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 2017

Author .....
Department of Electrical Engineering and Computer Science
April 15, 2017

Certified by .....
Prof. Dr. Odej Kao
Associate Professor
Thesis Supervisor

Certified by .....
Prof. Dr. Volker Markl
Associate Professor

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 optimimal 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 choosen level of consistency can severely affect algorithm performance, resulting in a non optimal convergence rate and therefore increased runtime.

This thesis introduces a novel framework for distributed machine learning, which is based on a state centric programming model with yield semantics. The programming model allows the user to focus on the key elements of developing distributed machine learning algorithms, namely parameter communication, parameter merging and adaptive consistency management among distributed workers, while the system takes care of utilizing the cluster resources in an optimal fashion. The experiments using elastic-net regularized 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.

Thesis Supervisor: Prof. Dr. Odej Kao
Title: Associate Professor

Acknowledgments

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

Eidesstattliche ErklaÌĺrung Hiermit erklaÌĺre ich Eides statt, dass ich dir vorliegende Arbeit selbststaÌĺndig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen woÌĺrtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Unterschrift Ort, Datum

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 tradeoff 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 approachs 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 dataflow pipeline based architecture, whileas 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 (ETL) 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 that have computed an update locally and send it to these state keepers. Which act as surrogates, taking state updates and incorporating 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 dataflow 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 algoritms while at the same time provide the ability to conscisely 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 conscisely 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 fast and efficient.

1.3 Thesis Outline

In this thesis we introduce a novel framework for large scale distributed machine learning. It improves upon currently available systems by providing a powerful programming abstraction that can conscisely 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-convegent 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 tradeoff between algorithm execution and progress communication the performance is improved as well. All of this can be easily customized for quick prototyping and finetuning, 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. I start 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 in Chapter 2. Furthermore 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 I am going to provide a more formal description. I introduce the common algorithm parallelization strategies in distributed machine learning. I conclude the chapter 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 differ-

ent framework architecture and abstraction is necessary to improve the performance and expressability 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. I further describe 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 choosen 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 $Q(z, w) = \ell(f_w(x), y)$. Emperical risk $E_n(f) = \frac{1}{n}\sum_{i=0}^{n} \ell(f(x_i), y_i)$ performance on training set, expected risk generalization performance.

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

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 Δ being an arbitrary update function.

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

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 Δw is computed and applied to the previous weight vector until a stopping condition is satisfied. E.g. the distance to the optimal solution or the objective difference between two iterations is monitored. When 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 η is a chosen gain, often refered to as learning rate. While this achieves linear convergence under sufficient regularity assumptions and a sufficiently small learning rate η [11] [12] a more simplified version is commonly used in practice, called stochastic gradient descent (SGD). 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 expectiation 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 Algorithm 1.

Algorithm 1 Stochastic Gradient Descent

1: $k \leftarrow 1$ and initialize $w^0 \in \mathbb{R}^d$
2: **repeat**
3: **for do**
4: $w^t \leftarrow w^{(t-1)} - \eta_t \nabla_w Q(z_t, w^{(t-1)})$
5: **until** termination criteria satisfied

Coordinate descent on the other hand iteratively tries to optimize a given objective by successively performing approximate minimization along a coordinate direction while keeping the other directions fixed. **NOTES:** give a more in detail explanation and provide algorithm description

2.1.3 Regularization

$$Q(z, w) = \ell(f_w(x), y) + r(w) \tag{2.5}$$

Algorithm 2 Stochastic Coordinate Ascent

1: $k \leftarrow 1$ and initialize $w^0 \in \mathbb{R}^d$
2: **repeat**
3: **for do**
4:
5: **until** termination criteria satisfied

$$Q(z, w) = \ell(f_w(x), y) + r(w) \tag{2.6}$$

$$Q(z, w) = \ell(f_w(x), y) + r(w) \tag{2.7}$$

NOTES: - regularization in general - L1, L2 regularization - elastic-net as special case - convexity properties - mini-batch setup algorithm

2.1.4 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

$$\min_{\alpha} Q(z, \alpha) = \ell(f_\alpha(x), y) + r(\alpha) \tag{2.8}$$

in a distributed setting. Where α denotes the weight vector, ℓ is convex and smooth and r is assumed to be seperable, which in this context means $r(x) = \sum_{i=0}^{n} r_i(x_i)$. Commonly the term ℓ is an emperical 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 λ 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, *nwaylesserthand*, 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 2.8. This leads to separability of the problem over the coordinates of α 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

$$\min_{\alpha_{[k]} \in \mathbb{R}^n} \zeta_k^{\sigma'}(\Delta\alpha_{[k]}, v, \alpha_{[k]}), \tag{2.9}$$

where

$$\zeta_k^{\sigma'}(\Delta\alpha_{[k]}, v, \alpha_{[k]}) = \frac{1}{K} f(v) + w^T A_{[k]} \Delta\alpha_{[k]} + \frac{\sigma'}{2\tau} \|A_{[k]} \Delta\alpha_{[k]}\|^2 + \sum_{i \in P_k} g_i(\alpha_i + \Delta\alpha_{[k]_i}), \tag{2.10}$$

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 $\alpha_{[k]_i} = 0$. In summary the framework provides a procedure to effectively combine the results

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 σ'
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** $k \in \{1, \ldots, K\}$ **do**
4: obtain an approximate solution $\Delta\alpha_{[k]}$ for the local subproblem 2.9
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

from local computation without having to deal with conflicts resulting from similar updates computed on other machines.

NOTES: - local solver (batch coordinate descent) - elastic-net in the cocoa setup - algorithm parameters sigma and gamma

2.2 Dataflow Systems

NOTES: - description of dataflow in general - architecture of dataflow systems, Apache Spark and Flink - description of RDDs

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 parallelized.

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 parameters). As depicted in Figure 2-1, 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,

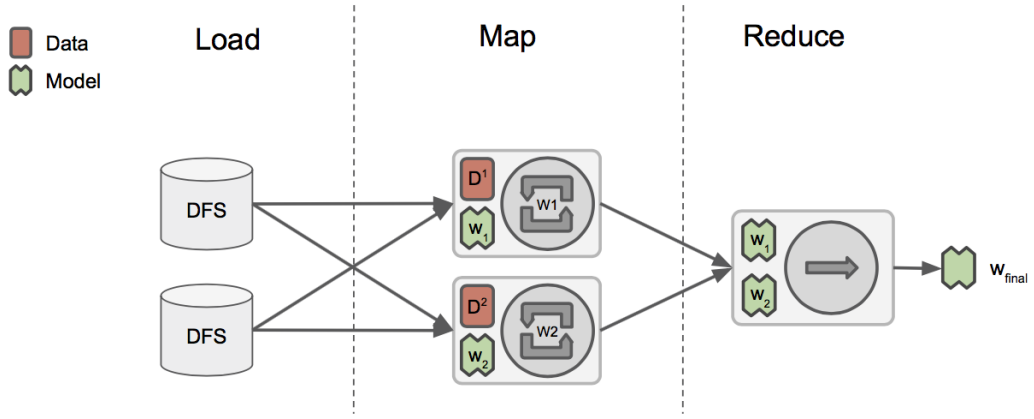

Figure 2-1: Data-parallelism via MapReduce

data-parallelism 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

$$w_k^t = w_k^{t-1} + \Delta(w_k^{t-1}, D^k) \tag{2.11}$$

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

$$w_{final} = \frac{1}{K} \sum_{k=1}^{K} w_k \tag{2.12}$$

Even though this approach works well in practice and gives considerable good results, it suffers from two limitations. First, if the size of a local model w_k exceeds the available memory on a single machine, the algorithm can 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 [16], 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 [17] 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]. In order to improve the performance, a system must be able to communicate more frequently and it must also be able to partition 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 2-2, 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 $\{w\}_{l=1}^{L}$ and making it accessible to the workers $\{W\}_{k=1}^{K}$, e.g. $W = \{W_1, W_2, W_3\}$, via a defined interface that is similar to a key-value store. The model is partitioned among the 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. 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

$$\Delta w_{k_i}^{t} = \Delta(w_{k_i}^{t-1}, D^k). \tag{2.13}$$

In the parameter server setup, the local state w acts as a cache for global parameters in order to reduce network usage. Therefore, depending on the caching policy, w_{k_i} can either be directly read either the local cache or must be retrieved from the parameter

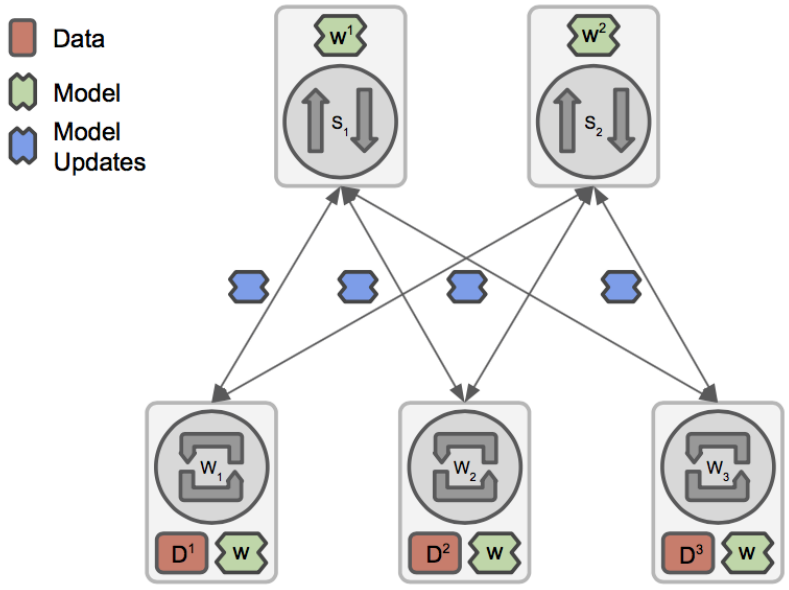

Figure 2-2: Parameter Server

server. Additionally to applying the update to the local model w, the difference $\Delta w_{k_i}^t$ is published to the parameter server as well, which takes care of applying it to the corresponding entry w_i to make the update 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. The procedure of retrieving, updating and publishing is 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. The procedure of updating the model in parallel is also known as model-parallelism.

While this schema has been proven to work well in practice (some numbers), a couple of issues still remain when running iterative-convergent algorithms at scale using the parameter server. According to [18] 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, net-

work communication is the bottleneck in distributed machine learning as well. Even though due to the stochastic nature of many machine learning algorithms, the communication can be reduce compared to the exact serial algorithm, it has been proven that fresher model parameters increase the algorithm throughput per iteration [19]. 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 is time consuming, 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 [20]. 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 among multiple machines concurrently accessing and updating some shared state. 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 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 ac-

cording to 2.14 and each worker retrieves the updated parameters before beginning the next iteration. This synchronization scheme guarantees consistency among all nodes at all times.

$$w^t = w^{t-1} + \frac{1}{K} \sum_{k=1}^{K} \Delta(w_k^{t-1}, Dk) \tag{2.14}$$

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 [19]. 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 [20] 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 2-3. Even though worker W_3 is a

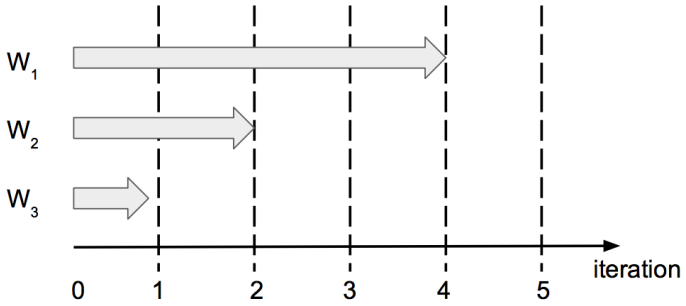

Figure 2-3: Straggler in TAP

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 [21]. This stems from the fact that no theoretical convergence bound can be established as the divergence in iteration between workers is unbound. A middle ground between bulk synchronous parallelization and total asynchronous

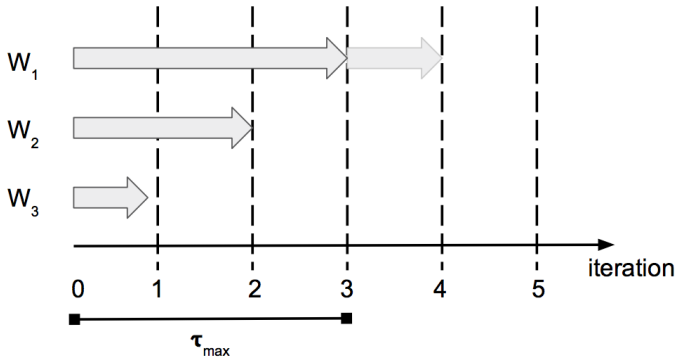

Figure 2-4: Straggler in SSP

parallelization is *stale synchronous parallel (SSP)* [22] or *bounded staleness (BS)*. As shown in Figure 2-3, SSP introduces a fixed maximum delay, or staleness threshold, of τ_{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 τ_{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

Framework

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. The 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 the parameter server paradigm provides 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 of distributed machine learning, namely an efficient parallel execution. This section introduces the general architecture of the framework and its main parts based on the example of iterative-convergent algorithms, though its application is not limited to this particular family of algorithms.

As depicted in Figure 3-1 the framework consists of three major parts, supporting the developer with the development and execution of distributed machine learning algorithms. First, it provides a collection of primitives that can used to describe the

[1] Application programming interface

algorithm with the help of a state centric programming model. The SCPM[2] 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 ensures a correct execution of each algorithm steps among all participating machines. For this purpose, additional primitives are offered by the framework to provide the consistency management with instructions on how to ensure a consistent distributed execution given a particular state and transformation. Secondly, the

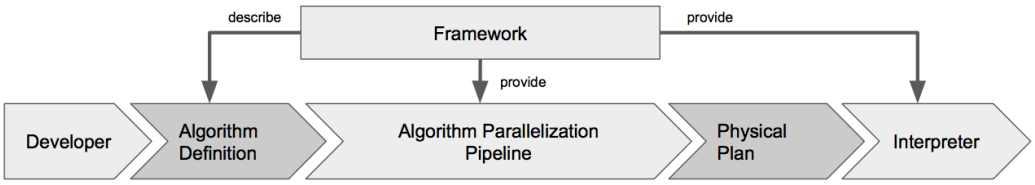

Figure 3-1: Framework Architecture

framework provides an algorithm parallelization pipeline which takes the algorithm definition as an input an creates a physical plan by applying a series of transformation or enrichment steps to it. The physical plan contains a detailed description on how to execute the algorithm in a distributed manner on a specific group of machines within the cluster. As the third part of the framework, the interpreter is responsible for translating the physical plan into control instructions for the machines part of the compute cluster, as can be seen in Figure 3-2. In this context, the driver is

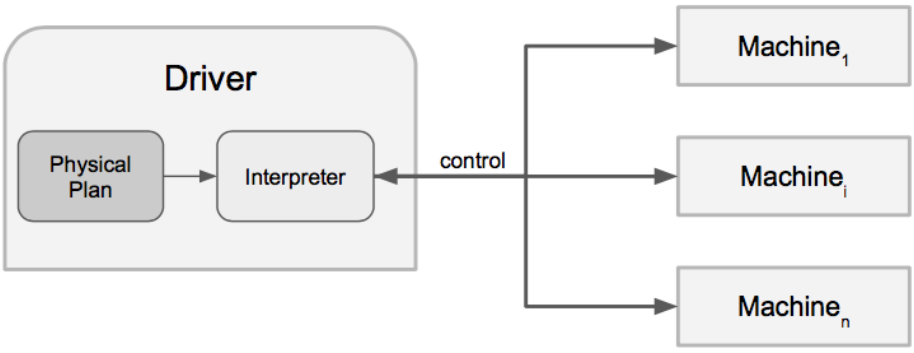

Figure 3-2: Algorithm Execution Architecture

[2]State-centric Programming Model

an arbitrary machine responsible for running the interpreter, which can be either a dedicated machine part of the cluster or the developers own machine.

3.1 State Centric Programming Model

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 derived by the help of the example algorithm definition in 4. The definition represents an iterative-convergent algorithm as it would be implemented by a developer. Due to the fact that the general algorithm definition is very similar among members of the ICA[3] family, only f_p and $\Delta(\ldots)$ must be replaced by the corresponding preprocessing transformation respectively the optimization technique used for iteratively approximating the optimal solution according to Section 2.1.2. Figure 3-3 shows the control flow of (4),

Algorithm 4 Generic iterative-convergent algorithm definition

State: Data tensor $D \in \mathbb{R}^{n \times d}$, weight tensor $w \in \mathbb{R}^{m \times d}$
Input: algorithm specific hyper-parameters θ, 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)

describing the training process of an ICA such as linear regression, support-vector machine or logistic regression. The process starts with loading the input data D from the 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 of transformation applied to them. The transformation process is depicted as a circle and can either be applied once

[3] iterative-convergent-algorithm

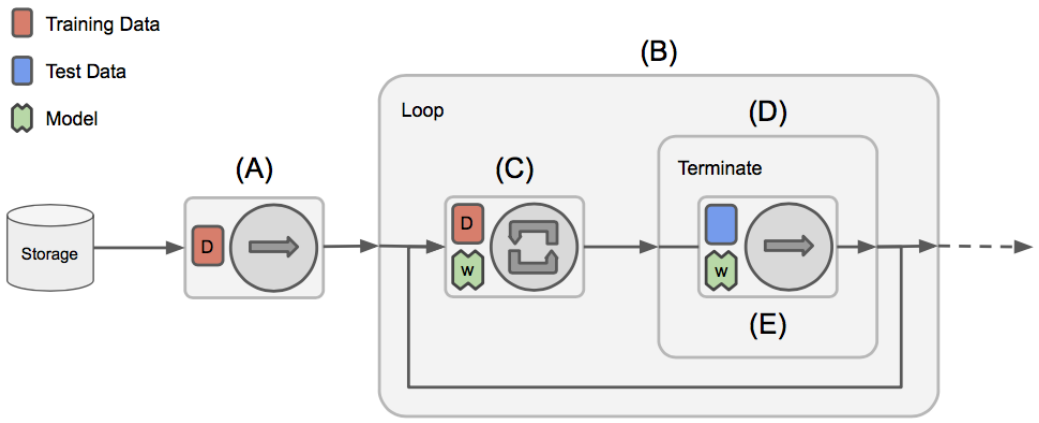

Figure 3-3: Logical plan for iterative-convergent algorithms

(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 symbolizes that 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. 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 the key elements of the programming model discussed by the help of the example algorithm.

3.1.1 State

Assuming the algorithm described in (4) should be parallelized with a dop[4] of two. In order to achieve this degree of parallelism, it is necessary to replicate each transformation step of the sequential algorithm on multiple machines, as shown in Figure 3-4. It is worth noting that the transformation logic stays the same independently

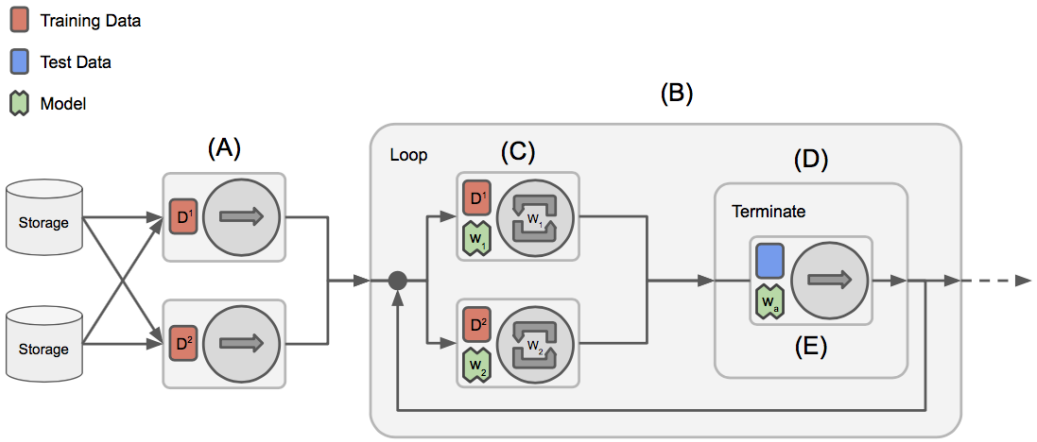

Figure 3-4: Distributed Machine Learning Pipeline for Iterative-convergent Algorithms

of the degree-of-parallelism, including a dop of one, which is essentially a single machine. Therefore the framework instead of parallelizing the transformation focuses on how to parallelize the state and keeping it consistent. Hence the programming model introduces the concept of state, which is essentially a container for an arbitrary, distributable data structure. A state γ 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

[4]degree of parallelism

and the model w. In order to parallelize the algorithm, the parallelization pipeline requires a partitioning $\{P\}_{k=1}^{K}$ for both states, which in machine learning can be divided into the following cases, shown in Table 3.1.1. Where $S \subset \{1, \ldots, M\}$ and

$\gamma_w \setminus \gamma_D$	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: Distributed Machine Learning Pipeline for Iterative-convergent Algorithms

$T \subset \{1, \ldots, M\}$, 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 means 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.

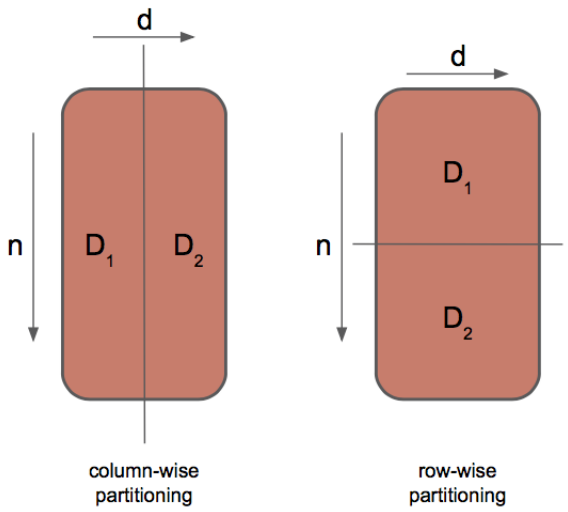

Figure 3-5: Partitioning of input data in distributed machine learning

3.1.2 Transformation

3.1.3 Consistency

Algorithm 5 SCPM iterative-convergent algorithm definition

State: Data tensor $D \in \mathbb{R}^{n \times d}$, weight tensor $w \in \mathbb{R}^{m \times d}$
Partitioning: $\{P_D\}_{k=1}^{K}$, $\{P_w\}_{k=1}^{K}$
Consistency:
Input: algorithm specific hyper-parameters θ, if any
Initialize: $t \leftarrow 0$, $w^{(0)} \leftarrow 0$

1: $D \leftarrow f_p(D)$
2: **repeat**
3: $\quad t \leftarrow t + 1$
4: $\quad w^{(t)} \leftarrow w^{(t-1)} + \Delta(w^{(t-1)}, \theta, D)$
5: **until** termination criteria satisfied

3.2 Algorithm Parallelization Pipeline

This intermediate representation (logical plan) can then be combined with information about the cluster resources and instructions for the consistency management to obtain a physical plan, which is then used by the framework to actually schedule the distributed execution of the algorithm in a cluster. The sequence of steps necessary to go from an algorithm definition to a distributed or physical plan is depicted in

Figure 3-6, where a light arrow resembles an intermediate representation and a dark arrow depicts a transformation step between those representations. At each step, the corresponding representation is enriched by additional information about the algorithm or cluster infrastructure. The following sections describe how the intermediate

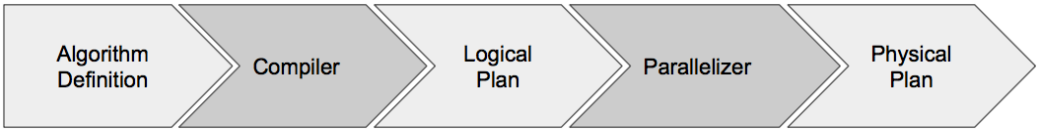

Figure 3-6: Algorithm parallelization pipeline

representations look like and what purpose they serve.

3.2.1 Compiler

In the first step, a logical plan is derived from the algorithm definition, the compiler identifies the building blocks of the given algorithm such as control flow, transformations and state present in the algorithm as well as the dependencies among them. The information used to derive the logical plan is solely based on the algorithm definition and does not take infrastructure related properties into account because this representation should be architecture independent.

NOTES: add caption for state, change figure, make termination more generic, it's better to call this control flow - describe the development flow: developer implements algorithm (specifies states, control flow, applied transformations -> functions), this needs to be identified and mapped into a logical plan by the compiler

3.2.2 Parallelizer

Therefore the actual concern in distributed machine learning is not how to distribute computation but how to distribute the state an keep it consistent in order to achieve optimal performance. As this thesis focuses on machine learning, a state is in general represented by a tensor (e.g. vector or matrix). The most efficient state distribution depends on properties of the state(s) such as size and sparsity of the input data and

model, algorithm used for optimization and infrastructure properties such as available memory and computational resources.

NOTES: add caption for state, change figure, make termination more generic, it's better to call this control flow

NOTES: - motivate the argumentation with an example machine learning pipeline (elastic-net linear regression) which includes preprocessing (e.g. standard-scaling), iterative model refinement which could be data/model parallel and depending on the size of the model it could either be replicated or partitioned among machines/workers, also a ML pipeline involves generalization error assessment via test/validation dataset, which should be included into the stopping criterion this should show: - what kind of workflow is generaly executed in practice - what structure a ml pipeline has and that there are parts that can be run in parallel and in parallel with relaxed constistency/synchronization - show what kind of information can be infered from knowledge about the architecture (computational and memory resources, bandwidth, bandwidth quota) and the problem size (size of input data, size of model) - what part of distributing a machine learning algorithm can be handled by the system (distributing data, distributing model and computation, scheduling of work) and what needs to be specified by the developer (parallelism, control flow and required states (?),)

- describe the limitations of current state-of-the-art distributed machine learning systems - low level primitives - not flexible nor expressible api for developing real word machine learning pipelines - inference of certain properties depending on hardware architecture (GPU, CPU, FPGA) and algorithm properties - distributed control flow - focus on optimizing distributed machine learning performance by quick prototyping - only concerned with the important parts for going from an exact single machine to a distributed algorithm implementation by specifying data partitioning, topology and constistency management properties (synchronzation requirements/schemes, filter, update/merging strategies) - dealing with algorithm related hyperparameters

Chapter 4

Consistency Management

Chapter 5

Experiments

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] A. Hadoop, "Hadoop," 2009.

[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] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[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.