# On improving Apache Spark's performance: a survey

Mario Becerra Contreras

Fall 2017

**Abstract**

Apache Spark is an open-source cluster computing framework that has grown and become one of the most active open-source projects. It is widely used in research and in industry, but because of this wide use, improvements have been made over the years. In this work, we attempt to give an overview of the modifications that have been made to Spark, whether by the official group or by third-party research groups. The improvements are classified in five areas: communication; file system, OS and JVM parameters; GPUs; performance prediction; and for HPC systems.

## 1 Introduction

In the last years there has been a growing amount of information in industry and research, mainly due to the internet and improvement of computers, which has given rise to different areas of research that did not exist before. One of these areas deals with the storing of massive amounts of information, and another one has to do with the processing of this information. MapReduce [9] is one of the most famous programming models that deals with large data sets on large clusters of commodity machines. Then Hadoop was introduced along with the Hadoop Distributed File System (HDFS) [30], allowing the analysis of large datasets, all using the MapReduce paradigm.

While the HDFS and its underlying technologies provided good performance for using large datasets, they were inefficient for a specific but quite used type of applications: applications that reuse a working set of data. Reusing of data is very common in machine learning applications where it is expected to minimize a loss function, so most algorithms iteratively optimize parameters using techniques such as gradient descent or other numerical optimization methods. Because of this need to have an efficient framework to do iterative algorithms in the same data, Apache Spark was introduced [37] [38].

In this work, it is intended to give a short review of the optimizations and improvements that have been made on Apache Spark over the years since its introduction.

The rest of the work is organized as follows: in section 2 an overview of Apache Spark is given, so that the reader has some context; section 3 surveys and summarizes different published papers on the subject of the improvement of Spark; and section 4 discusses the conclusions of the work and the direction of future work and improvement in this area.

## 2 Overview of Apache Spark

Apache Spark [38] is an open-source cluster computing framework that started in 2009 at the University of California, Berkeley. The project aimed at creating a unified engine for distributed data processing by using a model similar to MapReduce but with data-sharing capabilities. This data-sharing was implemented with an abstraction called Resilient Distributed Datasets (or RDDs) [37]. The idea was that in traditional MapReduce jobs, iterative jobs as the ones that surge naturally in machine learning, graph algorithms and interactive data mining suffer from a high overhead because of the continuous reading of the same data from disk. This could be improved by leveraging distributed memory. Back in the day, the only way to reuse data between computations was to write in a file system, which also incurred in a lot of overhead.

Today, Spark is commonly used with distributed file systems such as Hadoop Distributed File System (HDFS) [30]. It has grown and become one of the most active open-source projects, having over 1000 contributors and used in more than 1000 companies [35]. It is used widely in industry and has more than 350 third-party packages [1], and four main high-level libraries that facilitate the use of Spark: SparkSQL for relational queries [4], Spark Streaming for discretized streams [36], GraphX for graph computations [10], and MLlib for machine learning [23]. These four libraries can be easily combined in applications.

An RDD is a fault-tolerant representation of a read-only collection of partitioned objects across many machines. RDDs can be created through two operations: data from a file system or from other RDDs. These operations are called transformations, and examples include *map*, *filter* and *join*. These operations are evaluated lazily, this way an efficient plan to compute them can be executed. The evaluated is executed until an *action* (like *count*, *collect* or *save* operations) is called. RDDs are reliable because they can be efficiently rebuilt in case a partition is lost. They do this by keeping a log of the graph of transformations that were used to build an RDD (this is called the *lineage* of the RDD). Recovery is faster than rerunning the program because a failed has several partitions which can be rebuilt on other nodes in parallel.

There are two types of dependencies in RDDs: narrow and wide. In narrow dependencies, each partition of the parent RDD is used by at most one partition of the child RDD; whilst in wide dependencies multiple partitions may depend on the same partition of the parent RDD. Wide dependencies are communication intensive because they involve data shuffle across the network and thus may be a performance bottleneck.

By default, RDDs are recomputed every time an action is performed, but users have the option to tell Spark to keep an RDD in memory. If the RDD does not fit in memory, it is spilled to disk. This option is what can make Spark several orders of magnitude faster than plain MapReduce jobs on Hadoop in iterative problems. Since the RDDs can be kept in memory, the reused dataset is not loaded in every iteration. RDDs can express several cluster programming models in an efficient way; models such as MapReduce, SQL, Interactive MapReduce and Batched Stream Processing [37].

## 3 Survey

Even though Spark is a powerful computing framework, it is not perfect and has its caveats. There has been a handful of efforts to improve different areas of opportunity. Some are general improvements, whilst other are particularities that are suited for some tasks. For this work, the improvements are divided in five rather broad and somewhat overlapping areas: based on communication; based on file system, OS and JVM parameters; based on GPUs; based on performance prediction; and for HPC systems.

### 3.1 Improvement based on communication

As mentioned before, there is a performance bottleneck in the shuffling phase of wide dependencies. It is common to compress files before sending them in the network to alleviate this, but this is too dependent on the structure of the input data. Another problem with Spark is that it originally did not fully use the potential of cluster with high-speed connections. There has been work done to explicitly address these issues, and some of the efforts are shown in the following paragraphs.

Davidson and Or [8] propose some alternatives to solve the problems that stress the OS during the shuffle phase. Since the number of shuffle files may be in the order of the millions in a usual Spark work, the number of random writes is quite big. The main proposed solution to this problem is to create larger shuffle files. The results show that the I/O wait time is lower with this modification, but disk throughput is sustained, meaning that much less random I/O is done by creating fewer shuffle files. They suggest as future work to do an in-memory shuffle when it is possible to fit the data in the cluster's collective memory.

Liang and Lu tried to improve the shuffling phase with an event-driven MPI-based communication design called DataMPI-Iteration [18]. It separates the shuffle procedure into fine-grained operations thus permitting the overlap of data movement in an efficient matter, being able to achieve up to 3 times improvement in comparison with Spark's PageRank and K-means algorithms.

The shuffling issue was addressed some time later by the Spark group themselves. Armbrust et al. [3] recognize the problem of shuffling operations on many nodes, along with some other memory and usability issues. The former affair was resolved by changing the original NIO-based network module with a new implementation based on Netty. The latter problems were specifically that memory management needed to be improved, that performance debugging was hard to do, and that there was some confusion about Spark's API.

The memory management issue was dealt with by adding more caps to take care of compressed data that could scale from a few hundreds of MBs to a couple of GBs. As for the debugging, new metrics were added to the monitoring UI; allowing users to see things such as time taken to schedule, run and receive results from tasks. They also added new data visualization tools to monitor the jobs. The confusion about the API was addressed by developing a more declarative API, one that is based on data frames, such as the ones in Python and R. With this new API, the standard data frame interface is compiled using the optimizer in SparkSQL [4]. These data frames are also now being used as input and output between Spark's high level libraries. These changes were introduced in Spark's version 1.2.0.

Nicolae et al. argue that acceleration of shuffling requires the use of memory, which is a precious source and hence its effective utilization is very important [26]. They propose a data transfer strategy that intends to minimize auxiliary memory utilization in the shuffling phase, they achieve it with different design principles such as node-responsiveness based prioritization, load balancing of fetch requests, and static circular allocation of initial requests. That same group later published in 2017 the results of their design in different shuffle-intensive experiments in a cluster, achieving up to 40% better performance with 50% less memory utilization in the shuffle phase [25].

One early work was to adapt RDMA (Remote Direct Memory Access) to Spark in an InfiniBand Architecture [19]. The InfiniBand Architecture is "a new industry-standard architecture for server I/O and inter-server communication" [28]. The authors claim that Spark cannot fully use InfiniBand's capabilities to obtain optimal performance, so they adapt RDMA to be a part of the communication infrastructure by overriding Spark's native features so RDMA calls would be used instead of the Java Socket interface. Their adaptation achieved an 83% performance improvement in the test tasks that they used and conclude that with their RDMA-base design Spark applications could be improved.

This same group continued with this study [20]: in 2016 they tested an advanced RDMA-based design in the changed Apache architecture that came with version 1.2.0 [3]. They particularly check for improvements based on RDMA in the new shuffle architecture based on Netty instead of NIO. They implement an advanced design in three clusters with the most recent InfiniBand technologies and test several workloads on RDD Benchmarks, graph processing and SQL queries on top of Spark scaling up to 1536 cores. The results show that they achieve up to 79% improvement in RDD performance, 46% for graph workloads and 32% for SQL queries.

Some of the subjects that are left as future work are synchronization across nodes, interference between independent shuffles [25], optimizing independent shuffles or minimizing interference with other workloads [26]. The Spark group also mentions that they are currently working on memory management outside the JVM and runtime code generation [3]. There also can be more investigation about architectural bottlenecks in Spark and how to improve them [20].

## 3.2 Improvement based on file system, OS and JVM parameters

Spark is built in Scala, so it heavily uses the Java Virtual Machine to run tasks, and it also has to run on top of a file system and an operating system. Many clusters nowadays come with some Linux-based OS, and HDFS is a very common file system to use in these clusters. It is natural to try to accelerate the execution of Spark tasks by fine-tuning certain parameters that are based on the OS and file system used, or in the configuration of the JVM.

One of the first efforts was done by Islam et al. who focused on studying the impacts of two file systems (Tachyon and Triple-H) on Spark applications [15]. In particular, they identified critical parameters on the performance of the applications that run on top of these file systems, such as BlockSize, concurrent containers and concurrent tasks. They also propose selective caching into high performance memory layers (such as RAMDisks or SSDs).

This selective caching refers to caching the data that is the output of an iteration coming from a map or reduce task. One more optimization comes from improving the cache eviction algorithm after each algorithm iteration. They see that CPU-bound workloads do not show as much benefit as I/O intensive ones, and that iterative applications gain performance over the Triple-H file system.

Wang et al. [34] study the behaviour of Linux virtual memory to find out the effect of kernel parameters. They test this with Spark workloads that are especially sensitive to memory parameter tuning. They test the effect of transparent huge page but find none. They find that enabling Non-uniform memory access (NUMA) yields better results for some workloads, though not all. The conclusion of this work is that better results can be achieved by tuning certain parameters, although this study was small and not generalizable.

Since Spark runs on top of Java Virtual Machines (JVMs), another way to improve performance is to tune JVM parameters. Chiba and Onodera use different JVM and OS parameters to try to take full advantage of computing resources [7]. They argue that Spark creates many objects in heaps, so garbage collection (GC) pause time is a parameter that may cause execution time to be too long. They propose a series of optimizations (such as reducing the nursery space in the heap, changing the number of JVMs in a single node and applying NUMA affinity) and then test them in the TPC-H query benchmark, which is "a decision support benchmark [that] consists of a suite of business oriented ad-hoc queries and concurrent data modifications" [31]. They find that these optimizations performed up to 5 time faster and ran 30% to 40% faster on average.

Another similar work was done by Hema et al., testing the behaviour of Spark under different JVM parameters (heap size and GC frequency), testing it with machine learning case studies, particularly k-means, matrix factorization and logistic regression [12]. They found that k-means and logistic regression did not have performance issues in their tests because heap size was sufficient and it did not need much garbage collection, whilst matrix factorization had a heap memory bottleneck because of the large number of created objects, and consequently GC is called rather frequently. They say that performance can be improved by increasing the executor memory.

The work done on how Spark can be improved by tuning Linux parameters was very specific [34], so they can investigate further by doing more tests and finding out why certain parameters change the behaviour of Spark. As for the JVM-based improvements, the direction in which research is heading is about generalizing results by doing tests on more datasets and workloads, as well as integration of JVM, OS and file system [7] [12].

## 3.3 Improvement based on GPUs

Another line of improvement is to make use of the power of Graphics Processing Units (GPUs) along with Spark. GPUs have thousands of cores to process parallel workloads efficiently, which can be especially useful for many machine learning applications, so an intention to unite GPUs with Spark cluster computing was only natural.

The earliest work was done by Li et al. in 2015, who proposed combining GPUs with CPU-based cluster computing framework [17]. To do so, they implement HeteroSpark, a framework that augments the usual Spark framework by using GPUs in Spark worker nodes. The function calls from Spark are read by HeteroSpark and directed to the GPU via Java's Remote Method Invocation (RMI). If a function call is not implemented in the GPU, the developer may opt to use the original implementation or decide to implement it themselves. The results show that, not surprisingly, performance is much better with Heterospark. This is because of the GPU acceleration. What the authors remark is that this improvement was achieved without sacrificing neither programmability nor portability.

Another effort was made by Manzi and Tompkins, who explore the feasibility and potential benefits of combining GPUs and Spark. Since most non-shuffling operations can be completed on GPUs, their implementation focuses on porting these operations to the GPUs [21]. Their implementation first has to convert original data to a GPU-compatible format, and their conclusion from the analysis is that "the performance improvement for any application will depend on how much the gain from GPU thread-level parallelism overshadows the data conversion cost", e.g., in an implementation for WordCount, the CPU converts from unicode characters to ASCII, and this step takes 57% of the total time [21]. In the K-means algorithm, data shuffling is a minor part of the process, so the gains are much higher; so it is better to use GPUs with algorithms that make many iterations over the same dataset so that the data preparation cost is offset.

Hassaan and Elghandour introduce DAMB, a system for streaming data in a heterogeneous cluster with GPUs and CPUs, and SparkGPU, its core Spark extension that manages and delegates task in the cluster [11]. Their work is tested in a real-time lightning monitoring station that generates 1.6 GB of data per second and achieve very good results.

Ohno, Morishima, and Matsutani accelerate RDD operations for a single computer using GPUs that are either mounted locally or remotely, leveraging how many partitions of RDDs are cached in local memory and how many in remote memory. Their implementation shows a speed-up of up to 21 times compared to Vanilla Spark [27].

Despite these efforts, caching data in GPU memory for iterative processes had not yet been fully explored, until the work of Hong, Choi, and Jeong came along [13]. They noted that in most of the GPU-accelerated applications came from the system overhead and not from the computations themselves. They present a PySpark based extension for Spark that does efficient GPU in-memory caching and processing. They show that their system improves the performance of different iterative algorithms, such as logistic regression and k-means.

In the future, research will be aimed towards the reduction of the overhead of the tools and towards support for RDMA so GPUs have direct P2P communication [13], as well as implementing more sophisticated applications in GPU-based systems [27].

## 3.4 Improvement based on performance prediction

Performance prediction refers to approximating a performance metric (such as execution time or memory consumption) based on the input data fed to the Spark workers. Performance can vary depending on different things such as input data size, type of work done, computing capability, computing architecture, etc. Knowing how much resources a given task will take is useful for the allocation of resources and load balancing in computer clusters.

Wang and Khan [33] present a model that predicts execution time, memory consumption and I/O cost. The model first executes the Spark program on a cluster using a sample of the data, thus collecting information about execution time, memory consumption and I/O costs. They find that their predictions are accurate for execution time and memory consumption, but not so for I/O cost. They conclude that this approach can help better allocate resources in a cluster running Spark.

In 2017, Ulanov, Simanovsky, and Marwah developed a model to that captures the processing time of a given machine learning algorithm in a parallelized computing cluster [32]. Their study aims at modeling the scalability of an algorithm so that the modeler can have an idea in advance of how much the task will take in a given cluster architecture.

In a similar line of improvement, Marco et al. [22] developed a machine learning algorithm that predicts the memory requirement of any Spark application in order to use effective task co-location strategies. The machine learning model is trained for each specific task by taking a subset of the dataset that is going to be used and then fitting a set of linear and non-linear regressions. Their approach was tested with a range of Spark applications that cover different application domains and show an accurate system modeling model. Because of these accurate predictions, the memory resources are better utilized and thus the system achieves performance improvements by effectively using the runtime task scheduler.

Brandón Hernández et al. develop an algorithm to predict a good parallelization strategy so that users can have an optimal configuration before starting the cluster work [5]. Their method was tested in 15 different applications and find that they could achieve up to 51% improvement. Since they used simple models such as linear regression and decision trees, their models can be interpreted for recommendations and explanations as how the number of tasks per nodes affects certain workload.

The applications presented here have good accuracy in their predictive power, but many of the authors claim that accuracy can still be improved, so that is a future line of work. And in particular, Brandón Hernández et al. plan to apply their concepts to heterogeneous environments and make more sophisticated predictions on systems when there are other applications running and model their interference [5]. Ulanov, Simanovsky, and Marwah propose to integrate their software with mainstream tools like Spark, Hadoop, and Tensorflow [32].

## 3.5 Improvements for HPC systems

Even though Spark is most commonly used in large data centres, it can also be used in an HPC context, with supercomputers, such as Cray supercomputers. In 2015 Segal, Colangelo, and Nasiri presented SparkCL [29], a framework that intended to bring unconventional computing power such as GPUs, FPGAs, APUs and DSPs usually found in heterogeneous systems into a mainstream programming use (such as Spark), thus avoiding the inconvenience of working with different type of programming languages that come with hardware specialization. To use the framework, the programmer must write an algorithm in Java deriving from SparkCL-based classes and must override and define different support functions and the kernel function that is meant to be accelerated in the cluster. They implemented three functions to test the operation of SparkCL and conclude that their tool can be used through a high-level Java-OpenCL code in a heterogeneous cluster, although they are missing more detailed results such as speed-up of operations.

In 2016 Chaimov et al. study the performance of Spark in two Cray XC systems in a large supercomputing centre [6]. They first test the performance of a single workstation with SSDs with a single node of a Cray XC and found that the Cray node was slower. They attributed this to the latency involved in opening files (in `fopen`). To calibrate the performance in the Cray they propose using a local file system in a Lustre file or main memory and to use pooling when opening files so that metadata is cached.

HPC clusters are increasingly being equipped with fast high-performance storage, such as SDDs and RAMDisks, and with high-performance interconnects; although not all nodes in the same HPC cluster share the same characteristics: it is common to have SDDs in some nodes and normal HDDs in others. Since the Spark scheduler assumes homogeneity in the nodes, it schedules taking into account the locality of the data, which may not be the best idea for a heterogeneous system [14]. Islam et al. consider some approaches to solve the challenge of choosing a criteria for efficient data access and redesign the HDFS to include their strategies, leading to performance improvement compared to the locality based data access in the benchmarks tested [14].

Another recent effort that has been made is to combine MPI with Spark. Morris and Skjellum present MPIgnite [24], a framework that introduces peer-to-peer communication concepts into Spark. And the most recent work in the area of improving spark for HPC systems was made by Kim et al. In their article they introduce Sparkle [16], a library that enhances Spark for scale-up servers, such as HPC systems, leveraging the shared memory these systems have. With shared memory, data shuffling is less inefficient because there is no dependence on data transfer between distant nodes. Vanilla Spark makes the JVM garbage collector work on deleting a lot of objects in the heap memory of the worker nodes, but Sparkle takes care of this by efficiently storing data in the global shared memory. In the performance tests, Sparkle in a scale-up system showed an improvement of 1.6 up to 5 times in comparison to Vanilla Spark running on a scale-out cluster with equivalent hardware, this mainly due to the faster communication between the nodes.

Finally, in 2017, Anderson et al. pretend to bridge the gap between Spark and HPC frameworks such as MPI and take the both of both: MPI's performance and Spark's fault tolerance [2]. They argue that although Spark has done a very good job in accelerating the performance gotten from using Hadoop alone, it still falls short in performance in comparison with certain HPC environments with native code and optimized communication (e.g., MPI). In their work, they serialize the data from Spark's RDDs and transfer it for MPI processing. They tested their framework with four applications and found that they can achieve between 3 and 17 times speedups compared to Spark-based implementations of the same applications. Although they do put an emphasis on the fact that their framework will only make sense for certain operations in which Spark takes longer than some fixed overheads inherent from the data transfer from RDDs to MPI, i.e., operations which have more complex communication.

Although there has been good work in this area, most, if not all, of the improvements mentioned here seem to involve great programming efforts to get the programs running, and each of them have been ad-hoc improvements for certain type of HPC clusters. Authors also mention repeatedly the need of diminishing overhead in this type of architectures, so that the burden of using their tools is worthy of the results and performance.

## 4 Conclusions and future work

Apache Spark is a widely used framework for data analysis, both in industry and in research. It is one of the most contributed open-source projects with many third-party packages and libraries. Because of this wide use, there is room for improvement. The big data trend is not stopping any time soon, so there is a need to keep filling holes in data processing technologies. The contributors of Apache Spark continue to make improvements, like extending Spark for other languages like R, creating new high level libraries and enhancing performance [35]; but they cannot cover all the niche areas in which Spark can be used, so there are a lot of efforts to mix different technologies with Vanilla Spark.

Most of the improvements are for particular systems and, hence, lack generality; and in general they cannot be so easily integrated with the usual Spark installation. Future work should perhaps be aimed at making more user-friendly, like in high-level libraries, some of the tools presented here, so that less specialized scientists can make use of them without so much burden.

Also, in each of the papers presented here, authors recognize the particular areas in which they can improve their contributions, and most of them are aimed at reducing some sort of overhead related to their adaptations. This is a quite important subject which, even though is very general, it is paramount that if a potential improvement is implemented, the overhead induced by this shall not reduce the benefit that could be obtained by the implementation. For instance, in [2] it is mentioned that their implementation has an overhead of several seconds, so it should be used only when the potential benefit is larger than those seconds that are introduced as overhead.

Finally, there are always new developments in computer architecture, so the improvement of Apache Spark is a continuous work that will never be complete because there are always new areas in which it can be polished.

## References

[1] *A community index of third-party packages for Apache Spark*. `https://spark-packages.org/`. Accessed: 2017-11-04.

[2] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. “Bridging the gap between HPC and big data frameworks”. In: *Proceedings of the VLDB Endowment* 10.8 (2017), pp. 901–912. ISSN: 21508097. DOI: `10.14778/3090163.3090168`. URL: `http://dl.acm.org/citation.cfm?doid=3090163.3090168`.

[3] Michael Armbrust, Matei Zaharia, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, and Reynold Xin. “Scaling spark in the real world”. In: *Proceedings of the VLDB Endowment* (2015), pp. 1840–1843. ISSN: 21508097. DOI: `10.14778/2824032.2824080`.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. “Spark sql: Relational data processing in spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[5] Álvaro Brandón Hernández, ıAÁa S Perez, Smrati Gupta, and Victor Muntés-Mulero. “Using machine learning to optimize parallelism in big data applications”. In: *Future Generation Computer Systems* (2017). DOI: `10.1016/j.future.2017.07.003`. URL: `http://dx.doi.org/10.1016/j.future.2017.07.003.`.

[6] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. “Scaling Spark on HPC Systems”. In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing - HPDC ’16*. 2016, pp. 97–110. ISBN: 9781450343145. DOI: `10.1145/2907294.2907310`.

[7] Tatsuhiro Chiba and Tamiya Onodera. “Workload characterization and optimization of TPC-H queries on Apache Spark”. In: *ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software*. 2016, pp. 112–121. ISBN: 9781509019526. DOI: `10.1109/ISPASS.2016.7482079`.

[8] Aaron Davidson and Andrew Or. “Optimizing shuffle performance in spark”. In: *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* (2013).

[9] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[10] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. “GraphX: Graph Processing in a Distributed Dataflow Framework.” In: *OSDI*. Vol. 14. 2014, pp. 599–613.

[11] Mohamed Hassaan and Iman Elghandour. “A Real-Time Big Data Analysis Framework on a CPU / GPU Heterogeneous Cluster”. In: (2016), pp. 168–177.

[12] N. Hema, K. G. Srinivasa, Saravanan Chidambaram, Sandeep Saraswat, Sujoy Saraswati, Ranganath Ramachandra, and Jayashree B. Huttanagoudar. “Performance Analysis of Java Virtual Machine for Machine Learning Workloads using Apache Spark”. In: *Proceedings of the International Conference on Informatics and Analytics - ICIA-16*. 2016, p. 125. ISBN: 9781450347563. DOI: 10.1145/2980258.2982117.

[13] Sumin Hong, Woohyuk Choi, and Won Ki Jeong. “GPU in-memory processing using spark for iterative computation”. In: *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017* (2017), pp. 31–41. DOI: 10.1109/CCGRID.2017.41.

[14] Nusrat Sharmin Islam, Md Wasi-Ur-Rahman, Xiaoyi Lu, and Dhabaleswar K.D.K. Panda. “Efficient data access strategies for Hadoop and Spark on HPC cluster with heterogeneous storage”. In: *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*. 2016, pp. 223–232. ISBN: 9781467390040. DOI: 10.1109/BigData.2016.7840608.

[15] Nusrat Sharmin Islam, Md Wasi-Ur-Rahman, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. Panda. “Performance characterization and acceleration of in-memory file systems for Hadoop and Spark applications on HPC clusters”. In: *Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015*. 2015, pp. 243–252. ISBN: 9781479999255. DOI: 10.1109/BigData.2015.7363761.

[16] Mijung Kim, Jun Li, Haris Volos, Manish Marwah, Alexander Ulanov, Kimberly Keeton, Joseph Tucek, Lucy Cherkasova, Le Xu, and Pradeep Fernando. “Sparkle: Optimizing Spark for Large Memory Machines and Analytics”. In: *arXiv preprint arXiv:1708.05746* (2017).

[17] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. “HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms”. In: *Proceedings of the 2015 IEEE International Conference on Networking, Architecture and Storage, NAS 2015*. 2015, pp. 347–348. ISBN: 9781467378918. DOI: 10.1109/NAS.2015.7255222.

[18] Fan Liang and Xiaoyi Lu. “Accelerating Iterative Big Data Computing Through MPI”. In: *Journal of Computer Science and Technology* 30.2 (2015), pp. 283–294. ISSN: 10009000. DOI: 10.1007/s11390-015-1522-5.

[19] Xiaoyi Lu, Md Wasi Ur Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K. Panda. “Accelerating spark with RDMA for big data processing: Early experiences”. In: *Proceedings - 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects, HOTI 2014*. 2014, pp. 9–16. ISBN: 9781479958603. DOI: 10.1109/HOTI.2014.15.

[20] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K.D.K. Panda. “High-performance design of apache spark with RDMA and its benefits on various workloads”. In: *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*. 2016. ISBN: 9781467390040. DOI: 10.1109/BigData.2016.7840611.

[21] Dieudonne Manzi and David Tompkins. “Exploring GPU acceleration of apache spark”. In: *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016* (2016), pp. 222–223. DOI: 10.1109/IC2E.2016.30.

[22] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. “Improving Spark Application Throughput Via Memory Aware Task Co-location: A Mixture of Experts Approach”. In: (2017). arXiv: 1710.00610.

[23] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. “MLlib: Machine Learning in Apache Spark”. In: *Journal of Machine Learning Research* 17 (2016), pp. 1–7.

[24] Brandon L Morris and Anthony Skjellum. “MPIgnite: An MPI-Like Language and Prototype Implementation for Apache Spark”. In: (2017). arXiv: arXiv:1707.04788v1.

[25] Bogdan Nicolae, Carlos HA Costa, Claudia Misale, Kostas Katrinis, and Yoonho Park. "Leveraging adaptive I/O to optimize collective data shuffling patterns for big data analytics". In: *IEEE Transactions on Parallel and Distributed Systems* 28.6 (2017), pp. 1663–1674.

[26] Bogdan Nicolae, Carlos Costa, Claudia Misale, Kostas Katrinis, and Yoonho Park. "Towards Memory-Optimized Data Shuffling Patterns for Big Data Analytics". In: *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016* (2016), pp. 409–412. ISSN: 2376-4414. DOI: 10.1109/CCGrid.2016.85.

[27] Yasuhiro Ohno, Shin Morishima, and Hiroki Matsutani. "Accelerating spark RDD operations with local and remote GPU devices". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* (2017), pp. 791–799. ISSN: 15219097. DOI: 10.1109/ICPADS.2016.0108.

[28] Gregory F Pfister. "An introduction to the infiniband architecture". In: *High Performance Mass Storage and Parallel I/O* 42 (2001), pp. 617–632.

[29] O Segal, P Colangelo, and N Nasiri. "SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters". In: *arXiv preprint arXiv: ...* (2015). arXiv: 1505.01120. URL: http://arxiv.org/abs/1505.01120.

[30] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop distributed file system". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010*. 2010, pp. 1–10. ISBN: 9781424471539. DOI: 10.1109/MSST.2010.5496972.

[31] *TPC-H benchmark*. http://www.tpc.org/tpch/. Accessed: 2017-11-05.

[32] Alexander Ulanov, Andrey Simanovsky, and Manish Marwah. "Modeling scalability of distributed machine learning". In: *Proceedings - International Conference on Data Engineering*. 2017, pp. 1249–1254. ISBN: 9781509065431. DOI: 10.1109/ICDE.2017.160. arXiv: 1610.06276.

[33] Kewen Wang and Mohammad Maifi Hasan Khan. "Performance prediction for apache spark platform". In: *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE. 2015, pp. 166–173.

[34] Li Wang, Tianni Xu, Jing Wang, Weigong Zhang, Xiufeng Sui, and Yungang Bao. "Understanding the Behavior of Spark Workloads from Linux Kernel Parameters Perspective". In: *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference on ZZZ - Middleware Posters and Demos '16*. 2016. ISBN: 9781450346665. DOI: 10.1145/3007592.3007593.

[35] Matei Zaharia, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, and Shivaram Venkataraman. "Apache Spark: a unified engine for big data processing". en. In: *Communications of the ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 00010782. DOI: 10.1145/2934664. URL: http://dl.acm.org/citation.cfm?doid=3013530.2934664 (visited on 10/13/2017).

[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: Fault-tolerant streaming computation at scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 423–438.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion 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*. USENIX Association, 2012, p. 2. URL: http://dl.acm.org/citation.cfm?id=2228301 (visited on 10/13/2017).

[38] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95. URL: http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf (visited on 10/13/2017).