Spark-DIY: A Framework for Interoperable Spark Operations with High Performance Block-Based Data Models

Silvina Caíno-Lores*✉, Jesús Carretero*, Bogdan Nicolae†, Orcun Yildiz†, and Tom Peterka†
* *Computer Architecture and Technology Area (ARCOS)*
Department of Computer Science and Engineering, University Carlos III of Madrid
Leganés, Spain
Email: {scaino,jcarrete}@inf.uc3m.es
†*Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA*
Email: bnicolae@anl.gov,oyildiz@anl.gov, tpeterka@mcs.anl.gov

***Abstract*—Today scientific applications are increasingly relying on a variety of data sources, storage facilities, and computing infrastructures, and there is a growing demand for data analysis and visualization for these applications. In this context, exploiting Big Data frameworks for scientific computing is an opportunity to incorporate high-level libraries, platforms, and algorithms for machine learning, graph processing and streaming; inherit their data awareness and fault-tolerance; and increase productivity. Nevertheless, limitations exist when Big Data platforms are integrated with an HPC environment, namely poor scalability, severe memory overhead, and huge development effort. This paper focuses on a popular Big Data framework –Apache Spark– and proposes an architecture to support the integration of highly scalable MPI block-based data models and communication patterns with a map-reduce-based programming model. The resulting platform preserves the data abstraction and programming interface of Spark, without conducting any changes in the framework, but allows the user to delegate operations to the MPI layer. The evaluation of our prototype shows that our approach integrates Spark and MPI efficiently at scale, so end users could can take advantage of the productivity facilitated by the rich ecosystem of high-level Big Data tools and libraries based on Spark, without compromising efficiency and scalability.**

***Keywords*-HPC, Big Data, Spark, MPI, High-Performance analytics, programming environments.**

I. Introduction

Convergence between high-performance computing (HPC) and Big Data is now an established research area that has spawned new research topics such as data-intensive scientific computing, high-performance data analytics, and hybrid platforms and infrastructures based on virtualization techniques and novel storage hierarchies. Therefore, industry-wide consortia as ETP4HPC [1] and BDV [2], and the international scientific HPC community [3], [4] have recognized new opportunities in unifying the platform layer and data abstractions for both HPC and Big Data. In this context, there exists an opportunity to incorporate existing high-level libraries and algorithms for machine learning and data streaming and inheriting the data awareness and fault-tolerance mechanisms of Big Data frameworks and applying them to scientific computing.

Nonetheless, Big Data and HPC frameworks today remain largely incompatible: programming models and software development tools are inconsistent [5]; trying to mix both models out-of-the-box generates memory overheads and poor scalability in a HPC environment [6]; the disparity between collocated and distributed storage architectures in Big Data and HPC systems, respectively, degrades performance when running Big Data applications on HPC systems [7]; and the usage of merged Big Data models presents limitations, such as high memory consumption and low efficiency in communication between cooperating processes [8].

In previous works, the authors analyzed several use cases and compared frameworks and platforms to conclude that tools like Apache Spark provide an interesting baseline for intregration of scientific simulations in Big Data environments comprising clouds and sensor networks. However, the data abstractions and application model of Spark are not easily supported using MPI, which is the main programming model in HPC [9]

To overcome these problems, in this paper we introduce a framework named *Spark-DIY*, that allows the usage of native Big Data programming models using the highly-scalable data-intensive communication pattern library DIY (Do It Yourself Block Parallelism) [10]. Spark-DIY runs on top of MPI to enable the execution of data analysis applications in a supercomputer. Spark-DIY can also be used to assist in the integration of existing scientific codes into a Big Data environment. Consequently, our main goal is to preserve the usability and flexibility of Big Data tools.

The main contributions of this paper are the implementation of our Big Data-HPC framework, the definition of an interoperable data model between Spark and DIY, and the ability to offload parts of the Spark application to DIY, which can also be used to incorporate MPI programs into such application. The *Spark-DIY* framework allows users to move data freely between Spark and DIY data structures while maintaining the Spark programming model. The user can choose which tasks will run natively on Spark, and which

ones will be delgated to the DIY layer.

The rest of this paper is organized as follows: Section II motivates our proposal by depicting the needs of a real application example. Section III introduces the main aspects of Spark and DIY, and develops the open challenges regarding the convergence of Big Data and HPC paradigms. Section IV describes the design and architecture of the proposed framework. Section VI evaluates how Spark-DIY performs against Spark as the problem size scales. Section VII analyzes relevant work in the literature targeting similar goals, and Section VIII summarizes the contributions of this paper and previews directions for future work.

II. Motivation

Although the convergence problem is interesting by the many technical challenges it supposes, first we focus on why convergence is actually needed. To clarify this and motivate the scope of our work, we introduce a data assimilation use case from the hydrogeology domain (EnKF-HGS).

EnKF-HGS is a tool used to predict the state of hydrogeological systems (precipitation, surface water, etc.). Besides specific models for pre-alpine valleys in the Swiss Emmental region, the tool can also incorporate sensor data to refine these predictions, thus resulting in an iterative data assimilation process. Figure 1 depicts the elements involved in EnKF-HGS operations: the user provides a base model that will be distributed, simulated with EnKF-HGS kernels, and updated with the data fed by the sensor network; after each step, results are stored in a distributed manner in cloud storage for subsequent iterations.

This use case relies on cloud services for computation, data assimilation and storage. In addition, Big Data computing frameworks constitute a natural fit for EnKF-HGS because they provide facilities to collect data from streaming sources. On the other hand, this tool must handle many MPI simulations running in parallel, and high-performance is required as in any other scientific application. The combination of these requirements and features makes a case for the need of convergence for the family of scientific applications represented by EnKF-HGS. Similar data assimilation tasks using ensemble Kalman filters to fuse sensor and simulation data include weather forecasting [11], and carbon cycle [12] studies.

In previous works [13] we reported our experience combining traditional HPC with Big Data-inspired paradigms and platforms, in the context of scientific ensemble workflows like EnKF-HGS. Our goal was to provide a suitable environment that combined the HPC and Big Data elements required by EnKF-HGS, so we integrated the simulation kernels with the Apache Spark framework, which also supports streaming. We found that Spark was unable to scale due the memory and communication requirements of the kernels during the shuffle phase, combined with the platform's overhead. Similar results have been corroborated by other researchers such as [14], [15], who identified inefficiencies of Spark shuffle like explosion of files, high I/O contention, and TTL cleaner overhead.

In this context, achieving a data model fully compatible for Spark and MPI that provides scalability, performance and interoperability suitable for scientific data assimilation remains a challenge not fully satisfied by any existing platform, and this is the goal of our framework.

III. Background

A. Spark

Spark is arguably the most popular Big Data processing framework for data analysis, and it also supports numerous other tools for machine learning, graph analytics, and stream processing, among others. Being initially inspired by the Map-Reduce model, Spark supports extended functionality and operates primarily in memory by means of its core data abstraction: the *resilient distributed dataset* (RDD)[16]. An RDD is a read-only, resilient collection of objects partitioned across multiple nodes that holds provenance information (lineage) and can be rebuilt in case of failures by partial recomputation from ancestor RDDs. RDDs are by default ephemeral, which means that once computed and consumed, they are discarded from memory. However, since some RDDs might be repeatedly needed during computations, the user can explicitly mark them as persistent, which moves them in a dedicated cache for persistent objects.

Two types of operations can be executed in Spark: *transformations* that execute a function independently in each partition, and *actions* that trigger data shuffles between the partitions. Transformations are executed in a lazy manner and are triggered by actions. The operations that are contained between two communication points are called *stages*.

B. DIY

DIY is an MPI-based library that offers efficient and highly scalable communication patterns over a generic block-based data model. In DIY, algorithms are written in terms of data blocks that constitute the basic units of domain decomposition and parallel work. Blocks are linked forming neighborhoods that represent the domain in a distributed manner. The assignment of blocks to MPI processes, often multiple DIY blocks per MPI rank, is controlled by the DIY runtime transparent to the user. Given a block decomposition and assignment to MPI processes, the user is able to run reusable communication patterns between local blocks in a neighborhood and global operations such as reductions over all blocks. Therefore, DIY users can execute common communication patterns just by defining the block type and domain topology, without knowledge of the underlying communication details. Thus, for analytics, one can decompose the analysis problem among a large number of data-parallel sub-problems and efficiently exchange data among

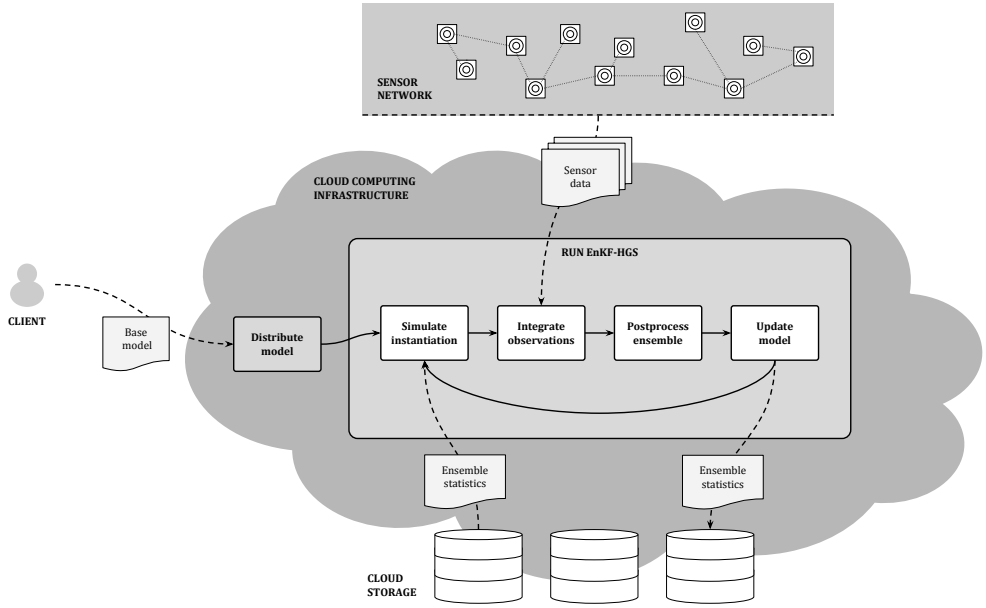


Figure 1. Interoperation of EnKF-HGS with the data assimilation sensor network and its supporting cloud infrastructure.

them using regular local and global communication patterns whose implementation has been tuned for HPC.

DIY has demonstrated efficient scaling on leadership-class supercomputers in a diverse array of science and analysis codes, including cosmology, molecular dynamics, nuclear engineering, astrophysics, combustion, and synchrotron light source imaging. For example, benchmarks of strong and weak scaling of parallel Delaunay tessellations [17], one of the libraries built on top of DIY, demonstrated parallel efficiency of over 90% on up to 128K MPI processes.

The similarity between Spark RDDs and DIY block parallelism, and the resemblance between Spark map-shuffle-reduce and DIY merge-reduce communication patterns are the basis for our integration of these two models.

IV. Spark-DIY Architecture

Our approach is to integrate Spark with DIY, without enforcing the usage of one model or the other, by allowing the user to freely switch between the two models and select the one that adapts better to each stage of the problem. There are three motivations for pursuing the interoperability between Spark and DIY. (1) Spark users can use HPC platforms to scale their workloads; (2) HPC users gain access to Spark libraries and associated projects, which increases productivity and interoperability with other elements in the Big Data ecosystem; and (3) both types of users can benefit from additional data patterns exposed by DIY (e.g. local neighborhood exchange).

Guided by our objective to offer the user the best features of both computing models, we formulate the following design goals for the integrated framework.

Interoperability: DIY and Spark target different canonical problems; therefore adapting a problem from Spark to DIY and vice versa should be explicit. To make the user aware of which model is currently active, we keep both platforms separated, but interoperable through explicit conversions.

Production-readiness: We believe that the viability of our solution depends on being able to use standard versions of Spark and DIY without any changes required to those platforms. Thus, the adaptation must be made to both of them using a middleware layer, transparently to the user, so that applications for Spark or DIY should run almost immediately.

Usability: Although the user must be aware of the explicit interoperability (including overheads associated with switching contexts), the knowledge of the underlying data model should be minimal to preserve the nature of the Spark programming and data interface. This would reduce the learning curve and minimize the impact in existing code.

Flexibility: We want to support multiple data types and provide flexibility for different datasets to coexist in the same application.

Performance: The data locality capability of Spark is one of its key features and must be enforced as much as possible. On the other hand, the efficiency and scalability of the communication patterns of DIY should be exploited whenever possible to accelerate communication-intensive (e.g., shuffle) operations.

Given the previous design goals, three aspects of Spark and DIY need to be connected: data abstraction, program-

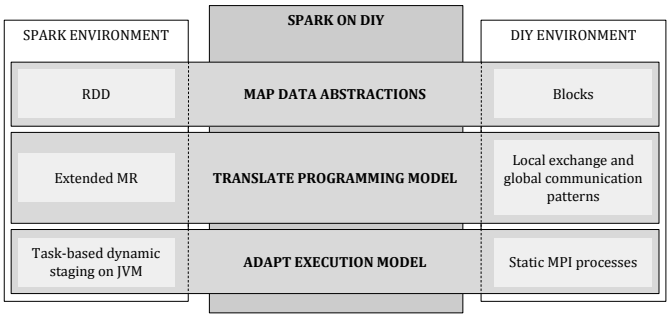


Figure 2. Interoperation Mechanisms between SPARK and DIY

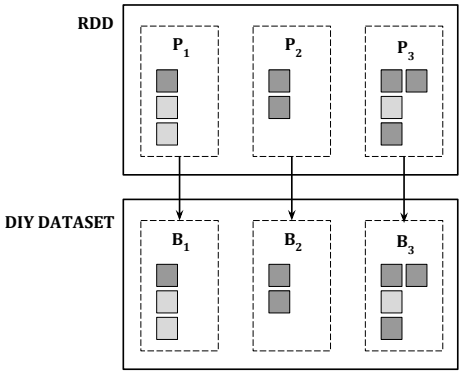


Figure 3. Mapping of data partitions in an RDD to DIY blocks

ming model, and execution model (see Fig. 2). The adaptations needed to connect each of these components between the two models are detailed below.

A. Data Abstraction

The first aspect that must be aligned is the way in which both frameworks represent their data abstractions. Both in the case of Spark and DIY, the way data are arranged determines the development of algorithms and the behavior of the runtime.

Since preserving the RDD abstraction of Spark is key to maintaining the usability and interoperability with upper layers, it is necessary to map RDDs to a block-based data structure in DIY. If we think of the RDD as the equivalent of the global DIY (distributed) domain, the data partition in a RDD maps directly to a data block in DIY. In this context, the RDD dataset is partitioned into independent DIY blocks, as shown in Fig. 3, where each partition P_i maps to a corresponding block B_i, preserving the same data elements inside the partition and respecting locality, since no data transfers occur to build the DIY dataset. As a consequence, the DIY dataset constitutes a distributed collection that reflects the inner structure of an RDD, while adding topology information for the DIY-based communication patterns. Data are moved among Spark and DIY, transparently holding the bindings for each partition and interacting with the Spark context to control partitioning.

B. Programming Model

Once the data abstractions are mapped, the translation of the programming model from the Spark interface to the underlying DIY communication patterns follows naturally. The way we mapped data abstractions facilitates the algorithmic mapping because we are able to preserve the independence between partitions and map data shuffles to underlying DIY communication patterns.

Spark operations on RDDs are internally expressed as algorithms built on top of DIY patterns to mimic the functionality expected from Spark. For example, the *map* and *filter* transformations in Spark can be translated to a *foreach* pattern in DIY, since both of them represent parallel and independent operations on the dataset; *reduceByKey* in Spark was translated to an algorithm based on the *swap-reduce* DIY pattern, which conducts several rounds of data exchanges between blocks, effectively shuffling data across the partitions; analogously, Spark's *reduce* corresponds to a *merge-reduce* pattern, similar to *swap-reduce* but merging the results in a single value.

To preserve the programming interface of Spark as much as possible, operations on partitions are triggered by the inner algorithms in DIY, but expressed as user-defined callbacks written by the user in Scala, who also defines the data type of the records and the supported operators (e.g. *unary* for independent transformations, *binary* for reductions, and *hash* for partitioning).

C. Execution Model

Besides translating the programming model into DIY patterns, it is necessary to provide the proper execution support. In this particular case, we must connect the dynamic task-based execution model from the Spark framework to the set of MPI processes that DIY assumes to exist at the beginning of its execution.

To achieve this, we wrap each Spark worker into an MPI process that forms a basic communicator for DIY. Since executors are spawned inside these processes, we can update the global communicator to include these children processes using MPI, in a similar way as depicted in Spark-MPI [18], a solution that extends the Spark ecosystem with the MPI applications using the Process Management Interface to allow the creation of MPI processes from Spark. Our work builds on this solution incorporating the data abstractions and topologies offered by DIY.

V. Deployment and Usage

Figure 4 shows the interaction between the main components of the proposed architecture. The following sections explain their role from the end user's perspective, and the accompanying internal behavior of the system.

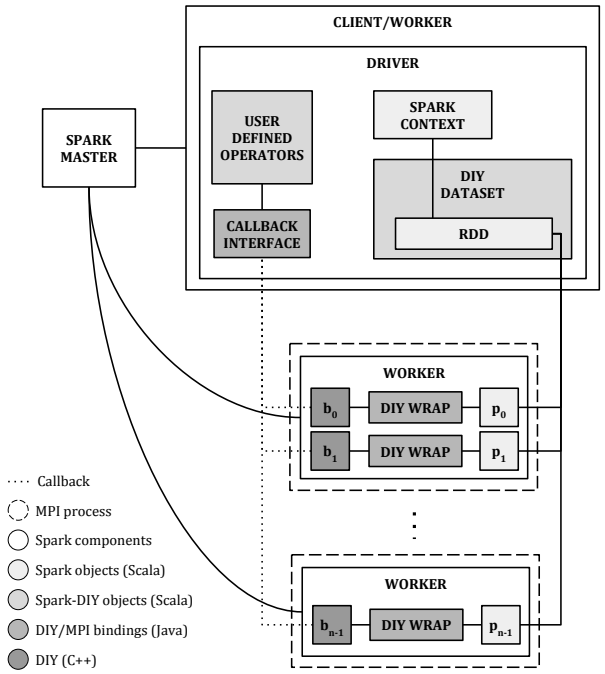


Figure 4. Architecture overview including Spark core objects and deployment units in cluster mode, Spark-DIY binding elements, and DIY MPI processes and block distribution.

A. User View

The end user is exposed to a limited number of additional elements of the interoperation layer in addition to the basic Spark interface. The driver code of the Spark application (in Scala or Java) must define and use these components as follows:

1) Select the record data type: The RDDs to be processed through DIY are collections of data records that we can convert to C++ data types through the Java Native Interface (JNI). To ease this process, a catalogue is offered where users can select a pre-built data type that handles type conversion and memory management from and to the C++ code. Since users may want to use a custom data type not present in the catalogue, we have also developed the internals of Spark-DIY in a generic manner. New data types can be defined in a helper file later used by the JNI code generation utility of choice, which is SWIG in our particular case. New data types must define a serialization function since both RDD and DIY block elements need to be serializable. In addition, collections of primitive data types (*byte*, *short*, *int*, *long*, *char*, *float*, *double*) are offered with reduced overhead, since serialization between Spark and DIY is not required. This is especially useful for scientific tasks, since most of the data there are numeric.
2) Define the callback operators for the record: Similarly as in Spark, the operations to be conducted on the data must be defined. In order to access these operators from DIY, users must implement the proper method as an object that extends the callback interface. For example, the interface exposes a unary operator for map-like transformations, a binary operator for reductions, and a hash operator for partitioning. Table I shows examples of Spark-DIY function invocation in simplified Scala code.
3) Delegate execution on a DIY dataset: A DIY dataset contains an RDD and mimics the operations the user would normally run on the RDD. Once an RDD is created along with its operators, we can run the Spark-equivalent transformations and actions implemented using the communication patterns of DIY, running on MPI. The result of this operation is a new RDD that can be further used in the driver with subsequent combinations of Spark functions or DIY algorithms.

B. Internals

Upon invoking a function that is delegated to DIY, several tasks are conducted internally to pass data from the Java to the C++ side:

1) Spawn executors: Since DIY algorithms are block-parallel, we exploit the one-to-one association between each partition of an RDD and the corresponding block in the DIY domain. We let Spark handle data serialization, partitioning, and executor creation by wrapping the partition-block conversion in a function that is passed to a *mapPartitions* Spark operator. This creates executors that live in the MPI environment and contain the data of the corresponding partition, which enforces locality. For datasets containing primitive data types, data are shared between the partition and the DIY block, which reduces the number of copies conducted during the delegation process.
2) Convert each partition to a DIY block: The partition set is converted to a DIY domain, where each partition corresponds to a block. Transformations can be conducted with independent blocks following a similar approach to the Spark counterpart, while shuffle operations are translated to DIY communication patterns.
3) Delegate algorithm to DIY: Once the domain is established, we can run the DIY operations through a wrapper in JNI that executes the user-defined callbacks for computation. The results are retrieved afterwards and converted back to an RDD, and the execution is resumed in Spark.

VI. Evaluation

We have evaluated a prototype of the framework on bare metal nodes of the Chameleon cloud at the University of Chicago. Each node has an Intel Xeon CPU E5-2670v3@2.30GH processor with 12 physical cores and

Table I
COMPARISON OF SPARK AND SPARK-DIY USAGE FOR *map*, *filter*, *reduce* AND *reduceByKey* OPERATIONS.†

Spark	Spark-DIY
`map(x => f(x))`	`Callback extends DIYCallback { override unary(x) = {f(x)} } map_DIY(new Callback())`
`filter(x => f(x))`	`Callback extends DIYCallback { override unary(x) = {f(x)} } filter_DIY(Callback())`
`reduce((x,y) => f(x,y))`	`Callback extends DIYCallback { override binary(x,y) = {f(x,y)} } reduce_DIY(Callback())`
`reduceByKey((x,y) => f(x,y))`	`Callback extends DIYCallback { override binary(x,y) = {f(x,y)} } reduceByKey_DIY(Callback())`

†The syntax is purely illustrative and does not reflect minor Scala-specific details.

135GB of RAM each. Both the Spark and Spark-DIY clusters were configured with single-core workers to limit the number of executors in order to obtain a fair comparison against the MPI deployment. Therefore, each executor is mapped to one worker, and each worker is mapped to a MPI process.

As indicated by our use case analysis, and also by the literature, communication-intensive operations generate most of the scalability issues. For example, EnKF-HGS makes extensive use of reductions in the post-processing stage, since the simulation results of each instantiation of the model need to be shared among them.

Therefore, our experiments will focus on *reduceByKey* operations, as a canonical example of a Spark operation requiring shuffles. We evaluated Spark-DIY for *reduceByKey* on synthetic data generated in the driver that is evenly distributed across a number of partitions, which is equal to the number of workers in the deployment. Results for the generic and primitive type implementations of *reduceByKey* in Spark-DIY are shown in comparison with Spark's native method as the number of workers varies from 8 to 128.

Weak scaling was tested on a dataset holding a constant problem-per-worker of two million records per partition. The objective is to determine how the behaviour of both frameworks evolves as communication for data distribution increases between workers. Additionally, strong scaling was analyzed on datasets ranging 8 to 128 millions of records in total in order to assess the impact of the partition size on the execution time.

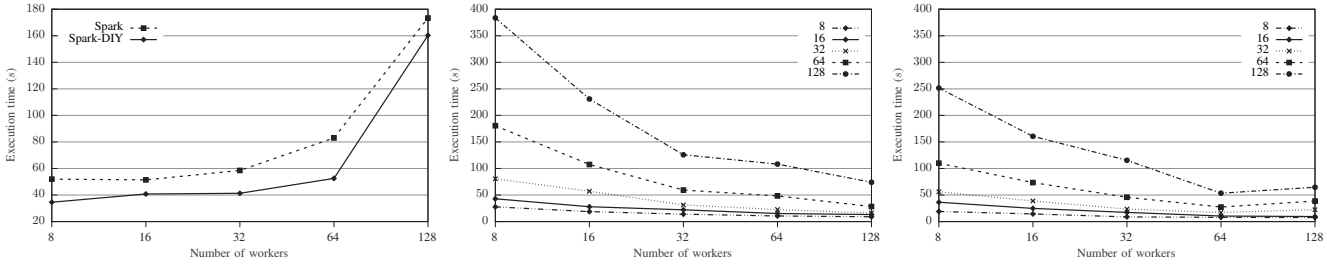
A. Evaluation of Generic Data Types

Figure 5 depicts the evaluation results for *reduceByKey* on *(string, integer)* pairs, thus showing the behaviour of the generic implementation of Spark-DIY against a Spark application that uses the same data interface. As indicated by (a), Spark-DIY offers competitive performance and a similar scaling trend against Spark, although they both fail to scale linearly as the problem size increases. Besides preserving the scaling trend of Spark, Spark-DIY reduces the execution time an average of 25.6%, but this improvement is reduced in the case of 128 workers and 256 millions of records. This shared trend and the reduction in the speed-up provided by Spark-DIY indicates an issue in the Spark platform, which is in charge of parallelization and task generation in both cases, and this is the price we pay for keeping compatibility and native Spark and DIY frameworks unmodified.

Since we have shown the behavior of the Spark-DIY *reduceByKey* is comparable to the Spark counterpart, we now focus on its scalability as the problem size increases for a fixed number of workers. Figure 5 shows execution times as the number of workers and the problem size increases for Spark (b) and Spark-DIY (b). The beneficial effects of DIY communication can be clearly appreciated in the figure, in comparison to the lower scale cases. As seen in the weak scaling results, data parallelization and task management take a large portion of the overall execution time. Therefore, Spark-DIY operations are meaningful in those cases where there is communication involved, and it represents a significant portion of the problem. This effect is clearer as the dataset size increases, which again is a good feature of Spark-DIY, as it is intended for very large datasets.

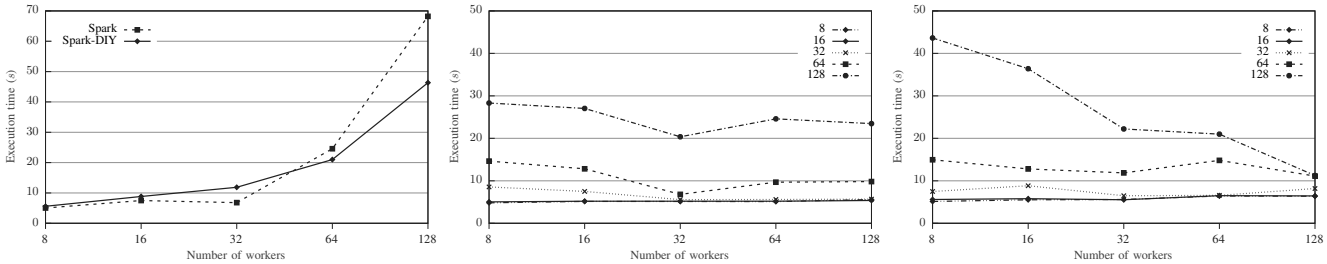
B. Evaluation of Primitive Data Types

Although the results in the previous section show promising performance, even considering the need for interoperability, a pure Spark application written with data types native to the selected programming language will deliver much better performance since less conversion and serialization



(a) Weak scaling. (b) Strong scaling (Spark). (c) Strong scaling (Spark-DIY).

Figure 5. Evaluation results for Spark and generic Spark-DIY in terms of weak scaling (a) and strong scaling with variable dataset size (b) and (c). Records are collections of strng-integer pairs.



(a) Weak scaling. (b) Strong scaling (Spark). (c) Strong scaling (Spark-DIY).

Figure 6. Evaluation results for Spark and Spark-DIY for primitive data types in terms of weak scaling (a) and strong scaling with variable dataset size (b). Records are collections of 4-byte integers.

steps would be needed. With this in mind, and considering that our target use cases (namely applications from the scientific domain) typically rely on primitive data types, we now compare a native Spark implementation against a Spark-DIY implementation using the optimizations for primitive data types described in Sec. V-A. These experiments using *reduceByKey* on *(string, integer)* pairs are reflected on Fig. 6.

Interestingly, the scaling curves of both platforms do not indicate the same trend as it occurred in the generic case. Although at a smaller scale Spark performs better, Spark-DIY delivers 14.66% and 32% less execution time for 64 and 128 workers respectively. This is also supported by the strong scaling results portrayed in (b) and (c): Spark shows a flat curve, which contrasts with the rough slope in Spark-DIY for 128 millions of records. As a result, Spark-DIY is 52.1% and 14.6% faster than Spark using 128 and 64 workers respectively, but slower if the number of workers is less.

Consequently, there is a trade-off between the scale of the problem and the platform and the interoperability and performance expected by the end user. At scale, performance is at least equivalent and the user still has the flexibility and interoperability offered by Spark-DIY, which enables the usage of higher level libraries and other associated services like streaming within their HPC toolchain.

VII. Related Work

Instances of Big Data and HPC convergence are evident in the literature of computer science, physical sciences, and business. Malitsky et al. [18] developed Spark workflows over MPI to parallelize and visualize reconstructions of synchrotron light source X-ray microscopy. PayPal relies on the high concurrency and low latency of HPC systems for fraud detection in Big Data [19]. Convergence in the opposite direction—big data tools for HPC applications—also appears, for example, in the usage of machine learning libraries, specifically TensorFlow, for HPC ptychographic reconstruction by Nashed et al. [20] or in the creation of a MapReduce framework over MPI, called Trace, for tomographic reconstruction [21].

Because Spark underlies many Big Data tools, the performance of Spark for scientific computing has been studied in several works. A study on Kira [22], a flexible and distributed astronomy image processing toolkit using Apache Spark, showed that Spark may be an alternative to an equivalent C program for many-task applications. The performance of a Spark implementation of a classification algorithm in the domain of High Energy Physics (HEP) was evaluated in [23], showing good scalability, but poor performance was compared with the results of an untuned MPI implementation of the same algorithm.

To overcome the former problems, three main approaches have been proposed: developing tailored frameworks, imple-

menting a MapReduce framework using MPI, and executing the Spark framework using MPI as the communication engine.

Several tailored MapReduce and data analytics frameworks have been developed. All of them target a particular family of applications or processor architecture, but they are not generalized for reuse in other contexts. A preliminary work was ROOT [24], an object-oriented C++ high-energy physics (HEP) framework designed for storing and analyzing petabytes of data efficiently by using a TTree object container optimized for statistical data analysis over very large data sets. A proposal to accelerate Spark communication was presented in [25], which used a high-performance RDMA-accelerated data shuffle in the Spark framework on high-performance networks and provided a performance improvement of 80%. An adaptation of the MapReduce framework for specific heterogeneous architectures has been proposed in IBMSparkGPU [26], but it is valid for local tasks only. Trace [21], mentioned earlier, is a high-throughput tomographic reconstruction engine for large-scale datasets using both (thread-level) shared memory and (process-level) distributed memory parallelization using a special data structure called a replicated reconstruction object. Fox et al. studied in [27] various frameworks for deep learning networks that can scale across multiple machines with full parallel support and distributed execution, such us Tensorflow, CNTK, Deeplearning4j, MXNet, H2O, Caffe, Theano, and Torch.

There are some implementations of MapReduce frameworks using MPI. Plimpton et al [28] created a parallel library written with message-passing (MPI) calls that allows algorithms to be expressed in the MapReduce paradigm, simplifying programming by using map and reduce operations callable from C++, C, Fortran, or scripting languages such as Python . Wang et al. [29] proposed a MapReduce-like framework, called Smart, to execute data analytics algorithms online alongside computational simulations (in situ analytics) in time-sharing or space-sharing modes. A more recent MapReduce framework over MPI is Mimir [30], which provides a redesign of the execution model with optimization techniques to increase performance and to reduce memory usage, thus increasing scalability to allow significantly larger problems to be executed. Another variant is FT-MRMPI [31], an extension to provide a fault tolerant MapReduce framework on MPI for HPC clusters. The main limitation of these solutions is that significant reimplementation effort is required to modify tools, libraries and applications to use these frameworks, which can impede adoption.

Due to the aforementioned limitations, executing the Spark framework using MPI as the communication engine is becoming the most feasible way to bridge the gap between HPC and Big Data frameworks. This approach allows users to benefit from efficient MPI libraries—such as DIY and others—in Spark with little effort on their parts. In [32], Liang and Lu proposed an event-driven pipeline and in-memory shuffle using DataMPI-Iteration, showing a speedup of 9X - 21X over Apache Hadoop, and 2X - 3X over Apache Spark for PageRank and k-means clustering. Anderson et al. [33] proposed a system for integrating MPI with Spark by offloading computation to an MPI environment from within Spark. The evaluation made with four distributed graph and machine learning applications shows speedups between 3X and 17X, including all of the overheads.

Spark-DIY provides advanced capability compared with the previously described works. For example, compared with [25], we provide compatible block management of Spark for DIY by using JNI and RDMA deployed on high-speed interconnections. Compared with [34], our solution provides not only powerful I/O through DIY, but also computing scalability. Moreover, Spark-DIY is a general solution, not domain-specific, like the work presented in [35]. Our approach is more similar to the solution proposed in [33], but Anderson et al. use HDFS to exchange data among Spark and MPI, while we use memory directly. Moreover, DIY manages the block communication graph, which avoids the burden of direct MPI usage.

VIII. Conclusion

In this paper we have explored the potential benefits of integrating a popular Big Data platform like Apache Spark, with HPC-oriented communication techniques represented by DIY block parallelism. We analyzed the literature to derive the key design features that would interest both the HPC and Big Data communities, and proposed an architecture to reflect these goals.

We developed the Spark-DIY framework, which preserves the programming interfaces and Spark environment, thus making it compatible with any Spark-based application and tool, while providing efficient shuffle and collectives by using DIY, a powerful library built on top of MPI. This framework shows good performance and scalability for communication-intensive operations in comparison to Spark, and enables the integration of elements from both the Big Data and HPC ecosystems for applications with diverse requirements without sacrificing productivity.

The work presented is relevant for the Big Data community since we offer improved performance and reduced latency for shuffle and other communication-intensive phases of Spark workflows. In addition, we expose the benefits of using supercomputing infrastructures without changing the Spark framework because we exploit MPI-based communication. Future works could enhance the architecture to support heterogeneity and accelerate independent transformations, and even extend the Spark programming model to exploit other DIY communication patterns such as local neighborhood block exchanges that are available in DIY, but have no Spark counterpart. On the other hand, the HPC

community can benefit from the myriad libraries and platforms built on top of Spark without giving away scalability. Spark's resilience, provenance, and ease of use are lacking in the HPC software stack, and Spark-DIY affords HPC practitioners of such characteristics that are commonplace in the Big Data world.

In the future, we plan to integrate Spark's elasticity into our architecture, along with MPI-I/O support to benefit from highly optimized parallel I/O in HPC systems as an alternative to current storage systems like HDFS.

ACKNOWLEDGMENT

This work was partially funded by the Spanish Ministry of Economy, Industry and Competitiveness under the grant TIN2016-79637-P "Towards Unification of HPC and Big Data Paradigms"; the Spanish Ministry of Education under the FPU15/00422 Training Program for Academic and Teaching Staff Grant; the Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357; and by DOE with agreement No. DE-DC000122495, program manager Laura Biven. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] J. Lavignon, D. Lecomber, I. Phillips, F. Subirada, F. Bodin, J. Gonnord, S. Bassini, G. Tecchiolli, G. Lonsdale, A. Pflieger *et al.*, "Etp4hpc strategic research agenda achieving hpc leadership in europe," 2017.

[2] B. D. V. Association, "Big data value strategic research and innovation agenda," October 2017.

[3] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[4] Dongarra, Jack et al. (2014) Big Data and Extreme-scale Computing Initiative (BDEC). [Online]. Available: http://www.exascale.org/bdec

[5] A. Ahmad, A. Paul, S. Din, M. M. Rathore, G. S. Choi, and G. Jeon, "Multilevel data processing using parallel algorithms for analyzing big data in high-performance computing," *International Journal of Parallel Programming*, vol. 46, no. 3, pp. 508–527, Jun 2018. [Online]. Available: https://doi.org/10.1007/s10766-017-0498-x

[6] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 97–110.

[7] O. Yildiz and S. Ibrahim, "On the performance of spark on hpc systems: Towards a complete picture," in *Asian Conference on Supercomputing Frontiers*. Springer, 2018, pp. 70–89.

[8] S. Caíno-Lores, A. G. Fernández, F. García-Carballeira, and J. C. Pérez, "A cloudification methodology for multidimensional analysis: Implementation and application to a railway power simulator," *Simulation Modelling Practice and Theory*, vol. 55, pp. 46–62, 2015.

[9] S. Caíno-Lores, A. Lapin, P. G. Kropf, and J. Carretero, "Lessons learned from applying big data paradigms to large scale scientific workflows." in *WORKS@ SC*, 2016, pp. 54–58.

[10] D. Morozov and T. Peterka, "Block-parallel data analysis with diy2," in *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, Oct 2016, pp. 29–36.

[11] K. Kondo, K. Terasaki, and T. Miyoshi, "Assimilating satellite radiances without vertical localization using the local ensemble transform kalman filter with up to 1280 ensemble members," in *EGU General Assembly Conference Abstracts*, vol. 19, 2017, p. 2170.

[12] M. Williams, P. A. Schwarz, B. E. Law, J. Irvine, and M. R. Kurpius, "An improved analysis of forest carbon dynamics using data assimilation," *Global change biology*, vol. 11, no. 1, pp. 89–105, 2005.

[13] J. P. S.Cano-Lores, A. Lapin, "Applying big data paradigms to a large scale scientific workflow: Lessons learned and future directions," *Future Generation Computer Systems*, no. April, 2018.

[14] A. Davidson and A. Or, "Optimizing shuffle performance in spark," *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, 2013.

[15] B. Nicolae, C. H. A. Costa, C. Misale, K. Katrinis, and Y. Park, "Leveraging adaptive i/o to optimize collective data shuffling patterns for big data analytics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1663–1674, June 2017.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[17] T. Peterka, D. Morozov, and C. Phillips, "High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 997–1007.

[18] N. Malitsky, A. Chaudhary, S. Jourdain, M. Cowan, P. OLeary, M. Hanwell, and K. K. V. Dam, "Building near-real-time processing pipelines with the spark-mpi platform," in *2017 New York Scientific Data Summit (NYSDS)*, Aug 2017, pp. 1–8.

[19] I. Lopez, "Idc talks convergence in high performance data analysis, 2013."

[20] Y. S. Nashed, T. Peterka, J. Deng, and C. Jacobsen, "Distributed automatic differentiation for ptychography," *Procedia Computer Science*, vol. 108, pp. 404–414, 2017.

[21] T. Bicer, D. Gürsoy, V. D. Andrade, R. Kettimuthu, W. Scullin, F. D. Carlo, and I. T. Foster, "Trace: a high-throughput tomographic reconstruction engine for large-scale datasets," *Advanced Structural and Chemical Imaging*, vol. 3, no. 1, p. 6, Jan 2017. [Online]. Available: https://doi.org/10.1186/s40679-017-0040-7

[22] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter, "Scientific computing meets big data technology: An astronomy use case," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 918–927.

[23] S. Sehrish, J. Kowalkowski, and M. Paterno, "Exploring the performance of spark for a scientific use case," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1653–1659.

[24] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. G. Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. M. Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel, "Root a c++ framework for petabyte data storage, statistical analysis and visualization," *Computer Physics Communications*, vol. 180, no. 12, pp. 2499–2512, 2009.

[25] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with rdma for big data processing: Early experiences," in *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, Aug 2014, pp. 9–16.

[26] IBM. (2017) GPU Enabler for Spark. [Online]. Available: https://github.com/IBMSparkGPU/GPUEnabler

[27] J. Fox, Y. Zou, and J. Qiu, "Software frameworks for deep learning at scale," *Internal Indiana University Technical Report*, 2016.

[28] S. J. Plimpton and K. D. Devine, "Mapreduce in mpi for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, pp. 610–632, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.02.004

[29] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: A mapreduce-like framework for in-situ scientific analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 51.

[30] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1098–1108.

[31] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault tolerant mapreduce-mpi for hpc clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 34:1–34:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807617

[32] F. Liang and X. Lu, "Accelerating iterative big data computing through mpi," *Journal of Computer Science and Technology*, vol. 30, no. 2, pp. 283–294, 2015.

[33] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 901–912, 2017.

[34] M. W. ur Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, and D. K. Panda, "High-performance design of yarn mapreduce on modern hpc clusters with lustre and rdma," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 291–300.

[35] S. Sehrish, J. Kowalkowski, and M. Paterno, "Spark and hpc for high energy physics data analyses," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 1048–1057.