Sparker: Efficient Reduction for More Scalable Machine Learning with Spark

Bowen Yu
Tsinghua University
Beijing, China
yubw15@mails.tsinghua.edu.cn

Huanqi Cao
Tsinghua University
Beijing, China
caohq18@mails.tsinghua.edu.cn

Tianyi Shan*
University of California San Diego
La Jolla, California, USA
tshan@eng.ucsd.edu

Haojie Wang
Tsinghua University
Beijing, China
wang-hj18@mails.tsinghua.edu.cn

Xiongchao Tang
Tsinghua Shenzhen International
Graduate School and Sangfor
Technologies Inc.
Shenzhen, Guangdong, China
txc13@tsinghua.org.cn

Wenguang Chen
Tsinghua University
Beijing, China
cwg@tsinghua.edu.cn

ABSTRACT

Machine learning applications on Spark suffers from poor scalability. In this paper, we reveal that the key reasons is the non-scalable reduction, which is restricted by the non-splittable object programming interface in Spark. This insight guides us to propose ***Sparker***, ***Spark*** with ***E***fficient ***R***eduction. By providing a split aggregation interface, Sparker is able to perform split aggregation with scalable reduction while being backward compatible with existing applications. We implemented Sparker in 2,534 lines of code. Sparker can improve the aggregation performance by up to 6.47× and can improve the end-to-end performance of MLlib model training by up to 3.69× with a geometric mean of 1.81×.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**; **Machine learning**; • **Information systems → Data analytics**.

KEYWORDS

Spark, Machine Learning, Aggregation, Reduction

ACM Reference Format:
Bowen Yu, Huanqi Cao, Tianyi Shan, Haojie Wang, Xiongchao Tang, and Wenguang Chen. 2021. Sparker: Efficient Reduction for More Scalable Machine Learning with Spark. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472499

*She involved in the research work of this paper when she was an intern in Tsinghua University in 2019.

ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9068-2/21/08.
https://doi.org/10.1145/3472456.3472499

1 INTRODUCTION

Spark is one of the most popular big data processing frameworks nowadays. Spark connects various data analytics domains via its *Resilient Distributed Dataset (RDD)* programming model [24] [25]. Spark's machine learning framework - MLlib [14], is widely used since it can be easily integrated into Spark's data analytics workflow.

However, we found that the scalability of machine learning through MLlib is far from satisfactory. For example, we tested the performance of three typical machine learning algorithms on real-world datasets with MLlib. Details of algorithms and datasets can be found in Section 2.1. As shown in Figure 1, on a 8-node cluster, the average speedup over a single-node is only 1.25×.

To understand the reasons behind MLlib's poor scalability, we further investigate the time breakdown of different stages of MLlib's execution. We found that `treeAggregate` (referred as *tree aggregation* later) occupies a geometric mean of 67.18% for end-to-end time, as shown in Figure 2. Thus, tree aggregation is a major hot-spot of MLlib is worth to investigate in detail.

Tree aggregation in Spark can be roughly divided into two successive steps: computation and reduction. We next analyze the performance data to see which part is the bottleneck. In computation step, the value of current partition is computed locally. In reduction step, the computed values, called aggregators are further reduced globally into a single aggregator. We found that although computation scales as expected, reduction time increases significantly, making overall performance not to scale, as illustrated in Figure 4.

Can we improve the scalabiltiy of redution in Spark significantly? The answer seems obvious. State-of-the-art reduction algorithms [2, 4, 7, 21] (referred as *Scalable Reduction* later), such as Rabenseifner's reduction algorithm, perform much better than `treeAggregate` because they employ larger parallelism than the tree reduction algorithm used by Spark. The question is why Spark uses a slow algorithm instead of a fast one?

We reveal that the current Spark programming interface is the reason. Scalable reduction requires to split the objects to be reduced to gain parallelism. However, in Spark, the objects are of *User-Defined Type (UDT)* represented by a generic type, which lacks information to be split. We believe that the key to improve the scalability problem is latent in this interface.

In this work, we design a new programming interface to enable scalable reduction in Spark. The key idea to allow programmers to specify how to split objects to be reduced into small pieces to gain more parallelism. The interface should also be general enough to support various machine learning models in MLlib.

Another opportunity we find to optimize the performance of Spark reduction is to reduce aggregators inside the same executor in memory locally to avoid unnecessary serialization and communication overhead. We call this technique *In-memory Merge (IMM)*.

To evaluate our idea, we make a prototype system called **Sparker**, which enhances Spark with split aggregation to support more scalable machine learning.

The contributions of this work are as follows:

- We observe that the scalability of MLlib is poor and the bottleneck lies in reduction.
- We reveal that the non-splittable object interface in Spark prevents MLlib from adopting fast reduction algorithms, and we propose a general splittable object interface to enable an fast reduction algorithm.
- We propose an in-memory merge approach to further improve the performance of reduction.

Our evaluation shows that Sparker can improve the aggregation performance by up to 6.47× and can improve the end-to-end performance of MLlib model training by up to 3.69× with a geometric mean of 1.81×.

The rest of the paper are organized as follows: Section 2 reveals the bottleneck of MLlib's poor scalability. Section 3 introduces Sparker's design. Section 4 introduces implementation details. Section 5 presents the evaluation of Sparker. Section 6 discusses Sparker's implication and limitation. Section 7 for related works and Section 8 for conclusion.

2 SCALABILITY ANALYSIS OF MLLIB

In this section, we present our observations on the scalability of MLlib. First, we illustrate the poor scalability in MLlib. We then observe that tree aggregation is a hot-spot and the non-scalable reduction algorithm used in tree aggregation is the bottleneck. We further reveal that the existing aggregation interface in Spark is unable to support scalable reduction algorithms due to the lack of object-splitting capabilities.

2.1 Experiment Setup

We perform the experiment on two clusters. The first cluster is denoted as *BIC* which is an in-house cluster connected with 100Gbps Infiniband. The second cluster is denoted as *AWS* which is a medium-scale cluster on Amazon EC2 with larger number of cores and connected with 25Gbps Ethernet.

Table 1 shows their configuration in detail. In the following subsections, we report our results of the experiments.

Table 2 shows all the real-world datasets we examined, denoted as LDA-E, LDA-N, LR-A, LR-C, LR-K, SVM-A, SMV-C, SMV-K, SVM-K12 respectively. LR-K12 runs out of memory under both of our configuration, thus excluded from our selection.

Table 1: Configuration of the two clusters used for experiment

Configuration	BIC	AWS
Cluster type	internal cluster	EC2 (m5d.24xlarge)
Number of nodes	8	10
Processor	Intel Xeon E5-2680 v4	Intel Xeon Platinum 8175M
Logical cores per node	56	96
Memory per node (GB)	256	384
Network	IPoIB EDR (100Gbps)	25Gbps Ethernet
Disks per node	6	4
Disk type	4TB HDD	900GB SSD
Executors per node	6	12
Executor cores	4	8
Executor memory (GB)	30	25
MPI Library	MPICH 3.2	

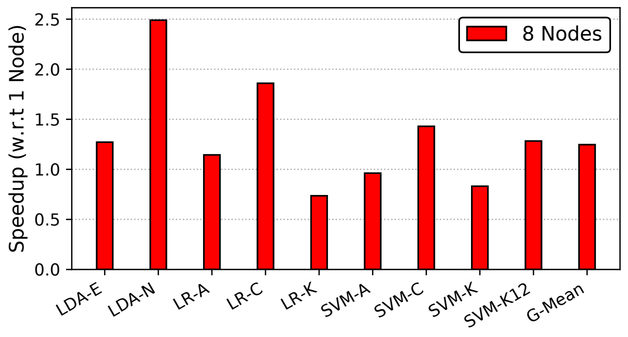

Figure 1: The speedups of different MLlib workloads on 8 nodes with respect to 1-node performance.

2.2 Poor Scalability of MLlib

Based on our prior usage on MLlib to train machine learning models, we experienced MLlib's unsatisfactory performance. Even worse, its performance seems not to scale as we add more computing power. This makes us wonder if MLlib is a highly-scalable machine learning framework. To answer this question, we design a set of experiments. The experiments include 9 combination of workloads including 3 machine learning models (LDA, SVM, LR, shown in Table 3) with and 6 real-world datasets (shwon in Table 2) on a 8-node proprietary cluster (shown in Table 1). We measure both the 1-node and the 8-node performance for each workload. The 8-node speedup over 1-node performance is shown in Figure 1. All of the workloads fall far from the perfect speedup 8. The highest speedup shows up in the workload LDA-N, which is only 2.49× of the performance of a single node. The workload LR-K scales the worst with a speedup of 0.73×. In other words, adding machines even slows down the performance of LR-K. The average speedup of all 9 workloads is 1.25×, which is only 15.58% to the perfect speedup.

2.3 Non-scalable Reduction is the Bottleneck

To find out the limiting factors preventing MLlib from scaling, we analyzed Spark's history logs produced by those workloads and found that the tree aggregation of Spark is a hot-spot. Among those 9 workloads with the 8-node configuration illustrated in Figure 2, the time spent in tree aggregation occupies 67.69% (geometric

Table 2: Real-world datasets used in the experiment

Dataset	Scale	Task	Source
avazu	45,006,431 samples with 1,000,000 features	classification	libsvm
criteo	51,882,752 samples with 1,000,000 features	classification	libsvm
kdd10	8,918,054 samples with 20,216,830 features	classification	libsvm
kdd12	149,639,105 samples with 54,686,452 features	classification	libsvm
enron	39,861 documents with 28,102 dictionary size	topic model	uci
nytimes	300,000 documents with 102,660 dictionary size	topic model	uci

Table 3: MLlib machine learning model used in the experiment

Name	Parameter	Task
Logistic Regression	regParam=0,elasticNetParam=0	classification
SVM	miniBatchFrac=1.0,regParam=0.01	classification
LDA	K=100	topic model

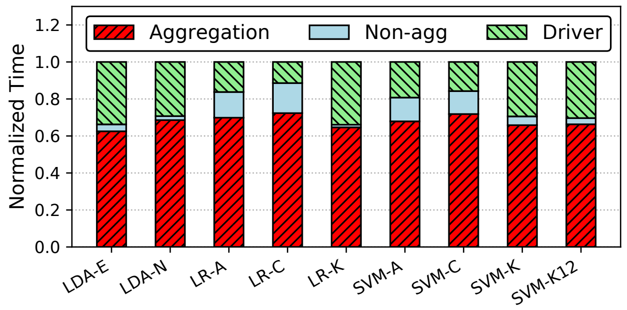

Figure 2: Time is decomposed into aggregation, non-aggregation and non-scalable computation per workload on MLlib.

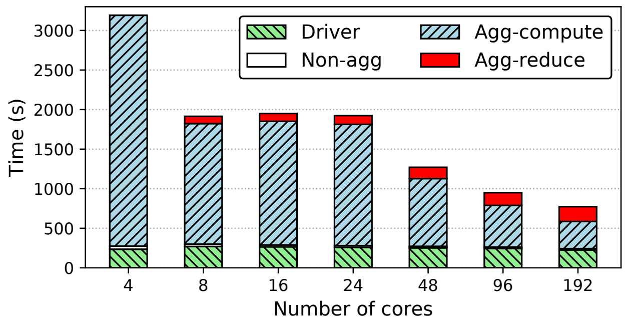

Figure 3: The decomposed end-to-end time of LDA-N with varying number of cores on BIC

mean) of the overall end-to-end time. Thus, tree aggregation is indeed a hot-spot and worth further investigation.

We first perform strong scalability tests on BIC illustrated in Figure 3. *Driver* and *Non-agg* denote non-scalable computation in the driver and scalable computation irrelevant to aggregation, respectively. We further decompose the tree aggregation time into computation and reduction. *Agg-compute* denotes the computation time, which is the time of the first stage of tree aggregation and represents roughly the time to compute the values of aggregators,

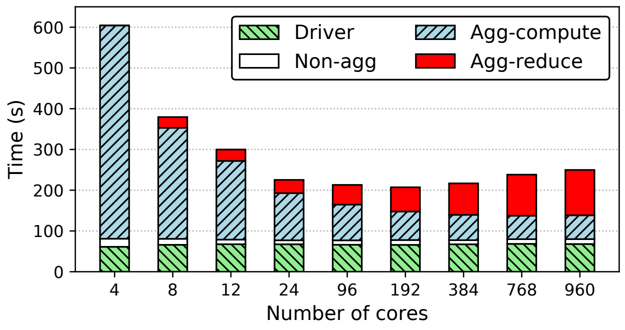

Figure 4: The decomposed end-to-end time of LDA-N with varying number of cores on AWS

while *Agg-reduce* denotes the reduction time, which is the time of the subsequent stages and represents the time to reduce the aggregators globally into a single aggregator in the driver. Figure 3 indicates that as the number of nodes increases from 1 (24 cores) to 8 (192 cores), Spark's computation time reduced from 1152.38*s* to 342.43*s* (4.47×), which suggests that the computation scales as expected. However, the reduction time has increased from 111.05*s* to 187.48*s* (1.69×), which is a scalability bottleneck.

Moreover, it shows that there is room for further scale. To shed some light, we conduct strong scalability tests on AWS ranging the number of cores from 4 to 960 for LDA-N and decompose the end-to-end time into 4 components for LDA-N, as shown in Figure 4. We have reduced the max number of iterations from 40 to 15, thus the AWS result has smaller absolute end-to-end time comparing to the BIC result. As the number of cores increases from 8 to 960, the computation has been reduced from 272.36*s* to 58.39*s* (4.66×), while the reduction time has increased from 26.38*s* to 111.23*s* (4.22×) and the portion of reduction time has increased from 6.95% to 44.55%. We observe that reduction gradually dominates the end-to-end time of the application and limits its scalability with the increasing of scale. From the trend showing in Figure 4, the reduction time would be a significant bottleneck at large scale.

Actually, reduction algorithms [17, 21] and using scalable reduction algorithms in a machine learning system [1, 12, 13] are well studied in earlier works. The fundamental difference between a non-scalable reduction algorithm and a scalable reduction algorithm is that scalable reduction can split the aggregators to extract more parallelism, as shown in Figure 5. Both sides represent the reduction of four aggregators V_1, V_2, V_3, V_4 into a single aggregator V_*. The left side shows that a non-scalable reduction algorithm treats the aggregators as non-splittable objects, while the right side

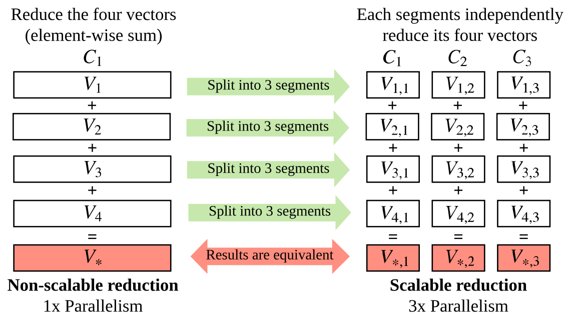
Reduce the four vectors (element-wise sum)
C_1
V_1
+
V_2
+
V_3
+
V_4
=
V_*
Non-scalable reduction
1x Parallelism
Split into 3 segments
Results are equivalent
Each segments independently reduce its four vectors
C_1 C_2 C_3
$V_{1,1}$ $V_{1,2}$ $V_{1,3}$
$V_{2,1}$ $V_{2,2}$ $V_{2,3}$
$V_{3,1}$ $V_{3,2}$ $V_{3,3}$
$V_{4,1}$ $V_{4,2}$ $V_{4,3}$
$V_{*,1}$ $V_{*,2}$ $V_{*,3}$
Scalable reduction
3x Parallelism

Figure 5: Demonstration showing how scalable aggregation splits aggregators to gain parallelism

shows that a scalable reduction algorithm will employ the fact that each aggregator is splittable and V_i can be split into three segments $V_{i,1}$, $V_{i,2}$, $V_{i,3}$. Each segment can be reduced independently to form the results $V_{*,1}$, $V_{*,2}$, $V_{*,3}$. Comparing to non-scalable reduction, scalable reduction gains 3× parallelism, thus more scalable.

2.4 The Aggregation Interface in Spark does not Support Fast Reduction

Splittable object is a prerequisite to use scalable reduction. However, we observe an interesting fact that Spark RDD's existing aggregation interface excludes us from using scalable reduction, due to its lacking in object-splitting functionalities. The `treeAggregate` in Figure 6 demonstrates the existing aggregation interface. The generic type `U` denotes the type of aggregators and two callbacks are provided. The `seqOp` describes how to compute an aggregator while the `reduceOp` describes how to reduce two aggregators into one aggregator. There are nothing to describe whether aggregators are of a splittable type and how to split the aggregators. Lacking in object-splitting functionalities, aggregators can only be treated as an indivisible object. As a result, only non-scalable reduction algorithms could be used.

In order to mitigate the MLlib's scalability bottleneck, we propose to create a new interface of aggregation and enable the new interface with object-splitting capability. In the next section, we will introduce our new interface and illustrate how this new interface enable us to fix the scalability issue of MLlib.

3 DESIGN OF SPARKER FOR FAST REDUCTION

In this section, we first introduce the design of `split aggregation interface` to enable fast reduction algorithm in Spark. Section 3.1 discusses the rationale behinds the interface design. Figure 6 shows the comparison between tree aggregation and our split aggregation. Figure 7 is an example of using split aggregation. Section 3.2 presents the technique of in-memory merge which is an additional optimization for fast reduction and reduces expensive serialization and communication's overheads.

3.1 Split Aggregation Design

The current reduction interfaces for fast reduction in paralle processing systems other than Spark, such as `MPI_Reduce` in the MPI standard [22], typically take an Array of Struct (AoS) from the user.

```
abstract class RDD[T] {
 def treeAggregate[U](zeroValue: U)(
  seqOp: (U, T) => U,
  reduceOp: (U, U) => U,
  depth: Int = 2): U

 def splitAggregate[U, V](zeroValue: U)(
  seqOp: (U, T) => U,
  splitOp: (U, Int, Int) => V,
  reduceOp: (V, V) => V,
  concatOp: IndexedSeq[V] => V,
  parallelism: Int = 4): V
}
```

Figure 6: API for tree aggregation and our proposed split aggregation

```
type AD = Array[Double]
def splitA(A: AD, i: Int, n: Int): AD = /* omitted */
def concatA(Aseq: Seq[AD]): AD = /* omitted */
abstract class Agg(val sum1: AD, val sum2: AD) {
  def add(elem: T): Agg
  def merge(agg: Agg): Agg = {/* omitted */}
}
class AggSeg(val sum1: AD, val sum2: AD) {
  def merge(agg: AggSeg): AggSeg = {/* omitted */}
}
val data: RDD[T] = /* dataset to be aggregated */
val resTA: Agg = data.treeAggregate(zeroValue)(
  (agg, t) => agg.add(t),
  (l, r) => l.merge(r))
val resSA: AggSeg = data.splitAggregate(zeroValue)(
  (agg, t) => {agg.add(t)},
  (agg, i, n) => new AggSeg(
    splitA(agg.sum1, i, n),
    splitA(agg.sum2, i, n)),
  (l, r) => {l.merge(r)},
  segs => new AggSeg(
    concatA(segs.map(_.sum1)),
    concatA(segs.map(_.sum2)))
  )
```

Figure 7: Example code adapted from the `RDDLossFunction` in MLlib to illustrate the concept of split aggregation interface and justify the design of split aggregation interface. For simplification, some of the code are omitted. `splitA` gets a slice from the given array. `concatA` concatenates a sequence of arrays into a single array. Both `merge` in `Agg` (aggregators type) and `AggSeg` (aggregator-segments type) perform the element-wise sum for each of their two arrays.

This interface allows object easily to be split. However, we found that this AoS style of interface is not general enough to represent objects in MLlib. An example is shown in Figure 7, the class for aggregator is denoted as `Agg` which is a struct with two arrays(`sum1` and `sum2`) and can not be presented by the AoS styled interface easily. A more general interface is wanted to support splittable objects for fast reduction in MLlib.

As shown in Figure 6, we design a general interface for aggregation called `Split Aggregation Interface (SAI)`. The rationale behinds this design is to allow users to define the the objects to be split then reduced. The type of aggregator is denoted as `U`, and the type of aggregator-segment is denoted as `V`. Different from the interface of tree aggregation, its `reduceOp` is defined on aggregator-segments, rather than aggregators. In addition, it takes another two callbacks from the users. First, `splitOp` describes how to get a segment with specified index from a given aggregator. Second, `concatOp` describes how to concatenate a sequence of segments into a single segment.

We illustrate the usage of our split aggregation interface with an example code, as shown in Figure 7. The example code is simplified from logistic regression for the purpose of demonstration. Val `resTA` at line 12 represents the result of tree aggregation while val `resSA` at line 15 represents the result of split aggregation. There are several points that worth attention.

First, the type of aggregator-segment (`V`) and the type of aggregator (`U`) can be different. This is because that if we require them to share the same type, MLlib is unable to perform split aggregation. As shown in the example, the aggregator class (`Agg`, line 4) is an abstract class that relies on its sub-classes to define `add` (line 5), which denotes the behavior of adding an sample into this aggregator. Because an abstract class is non-constructable, it is hard to define a `splitOp` that can split an instance of `Agg` and returns `Agg`-typed segments. As a result, if the type of aggregator-segment (`V`) is not distinguished from `U`, it is hard to define a `splitOp` under this scenario. With our design, this problem can be easily solved by introducing a new class `AggSeg` (line 8) that only concerns its role as a merge-only aggregator and its `merge` shares the same behavior to `Agg`'s `merge` (line 6). It is easy to implement `splitOp` as shown in line 17.

Second, the function `splitOp` is defined to return a single specified segment. An alternative design would be returning a sequence of segments within a single call. We choose the former one because multiple threads can split a single aggregator in parallel.

The split aggregation requires the user to provide splitting-related logic. Thus, framework users have to write more code in order to enjoy the benefit of split aggregation. The implication behind is a trade-off between code complexity and performance. However, we argue that such increment in code complexity would not affect the end users most of the time. A library, such as MLlib, can use split aggregation but does not expose such detail to the user. In our case of adapting MLlib to split aggregation, MLlib users only need a configuration parameter to control whether to use split aggregation or not.

We have successfully adapted 3 machine learning algorithms, LR, SVM and LDA, in MLlib for split aggregation by adding 327 lines of code totally.

3.2 In-Memory Merge

Spark allows a single executor to use multiple CPU cores in its resource model. As a result, there would be multiple tasks in the same stage scheduled to the same executor, as shown in Figure 8. Under existing Spark's execution model, each task serializes its result into byte array immediately upon task completion, which will be further fetched by the driver. However, serialization may dominate Spark's overhead [16]. Therefore, it is important to reduce serialization overhead to achieve better performance. We present *in-memory merge*, which is a method that merges task results in the same executor in memory before they are serialized. To be specific, each task updates its task result directly to an in-memory value which is shared among tasks. Thus, all the task results in the same executor will be merged into a single value before serialization to reduce overheads.

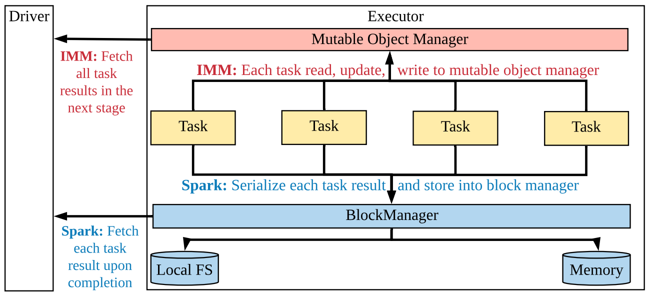
Driver
Executor
Mutable Object Manager
IMM: Fetch all task results in the next stage
IMM: Each task read, update, write to mutable object manager
Task
Task
Task
Task
Spark: Serialize each task result and store into block manager
BlockManager
Spark: Fetch each task result upon completion
Local FS
Memory

Figure 8: in-memory merge reduces serialization and communication overheads by deferring the driver's result fetching and merging task results within the same executor in memory

In case of the failure of any task with IMM, we simply clean up the failed stage which is stored in the shared in-memory value. Then, we re-submit the failed stage again. This is different from the existing RDD, which requires all tasks to be independent from each other. The benefit of this is that any failure of a task would only result in the restart of this task. However, we argue that workloads of in-memory machine learning typically consist with multiple iterations, and each has a short duration. Hence, to restart a whole stage would not bring in large overheads.

4 IMPLEMENTATION

In this section, we lay out the logic behind the implementation of split aggregation and in-memory merging. Then we integrate them into Spark to form our prototype system Sparker. Figure 9 gives an overview to Sparker. Sparker extends Spark executors with two components: scalable communicator and mutable object manager. Scalable communicator enables direct inter-executor communication and provides scalable communication primitives required by split aggregation. Mutable object manager stores intermediate states shared by tasks on the same executor, which is useful for in-memory merge to store currently merged values. First, we introduce how we build a ring-based communication infrastructure for scalable communicator to support scalable reduction based on JeroMQ, which is the Java-binding of the communication library ZeroMQ [8]. Second, we introduce the reduction algorithm and how we implement this algorithm for scalable communicator upon the ring-based topology, and discuss design choices such as topology-awareness and parallelism of the scalable communicator. Third, we discuss the Spark-specific implementation details, including the changes we make to support split aggregation and IMM.

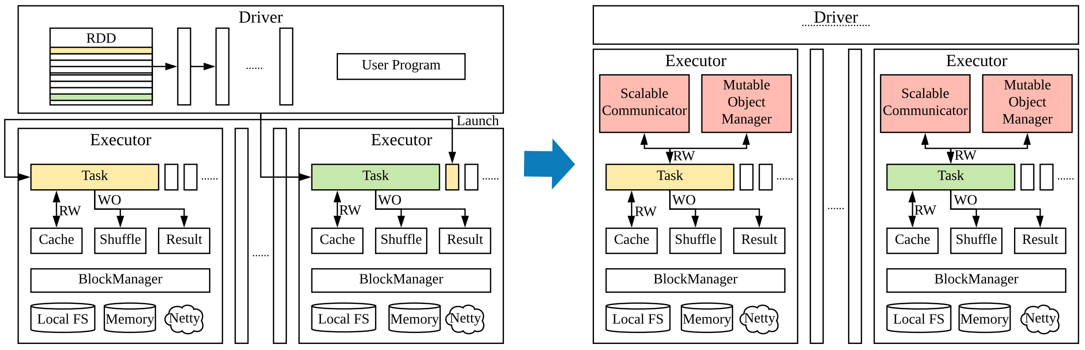

Figure 9: Overview of Spark (left) and Sparker (right)

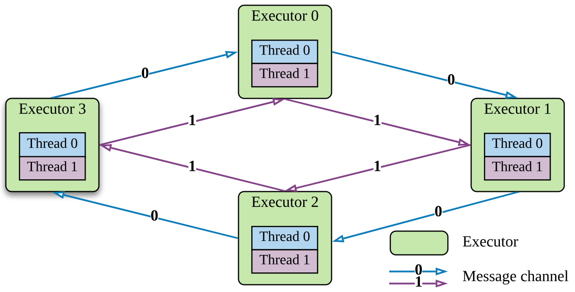

Figure 10: The topology of a scalable communicator with 2-parallelism. Executors are arranged in the form of a parallel directed ring (PDR). Arrow represents message channel, which represents a TCP/IP connection with a pair of sockets. Multiple parallel channels may be established to utilize the network bandwidth.

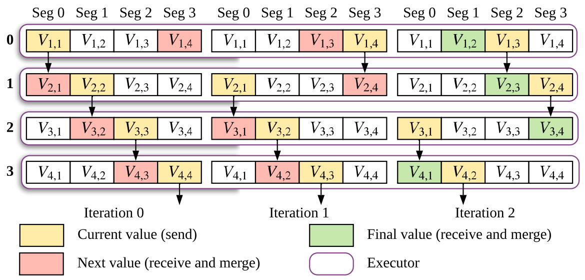

Figure 11: Ring-based reduce-scatter algorithm

4.1 Communication Infrastructure

Since existing Spark communication mechanisms are not suitable for split aggregation, we introduce a new inter-executor communication mechanism for Spark. Currently there are two kinds of mechanisms in Spark that could potentially used for communication: Spark RPC and Spark BlockManager. The former provides an abstraction of remote procedure call and is excluded from our candidates because it is primarily for driver-executor communication. The latter provides an abstraction of distributed key-value store and could be used for inter-executor communication. In the earlier stage of our work, we actually had adapted Spark BlockManager into a versatile communication library supporting send and receive, which provides similar semantic as MPI [22]'s point-to-point communication. However, the performance result, as shown in Figure 12, demonstrates that it has a latency of $3861.25us$, which is $242.24\times$ slower than the result measured by MPI. Its poor latency shadowed the benefit of scalable reduction because the ring-based scalable reduction is sensitive to latency. As a result, we have to make our own communication infrastructure from scratch.

Fortunately, there are existing libraries saving us from low-level network programming. We choose to use ZeroMQ, which is a cross-platform communication library that provides a high-level network abstraction. We use its pure Java binding called JeroMQ, which is fully implemented in JVM and can be used without any native library requirement. This is intended for not breaking Spark's portability. JeroMQ has a latency of $73.23us$, which performs much better compared with Spark BlockManager. As shown in Figure 10, executors are arranged in the form of a *parallel directed ring (PDR)*. Each executor is assigned a unique id called *rank* in the range of $[0, N-1]$ (N denotes the number of executors). Executor i can send to its next neighbor (ranked $(i+1) mod N$) and receive from its previous neighbor (ranked $(i-1+N) mod N$). There are multiple parallel connections between each pair of executors in order to fully utilize the bandwidth in a TCP/IP network. Figure 13 justifies the parallelism in PDR design and shows that certain amount of parallelism is necessary to fully utilize the bandwidth of TCP/IP.

4.2 Scalable Reduction

Since aggregators can now be split via split-aggregate interface, we are able to implement a scalable reduction based on this interface. We choose to use ring-based reduction algorithm [17] as our design foundation due to its high scalability when the data size is large. Scalable reduction involves two steps. The first step is to perform a communication pattern called *reduce-scatter*. We re-use Figure 5 to demonstrate reduce-scatter. Initially, there are 4 executors, and the aggregator V_i is inside executor i. The post-condition of *reduce-scatter* is that each executor i has a segment of the reduced aggregator $V_{*,i}$. The second step is to gather the segments in each executor into the driver. We can use action `collect` provided by Spark to implement the second step. Thus, the scalable communicator only concerns the scalable reduce-scatter. Figure 11 demonstrates the concept of ring-based reduce-scatter by an example to perform reduce-scatter for the 4 executors, each with a single aggregator. First, for each executor i, it splits the aggregator into 4 segments with *splitOp*, denoted as $V_{i,j}$. For each iteration, executors send the *current value* to their next executors, while at the same time, receive a value from their previous executors and merge it into the *next value*. After 3 iterations, each of the segments has traversed across all the 4 executors and forms a final value in one of the executors as shown in the green block in iteration 2.

There are several issues that need attention when fitting the above algorithm into the PDR topology. First, the PDR topology has multiple parallel message channels and those channels should be used in order to fully utilize the network bandwidth. Thus, given the number of parallel channels, i.e., *parallelism* of the scalable communicator (denoted as P), we split each aggregator into $P \times N$ segments, rather than N segments. P threads perform reduce-scatter in parallel. To be specific, thread i uses i-th channel to communicate and perform reduce-scatter on segments in the range of $[i * N, (i+1) * N - 1]$. Figure 10 shows the topology of scalable communicator with 2-parallelism. In our result, 8-parallelism reduce-scatter has a 3.06× speedup over 1-parallelism reduce-scatter. Second, the mapping between the executor to physical nodes matters for performance. Sorting the executors by their hostname, which is called topology-awareness, is an effective way to minimize inter-node communication amount. In our result, the reduce-scatter with topology-awarness has a 2.76× speedup over the reduce-scatter without topology-awareness.

4.3 Spark-specific Details

In order to implement In-Memory Merge upon Spark, we implement a new stage called *reduced-result stage* based on Spark's existing `ResultStage`. While a `ResultStage` serializes and reports each completed task's result to the driver, a reduced-result stage merges the task result locally into the mutable object manager and just returns the executor id and the object id to the driver. We integrate the creation of reduced-result stage into `SparkContext` with an interface similar with `runJob`, but with an additional closure-typed argument describing how to reduce the results.

In order to perform scalable reduction on those aggregators materialized in executors, we implement a new RDD class called `SpawnRDD`. `SpawnRDD` enables task creation with static scheduling. Given a closure describing the task and a list of executor ids describing the task locations, `SpawnRDD` will launch tasks exactly according to the executor list. Split aggregation begins with a reduced-result stage to materialize the results in memory while ensuring that there is exactly one aggregator in each executor. And a `SpawnRDD` is created to perform reduce-scatter and each partition in the `SpawnRDD` represents a segment of the final aggregator. Those segments are further concatenated via `concatOp`.

5 EVALUATION

5.1 Experiment Design

First, we measure the point-to-point latency and throughput between a pair of executors of scalable communicator. Second, we measure the scalability of the reduce-scatter primitive provided by the scalable communicator under different message sizes. We also measure the performance of the reduce-scatter primitive under the largest scale varying the parallelism number. Third, we make a micro-benchmark to evaluate the performance of RDD aggregation, then measure and compare the scalability of tree aggregation, tree aggregation with IMM, and split aggregation under different message sizes. We denote the length of each of the array as *message size*. Forth, we compare the end-to-end performance of Sparker and Spark with 9 combinations of MLlib applications as shown in Table 3 and real-world datasets as shown in Table 2. The configuration of the two clusters used for these experiments is listed in 1.

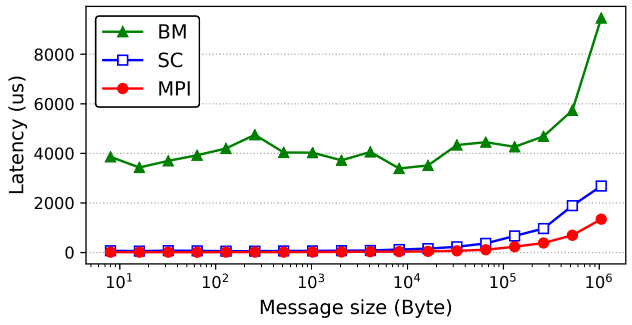

Figure 12: Comparing point-to-point communication latency of BlockManager-based message passing, scalable communicator and MPI

5.2 Micro-benchmarks

We evaluated both micro-benchmarks and end-to-end applications performance in both platforms. Due to space limitation, we only show micro-benchmark results on BIC. The result on AWS is similar.

5.2.1 Point-to-point performance. To evaluate the basic performance of underlying communication library, we make a micro-benchmark to measure the latency and throughput between a pair of executors. As a reference for the ideal performance of underlying network, we also measure the latency and bandwidth from OSU Micro-Benchmarks [15] as MPI's performance is considered closest to optimal network performance.

As shown in Figure 12, MPI achieves a latency of 15.94us on BIC. Scalable communicator (denoted as SC) achieves a latency of 72.73us on BIC, which is 4.56× slower than MPI's latency. The latency of communication library based on Spark BlockManager (denoted as BM) achieves a latency of 3861.25us on BIC, which is 242.24× slower than MPI's latency. The latency result justifies the necessity of building our own communication library from the ground up. Re-using existing Spark communication mechanisms does not satisfy the need of low-latency from split aggregation.

As shown in Figure 13, MPI achieves a maximum throughput of 1185.43 MB/s on BIC, while scalable communicator achieves a throughput of 1151.80 MB/s on BIC . Thus, in terms of throughput, the scalable communicator can achieve 97.1% of the line rate. Actually, the underlying network is capable to provide 100Gb/s bandwidth if RDMA can be used in a bare-metal environment, rather than TCP/IP protocol in a JVM environment. However, as

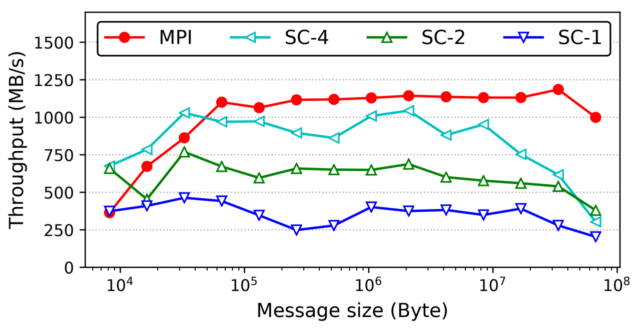

Figure 13: Comparing point-to-point communication throughput of scalable communicator and MPI; x-axis in log-scale. We vary the parallelism of scalable communicator among 1, 2 and 4.

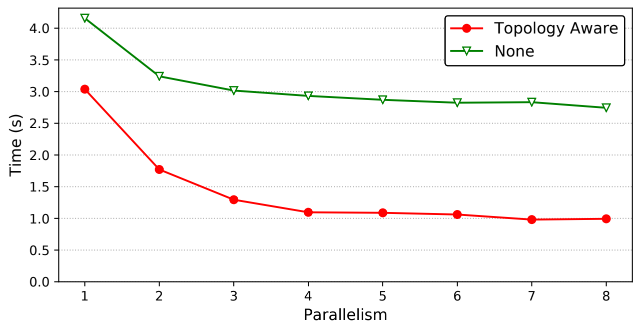

Figure 14: Reduce-scatter performance of scalable communicator varying the parallelism when the message size is 256MB and the number of executors are 48.

is irrelevant to this work, we opt to stick to a pure JVM-based communication library and not to step further. From the figure, we can observe that multiple pairs of sockets and threads are required to fully utilize the bandwidth if TCP/IP protocol is used. Another observation we can see is that with the increasing of message sizes, the scalable communicator's bandwidth changes unsmoothly. The bandwidth even gets worse when the message size is large. We believe this is due to the GC overheads in JVM.

5.2.2 *Reduce-scatter performance.*

As reduce-scatter is the fundamental communication primitive used by the split aggregation, its performance greatly determines the performance of split aggregation. We make a micro-benchmark based on scalable communicator and measure the time to perform reduce-scatter on a randomly-generated array of 8-byte long integers.

We measure reduce-scatter's performance varying the parallelism with 48 executors and 256MB message size. The result is shown in Figure 14. From the result, increasing parallelism inside an executor reduces the communication time from 3.04*s* to 0.99*s* (3.06×). Figure 14 also demonstrates the effectiveness of topology-awareness. Comparing with ordering executors by executor id, ordering executors by host name brings executors in the same node together in the ring topology and reduces the communication time from 2.77*s* to 0.99*s* (2.76×). Thus, in later experiments, we set 4 as the number of parallelism and sort the executors by host name.

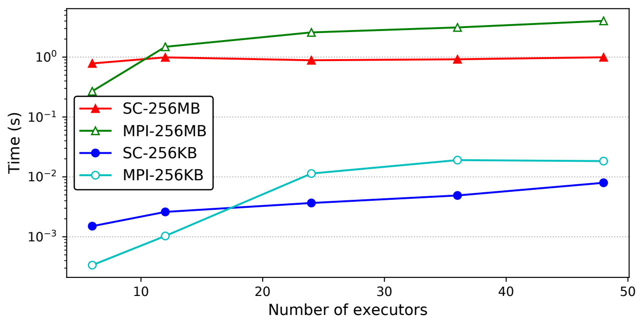

Figure 15: The scalability of reduce-scatter of scalable communicator, compared with MPI as a reference performance.

We measure the scalability of reduce-scatter of the scalable communicator. As a reference, MPI's reduce-scatter performance is also measured. The result is shown in Figure 15. When the message size is large (256MB), scalable communicator achieves a satisfactory scalability. The communication time increased from 784.13*ms* to 993.35*ms* (1.27×) when scaling the number of executors from 6 to 48. When the message size is small (256KB), scalable communicator's communication time increases nearly proportionally with the scaling of the number of executors. The communication time increased from 1.51*ms* to 7.98*ms* (5.30×) when scaling the number of executors from 6 to 48. Scalable communicator even scales better than MPI. One possible reason is that this MPI implementation chooses to use a sub-optimal algorithm, leading to worse scalability even with MPI's advantage in point-to-point communication bandwidth.

5.2.3 *Aggregation performance.*

To evaluate the performance of split aggregation, we implement a micro-benchmark that performs summation of an RDD of randomly-generated fixed-length arrays of 8-byte long integer. The storage level of that RDD is set to `MEMORY_ONLY` and pre-loaded by a `count` action, so that the arrays in that RDD will be materialized in memory. We measure the end-to-end time of calling tree aggregation, tree aggregation with IMM and split aggregation respectively on that RDD varying the message sizes and the number of executors involved in communication. Each experiment item is performed for three times and we take the average as the result.

Figure 16 compares the scalability of tree aggregation, tree aggregation with in-memory merge and split aggregation on small (1KB), medium (8MB) and large (256MB) message sizes by varying the number of nodes from 1 to 8. Figure 16 shows that split aggregation matches the performance of tree aggregation for small messages and significantly better than tree aggregation for large messages. When the message size is 1KB, those three methods have a similar performance. When the message size is 8MB, split aggregation starts to gain advantage to tree aggregation due to the splitting of aggregator. Split aggregation has a speedup of 1.91× over tree aggregation. However, IMM is still not effective at this size. When the message size is 256MB, split aggregation scales nearly constantly with the number of nodes, as the 8-node split aggregation time only increases to 1.12× to the 1-node time. As a result, split aggregation's advantage is amplified by the increasing of the number of nodes.

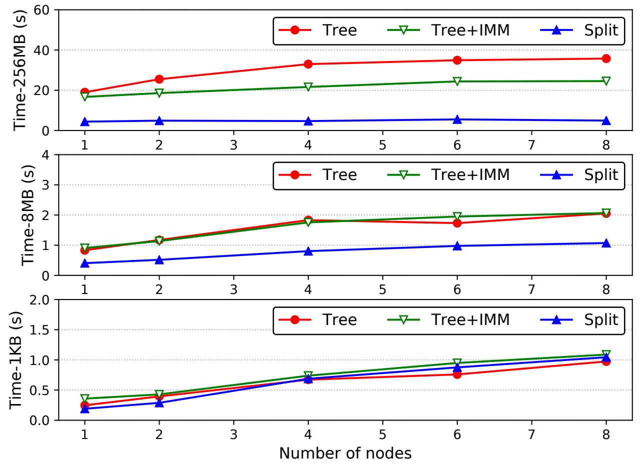

Figure 16: Comparing the scalability of tree aggregation (`Tree`), tree aggregation with in-memory merge (`Tree+IMM`) and split aggregation (`Split`) on 1KB, 8MB and 256MB message sizes.

Split aggregation has a speedup of 6.48× over tree aggregation. As to IMM, it becomes effective at this size, which has a speedup of 1.46× over tree aggregation. There are two messages from figure 16. First, split aggregation is significantly more scalable than tree aggregation. Second, although in-memory merge contributes to split aggregation's improvement, most of the improvement comes from the scalable reduction, which is enabled by our split aggregation interface.

5.3 Real-world applications

Results above show that split aggregation is indeed faster than tree aggregation. To further evaluate split aggregation, we need to investigate its end-to-end improvement in real-world workloads.

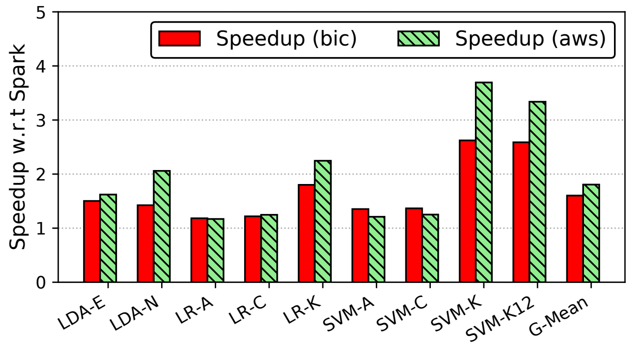

Figure 17: The end-to-end time speedup of Sparker over vanilla Spark on BIC and AWS clusters.

5.3.1 Overall Speedup. Figure 17 shows the speedup of Sparker over vanilla Spark with 9 combinations of machine learning models and real-world datasets on both BIC and AWS. Overall, Sparker achieves a geometric mean speedup of 1.60× on BIC and 1.81× on AWS. Sparker achieves a larger speedup on AWS in most of the workloads because Sparker has better scalability and the AWS cluster is larger than our proprietary cluster. For BIC, the benchmark SVM-K achieves the largest speedup of 2.62×. For AWS, the benchmark SVM-K achieves the largest speedup of 3.69×. Sparker performs well for LDA-N, LR-K, SVM-K and SVM-K12, as their end-to-end speedups on AWS are all above 2×, which is because both `kdd10` and `kdd12` have significantly a larger number of features and `nytimes` has a larger dictionary size, making the size of aggregators significantly large. As a result, reduction becomes a significant performance bottleneck in those workloads. A counter-intuitive result is that SVM-A and SVM-C have a slightly lower speedup on AWS. We investigate SVM-A's profile and find that the speedup of reduction alone is indeed improved from 2.40× on BIC to 2.67× on AWS. This is because that tests on AWS have fewer iterations that lowers the overall speedup on AWS. In summary, those results confirm the effectiveness of split aggregation in real-world applications.

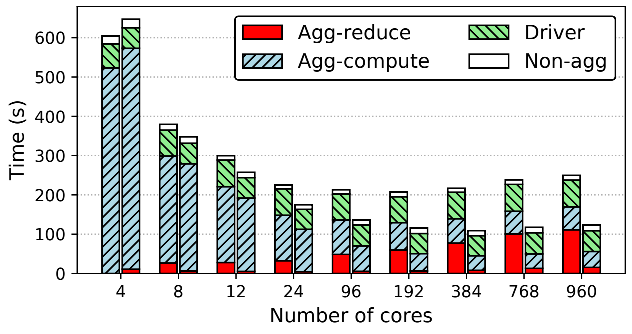

Figure 18: Strong scalability and time decomposition of LDA-N with varying number of cores on AWS. Left bar represents Spark while right bar represents Sparker.

5.3.2 Strong Scalability. To further investigate why split aggregation is effective and how it fixes the scalability issue, we perform the strong scalability tests of LDA-N under both vanilla Spark and Sparker on AWS. We shrink the number of cores for each executor to 4 for intra-node configuration in order to get the 4-core performance result. When the number of cores is less or equal to 96, all the executors are placed in the same node. The end-to-end time has been decomposed into 4 parts, which is similar to the approach previously used in Section 2. The result is shown in Figure 18. When the number of cores is 8, where only 2 executors on the same node involves, the computation time under vanilla Spark is 272.36*s* while the computation under Sparker is 272.95*s*. The computation time of Sparker is near the computation time of vanilla Spark. The reduction times of Spark and Sparker are 26.36*s* and 6.29*s* respectively, showing that the scalable reduction is 4.19× faster than the reduction in tree aggregation. When the number of cores is 960, where all the 10 nodes with 120 executors involve, the computation time under vanilla Spark is 58.39*s* while the computation time under Sparker is 40.49*s*. The computation time of Sparker is smaller than the computation time of vanilla Spark, which is because in-memory merge reduces the serialization overhead and improves the computation performance. The reduction times of Spark and Sparker are 111.26*s* and 15.41*s*, showing that the scalable reduction is 7.22× faster than the reduction in tree aggregation. The reduction time speedup over tree aggregation increases as the number of executors scales, and the reason is that Sparker has a better scalability and performs better at larger scale.

6 DISCUSSION AND LIMITATIONS

In this paper, we provide a case study on how programming interface design may restrict the system from adopting best-known algorithms. We propose a solution to get better performance and scalability with more programming efforts.

There are certainly other design choices by making different tradeoffs between system complexity, user efforts, and performance. For example, compiler techniques may be used to analyze the aggregator to generate split aggregation code without user-defined code. We plan to explore this approach in the future.

Another limitation of this work is that we just remove the reduction bottleneck for Spark. But as shown in Figure 18, the driver overhead becomes the new bottleneck, which deserves further investigation.

7 RELATED WORKS

Aggregation optimization Aggregation is a fundamental component in data-intensive applications and many works have proposed to optimize it. Yu et. al [23] discuss different interfaces, implementations, and optimization strategies of aggregation in different distributed computing systems, pointing out that the choice of programming interface has a significant effect on the aggregation performance. Cedar [11] proposes a solution on deciding the waiting time for aggregators. Camdoop [5] is a MapReduce-like system which improves the performance by using in-network aggregation to reduce the traffic during the shuffle phase. Amur et. al [3] propose a data structure called Compressed Buffer Tree (CBT) to improve the memory efficiency and performance for GroupBy-Aggregation. Symple [18] can automatically parallelize user-defined aggregations (UDAs) using symbolic execution to reduce the network communication and job latency. SwitchML [19] uses in-network aggregation to reduce the volume of exchanged data. However, the works above focus on GroupBy-Aggregation for a dataset of key-value pairs, assuming that sufficient parallelism exists among the elements, which is not applicable to global aggregation without a group-by-key. Sparker focuses on the scalability issue of Spark's global aggregation and provides a splittable aggregation interface to overcome the scalability issue.

Fast reduction algorithms There are a series of fast reduction algorithms. Thakur et. al [21] improve the reduce operations for MPI [20] by using multiple algorithms depending on the message size: for large messages, the ring algorithm is the one with best performance. Works like [2, 4, 7] also optimize the performance of reduction. Sparker's splittable aggregation interface makes it possible to accelerate Spark's global aggregation using those state-of-the-art reduction algorithms.

Machine learning on Spark Spark provides a unified engine for end-to-end machine learning pipelines. MLlib [14] is a widely used Spark-based machine learning library, based on non-splittable aggregation interface of Spark RDD, which limits its scalability as this paper suggests. BigDL [6] accelerates MLlib but relies on Spark's data shuffling for aggregation. Sparker provides a general interface and is not limited to shuffle-based aggregation and enables faster reduction algorithm, including the ring-based algorithm. Zhang et al. [26] supports machine learning workloads such as LDA on compressed data based on Spark.

Standalone machine learning frameworks TencentBoost [10] and Jiang et. al [9] propose Gradient Boosting Tree (GBT) frameworks that are faster than MLlib with a parameter-server architecture. Although standalone frameworks not based on Spark-like data-flow frameworks would have higher performance, they also abandon Spark's capability of unified big data processing pipelines and lineage-based fault tolerance.

8 CONCLUSION

Spark delivers poor scalability on machine learning applications. This paper examines its scalability bottleneck and reveals that the key to solve the inadequate scalability is the non-scalable reduction and the restricted interface. We present a new framework ***Sparker***, ***Spark*** with ***E***fficient ***R***eduction. Sparker is able to perform split aggregation with scalable reduction while being backward compatible with existing applications. We evaluated Sparker on a proprietary cluster and a cloud platform with three machine learning algorithms. The measurement shows that Sparker can easily improve the performance and the scalability of MLlib models. On average, Sparker improves the aggregation throughput by up to 6.47× and the end-to-end performance by up to 3.69× (with geometric mean 1.81×).

ACKNOWLEDGMENTS

This work is partially supported by NSFC U20B2044, BAAI scholar program, and China Postdoctoral Science Foundation (Grant No. 2020M682838). The corresponding author is Wenguang Chen.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Mohammed Alfatafta, Zuhair AlSader, and Samer Al-Kiswany. 2018. COOL: A Cloud-Optimized Structure for MPI Collective Operations. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 746–753.

[3] Hrishikesh Amur, Wolfgang Richter, David G Andersen, Michael Kaminsky, Karsten Schwan, Athula Balachandran, and Erik Zawadzki. 2013. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 18.

[4] Mohammadreza Bayatpour, Sourav Chakraborty, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. Scalable reduction collectives with data partitioning-based multi-leader design. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 64.

[5] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=2228298.2228302

[6] Jason (Jinquan) Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Li (Cherry) Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, and Guoqiong Song. 2019. BigDL: A Distributed Deep Learning Framework for Big Data. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'19)*. Association for Computing Machinery, 50–60. https://doi.org/10.1145/3357223.3362707

[7] Yifan Gong, Bingsheng He, and Jianlong Zhong. 2015. Network performance aware MPI collective communication operations in the cloud. *IEEE Transactions on Parallel and Distributed Systems* 26, 11 (2015), 3079–3089.

[8] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.".

[9] Jiawei Jiang, Bin Cui, Ce Zhang, and Fangcheng Fu. 2018. DimBoost: Boosting Gradient Boosting Decision Tree to Higher Dimensions. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1363–1376.

[10] Jie Jiang, Jiawei Jiang, Bin Cui, and Ce Zhang. 2017. TencentBoost: a gradient boosting tree system with parameter server. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 281–284.

[11] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. 2016. Hold’em or fold’em?: aggregation queries under performance variations. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 7.

[12] He Ma, Fei Mao, and Graham W. Taylor. 2017. Theano-MPI: A Theano-Based Distributed Training Framework. In *Euro-Par 2016: Parallel Processing Workshops*. Springer International Publishing, Cham, 800–813.

[13] Amith R. Mamidala, Georgios Kollias, Chris Ward, and Fausto Artico. 2018. MXNET-MPI: Embedding MPI parallelism in Parameter Server Task Model for scaling Deep Learning. *CoRR* abs/1801.03855 (2018). arXiv:1801.03855 http://arxiv.org/abs/1801.03855

[14] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[15] OSU Micro-Benchmarks. [n. d.]. http://mvapich.cse.ohio-state.edu/benchmarks/.

[16] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[17] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117 – 124. https://doi.org/10.1016/j.jpdc.2008.09.002

[18] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 153–167.

[19] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[20] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI–the Complete Reference: The MPI core*. Vol. 1. MIT press.

[21] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (Feb. 2005), 49–66. https://doi.org/10.1177/1094342005051521

[22] David W Walker and Jack J Dongarra. 1996. MPI: a standard message passing interface. *Supercomputer* 12 (1996), 56–68.

[23] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 247–260.

[24] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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 (NSDI’12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[25] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[26] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.