Performance Evaluation of Big Data Frameworks for Large-Scale Data Analytics

Jorge Veiga, Roberto R. Expósito, Xoán C. Pardo, Guillermo L. Taboada, Juan Touriño
Computer Architecture Group, Universidade da Coruña, Spain
{jorge.veiga, rreye, pardo, taboada, juan}@*udc.es*

***Abstract*—The increasing adoption of Big Data analytics has led to a high demand for efficient technologies in order to manage and process large datasets. Popular MapReduce frameworks such as Hadoop are being replaced by emerging ones like Spark or Flink, which improve both the programming APIs and performance. However, few works have focused on comparing these frameworks. This paper addresses this issue by performing a comparative evaluation of Hadoop, Spark and Flink using representative Big Data workloads and considering factors like performance and scalability. Moreover, the behavior of these frameworks has been characterized by modifying some of the main parameters of the workloads such as HDFS block size, input data size, interconnect network or thread configuration. The analysis of the results has shown that replacing Hadoop with Spark or Flink can lead to a reduction in execution times by 77% and 70% on average, respectively, for non-sort benchmarks.**

***Keywords*-Big Data; MapReduce; Hadoop; Spark; Flink**

I. Introduction

In the last decade, Big Data analytics has been widely adopted by many organizations to obtain valuable information from the large datasets they manage. This is mainly caused by the appearance of new technologies that provide powerful functionalities to the end users, who can focus on the transformations to be performed on the data rather than on the parallelization of the algorithms.

One of these technologies is Apache Hadoop [1], an open-source implementation of the MapReduce model [2]. The success of Hadoop is mainly due to its parallelization abstraction, fault-tolerance and scalable architecture, which supports both distributed storage and processing of large datasets. However, the performance of Hadoop is severely limited by redundant memory copies and disk operations that it performs when processing the data [3]. As a result, new Big Data frameworks have appeared on the scene in the last several years, claiming to be a worthy alternative to Hadoop as they improve its performance by means of in-memory computing techniques. Apache Spark [4] and Apache Flink [5] are the ones that have attracted more attention due to their easy-to-use programming APIs, high-level data processing operators and performance enhancements.

Although both authors of Spark and Flink provide experimental results about their performance, there is lack of impartial studies comparing the frameworks. This kind of analysis is extremely important to identify the strengths and weaknesses of current technologies and help developers to determine the best characteristics for future Big Data systems. Hence, this paper aims to assess the performance of Hadoop, Spark and Flink on equal terms, with the following contributions:

- Comparative performance evaluation of Hadoop, Spark and Flink with various batch and iterative workloads.
- Characterization of the impact of several experimental parameters on the overall performance.

The rest of this paper is organized as follows: Section II presents the related work. Section III briefly introduces the main characteristics of Hadoop, Spark and Flink. Section IV describes the experimental configuration and Section V analyzes the performance results. Finally, Section VI summarizes our concluding remarks and proposes future work.

II. Related Work

Despite the importance of having performance studies that compare Hadoop, Spark and Flink, there are still not many publications on the subject. Authors of Spark [4] and Flink [5] have shown that their frameworks provide better performance than Hadoop using several workloads. A few impartial references compare Spark with Hadoop. In [6], several frameworks including Hadoop (1.0.3) and Spark (0.8.0) are evaluated on Amazon EC2 using iterative algorithms. Results show that Spark outperforms Hadoop up to 48x for the PAM clustering algorithm and up to 99x for the CG linear system solver. But for the CLARA k-medoid clustering algorithm Spark is slower than Hadoop due to difficulties in handling a dataset with large number of small objects. In [7], Hadoop (2.4.0) and Spark (1.3.0) are evaluated using a set of data analytics workloads on a 4-node cluster. Results show that Spark outperforms Hadoop by 2.5x for WordCount and 5x for K-Means and PageRank. Authors point out the efficiency of the hash-based aggregation component for combine and the RDD caching as the main reasons. An exception is the Sort benchmark for which Hadoop is 2x faster than Spark, showing a more efficient execution model for data shuffling.

It was only recently that some comparisons between Spark and Flink have been published. In [8], both frameworks (versions are not mentioned) are compared on a 4-node cluster using real-world datasets. Results show that Spark outperforms Flink by up to 2x for WordCount, while Flink is better than Spark by up to 3x for K-Means, 2.5x for PageRank and 5x for a relational query. Authors conclude

that the processing of some operators like *groupBy* or *join* and the pipelining of data between operators are more efficient in Flink, and that the Flink optimizer provides better performance with complex algorithms. In [9], Flink (0.9.0) and Spark (1.3.1) are evaluated on Amazon EC2 using three workloads from genomic applications over datasets of up to billions of genomic regions. Results show that Flink outperforms Spark by up to 3x for the Histogram and Mapping to Region workloads and that, contrary to the results of the relational query in [8], Spark was better by up to 4x for the Join workload. Authors conclude that concurrent execution in Flink is more efficient because it produces less sequential stages and that the tuple-based pipelining of data between operators of Flink is more efficient than the block-based Spark counterpart.

Finally, and although not in the scope of this work, [10], [11] and [12] provide recent comparisons of Big Data frameworks from the streaming point of view.

III. Technological Background

The main characteristics of Hadoop [1], Spark [4] and Flink [5] are explained below.

Hadoop: As the de-facto standard implementation of the MapReduce model [2], Hadoop has been widely adopted by many organizations to store and compute large datasets. It mainly consists of two components: (1) the Hadoop Distributed File System (HDFS) and (2) the Hadoop MapReduce engine. The MapReduce model is based on two user-defined functions, map and reduce, which compute the data records represented by key-value pairs. The map function extracts the relevant characteristics of each pair and the reduce function operates these characteristics to obtain the desired result. Although it can provide good scalability for batch workloads, the Hadoop MapReduce engine is strictly disk-based incurring high disk overhead and requiring extra memory copies during data processing.

Spark: Spark increases the variety of transformations that the user can perform over the data, while still including several operators for key-based computations (e.g., sortByKey), which makes Spark particularly suited to implement classic key-based MapReduce algorithms. The programming model of Spark is based on the abstraction called Resilient Distributed Datasets (RDDs) [13], which holds the data objects in memory to reduce the overhead caused by disk and network operations [4]. This kind of processing is specially well suited for algorithms that carry out several transformations over the same dataset, like iterative algorithms. By storing intermediate results in memory, Spark avoids the use of HDFS between iterations and thus optimizes the performance of these workloads.

Flink: Evolved from Stratosphere [5], Flink uses a similar approach to Spark to improve Hadoop performance by using in-memory processing techniques. One of them is the use of efficient memory data structures that contain serialized data instead of Java objects, avoiding excessive garbage collections. Its programming model for batch processing is based on the notion of DataSet, which is transformed by high-level operations (e.g., FlatMap). Unlike Spark, Flink is presented as a true streaming engine as it is able to send data, tuple by tuple, from one operation to the next without executing computations on batches. For batch processing, batches are considered as finite sets of streaming data. It is worth noting that Flink includes explicit iteration operators. The bulk iterator is applied to complete DataSets, while the delta iterator is only applied to the items that changed during the last iteration.

Table I: DAS-4 node configuration

Hardware configuration	
CPU	2 × Intel Xeon E5620 Westmere
CPU Speed/Turbo	2.4 GHz/2.66 GHz
#Cores	8
Memory	24 GB DDR3
Disk	2 × 1 TB HDD
Network	IB (40 Gbps) & GbE
Software configuration	
OS version	CentOS release 6.6
Kernel	2.6.32-358.18.1.el6.x86_64
Java	Oracle JDK 1.8.0_25

IV. Experimental Setup

This section presents the characteristics of the system where the evaluations have been carried out, the configuration of the frameworks and the settings for the workloads, as well as the different experiments that have been performed.

A. Testbed configuration

The evaluations have been conducted on DAS-4 [14], a multi-core cluster interconnected via InfiniBand (IB) and Gigabit Ethernet (GbE). Table I shows the main hardware and software characteristics of this system. Each node has 8 cores, 24 GB of memory and 2 disks of 1 TB.

The experiments have been carried out by using the Big Data Evaluator tool (BDEv), which is an evolution of the MapReduce Evaluator [15]. BDEv automates the configuration of the frameworks, the generation of the input datasets, the execution of the experiments and the collection of the results.

B. Frameworks

Regarding software settings, the evaluations have used stable versions of Hadoop (2.7.2), Spark (1.6.1) and Flink (1.0.2). Both Spark and Flink have been deployed in the stand-alone mode with HDFS 2.7.2. The frameworks have been carefully configured according to their corresponding user guides and the characteristics of the system (e.g., number of CPU cores, memory size). Table II shows the most

Table II: Configuration of the frameworks

Hadoop		Spark		Flink	
HDFS block size	128 MB	HDFS block size	128 MB	HDFS block size	128 MB
Replication factor	3	Replication factor	3	Replication factor	3
Mapper/Reducer heap size	2.3 GB	Executor heap size	18.8 GB	TaskManager heap size	18.8 GB
Mappers per node	4	Workers per node	1	TaskManagers per node	1
Reducers per node	4	Worker cores	8	TaskManager cores	8
Shuffle parallel copies	20			Network buffers per node	512
IO sort MB	600 MB			TaskManager memory preallocate	false
IO sort spill percent	80%			IO sort spill percent	80%

Table III: Benchmark sources

Benchmark	Characterization	Input data size	Input generator	Hadoop	Spark	Flink
WordCount	CPU bound	100 GB	RandomTextWriter	Hadoop ex.	Adapted from ex.	Adapted from ex.
Grep	CPU bound	10 GB	RandomTextWriter	Hadoop ex.	Adapted from ex.	Adapted from ex.
TeraSort	I/O bound	100 GB	TeraGen	Hadoop ex.	Adapted from [16]	Adapted from [16]
Connected Components	Iterative (8 iter.)	9 GB	DataGen	Pegasus	Graphx	Gelly
PageRank	Iterative (8 iter.)	9 GB	DataGen	Pegasus	Adapted from ex.	Adapted from ex.
K-Means	Iterative (8 iter.)	26 GB	GenKMeansDataset	Mahout	MLlib	Adapted from ex.

important parameters of the resulting configuration. The network interface of the frameworks was configured to use IP over InfiniBand (IPoIB), except in the GbE experiments (see Section V-B).

C. Benchmarks

Table III describes the benchmarks used in the experiments, along with their characterization as CPU bound, I/O bound (disk and network) or iterative. The size of the input datasets and the generators used for setting them up are shown in the next two columns. The table also includes the source of the benchmark codes, which have been carefully studied in order to provide a fair performance comparison. Hence, each framework uses a benchmark implementation based on the same algorithm, taking the same input and writing the same output to HDFS. Although the algorithm remains unchanged, each framework employs an optimized version adapted to its available functionalities. Therefore, each benchmark uses a different implementation of the same algorithm in order to obtain the same result. Further details about each benchmark are given next.

WordCount: Counts the number of times each word appears in the input dataset. Both WordCount and its input data generator, RandomTextWriter, are provided as an example ("ex." in the table) in the Hadoop distribution. In the case of Spark and Flink, the source code has been adapted from their examples.

Grep: Counts the matches of a regular expression in the input dataset. It is included in the Hadoop distribution, and in the case of Spark and Flink it has been adapted from their examples. Its data generator is also RandomTextWriter.

TeraSort: Sorts 100 Byte-sized key-value tuples. Its implementation, as well as the TeraGen data generator, is included in Hadoop. However, TeraSort is not provided as an example for Spark and Flink, and so their source codes have been adapted from [16].

Connected Components: Iterative graph algorithm that finds the connected components of a graph. It is included in Pegasus [17], a graph mining system built on top of Hadoop. In the case of Spark and Flink, Connected Components is supported by Graphx [18] and Gelly [19], respectively, which are graph-oriented APIs. The input dataset is set up by using the DataGen tool, included in the HiBench [20] benchmark suite.

PageRank: Iterative graph algorithm which ranks elements by counting the number and quality of the links to each one. Pegasus includes PageRank for Hadoop, and the source codes for Spark and Flink have been adapted from their examples. Although there are implementations available in Graphx and Gelly, these versions did not improve the performance of the examples, and so they have not been used in the experiments. The input dataset of PageRank is also set up by DataGen.

K-Means: Iterative clustering algorithm that partitions a set of samples into K clusters. Apache Mahout [21] includes this algorithm for Hadoop and provides the dataset generator, GenKMeansDataSet, while Spark uses the efficient implementation provided by its machine learning library MLlib [22]. As the Flink machine library, Flink-ML, does not include an implementation of K-Means yet, its source code has been adapted from the example.

All these benchmarks are included in the last version (2.2) of our BDEv tool, which is available to download at http://bdev.des.udc.es.

D. Conducted evaluations

In order to perform a thorough experimental analysis, the evaluations have studied two different aspects: performance of the frameworks and impact of several configuration parameters.

The first set of experiments, shown in Section V-A, compares the performance and the strong scalability of the frameworks. For doing so, the benchmarks have been executed using 13, 25, 37 and 49 nodes. Each cluster size *n* means 1 master and *n-1* slave nodes. The input data size of each benchmark is shown in Table III.

Section V-B analyzes the impact of some configuration parameters on the overall performance of the frameworks. Experiments have been carried out using different HDFS block sizes, input data sizes, network interconnects and thread configurations with the maximum cluster size that has been considered (i.e., 49 nodes). Three benchmarks have been selected for these experiments: WordCount, TeraSort and PageRank that represent, respectively, three types of workloads: CPU bound, I/O bound, and iterative.

The HDFS block sizes that have been evaluated are 64, 128, 256 and 512 MB, using the same input data size as in Section V-A (see Table III). In the data size experiments, WordCount and TeraSort have processed 100, 150, 200 and 250 GB, whereas PageRank has processed 9, 12.5, 16 and 19.5 GB, which correspond with 15, 20, 25 and 30 million pages, respectively. The network interconnects that have been evaluated are GbE and IPoIB, configuring the frameworks to use each interface for shuffle operations and HDFS replications. These experiments (as well as the thread configuration experiments described later) have used the maximum data size considered, 250 GB for WordCount and TeraSort, and 19.5 GB for PageRank, in order to maximize their computational requirements.

The thread configurations of the frameworks determine how the computational resources of each node are allocated to the Java processes and threads. On the one hand, Hadoop distributes the CPU cores between mappers and reducers, which are single-threaded processes. The *#mappers/#reducers* configurations evaluated are 4/4, 5/3, 6/2 and 7/1. On the other hand, Spark and Flink use Workers and TaskManagers, respectively, which are multi-threaded manager processes that run several tasks in parallel. The *#managers/#cores per manager* configurations evaluated are 1/8, 2/4, 4/2 and 8/1.

V. Experimental results

This section presents the analysis of the evaluation of Hadoop, Spark and Flink in terms of performance and scalability (Section V-A), as well as the impact of configuration parameters (Section V-B). The graphs in this section show the mean value from a minimum of 10 measurements, while the observed standard deviations were not significant.

A. Performance and scalability

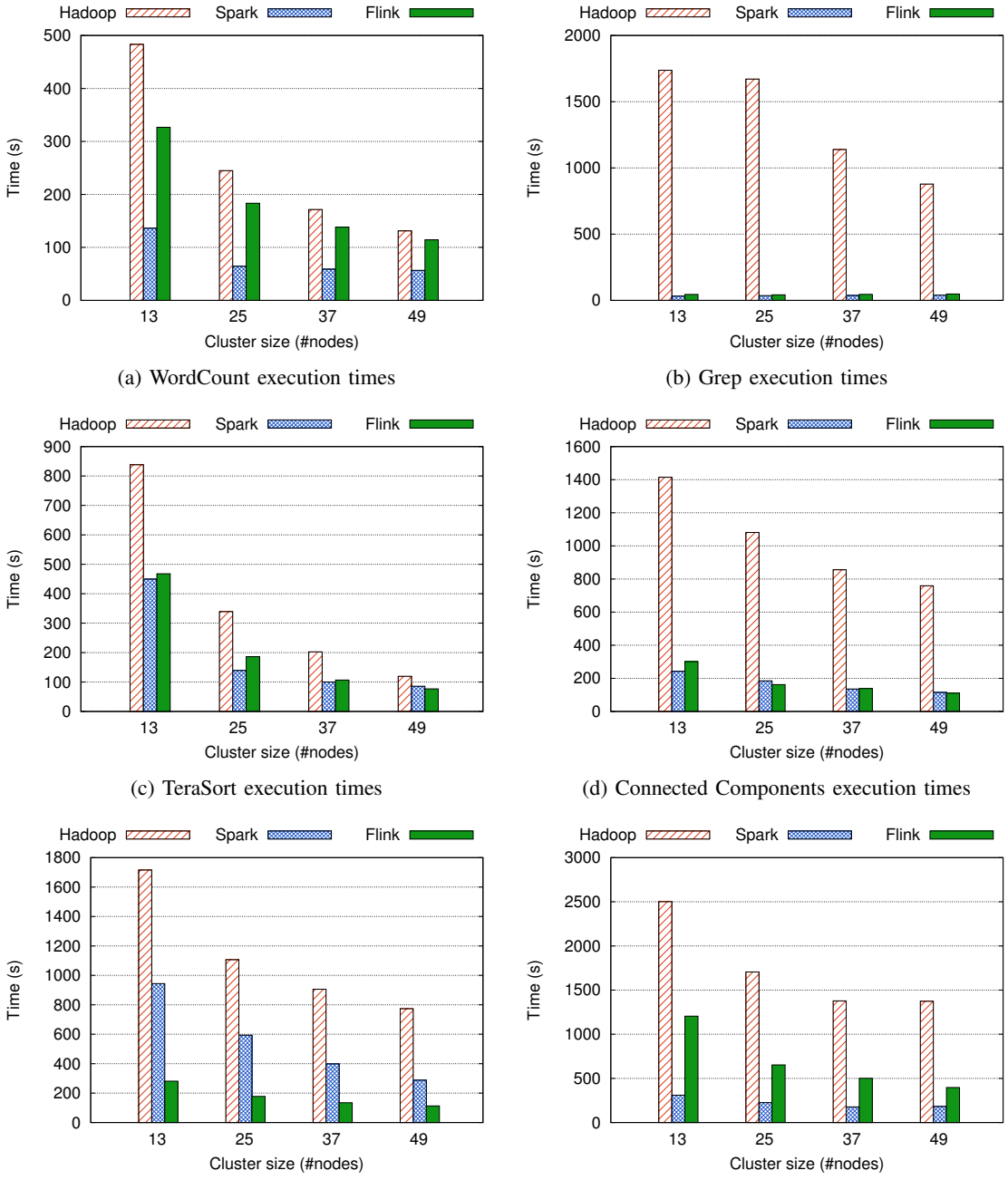
Execution times for all benchmarks are shown in Figure 1. As expected, these graphs demonstrate an important performance improvement of Spark and Flink over Hadoop. The comparison between Spark and Flink varies depending on the benchmark. With the maximum cluster size, Spark obtains the best results in WordCount and K-Means, while Flink is better for PageRank. Both obtain similar results for Grep, TeraSort and Connected Components.

In WordCount, Spark obtains the best results because of its API, which provides a *reduceByKey()* function to sum up the number of times each word appears. Flink uses a *groupBy().sum()* approach, which seems to be less optimized for this kind of workload. Furthermore, the CPU-bound behavior of WordCount makes the memory optimizations of Flink less significant compared to other benchmarks, and even introduce a certain overhead when computing the results.

In Grep, Spark and Flink widely outperform Hadoop due to several reasons. The most important is the inadequacy of the MapReduce API for this benchmark. In Hadoop, the benchmark uses two MapReduce jobs: one for searching the pattern and another for sorting the results. This produces a high number of memory copies and writes to HDFS. Spark and Flink take a different approach, selecting the matching input lines by means of a *filter()* function, without copying them. Next, the selected lines are counted and sorted in memory. Moreover, the pattern matching in Hadoop is performed within the *map()* function, which has only half of the CPU cores of the nodes. In Spark and Flink the parallelism of all the operations is set to the total number of cores in the cluster.

TeraSort is the benchmark which shows the smallest performance gap when comparing Hadoop with Spark and Flink. The main reason is that Hadoop was originally intended for sorting, being one of the core components of the MapReduce engine. Although Spark and Flink outperform Hadoop, its high scalability allows it to obtain competitive results, especially when using 49 nodes. Spark and Flink are in a statistical tie. A similar benchmark, Sort, has also been evaluated in the experiments. However, the results were very similar to those of TeraSort, and so they are not shown in the graphs due to space constraints.

The performance of Spark and Flink for iterative algorithms (Figures 1d-1f) is clearly better than that of Hadoop (up to 87% improvement with 49 nodes). As mentioned in Section IV-C, both frameworks provide optimized libraries for graph algorithms, Graphx and Gelly, obtaining very similar results for Connected Components. That is not the case of PageRank, whose implementation has been derived from the examples. In this benchmark, Flink obtains the best performance mainly due to the use of delta iterations, which only process those elements that have not reached their final



(a) WordCount execution times

(b) Grep execution times

(c) TeraSort execution times

(d) Connected Components execution times

(e) PageRank execution times

(f) K-Means execution times

Figure 1: Performance results

value. However, Spark obtains the best results for K-Means thanks to the optimized MLlib library, although it is expected that the support of K-Means in Flink-ML can bridge this performance gap.

To sum up, the performance results of this section show that, excluding TeraSort, replacing Hadoop with Spark and Flink can lead to a reduction in execution times by 77% and 70% on average, respectively, when using 49 nodes.

B. Impact of configuration parameters

This section shows how the performance of WordCount, TeraSort and PageRank is affected when modifying some of the configuration parameters of the experiments. Note that those parameters which were not specifically modified keep the values indicated in the experimental setup of Section IV (see Table II).

1) HDFS block size: The performance values when using different HDFS block sizes are shown in Figure 2. HDFS manages the data in blocks of a certain size, which determines the amount of data that is read in each task. In WordCount, the block size does not have a great impact on performance except for Hadoop, which obtains its best result with 128 MB. Spark achieves it with 64 MB, and Flink has almost the same result with 128 and 512 MB. TeraSort is the benchmark more affected by the block size, and the optimal value depends on the framework: 256 MB for Hadoop and 64 MB for Spark and Flink. In PageRank, Hadoop obtains its best results with 64 MB, with descending performance as the block size increases. In this benchmark, Spark and Flink are not affected by the HDFS block size. This is caused by the iterative behavior of PageRank. Hadoop stores the intermediate results (i.e., between iterations) in HDFS, and so it is more affected by its configuration. Spark and Flink, which store intermediate results in memory, are only influenced by the HDFS configuration when reading the initial input and writing the final output. Taking into account the results of the different benchmarks, the best option for each framework, on average, is 128 MB for Hadoop and Flink, and 64 MB for Spark.

2) Input data size: Figure 3 shows the performance when executing different problem sizes. Analyzing the scalability of WordCount, the slope is much steeper in Hadoop and Flink than in Spark, being Spark the most scalable framework. The same happens in TeraSort, enabling Spark to widen the gap with Flink as the input data size increases. Therefore, Spark is the most scalable framework for TeraSort, although it is not the best performer in the 100 GB case. For PageRank, Flink shows higher scalability than Hadoop and Spark, whose slope is steeper. As previously explained, Flink uses delta iterations to avoid re-processing the whole dataset. This, along with efficient memory management to avoid major garbage collections, makes Flink the most scalable framework for PageRank, obtaining execution times up to 6.9x and 3.6x faster than Hadoop and Spark, respectively.

3) Interconnection network: The performance of the frameworks when using GbE and IPoIB is shown in Figure 4. The impact of the network not only affects the internode communications during the shuffle phase, but also the writing operations to HDFS, which replicate the data blocks among the slave nodes. Generally, reading the input dataset does not imply any network traffic, as the data is read locally in each node. In WordCount, Spark is the only one which slightly benefits from using IPoIB. This is not the case of TeraSort, which is the most network-intensive benchmark under evaluation. Here, the use of IPoIB provides a performance improvement of up to 12%, 41% and 11% for Hadoop, Spark and Flink, respectively. In PageRank, Hadoop and Spark also improve their performance when using IPoIB, while Flink maintains the same values. Therefore, the high bandwidth provided by IPoIB seems to favor more the block-based data pipelining of Spark than the tuple-based Flink counterpart.

4) Thread configuration: Figure 5 displays the performance of the frameworks with different thread configurations. In Hadoop (see Figure 5a), the best configuration is 4 mappers and 4 reducers, except for WordCount, which is 7 mappers and 1 reducer. This is caused by the CPU-bound behavior of WordCount, where most of the computation is performed by mappers, and so increasing their number reduces the execution time. In Spark, 1 Worker with 8 cores is the best configuration except for PageRank, which is 4 Workers with 2 cores. Spark employs the same JVMs to compute the different iterations of PageRank, involving a lot of object creations/destructions. Therefore, the use of smaller JVMs reduces the overhead of garbage collection stops. However, it also decreases the parallelism within each Worker and replicates some of the services computed in them, like shuffle managers, causing the 8/1 configuration to perform clearly the worst. Although performance is poorer with this configuration, Spark executes all the benchmarks successfully. That is not the case of Flink, where the results for 8 TaskManagers and 1 core are not shown, as the experiments with this configuration did not finished successfully (not enough memory). It seems that Flink is not well suited to use small JVM sizes. The best configuration for Flink is 2/4 for WordCount and PageRank, while 1/8 for TeraSort. In Flink, iterative algorithms are not so affected by garbage collections, due to the memory optimizations conducted in order to avoid the creation/destruction of Java objects.

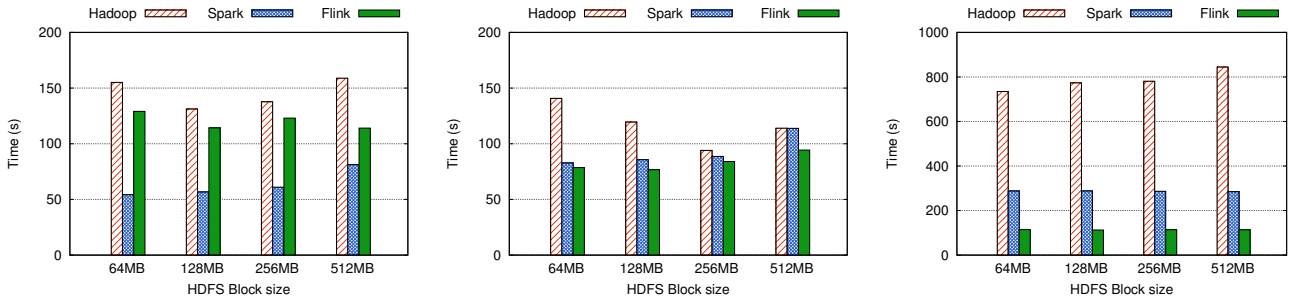
VI. Conclusions and Future Work

This paper has thoroughly evaluated Hadoop, Spark and Flink in terms of performance and scalability, taking also into account different framework configurations. The results show that replacing Hadoop with Spark or Flink can lead to significant performance improvements. However, the effort needed to rewrite the source code of MapReduce workloads to adapt them to the new APIs must be taken into account.

Currently, Spark is the framework which provides the best results in general terms. It is now a much larger Apache project comparatively to Flink, being a more mature framework and superior in terms of market share and community. Moreover, it includes a richer set of operations and a wide range of tools compared to Flink.

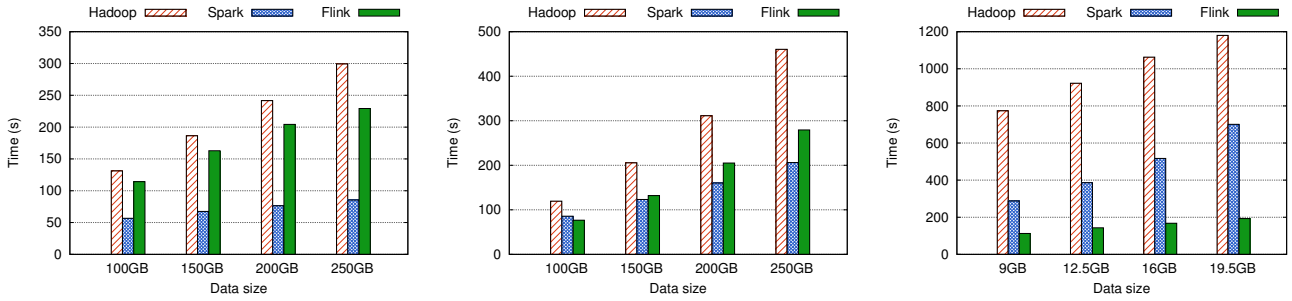
However, Flink has definitely contributed with several interesting and novel ideas, some of which are being adopted by Spark. The transparent use of persistent memory management using a custom object serializer for Flink operations minimizes the overhead of garbage collections. Furthermore, iterative algorithms can greatly benefit from the use of explicit iterators (e.g., for PageRank execution times are up to 6.9x and 3.6x faster than Hadoop and Spark, respectively).

As future work, we plan to investigate further on the impact of more configuration parameters on the performance of



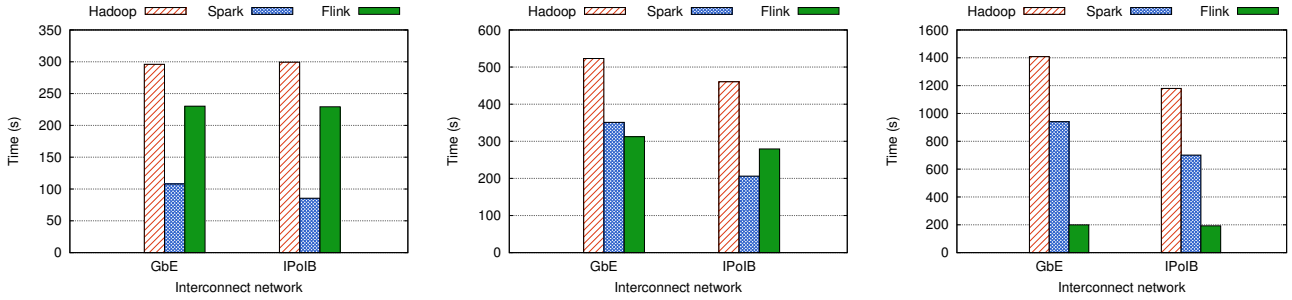
(a) WordCount (b) TeraSort (c) PageRank

Figure 2: HDFS block size impact



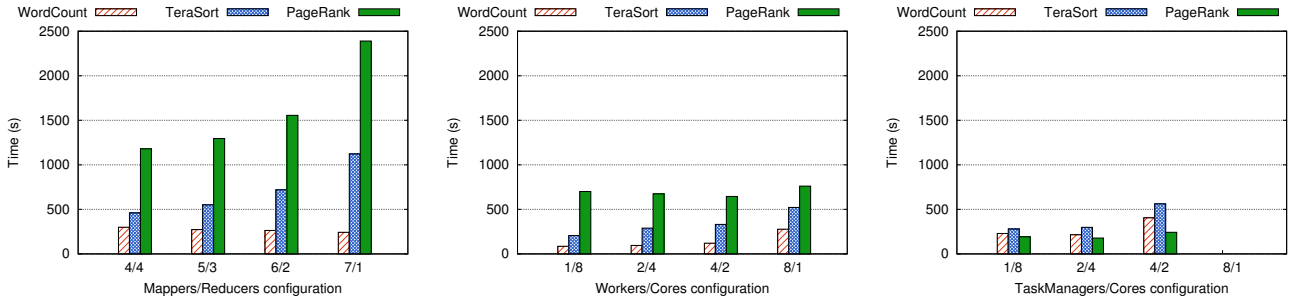
(a) WordCount (b) TeraSort (c) PageRank

Figure 3: Input data size impact



(a) WordCount (b) TeraSort (c) PageRank

Figure 4: Network impact



(a) Hadoop (b) Spark (c) Flink

Figure 5: Analysis of different thread configurations

these frameworks (e.g., spilling threshold, network buffers). We also intend to carry out a similar evaluation but focusing on streaming workloads using Spark, Flink and other recent streaming frameworks.

ACKNOWLEDGMENT

This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the European Union (TIN2013-42148-P) and by the FPU Program of the Ministry of Education (FPU14/02805). We thankfully acknowledge the Advanced School for Computing and Imaging (ASCI) and the Vrije University Amsterdam for providing access to the DAS-4 cluster.

REFERENCES

[1] Apache Hadoop, http://hadoop.apache.org/, [Last visited: July 2016].

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño, "Analysis and evaluation of MapReduce solutions on an HPC cluster," *Computers & Electrical Engineering*, vol. 50, pp. 200–216, 2016.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. of the 2nd USENIX Conference on Hot topics in Cloud Computing (HotCloud'10)*, Boston, USA, 2010, pp. 1–7.

[5] A. Alexandrov *et al.*, "The Stratosphere platform for Big Data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[6] P. Jakovits and S. N. Srirama, "Evaluating MapReduce frameworks for iterative scientific computing applications," in *Proc. of the International Conference on High Performance Computing & Simulation (HPCS'14)*, Bologna, Italy, 2014, pp. 226–233.

[7] J. Shi *et al.*, "Clash of the titans: MapReduce vs. Spark for large scale data analytics," in *Proc. of the Very Large Data Bases (VLDB) Endowment*, vol. 8, no. 13, 2015, pp. 2110–2121.

[8] N. Spangenberg, M. Roth, and B. Franczyk, "Evaluating new approaches of Big Data analytics frameworks," in *Proc. of the 18th International Conference on Business Information Systems (BIS'15)*, Poznań, Poland, 2015, pp. 28–37.

[9] M. Bertoni, S. Ceri, A. Kaitoua, and P. Pinoli, "Evaluating cloud frameworks on genomic applications," in *Proc. of the 2015 IEEE International Conference on Big Data (IEEE BigData 2015)*, Santa Clara, USA, 2015, pp. 193–202.

[10] S. Chintapalli *et al.*, "Benchmarking streaming computation engines: Storm, Flink and Spark streaming," in *Proc. of the 1st IEEE Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM'16)*, Chicago, USA, 2016.

[11] J. Samosir, M. Indrawan-Santiago, and P. D. Haghighi, "An evaluation of data stream processing systems for data driven applications," in *Proc. of the International Conference on Computational Science (ICCS'16)*, San Diego, USA, 2016, pp. 439–449.

[12] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking Modern Distributed Streaming Platforms," in *Proc. of the 2016 IEEE International Conference on Industrial Technology (ICIT 2016)*, Taipei, Taiwan, 2016, pp. 592–598.

[13] M. Zaharia *et al.*, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, USA, 2012, pp. 15–18.

[14] H. Bal *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[15] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño, "MREv: An automatic MapReduce Evaluation tool for Big Data workloads," in *Proc. of the International Conference on Computational Science (ICCS'15)*, Reykjavík, Iceland, 2015, pp. 80–89.

[16] TeraSort for Apache Spark and Flink, https://github.com/eastcirclek/terasort, [Last visited: July 2016].

[17] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. of the 9th IEEE International Conference on Data Mining (ICDM'09)*, Miami, USA, 2009, pp. 229–238.

[18] Spark Graphx: Apache Spark Graph API, http://spark.apache.org/graphx/, [Last visited: July 2016].

[19] Flink Gelly: Apache Flink Graph API, https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/gelly.html, [Last visited: July 2016].

[20] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW'10)*, Long Beach, USA, 2010, pp. 41–51.

[21] Apache Mahout: Scalable machine learning and data mining, http://mahout.apache.org/, [Last visited: July 2016].

[22] Spark MLlib: Apache Spark's scalable Machine Learning library, http://spark.apache.org/mllib/, [Last visited: July 2016].