# Towards Production-Grade, Platform-Independent Distributed ML

**Mikhail Bilenko, Tom Finley, Shon Katzenberger** {MBILENKO,TFINLEY,SHONK}@MICROSOFT.COM
**Sebastian Kochman, Dhruv Mahajan** {SEBASTKO,DHRUMAHA}@MICROSOFT.COM
**Shravan Narayanamurthy, Julia Wang, Shizhen Wang** {SHRAVAN,JUWANG,SHIZHEN}@MICROSOFT.COM
**Markus Weimer** MWEIMER@MICROSOFT.COM

## Abstract

Most existing frameworks for distributed machine learning are either tied to a specific data platform, or focus on novel computational and communication abstractions. The latter often neglect the constraints of shared-use clusters, such as fault tolerance, fair resource (network, CPU) usage, and isolation. This paper proposes a new distributed ML framework, SALMON, that abstracts the key components (control flow, partitioned data store, group communication) and relies only on above-resource-manager platform dependencies (via Apache REEF). The resulting framework is both expressive for common ML algorithm patterns (e.g., iterative MapReduce and parameter server), and flexible to operate on a variety of conventional, shared-use platforms (e.g., Apache Hadoop and HPC). Early experiments demonstrate the promise of this approach via comparisons with Apache Spark on a large-scale production dataset.

## 1. Introduction

Scaling up machine learning methods is an active research area motivated by the challenges of ever-increasing data volumes and velocity, and enabled by the availability of new, powerful distributed computing platforms. In recent years, a number of parallel ML frameworks have emerged, which can be grouped into two broad families:

- ML-centric: motivated by the desire to design elegant, efficient abstractions for ML algorithms, such as GraphLab (Yucheng Low & Hellerstein, 2012), Petuum (Xing et al., 2015), and Parameter Server (Li et al., 2014).
- Platform-centric: built for particular data platforms (e.g., Hadoop or Spark (Zaharia et al., 2010)), such as MAHOUT and MLLIB (Meng et al., 2015).

*Proceedings of the $33^{rd}$ International Conference on Machine Learning*, New York, NY, USA, 2016. JMLR: W&CP volume 48. 

There is an inherent trade-off between the two approaches: while ML-centric frameworks focus on architecture innovation and optimizing the control flow and communication for ML algorithms, they also require extensive adaptation for use in production, shared-use systems. Platform-centric approaches, in contrast, are able to leverage all functionality provided by the APIs of the underlying data platform, but are also limited by them. The dependence on platform-specific components inherently makes such frameworks less useful for heterogeneous-platform production environments, and potentially obsolete as platforms evolve.

Getting the best of both worlds – implementation flexibility of ML-centric platforms and pluggabilty into shared-use platforms – is the holy grail for production-grade machine learning frameworks. In this paper, we investigate a ground-up approach to designing such a framework, SALMON (Scalable Architecture for Learning Methods On Networks). We examine the patterns that should be enabled to implement state-of-the-art ML algorithms, and present early but promising experimental results.

Our investigation is framed by both recent developments in data platforms (the emergence of resource managers as their core abstractions) and ML methods (the rise of asynchronous, stochastic optimization methods). We identify a set of key building blocks for assembling a wide range of ML-specific systems, while facilitating deep platform integration. A flexible, but manageable container-based control flow forms the basis of this set. An abstraction for partitioned datasets standardizes data access across otherwise heterogeneous data platforms. Finally, group communications primitives provide the predominant communications patterns for modern ML systems. Using these building blocks, we assemble specific systems for established distributed ML approaches, such as iterative MapReduce and Parameter Server. Furthermore, the very same building blocks can be used to assemble *algorithm-specific* systems. In all of this, we expect to improve upon the platform-centric frameworks both in terms of raw performance and elegance of expression of the machine learning methods.

In the remainder of this paper, we provide background on modern cluster environments, describe the building blocks

introduced above and how they can be used to assemble ML systems. We present initial experiments comparing SALMON favorably to Spark's MLlib and conclude the paper by outlining a number of promising directions for future work that we are currently pursuing.

## 2. Background

Historically, cluster environments have exposed a single programming abstraction – e.g., in the case of Hadoop, that was MapReduce. This constraint forced an uncomfortable compromise between the needs of machine learning algorithms, which favored other abstractions, and the ease of integration into production systems built on those clusters.

Resource managers like Apache Mesos (Hindman et al., 2011) and Apache Hadoop YARN (Vavilapalli et al., 2013) have emerged to address this challenge. They provide high efficiency under multi-tenant workloads, and, more importantly, allow programming via multiple abstractions: production batch jobs of one team can run right alongside the exploratory interactive jobs of other teams. For example, MapReduce now becomes one of the multiple applications running on top of a resource manager (e.g., YARN). Resource managers provide the performance and security isolation required in this architecture. Applications are split into *containers*, each of which represents an isolated fraction of a physical machine. Depending on the cluster in question, containers are implemented as well-insulated processes, fast-starting virtual machines, or hybrids between the two. Different resource managers implement different container allocation schemes, but they all agree on the basic container abstraction.

Within the confines of the container abstraction, applications are free to implement whichever data and control plane they choose. This allows any of the ML-centric frameworks to be ported to a resource manager. At the same time, resource managers standardize the allocation, billing and general management of the cluster resources. This facilitates the easier integration of ML-centric frameworks into production control and dataflows.

## 3. SALMON Architecture

### 3.1. SALMON Building Blocks

While resource managers provide excellent flexibility to applications, their low position in the distributed platform stack makes programming against them suboptimal, since they provide no data management APIs. The support for control flow can also be problematic, as resource managers typically only facilitate a set of very basic operations: allocation of containers, delivery of resources such as program binaries, and launch and exit notification for tasks.

SALMON builds on the insight that while resource allocators provide a strong base layer for platform-independent machine learning systems, they also require additional building blocks to support commonly-used ML-centric abstractions. It is an open research problem to identify a sufficient basis set of such building blocks, and in this work, we conjecture that the following three are necessary: (1) control flow support, (2) group (or collective) communications, and (3) partitioned data representations.

#### 3.1.1. Control Flow: Apache REEF

ML systems need control flow primitives beyond those provided by the resource manager. For example, we need to send tasks to individual containers, maintain state (e.g. training data) on those containers, and receive notifications of progress as well as failure. However, different ML systems are not in agreement about either their specific scheduling policies, nor their approach to data management and fault handling. Hence, SALMON needs to provide a sufficiently high-level abstraction to simplify the shared aspects of different ML systems, while maintaining the flexibility to allow bespoke control flows for specific (classes of) algorithms (e.g., early stopping of splits for tree learners).

Because of these requirements, we choose Apache REEF (Weimer et al., 2015) to provide the control-flow layer of SALMON. REEF provides the basic primitives for developing portable applications on multiple resource managers. It provides a control-flow master – the Job Driver – which provides the reactive framework for handling the events of a distributed system in a centralized fashion. Such events include the allocation of containers, completion of tasks, or the failure of a container. The *detection* of those events as well as default behaviors for common patterns is provided by REEF. On each container, REEF instantiates an Evaluator – a runtime for the application tasks. Each Evaluator provides services such as messaging between tasks on different containers. REEF is language-neutral and portable across resource managers: it supports JVM- and CLR-based programs on YARN, Mesos, and execution on individual servers.

#### 3.1.2. Group Communications

ML systems require communications between their containers. Beyond low-latency peer-to-peer communications, this often includes collective or group communications across many containers. Examples include the MPI operators BROADCAST (which sends a datum from one to many containers), REDUCE (which aggregates the inputs from many containers to one) and ALLREDUCE (which merges a REDUCE followed by a BROADCAST for better performance). In SALMON, we provide all these opera-

tions. Different from MPI, they are *elastic*: containers can be added and removed from the communication groups at runtime; either by choice of the algorithm or because of machine failure. This enables algorithmic fault tolerance and elastic work scheduling for ML algorithms (Narayanamurthy et al., 2013; Beutel et al., 2014).

#### 3.1.3. Partitioned Data

Training data in distributed machine learning is partitioned. For data-parallel learning, each partition contains a subset of the examples. For model-parallel learning, each partition contains a subset of the features of the examples. Both of these approaches are frequently combined for the utmost in scalability. In SALMON, we provide abstractions to access data partitions in each container. Further SALMON provides an abstraction for the scheduling of containers that takes data locality and data statistics into account.

Using these building blocks, we assemble ML-specific abstractions. The list of supportable abstractions is our ongoing research agenda. Here, we describe a simple abstraction, Iterative Map-Reduce-Update in some detail and summarize the SALMON Parameter Server.

### 3.2. Iterative Map-Reduce-Update

Many machine learning algorithms can be expressed in the Statistical Query Model (SQM) (Kearns, 1998). Algorithms in this model include linear models and SVMs. In this model, an algorithm is expressed purely in terms of statistical queries over the training data. Such queries can be thought of as the sum of a function applied to all data points. This formulation is easy to implement in MapReduce (Chu et al., 2006), which spawned Apache Mahout and is at the core of Apache Spark's attraction for machine learning. However, MapReduce systems only manage the query execution, not the overall data flow of the SQM algorithm: Each query can be executed in MapReduce, but the overall flow requires many of those to be executed, and subsequent queries depend on the results of earlier ones. In MapReduce systems, this is awkward as the result of a query is sent to a single driver process which then submits another MapReduce round, potentially incurring heavy scheduling and data loading overheads each time.

In SALMON, we provide first-class support for this algorithm structure via the Iterative Map-Reduce-Update abstraction (IMRU), where an algorithm consists of 3 functions:

$$\begin{aligned} map &: T_{mapIn} \rightarrow T_{mapOut} \\ reduce &: [T_{mapOut}]^* \rightarrow T_{mapOut} \\ update &: [NULL|T_{mapOut}] \rightarrow [T_{mapIn}|T_{result}] \end{aligned}$$

Here, the map input $T_{mapIn}$ denotes the side information for the mappers, e.g. the current model in a gradient descent algorithm. The training data is assumed to be pre-partitioned and available to the mappers. The map output $T_{mapOut}$ is aggregated by $reduce$, which is assumed to be associative and cumulative. The $update$ function decides if there is another query, in which case it returns a new $T_{mapIn}$ to the mappers or if the job shall emit a result $T_{result}$, e.g. the model. In practice $T_{mapIn}$ often is a composite type which encodes both the query and its parameters, e.g. `compute_gradient` and the current model.

In SALMON, IMRU is implemented using the group communications operators introduced above: The output of $update$ is BROADCAST to the $map$ function whose output is fed to $reduce$ via the $REDUCE$ operator. This approach is much more efficient than a MapReduce implementation, as scheduling and communications can be optimized for the known data flow instead of having to be solved ad-hoc in each iteration.

### 3.3. Parameter Server and beyond

IMRU is an example of a Bulk-Synchronous-Parallel (BSP) approach. There is a hard synchronization barrier, as $update$ requires input from all of the $map$ functions to continue. This can be problematic when the machines involved don't have uniform performance or when they get different workloads, as is common in distributed bayesian learning. Parameter Servers emerged as a successful approach to learning problems in that regime. Instead of updating parameters in bulk as part of the $update$ function, they are continuously updated. To do so, some containers assume the role of *servers* responsible for managing partitions of the model, while the remaining containers are *workers* that perform passes over their respective partition of training data. Updates are exchanged between workers and servers on a continuous basis. A Parameter Server can be assembled from the building blocks above: the workers communicate with the servers via the communications library and get scheduled based on the partitioned data set abstraction. The servers need local state management which is beyond the scope of this paper.

Beyond established models of distributed machine learning, SALMON also facilitates the assembly of systems specific to a learning algorithm. For instance, in (Beutel et al., 2014), we used the partitioned data abstraction and the communications library to implement a work-stealing elastic approach to the scheduling of model updates: The model is partitioned into a set of small blocks. Non-overlapping blocks of that model can be scheduled for update in parallel. Worker containers pull these blocks off of a central queue. This approach scales naturally to different numbers of workers. And, more importantly, the number of workers can change during the runtime of the algorithm to account for resource availability.

### 3.4. Integration with Data Platforms

SALMON integrates with the underlying data platforms in several ways. Most importantly, it executes on the data platform's nodes, allowing tight integration with the storage layer via the partitioned dataset abstraction, thus minimizing data transfer costs. Beyond integration with the runtime and storage layers, SALMON also provides crucial hooks for operationalization of ML systems on public cloud. SALMON provides metrics, counters and logs for the ML layer, facilitating deep integration with the debugging, monitoring and billing features of the data platform. Lastly, SALMON provides a central way to impose cluster policies across a wide range of specific ML systems, such as custom restrictions on network connection and local I/O preferences (e.g., preferences for SSD storage).

## 4. Experimental Results

**Algorithm:** We implemented the OWL-QN optimizer (Andrew & Gao, 2007) based on L-BFGS (Liu & Nocedal, 1989) using the IMRU framework described in the previous section, allowing us to train logistic regression models with $L_1$ and $L_2$ regularization. For the experiments described below, the regularization parameters were set to $10^{-6}$ for $L_1$ and 0 for $L_2$, with history size $m = 10$. For experiments with high-dimensionality (over 70 million features), we use VL-BFGS which distributes the L-BFGS state (gradient and update history) and avoids the dot products in the two-loop recursion (Chen et al., 2014).

**Dataset:** All experiments are performed on the production dataset for training click prediction models of a leading search engine. The data set consists of around 1.1 billion examples with $75M$ features. To study the effects of dimensionality and to allow direct comparisons with Spark MLLIB, we applied MurmurHash3 (Appleby) to generate derived data sets with $1M, 500k$ and $1k$ features respectively. Note that since original dimensionality is very large and feature vectors are extremely sparse, hashing has minor impact on the original data size. Train and test sets were obtained via an 80-20 split.

**Baseline:** We compare SALMON to Apache Spark (Zaharia et al., 2010) 1.6.1. We use the logistic regression in MLLIB (Meng et al., 2015) included in Spark in "org.apache.spark.ml" package that uses the OWL-QN optimizer, similar to the one implemented in SALMON. Significant effort was made to try to find the best possible configuration for Spark experiments. For instance, we configured Spark to use the KRYO SERIALIZER, instead of the default JAVA SERIALIZER, which improved Spark's performance by around 40% in our experiments.

**Infrastructure:** We report experiments on a 112-node YARN cluster running on Windows Server 2012R2. Each node has two Intel 8-core Xeon E5-2660 2.20GHz CPUs and 104GB of RAM. On each node, 20 virtual cores and 100GB of memory were available to YARN containers.

| | Number of features | | |
|---|---|---|---|
| Platform | 100k | 500k | 1M |
| Spark MLLIB | 17.5 | 39.0 | 148.5 |
| TLC++ | 9.2 | 11.5 | 15.0 |
| speed-up | 1.9x | 3.39x | 9.9x |

*Table 1.* Comparison of total training time (in minutes) for 100 iterations, with limited feature space dimensionality, using the OWL-QN optimizer on different platforms. Reported times measure only training phase and don't include initialization, reading data, writing output etc.

**Results:** Table 1 compares the time of 100 iterations of SALMON and MLLIB for different dimensionalities, using 100 containers with 20 virtual cores each. In each case, both trainings converged to similar models with almost the same Areas Under an ROC Curve when validated against a test set. Results demonstrate SALMON to be 1.9-9.9x faster on this dataset, with speedups growing with increased dimensionality, demonstrating the communication efficiency benefits of SALMON.

We also observed Spark to require more memory. While for SALMON it was sufficient to allocate only 10GB per container to run efficiently, attempts to run Spark with similar amount of memory caused numerous issues. To resolve this problem and ensure clean results, we allocated 80GB per container (8x more than SALMON) to Spark.

Finally, we attempted to train both MLLIB and SALMON on the non-hashed, 75M-dimensional dataset. MLLIB runs have failed due to excessive communication. In contrast, the SALMON implementation of VL-BFGS completes 100 iterations on that data set in 50 minutes. Using a proprietary implementation of VL-BFGS on Spark that incorporates improvements and optimizations to Spark Core, training completed, however it took 96 minutes – 1.9 times slower.

**Discussion:** These initial experiments verify that our approach can outperform the platform-centric approach, exemplified by MLLIB. This is to be expected, as IMRU can utilize efficient, persistent group communications where MLLIB has to rely on the flexible, but comparatively slow data movement primitives in Spark. Furthermore, SALMON facilitates optimizations such as the use of lower precision floating point representations, which aren't readily accesible to Spark developers. Future work shall not only investigate the contributions of each of these, but also compare SALMON with other ML-centric frameworks.

## 5. Conclusions and Future Work

Defining the basic abstraction layers for a platform-agnostic distributed ML framework has allowed us to com-

bine the benefits of ML-centric and platform-centric systems, while leveraging the flexibility of Apache REEF to allow deployment in shared-use production clusters. The initial performance results of the proposed SALMON framework in comparison to Spark's MLlib are highly encouraging. In ongoing work, we are implementing additional algorithms and integrating SALMON with leading parallel data platforms, and are highly optimistic that the presented approach yields a practical, highly-performant architecture for the next generation of distributed ML systems.

## References


Andrew, Galen and Gao, Jianfeng. Scalable training of L1-regularized log-linear models. In *ICML*, 2007.

Appleby, Austin. MurmurHash. URL https://sites.google.com/site/murmurhash/.

Beutel, Alex, Weimer, Markus, Minka, Tom, Zaykov, Yordan, and Narayanan, Vijay. Elastic distributed Bayesian collaborative filtering. In *NIPS Distributed Machine Learning and Matrix Computations Workshop*, 2014.

Chen, Weizhu, Wang, Zhenghao, and Zhou, Jingren. Large-scale L-BFGS using MapReduce. In *NIPS*. 2014.

Chu, Cheng-Tao, Kim, Sang Kyun, Lin, Yi-An, Yu, YuanYuan, Bradski, Gary R., Ng, Andrew Y., and Olukotun, Kunle. Map-Reduce for machine learning on multicore. In *NIPS*, 2006.

Hindman, Benjamin, Konwinski, Andy, Zaharia, Matei, Ghodsi, Ali, Joseph, Anthony D, Katz, Randy, Shenker, Scott, and Stoica, Ion. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

Kearns, Michael. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, 1998.

Li, Mu, Andersen, David G, Park, Jun Woo, Smola, Alexander J, Ahmed, Amr, Josifovski, Vanja, Long, James, Shekita, Eugene J, and Su, Bor-Yiing. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

Liu, D. C. and Nocedal, J. On the Limited Memory BFGS method for large-scale optimization. *Math. Program.*, 45 (3):503–528, December 1989.

Meng, Xiangrui, Bradley, Joseph, Yavuz, Burak, Sparks, Evan, Venkataraman, Shivaram, Liu, Davies, Freeman, Jeremy, Tsai, DB, Amde, Manish, Owen, Sean, et al. MLlib: Machine learning in Apache Spark. *arXiv preprint arXiv:1505.06807*, 2015.

Narayanamurthy, Shravan, Weimer, Markus, Mahajan, Dhruv, Condie, Tyson, Sellamanickam, Sundararajan, and Keerthi, S. Sathiya. Towards resource-elastic machine learning. In *NIPS 2013 BigLearn Workshop*, 2013.

Vavilapalli, V., Murthy, A., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, 2013.

Weimer, Markus, Chen, Yingda, Chun, Byung-Gon, Condie, Tyson, Curino, Carlo, Douglas, Chris, Lee, Yunseong, Majestro, Tony, Malkhi, Dahlia, Matusevych, Sergiy, et al. REEF: Retainable evaluator execution framework. In *SIGMOD*, 2015.

Xing, Eric P, Ho, Qirong, Dai, Wei, Kim, Jin Kyu, Wei, Jinliang, Lee, Seunghak, Zheng, Xun, Xie, Pengtao, Kumar, Abhimanu, and Yu, Yaoliang. Petuum: a new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola Danny Bickson Carlos Guestrin and Hellerstein, Joseph M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J, Shenker, Scott, and Stoica, Ion. Spark: Cluster computing with working sets. *HotCloud*, 10: 10–10, 2010.