixed schedule. A more data-driven approach takes the absolute magnitude of the change

$$\mid \Delta w \mid = \mid w - w_{cache} \mid \tag{4.1}$$

into account when deciding which parameters to communicate first. Depending on the algorithm, the relative change in magnitude

$$\mid \frac{\Delta w}{a} \mid = \mid \frac{w - w_{cache}}{a} \mid \tag{4.2}$$

might be a better prioritization strategy, where $a$ is the current value of the parameter.

### 4.4.2 Merge

In addition to prioritization, the merging strategy must be specified as well. Merging refers to the process of combining local updates $w_{local}$, remote updates $w_{remote}$ and the current state $w$ to form an updated version of this particular state. Merging in general can be described by

$$w_i^t = w_i^{t-1} + \gamma \sum_{k=0}^{K} \Delta w_k \tag{4.3}$$

where $w_i^t$ is the updated state at time $t$, computed by updating the previous version of the state with a weighted sum of the remote states updates $\Delta w_k, k \neq i$ and the local updates $\Delta w_k, k = i$. Depending on the partitioning of the state and the algorithm employed, the parameter $\gamma$ is commonly chosen to be either $\frac{1}{K}$ (averaging) or 1 (adding).

# Chapter 5

# Experiments

In order to evaluate the current performance of the framework, a series of experiments has been conducted. The experiments are divided into two parts, where the first part focuses on the assessment of the overall algorithm performance when executing machine learning algorithms in a distributed fashion. For comparison Apache Spark is used as the system is currently the state-of-the-art in distributed machine learning as well as an non-parallel baseline implementation. Besides the evaluation of the current status of the framework another motivation behind this set of experiments is to prove the usefulness of the programming model. The second set of experiments tries to quantify how different facets of consistency management affect the algorithm performance. Therefore the impact of communication frequency, synchronization strategy as well as various filter and merging strategies on the overall algorithm performance is evaluated.

## 5.1 Experiment Setup

As the algorithm for comparison, distributed lasso linear regression within the CoCoA framework is used as described in section 2.1.3. Based on the generic form of convex optimization problems described in (2.5), lasso linear regression is represented by

$$f_\alpha = \frac{1}{2} \parallel D\alpha - y \parallel_2^2 \tag{5.1}$$

and

$$r(\alpha) = \lambda \parallel \alpha \parallel_1 \tag{5.2}$$

where $\alpha$ is the weight vector, $D$ resembles the input data together with the output $y$ and $\lambda$ is the regularization constant. Simple randomized coordinate descent is used as the local solver within the CoCoA framework, as described in algorithm 2. As a measure for the overall algorithm performance the distance to the optimal primal solution is used. This optimal value is calculated by running all methods for a large number of iterations (until the algorithm is converged) and then selecting the smallest primal value amongst the results.

In order to show the algorithm performance from different perspectives, two datasets with different properties such as ratio between number of examples and feature size as well as sparsity are used, as can be seen in Table 5.1. In BSP the runtime is measured on a per iteration basis on the driver for Apache Spark as well as the framework. This includes scheduling of computation and execution. When running stale synchronous parallel (SSP) and total asynchronous parallel (TAP) the runtime is measured for each node and iteration independently and the overall runtime is set by the best performing node. For a better comparison between systems the data loading and setup stage is not accounted to the overall runtime.

| **Dataset** | **Training Size** | **Feature Size** | **Sparsity** |
|---|---|---|---|
| epsilon | 400 K | 2 K | 1.0 |
| url | 2.4 M | 3.2 M | 3.5e-5 |

Table 5.1: Datasets for Empirical Evaluation

The cluster itself consists of 8 machines with each machine equipped with 16 GB of RAM and 8 Intel Xeon E3-1230 V2 CPUs running at 3.30 GHz. All nodes are connected via 1 GBit Ethernet. Distributed experiments running Apache Spark and the framework use 8 machines if not stated otherwise. As a baseline, a single-core implementation of the algorithm is used.

## 5.2 Algorithm Performance

In order to asses the algorithm performance in comparison to Apache Spark and the baseline, lasso regression is run on the epsilon and url dataset with bulk synchronous parallel synchronization (BSP) (Section 3.1.3). The algorithm hyper-parameters $\gamma$ and $\lambda$ have been tuned for best performance and are set to 1.0 and 1e-5, respectively.

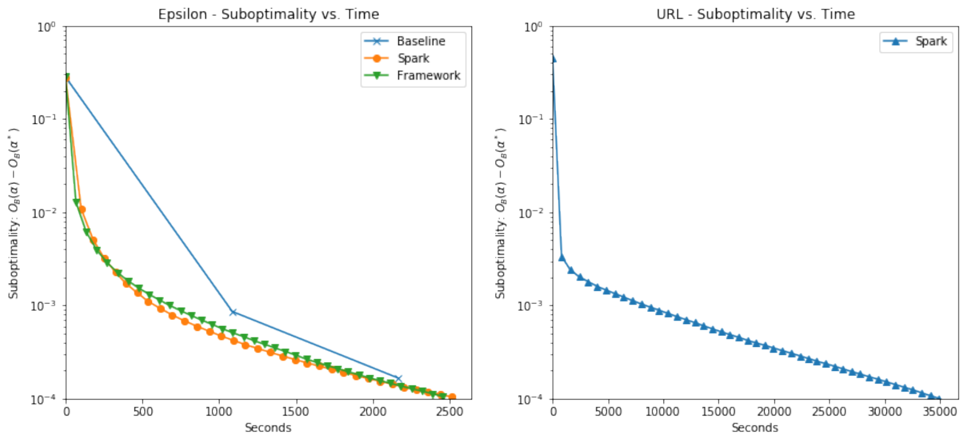

Figure 5-1: Overall Algorithm Performance Comparison

## 5.3 Consistency Management

### 5.3.1 Communication Frequency

### 5.3.2 Synchronization Strategy

### 5.3.3 Filtering Strategy

### 5.3.4 Merging Strategy

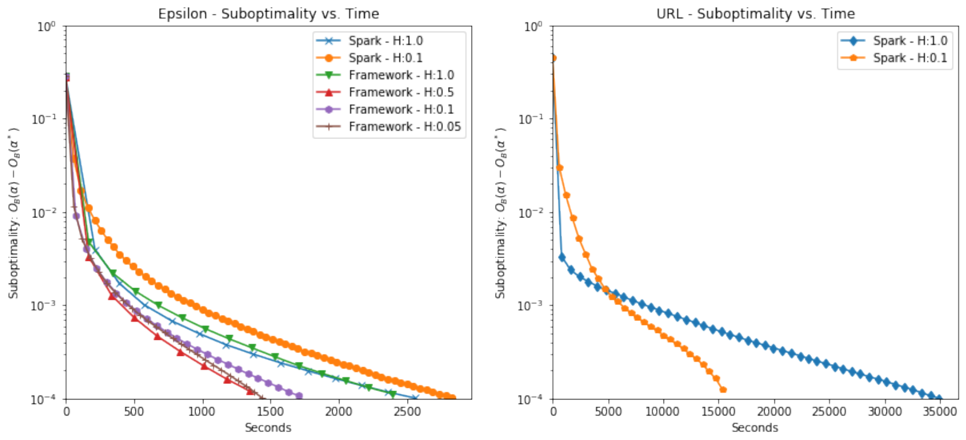

Figure 5-2: Communication Frequency Comparison

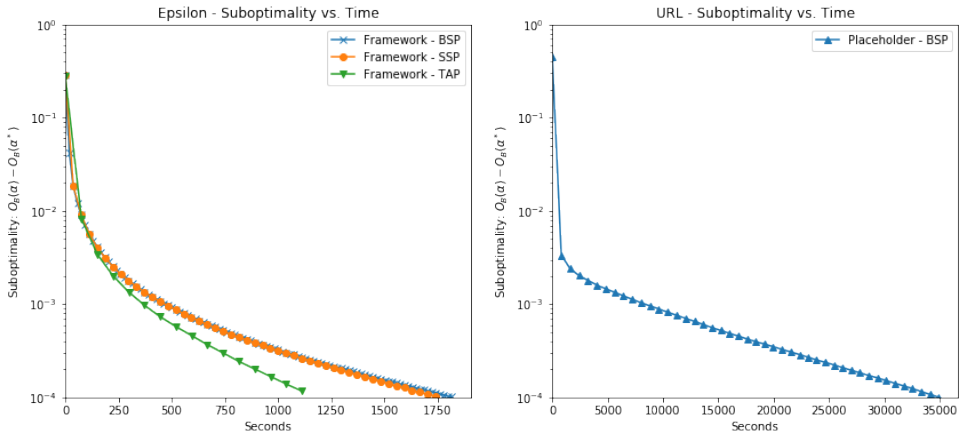

Figure 5-3: Synchronization Strategy Comparison

# Chapter 6

# Conclusions and Outlook

# Bibliography

[1] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pp. 137–149, 2004.

[2] "Apache hadoop." `https://hadoop.apache.org`.

[3] V. Kumar Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O 'malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," *SOCC '13 Proceedings of the 4th annual Symposium on Cloud Computing*, vol. 13, pp. 1–3, 2013.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, p. 10, 2010.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J. C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere platform for big data analytics," *VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin, "GraphLab: A Distributed Framework for Machine Learning in the Cloud," *The 38th International Conference on Very Large Data Bases*, vol. 5, no. 8, pp. 716–727, 2012.

[7] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, and X. Zheng, "Petuum : A New Platform for Distributed Machine Learning on Big Data," 2015.

[8] M. Li, D. G. Andersen, J. W. Park, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, B.-y. Su, M. Li, D. G. Andersen, J. W. Park, A. J. Smola, and A. Ahmed, "Scaling Distributed Machine Learning with the Parameter Server," *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[9] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "MALT : Distributed Data-Parallelism for Existing ML Applications," 2015.

[10] M. Jaggi, V. Smith, M. Tak??, J. Terhorst, S. Krishnan, T. Hofmann, and M. Jordan, "Communication-efficient distributed dual coordinate ascent," *Advances in Neural Information Processing Systems*, vol. 4, no. January, 2014.

[11] J. E. Dennis Jr and R. B. Schnabel, *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM, 1996.

[12] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, 2010.

[13] V. Smith, S. Forte, M. I. Jordan, and M. Jaggi, "L1-regularized distributed optimization: A communication-efficient primal-dual framework," *arXiv preprint arXiv:1512.04011*, 2015.

[14] V. Smith, S. Forte, C. Ma, M. Takac, M. I. Jordan, and M. Jaggi, "Cocoa: A general framework for communication-efficient distributed optimization," *arXiv preprint arXiv:1611.02189*, 2016.

[15] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, pp. 2595–2603, 2010.

[16] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones.," in *NSDI*, vol. 13, pp. 185–198, 2013.

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

[18] E. P. Xing, Q. Ho, P. Xie, and W. Dai, "Strategies and principles of distributed machine learning on big data," *arXiv preprint arXiv:1512.09295*, 2015.

[19] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameterserver consistency models," *arXiv preprint arXiv:1410.8043*, 2014.

[20] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pp. 381–394, ACM, 2015.

[21] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," *arXiv preprint arXiv:0911.0491*, 2009.

[22] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, pp. 1223–1231, 2013.

[23] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing, “Solving the straggler problem with bounded staleness,” 2013.

[24] T. Herb, T. Jungnickel, and C. Alt, “Weak consistency and stochastic environments: harmonization of replicated machine learning models,” in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, p. 8, ACM, 2016.