Understanding and Optimizing the Performance of Distributed Machine Learning Applications on Apache Spark

Celestine Dünner
IBM Research
Zürich, Switzerland
cdu@zurich.ibm.com

Thomas Parnell
IBM Research
Zürich, Switzerland
tpa@zurich.ibm.com

Kubilay Atasu
IBM Research
Zürich, Switzerland
kat@zurich.ibm.com

Manolis Sifalakis
IBM Research
Zürich, Switzerland
emm@zurich.ibm.com

Haralampos Pozidis
IBM Research
Zürich, Switzerland
hap@zurich.ibm.com

***Abstract*—In this paper we explore the performance limits of Apache Spark for machine learning applications. We begin by analyzing the characteristics of a state-of-the-art distributed machine learning algorithm implemented in Spark and compare it to an equivalent reference implementation using the high performance computing framework MPI. We identify critical bottlenecks of the Spark framework and carefully study their implications on the performance of the algorithm. In order to improve Spark performance we then propose a number of practical techniques to alleviate some of its overheads. However, optimizing computational efficiency and framework related overheads is not the only key to performance – we demonstrate that in order to get the best performance out of any implementation it is necessary to carefully tune the algorithm to the respective trade-off between computation time and communication latency. The optimal trade-off depends on both the properties of the distributed algorithm as well as infrastructure and framework-related characteristics. Finally, we apply these technical and algorithmic optimizations to three different distributed linear machine learning algorithms that have been implemented in Spark. We present results using five large datasets and demonstrate that by using the proposed optimizations, we can achieve a reduction in the performance difference between Spark and MPI from 20x to 2x.**

I. Introduction

Machine learning techniques provide consumers, researchers and businesses with valuable insight. The rapid proliferation of machine learning in these communities has been driven both by the increased availability of powerful computing resources as well as the large amounts of data that are being generated, processed and collected by our society on a daily basis. While there exist many small and medium-scale problems that can be easily solved using a modern laptop, there also exist datasets that simply do not fit inside the memory capacity of a single machine. In order to solve such problems, one must turn to distributed implementations of machine learning: algorithms that run on a cluster of machines that communicate over a network interface. There are two main challenges that arise when scaling out machine learning to tackle large-scale problems. The first challenge relates to algorithm design: in order to learn in a distributed environment one must determine how the training data should be partitioned across the worker nodes, how the computations should be assigned to each worker and how the workers should communicate with one another in order to achieve global convergence. The second challenge relates to implementation and accessibility. Well-established high performance computing frameworks such as Open MPI provide rich primitives and abstractions that allow flexibility when implementing algorithms across distributed computing resources. While such frameworks enable high performance applications, they require relatively low-level development skills, making them inaccessible to many. In contrast, more modern frameworks such as Hadoop and Spark adhere to well-defined distributed programming paradigms, provide fault tolerance and offer a powerful set of APIs for many widely-used programming languages. While these abstractions certainly make distributed computing more accessible to developers, they come with poorly understood overheads associated with communication and data management which can severely affect performance.

In this work we aim to quantify and understand the different characteristics of Spark- and MPI-based implementations and, in particular, their implications on the performance of distributed machine learning workloads. Our goal is to provide guidance to developers and researchers regarding the best way to implement distributed machine learning applications. Therefore we will proceed as follows:

1) We analyze the performance of a distributed machine learning algorithm implemented from scratch in both Spark and MPI. We clearly decouple framework-related overheads from computational time in order to study Spark overheads in a language agnostic manner.
2) We propose two techniques for extending the functionality of Spark specifically designed to improve the performance of machine learning workloads. We demonstrate that these techniques, combined with C++ acceleration of the local solver, provide over an order of magnitude improvement in performance.
3) We demonstrate that, in order to achieve optimal performance using either framework as well as the proposed extensions, it is crucial to carefully tune the algorithm to the communication and computation characteristics of the specific framework being used.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

4) Finally, we study the effect of the proposed Spark optimizations across five large datasets and apply them to three different distributed machine learning algorithms. We show that the performance of Spark can be improved by close to $10\times$ in all cases.

II. Distributed Machine Learning

In distributed learning we wish to learn a best-fit classification or regression model from the given training data, where every machine only has access to its own part of the data and some shared information that is periodically exchanged over the network. The necessity of this periodic exchange is what makes machine learning problems challenging in a distributed setting and distinguishes them from naively parallelizable workloads. The reason is that the convergence of machine learning algorithms typically depends strongly on how often information is exchanged between workers, while sending information over the network is usually very expensive relative to computation. This has driven a significant effort in recent years to develop novel methods enabling communication-efficient distributed training of machine learning models. In what follows we will focus on the class of synchronous learning algorithms. Such algorithms are more suitable for the purposes of this study than their asynchronous counterparts [1], [2], [3] since they can be naturally expressed as a sequence of *map* and *reduce* operations and can thus be implemented using the Spark framework.

A. Algorithmic Framework

We consider algorithms that are designed to solve regularized loss minimization problems of the form

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}^n} \ell(A\boldsymbol{\alpha}) + r(\boldsymbol{\alpha}), \tag{1}$$

where $A \in \mathbb{R}^{m\times n}$ denotes the data matrix consisting of m training samples and n features. The model is parameterized by the vector $\boldsymbol{\alpha} \in \mathbb{R}^n$ and $\ell : \mathbb{R}^m \to \mathbb{R}$ and $r : \mathbb{R}^n \to \mathbb{R}$ are convex loss and regularization functions respectively. The data matrix A contains the training samples $\{\mathbf{r}_i^\top\}_{i=1}^m$ in its rows and hence the features $\{\mathbf{c}_i\}_{i=1}^n$ in its columns. The matrix can be partitioned column-wise according to the partition $\{\mathcal{P}_k\}_{k=1}^K$ such that features $\{\mathbf{c}_i\}_{i\in\mathcal{P}_k}$ reside on worker k where $n_k := |\mathcal{P}_k|$ denotes the size of the partition. Alternatively, the matrix can be partitioned row-wise according to the partition $\{\hat{\mathcal{P}}_k\}_{k=1}^K$, such that the samples $\{\mathbf{r}_i\}_{i\in\hat{\mathcal{P}}_k}$ reside on worker k, where $m_k := |\hat{\mathcal{P}}_k|$.

The model is then learned in an iterative fashion as illustrated in Figure 1: During one round of the algorithm, a certain amount of computation is performed on each worker (1) and the results of this work are communicated back to the master (2). Once the master has received the results from all workers it performs an aggregation step (3) to update some global representation of the model. This information is then broadcast to all workers (4) and the next round can begin.

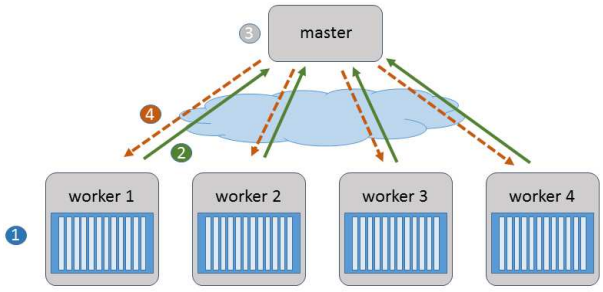


Figure 1: Four-stage algorithmic pattern for synchronous distributed learning algorithms. Arrows indicate the synchronous communication per round.

Let us introduce the hyperparameter H which quantifies the number of local data vectors that are processed on every worker during step (1). This will prove to be a useful tuning parameter allowing users to optimally adapt the algorithm to a given system. In this paper we will cover three prominent algorithms that adhere to this strategy in more detail:

CoCoA: We have implemented the CoCoA algorithm as described in [4]. The data is partitioned feature-wise across the different workers and the regularization term in (1) is assumed to be separable over the coordinates of $\boldsymbol{\alpha}$. Given this partitioning, every worker node repeatedly works on an approximation of (1) based on its locally available data. The CoCoA framework is flexible in the sense that any algorithm can be used to solve the local sub-problem. The accuracy to which each local subproblem is solved is directly tied to the choice of H. For more detail about the algorithm framework and sub-problem formulations, we refer the reader to [4]. In our implementation we use stochastic coordinate descent (SCD) as the local solver, where every node works on its dedicated coordinates of $\boldsymbol{\alpha}$. Hence, in every round, each worker performs H steps of SCD after which it communicates to the master a single m-dimensional vector: $\Delta\mathbf{v}_k := A\,\Delta\boldsymbol{\alpha}_{[k]}$, where $\Delta\boldsymbol{\alpha}_{[k]}$ denotes the update computed by worker k to its local coordinate vector during the previous H coordinate steps. $\Delta\mathbf{v} \in \mathbb{R}^m$ is a dense vector, encoding the information about the current state of $\boldsymbol{\alpha}_{[k]}$, where $\boldsymbol{\alpha}_{[k]}$ itself can be kept local. The master node then aggregates these updates and determines the global update $\mathbf{v}^{(t+1)} = \mathbf{v}^{(t)} + \gamma\sum_k \Delta\mathbf{v}_k$ which is then broadcast to all workers and all local models are synchronized. We use $\gamma = 1$ for our implementations of CoCoA and tune remaining algorithmic parameters accordingly.

Distributed mini-batch SCD: This algorithm differs from CoCoA only in how the local updates $\Delta\boldsymbol{\alpha}_{[k]}$ are computed. While in CoCoA local SCD updates are immediately applied, for mini-batch SCD, every worker computes the gradient g_i for a subset $\mathcal{S}_k \subset \mathcal{P}_k$ of size H of its local coordinates and then determines the update as $\Delta\mathbf{v}_k = -\gamma\sum_{i\in\mathcal{S}_k} g_i\mathbf{c}_i$, where $\gamma \in \mathbb{R}^+$ is the stepsize. The master node aggregates these updates, updates $\mathbf{v}$ and broadcasts this vector back to the workers.

Distributed mini-batch SGD: In contrast to the former two algorithms, mini-batch stochastic gradient descent (SGD) requires separability of the loss term in (1) over the samples and assumes the data to be distributed row-wise. Then, in every round, each worker computes a gradient over a subset $\mathcal{S}_k \subset \{\mathbf{r}_i\}_{i\in\hat{\mathcal{P}}_k}$ of size H of its local samples. These gradients are then communicated to the master node which aggregates them to compute an approximation of the global gradient $\tilde{\mathbf{g}} \in \mathbb{R}^n$ and perform a gradient step on $\boldsymbol{\alpha}$: $\boldsymbol{\alpha}^+ := \boldsymbol{\alpha} - \gamma\tilde{\mathbf{g}}$ before the new parameter vector is broadcasted to the workers.

B. Performance Model

After introducing the hyper-parameter H, the execution time of a distributed algorithm can be modelled as follows: Let us denote $N_\epsilon(H)$ the number of rounds needed to achieve a suboptimality of ϵ given H and $T_\epsilon(H)$ the corresponding execution time. Then, we can write

$$T_\epsilon(H) = N_\epsilon(H)\,(t_1 + t_2 H)\,, \tag{2}$$

where t_1 denotes the fixed overhead of a single round (including communication and aggregation cost), and t_2 denotes the execution time to perform a single update on the worker. Since $N_\epsilon(H)$ is a decreasing function in H there is an optimal value $H^\star$ minimizing the execution time T_ϵ in a given setting. As we will demonstrate this optimal value is very sensitive to the specific infrastructure the algorithm is executed on. While finding a general function from for $N_\epsilon(H)$ that can be used to model convergence for any dataset and/or algorithm remains an interesting research topic we will provide a model for CoCoA in our experimental setup in Section VI.

III. Programming Frameworks for Distributed Computing

There exist many different programming frameworks and libraries that are designed to simplify the implementation of distributed algorithms. In this work we will focus on Spark, due to its widespread use, and compare it to the well established MPI framework.

A. Spark

Apache Spark [5] is an open source general-purpose cluster computing framework developed by the AMP lab at the University of California, Berkeley. The core concept underpinning Spark is the resilient distributed dataset (RDD) which represents a read-only collection of data elements, spread across a cluster that can be operated on in parallel. With this abstraction, Spark allows the developer to describe their distributed program as a sequence of high level operations on RDDs without being concerned with scheduling, load-balancing and fault tolerance. The core of Spark is written in Scala, runs on the Java virtual machine (JVM) and offers a functional programming API to Scala, Python, Java and R. The Python API is called *pySpark* and exposes the Spark programming model to Python. Specifically, the local driver consists of a Python program in which a Spark context is created. The Spark context communicates with the Java virtual machine (over py4J) which in turn is responsible for initiating and communicating with Python processes.

B. MPI

Message Passing Interface (MPI) [6] is a language-independent communication protocol for parallel computing that has been developed for high-performance computing (HPC) platforms. It offers a scalable and effective way of programming distributed systems consisting of up to tens of thousands of nodes. MPI allows application programmers to take advantage of problem-specific load-balancing, communication optimization techniques and various different ways of enabling fault-tolerance for distributed applications. However, this typically requires a significant amount of manual work and advanced understanding of the algorithms, MPI's library functions, and the underlying network architecture.

IV. Implementation Details

To understand the characteristics of the aforementioned programming frameworks and their implications on the performance of distributed learning, we have chosen the CoCoA algorithm [4] as a representative example and implemented it from scratch on Spark, pySpark and MPI. In our implementations these programming frameworks are used to handle the communication of updates between workers during the training of CoCoA. Mathematically, all of the following implementations are equivalent but small differences in the learned model can occur due to randomization and slight variations in data partitioning, which needs to be implemented by the developer in the case of MPI. As an application we have chosen ridge regression because the least squares loss term, as well as the Euclidian norm regularizer, are separable functions, which allows us to apply and later compare all of the three algorithms mentioned in Section II-A.

A. Implementations

(A) Spark: We use the open source implementation of Smith et al. [7] as a reference implementation of CoCoA. This implementation is based on Spark and entirely written in Scala. The Breeze library [8] is used to accelerate sparse linear algebra computations. As Spark does not allow for persistent local variables on the workers, the parameter vector $\boldsymbol{\alpha}$ needs to be communicated to the master and back to the worker in every round, in addition to the shared vector $\mathbf{v}$ – the same applies to the following three Spark implementations.

(B) Spark+C: We replace the local solver of implementation (A) by a Java native interface (JNI) call to a compiled and optimized C++ module. Furthermore, the RDD data structure is modified so that each partition consists of a flattened representation of the local data. In that manner,

one can pass the local data into the native function call as pointers to contiguous memory regions rather than having to pass an iterator over a more complex data structure. The C++ code is able to directly operate on the RDD data (with no copy) by making use of the *GetPrimitiveArrayCritical* functions provided by the JNI.

(C) pySpark: This implementation is equivalent to that of (A) except it is written entirely in Python/pySpark. The local solver makes use of the *NumPy* package [9] for fast linear algebra.

(D) pySpark+C: We replace the local solver of implementation (C) with a function call to a compiled and optimized C++ module, using the Python-C API. Unlike implementation (B) we did not flatten the RDD data structure since this was found to lead to worse performance in this case. Instead, we iterate over the RDD within the local solver in order to extract from each record a list of *NumPy* arrays. The list of *NumPy* arrays is then passed into the C++ module. The Python-C API allows *NumPy* arrays to expose a pointer to their raw data and thus the need to copy data into any additional C++ data structures is eliminated.

(E) MPI: The MPI implementation is entirely written in C++ using the same code for the local solver used in (B) and (D). To initially partition the data we have developed a custom load-balancing algorithm to distribute the computational load evenly across workers, such that $\sum_{i\in\mathcal{P}_k} \|\mathbf{c}_i\|_0$ is roughly equal for each partition. This was found to perform comparable to the Spark partitioning.

B. Infrastructure

All our experiments are run on a cluster of 4 physical nodes interconnected in a LAN topology through a 10Gbit-per-port switched inter-connection. Each node is equipped with 64GB DDR4 memory, an 8-core Intel Xeon* E5 x86_64 CPU and solid-state disks. The software configuration of the cluster is based on Linux* kernel v3.19, MPI v3.2, and Apache Spark v2.2. We use the Open MPI branch of MPI. Spark is configured to use 8 Spark executors with 24 GB of memory each, 2 on each machine. Furthermore, Spark does not use the HDFS filesystem; instead SMB sharing directly over ext4 filesystem I/O is employed. While this decision may occasionally lead to reduced performance in Spark, it eliminates I/O measurement delay-variation artifacts which enables a fairer comparison with MPI since all overheads measured are strictly related to Spark.

V. Analysis and Optimization of Spark

In the first part of this section we analyze the performance of the different implementations of the CoCoA algorithm discussed in Section IV-A by training the ridge regression model on the publicly available *webspam* dataset [10].

A. Spark overheads

We start by extracting the computational time from the total run time for the individual implementations. Therefore we fixed the number of rounds, as well as H, and measured, for each implementation, the total execution time (T_{tot}), as well as the time spent computing on each worker (T_{worker}) and the time spent computing on the master (T_{master}). We denote $T_{\text{overhead}} := T_{\text{tot}} - T_{\text{worker}} - T_{\text{master}}$ which measures overheads related to communications including data transfer as well as serialization/deserialization overheads. The results are displayed in Figure 2.

We observe that the performance of the Spark (A) and pySpark (C) implementations is vastly dominated by the time spent in the local solver. While the code written in Scala performs significantly better than the equivalent Python implementation, both can be accelerated significantly by replacing the local solver with C++ modules. Thereby, the local execution time of the Spark implementation is reduced by a factor of 6 and the execution time of the pySpark implementation by more than 2 orders of magnitude. The local execution time of the C++ code is roughly the same for implementation (B), (D) and (E) up to some internal overheads of the JNI. Leaving the language-dependent differences in execution time aside, focusing on the overheads and subtracting the actual communication cost, as measured in the MPI implementation (3% of the total execution time), we can accurately quantify the framework-related overheads of Spark (and pySpark). We can see that the overheads of the pySpark implementation (C) are 2× larger than those of the reference Spark implementation (A) written in Scala. This performance loss of pySpark was also observed in earlier work [11] and can be attributed to the Python API which introduces additional serialization steps and overheads associated with initializing Python processes and copying data between the JVM and the Python interpreter. Furthermore, we see that calling the C++ modules from Python adds some additional overhead on top of the pySpark overhead, which can be attributed to the large number of Python-C API calls that are required. However, despite the slight increase, these additional overheads are negligible compared to the gain in execution time achieved by offloading the local solver into C++. For Scala we do not see the same increase in overheads and, in fact, observe the opposite behavior. We believe that

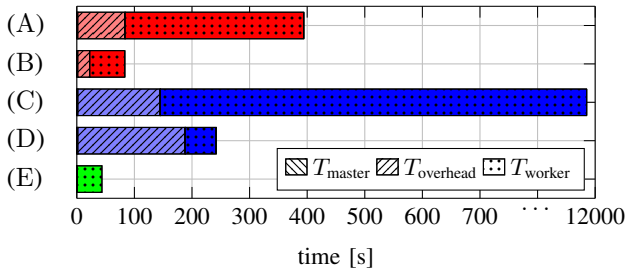


Figure 2: Total run time for 100 iterations with $H = n_k$ split into compute time and overheads for the Spark implementations (A) and (B), the pySpark implementations (C) and (D) and the MPI implementation (E).

this improvement can be attributed to the flattened RDD data format that was implemented when adding the C++ modules in Scala. This structure was explicitly designed to minimize the number of JNI calls. We can see that this flattened data format brings a large benefit: overheads are reduced by a factor of 3. We have also implemented the flattened format in Python but we were not able to achieve a similar improvement.

B. Reducing Spark Overheads

Before we further analyze the implications of the overheads of the Spark framework on the achievable performance of CoCoA we will propose two techniques for extending the functionality of Spark so that these overheads can be somewhat alleviated for distributed learning algorithms.

Persistent Local Memory: Spark does not allow for local variables on the workers that can persist across stage boundaries, that is, the algorithm rounds. Thus the CoCoA algorithm, as well as mini-batch SCD, require additional communication when implemented in Spark since it is not possible for workers to store their dedicated coordinates of $\boldsymbol{\alpha}$ locally. As a consequence, in addition to the shared vector, the $\boldsymbol{\alpha}$ vectors need to be communicated to the master and back in every stage of the algorithm, thus increasing the overhead associated with communication. However, it is relatively straightforward to provide such functionality from within the C++ extension modules. Globally-scoped arrays can be allocated upon first execution of the local solver that store the local $\boldsymbol{\alpha}$ vectors. The state of these arrays persists into the next stage of execution in Spark, and thus the additional communication is no longer necessary. It should be noted that this extension comes at the expense of a violation of the Spark programming model in terms of consistency of external memory with the lineage graph.

Meta-RDDs: For the Python implementations in particular, there is a significant overhead related to the RDD data structure. It is possible to overcome this overhead by following an approach similar to that in [12] and working with RDDs that consist purely of meta-data (e.g. feature indices) and handling all loading and storage of the training data from within underlying native functions. While some additional effort is required to ensure data resiliency, Spark is still being used to schedule execution of the local workers, to collect and aggregate the local updates and broadcast the updated vectors back to the workers.

We have implemented these two features for both the Scala and the Python-based implementations of CoCoA. In Figure 3 we compare the execution time and the overheads of these optimized implementations (B)* and (D)* with the corresponding implementations that only make use of native functions. We observe that our two modifications reduce overheads of the Scala implementation (B) by 3× and those of the Python implementation (D) by 10×. For the Scala implementation, the overall improvement due to using the meta-RDDs is small and most of the gain comes from introducing local memory and thus reducing the amount of data that needs to be communicated. However, for the Python implementation the effect of using meta-RDDs is far more significant. This is most likely due to the vast reduction in inter-process communication that has been achieved. It is worth pointing that the concept of meta-RDDs has additional applications and implications since similar techniques have been used to overcome some of the limitations of Spark, such as using GPUs inside Spark [13].

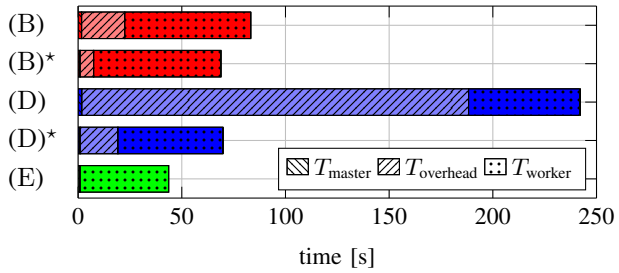
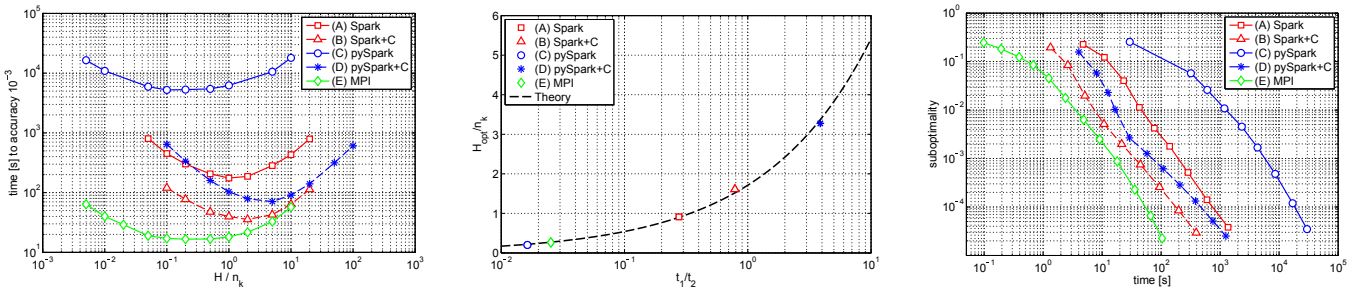


Figure 3: The performance of the optimized implementations (B)⋆ and (D)⋆: by introducing persistent local variables and meta-RDDs we are able to significantly reduce the overheads of Spark relative to MPI.

VI. Trading off Communication vs. Computation

We have seen in Figure 2 that the implementations (A)-(E) suffer from different computational efficiency and overheads associated with communication and data management. These are reflected by t_2 resp. t_1 in our model objective (2). Hence, to optimize performance, i.e., minimize T_ϵ, for the different implementations, it is essential to optimize the hyper-parameter H separately for each implementation to account for these different costs of communication and computation. In this section we will study how the parameter H can be used to control this trade-off for distributed implementations of machine learning and demonstrate the range of improvement an algorithm developer can expect when taking this approach.

In Figure 4a we show the time measured to achieve a suboptimality of 10^{-3} as a function of H for the five different implementations of CoCoA (A)-(E). We see that there is indeed an optimal trade-off for every implementation and the optimal value of H varies significantly among the different implementations of the same algorithm on the same hardware. Hence, in order to get the best performance out of every implementation, it is crucial that H be tuned carefully. Failure to do so may degrade performance dramatically. Indeed, we can see that the best performance of the pySpark implementation is achieved for $H = 0.2n_k$, i.e., every worker performs $0.2n_k$ coordinate updates in every round. However, for the accelerated pySpark implementation (D) the optimal value of H is more than 25× larger. This is because, in implementation (D), the computational cost is significantly reduced relative to the vanilla pySpark



(a) Time to achieve training suboptimality 10^{-3} for implementations (A)-(E) as a function of H.

(b) The optimal value of H as a function the ratio t_1/t_2 of communication time and normalized computation time.

(c) Evolution of suboptimality over time for the implementations (A)-(E) using optimized values for H in all cases.

Figure 4: Optimization of the parameter H: trading-off communication vs. computation.

implementation (see Figure 2) and we can afford to do more updates between two consecutive rounds of communication, thus obtaining a more accurate solution to the local sub-problems. Hence, if the algorithm was not adapted when replacing the local solver by C++ modules, the gain observed would be only $15\times$ instead of $75\times$. Also, comparing implementation (D) to the MPI implementation (E) for which the computation cost is the same but communication is much cheaper, we see a similar difference. While the overheads are less significant, the same reasoning applies to the Scala implementations. These results demonstrate that introducing a tunable hyper-parameter to trade-off communication and computation when designing a distributed learning algorithm is crucial for its applicability in practical environments.

A. Theoretical Analysis

To better understand this trade-off illustrated in Figure 4a we recall the performance model introduced in Section II-B. For the algorithm and dataset under discussion, $N_\epsilon(H)$ can accurately be modelled by:

$$N_\epsilon(H) = \frac{a}{H} + b, \tag{3}$$

where $a, b \in \mathbb{R}$ are constants. Combining (2) with (3) and optimizing for H yields

$$H_{opt} = c\sqrt{t_1/t_2}$$

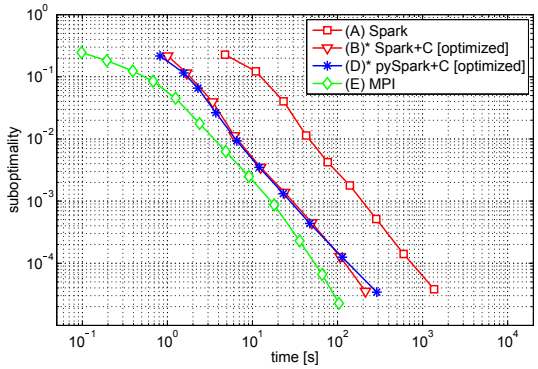
for some constant $c \in \mathbb{R}$. Hence, for this algorithm and dataset, the optimal value H is proportional to the square-root of the ratio between communication and computation cost, which could easily be measured as part of a pre-training phase. In Figure 4b we show that this theoretical estimate precisely agrees with the measurements from Figure 4a.

VII. Performance Evaluation

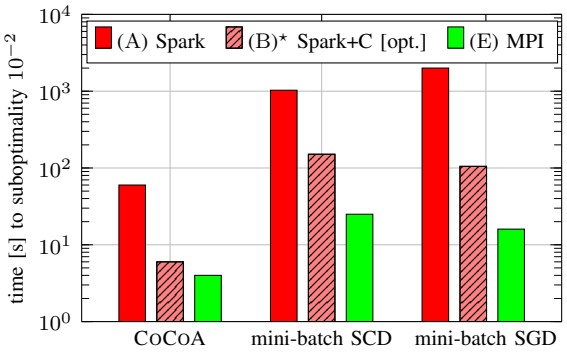
We start by comparing the performance of the five implementations (A)-(E) presented in Section IV-A for individually optimized values of H, see Figure 4c. We observe that this optimization amplifies the performance differences observed in Figure 2. The comparison between implementation (B), (D) and (E) is of particular interest, because in these implementations the computations on the workers are unified in order to eliminate language dependent differences in computational efficiency. Hence, the gap in performance between (B) and (D) can solely be attributed to the overheads of using the Python API to Spark. Similarly, the performance difference between implementations (B), (D) and the MPI implementation (E) can be attributed to framework related overheads of Spark resp. pySpark over MPI. It is worth mentioning that when comparing (E) to (A) instead, more than half of the performance gap is due to the local solver computation being more efficient in C++ than Scala.

By implementing the extensions suggested in Section IV-A in addition to the C++ modules we managed to further improve the Spark performance by 25% and the pySpark performance by 63%. This is shown in Figure 5a. We would like to emphasize that while reducing overheads improves performance by reducing the absolute time spent communicating, it provides the additional benefit that the value of H can be reduced and thus communication frequency is increased, resulting in faster convergence of the algorithm. Without being offered the possibility of tuning H we would only be able to achieve 50% of the performance gain observed by implementing our extensions. Thus, by combining our optimizations we can reduce the performance gap between Spark, resp. pySpark, and MPI from $10\times$, resp. $20\times$ to an acceptable level of less than $2\times$. While we acknowledge that this performance improvement has come at the expense of implementation complexity, these extensions could be integrated within a new or existing Spark library and thus effectively hidden from the developer building a larger machine learning pipeline.

The proposed techniques to improve the performance of distributed machine learning executed on frameworks such as Spark do not only apply to CoCoA. In fact, these techniques can be useful for any algorithm fitting the synchronous pattern of communication described in Figure 1. To illustrate this, we have implemented mini-batch SGD and mini-batch SCD using the proposed optimizations and, in Figure 5b, compare the performance to a reference Spark



(a) Optimized implementations of the CoCoA algorithm to train the ridge regression model on the webspam dataset.



(b) Training the ridge regression model on the webspam dataset using CoCoA, mini-batch SGD and mini-batch SCD.

Figure 5: Proposed Spark optimizations: Performance results

implementation of both algorithms as well as the CoCoA implementations (A), (B)⋆ and (E) that have been previously examined. To implement mini-batch SCD, we modified the local solver of the CoCoA implementations so that a mini-batch coordinate update is computed in each round. For mini-batch SGD, we modified the data partitioning to distribute the data by samples, and used as a reference the implementation provided by Spark MLlib. Optimized Spark and MPI implementations were also developed. The stepsize has been carefully tuned for mini-batch SGD and mini-batch SCD. We observe that CoCoA performs better than the two other algorithms, which is consistent with the results in [4]. The gain from our proposed improvements to Spark is significant for all three algorithms, but for CoCoA we get significantly closer to the performance of MPI than for mini-batch SCD and mini-batch SGD. This is because the two other algorithms have different convergence properties (captured by $N_\epsilon(H)$) and require more frequent communication to achieve convergence. Hence, the overheads associated with communication – which are larger for Spark than for MPI – have a bigger effect on performance.

Finally, we evaluate the performance for the best of the three algorithms (CoCoA) across a range of different datasets. In Table I we present the training time for the Spark reference implementation (A), our optimized implementation (B)⋆ and the MPI implementation (E) for five different datasets. We could not run the reference Spark code for the kdda dataset on our cluster because it ran out of memory due to the large number of features. We observe that by using the proposed optimizations the average performance loss of Spark relative to MPI has been reduced from approximately $20\times$ to around $2\times$.

VIII. Related Work

There have been a number of previous efforts to study the performance of Spark and its associated overheads. In [14] a study was performed comparing the performance of large-scale matrix factorization in Spark and MPI. It was found that overheads associated with scheduling delay, stragglers, serialization and deserialization dominate the runtime in Spark, leading to significantly worse performance relative to MPI. The performance of Spark was also studied in [15] for a number of data analytics benchmarks and it was found that time spent on the CPU was the bottleneck and the effect of improved network performance was minimal. The difference in performance between Spark and MPI/OpenMP was further examined in [16] for the k-nearest neighbors algorithm and support vector machines; the authors concluded that MPI/OpenMP outperforms Spark by over an order of magnitude. Our work differs from these previous studies [14], [15], [16] in that they consider a fixed algorithm running on different frameworks, whereas we optimize the algorithm to achieve optimal performance for any specific framework and implementation, which we have demonstrated to be crucial for a fair analysis of machine learning workloads.

An approach to address Spark's computational bottlenecks, in a similar spirt to our extensions suggested in Section V-B. was proposed in [17]. The authors suggest a high-performance analytics system which they call Tupleware. Tupleware focus on improving the computation bottleneck of analytics tasks by automatically compiling user-defined function (UDF) centric workflows. In this context, a detailed comparison to Spark is provided in a single node setting, demonstrating the inefficiencies introduced by high-level abstractions like Java and iterators. While they suggest a novel analytics platform, our extensions aim to improve the performance of algorithms within a given framework.

The fundamental trade-off between communication and computation of parallel/distributed algorithms has been widely studied. It is well known that there is a fundamental limit to the degree of parallelization, after which adding nodes slows down performance. In the context of large-scale machine learning this behavior has been modelled in [18] aiming to predict a reasonable cluster size. While such a model assumes increasing framework and communication overheads with the number of nodes in a cluster, their assumptions about algorithmic behavior are not reflective of the properties of iterative distributed algorithms, where convergence strongly depends on the communication frequency.

Table I: Spark optimizations of CoCoA for different datasets

Dataset	# samples	# nonzero features	time [s] to reach suboptimality 10^{-3}			slow-down vs. MPI	
			Spark	Spark optimized	MPI	Spark	Spark optimized
news20-binary	19996	1355191	29.92	2.26	0.70	42.73	3.23
webspam	262938	680715	205.24	29.11	16.39	12.52	1.78
E2006-log1p	16087	4265669	610.444	83.04	66.06	9.24	1.26
url	2396130	3230442	1582.78	216.96	118.22	13.39	1.84
kdda	8407752	19306083	–	184.51	79.68	–	2.32

IX. Conclusions

In this work we have demonstrated that vanilla Spark implementations of distributed machine learning can exhibit a performance loss of more than an order of magnitude relative to equivalent implementations in MPI. A large fraction of this loss is due to language dependent overheads. After eliminating these overheads by offloading critical computations into C++, combining this with a set of practical extensions to Spark and effective tuning of the algorithm, we demonstrated a reduction in this discrepancy with MPI to only $2\times$. We conclude that in order to develop high-performance, distributed machine learning applications in Spark as well as other distributed computing frameworks, it is not enough to optimize the computational efficiency of the implementation. One must also carefully adapt the algorithm to account for the properties of the specific system on which such an application will be deployed. For this reason, algorithms that offer the user a tuning parameter to adapt to changes in system-level conditions are of considerable interest from a research perspective.

Acknowledgment

The authors would like to thank Frederick R. Reiss from the IBM Spark Technology Center for highly constructive advice regarding this work and Martin Jaggi from EPFL for his help and inspiring discussions.

* Intel Xeon is a trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

References

[1] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 583–598.

[2] Microsoft, "Multiverso," https://github.com/Microsoft/multiverso, 2015.

[3] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

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

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[6] M. P. Forum, "MPI: A message-passing interface standard," *University of Tennessee*, 1994.

[7] V. Smith and M. Jaggi, "ProxCoCoA+," https://github.com/gingsmith/proxcocoa, 2015.

[8] D. Hall and D. Ramage et al., "Breeze: A numerical processing library for Scala," https://github.com/scalanlp/breeze, 2009.

[9] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation."

[10] S. Webb, J. Caverlee, and C. Pu, "Introducing the webb spam corpus: Using email spam to identify web spam automatically," in *Proceedings of the Third Conference on Email and Anti-Spam (CEAS)*, 2006.

[11] H. Karau, "Improving pyspark performance: Spark performance beyond the jvm," *PyData, Amsterdam*, 2016.

[12] T. Hunter, "TensorFrames on Google's TensorFlow and Apache Spark," *Bay Area Spark Meetup*, Aug 2016.

[13] J. Samuel, K. Ishizaki, and et al., " IBMSparkGPU," https://github.com/IBMSparkGPU/GPUEnabler, 2016.

[14] J. L. Reyes-Ortiz, L. Oneto, and A. Davide, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," in *Procedia Computer Science*, ser. NSDI'12. Elsevier, 2015, pp. 121–130.

[15] A. Gittens, A. Devarakonda, E. Racah, M. F. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. J. Maschhoff, S. Canon, J. Chhugani, P. Sharma, J. Yang, J. Demmel, J. Harrell, V. Krishnamurthy, M. W. Mahoney, and Prabhat, "Matrix factorization at scale: a comparison of scientific data analytics in spark and C+MPI using three case studies," *CoRR*, 2016.

[16] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015, pp. 293–307.

[17] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. B. Zdonik, "Tupleware:" big" data, big analytics, small clusters." in *CIDR*, 2015.

[18] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska, "Automating model search for large scale machine learning," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 368–380.