IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 23, 2019, accepted October 15, 2019, date of publication October 28, 2019, date of current version November 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2949836

Toward High-Performance Computing and Big Data Analytics Convergence: The Case of Spark-DIY

SILVINA CAÍNO-LORES[1], (Student Member, IEEE), JESÚS CARRETERO[1], (Senior Member, IEEE), BOGDAN NICOLAE[2], (Member, IEEE), ORCUN YILDIZ[2], AND TOM PETERKA[2], (Member, IEEE)

[1]Computer Architecture and Technology Area (ARCOS), Department of Computer Science and Engineering, University Carlos III of Madrid, 28911 Leganés, Spain
[2]Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439, USA

Corresponding author: Silvina Caíno-Lores (scaino@inf.uc3m.es)

This work was supported in part by the Spanish Ministry of Economy, Industry and Competitiveness under Grant TIN2016-79637-P (toward Unification of HPC and Big Data Paradigms), in part by the Spanish Ministry of Education under Grant FPU15/00422 Training Program for Academic and Teaching Staff Grant, in part by the Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, and in part by the DOE with under Agreement DE-DC000122495, Program Manager Laura Biven.

ABSTRACT Convergence between high-performance computing (HPC) and big data analytics (BDA) is currently an established research area that has spawned new opportunities for unifying the platform layer and data abstractions in these ecosystems. This work presents an architectural model that enables the interoperability of established BDA and HPC execution models, reflecting the key design features that interest both the HPC and BDA communities, and including an abstract data collection and operational model that generates a unified interface for hybrid applications. This architecture can be implemented in different ways depending on the process- and data-centric platforms of choice and the mechanisms put in place to effectively meet the requirements of the architecture. The Spark-DIY platform is introduced in the paper as a prototype implementation of the architecture proposed. It preserves the interfaces and execution environment of the popular BDA platform Apache Spark, making it compatible with any Spark-based application and tool, while providing efficient communication and kernel execution via DIY, a powerful communication pattern library built on top of MPI. Later, Spark-DIY is analyzed in terms of performance by building a representative use case from the hydrogeology domain, EnKF-HGS. This application is a clear example of how current HPC simulations are evolving toward hybrid HPC-BDA applications, integrating HPC simulations within a BDA environment.

INDEX TERMS Big data analytics, high performance computing, spark, DIY, MPI.

I. INTRODUCTION

Convergence between high-performance computing (HPC) and big data analytics (BDA) 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. HPC-BDA convergence became a hot topic as applications and their associated data evolved outside of their original ecosystems. At that time, the problem for HPC was *How can we cope with increasing datasets?*, while BDA was wondering *How can we run analytics faster?* Since then, the HPC [1] and BDA [2] communities have recognized new opportunities in unifying the platform layer and data abstractions for both HPC and BDA [3]. This situation led to the advent of specific research topics, such as high-performance data analytics and data-driven science.

The divergence between HPC and BDA software ecosystems emerged early this century when software infrastructure and tools for data analytics that had been developed by online service providers were open sourced and picked up

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

by various scientific communities to solve their own data analysis challenges [4]. Major technical differences between HPC and BDA ecosystems include software development paradigms and tools, virtualization and scheduling strategies, storage and networking models, resource allocation policies, and strategies for redundancy and fault tolerance [5]. These technical differences, in turn, tend to make future cross-boundary collaboration and progress increasingly problematic. This leads to a challenging scenario that involves understanding a different community and computing model in order to inspire new approaches to replicate features that become necessary, and managing a computing infrastructure built for a completely different paradigm. While each of these domains has its set of unique requirements in terms of the underlying infrastructure, there is an increased pressure for leveraging technology, methods and tools from across these domains.

However, many issues remain unsolved, and this situation has been worsened by the appearance of new application domains that are completely hybrid in nature, such as autonomous vehicles, surveillance, e-science with big data sources, monitoring of large scale infrastructures, and smart cities. These domains have in common the need to support the simulation of complex models, assimilating voluminous and variable real-time data in order to generate refined models for better understanding of the domain, to prescribe pattern-based control actions, or to predict a future behavior. Under these circumstances, borrowing features from the other paradigm proves insufficient, and deeper convergence becomes necessary in order to cope with mixed requirements, new infrastructures, and upcoming performance expectations. Major technical requirements involved in this process include highly scalable performance, high memory bandwidth, low power consumption, and excellent short arithmetic performance. Consequently, BDA and HPC platforms today remain largely incompatible.

The objective in this work is *to architect an abstract system that enables the interoperability of established BDA and HPC execution models, in light of their canonical underlying infrastructures and considering the requirements of hybrid applications*, which we analyze in depth. We summarize our contributions as follows:

1) *A definition of a generic unified distributed data abstraction (UDDA) and its associated unified operational model (UOM)*, which sets the foundation of a theoretical frame for the analysis and definition of composite HPC-BDA applications. This data abstraction embodies a careful selection of the features required to interoperate BDA and HPC operations and generates the guidelines that must be enforced by implementations of the architecture in order to preserve interoperability.
2) *A generalist execution model interoperability architecture for HPC-BDA applications* based on the UDDA and UOM definitions. It includes a transparent execution delegation system (EDS) that transfers each stage of the composite application to the appropriate execution model (process- or data-centric).
3) *An implementation of the former architecture*, based on Spark and MPI, which we named Spark-DIY. This framework is suitable for stateful and stateless operations on generalist data types, and it is optimized for primitive data types as well. It allows the composition of applications with HPC and BDA stages, including different mechanisms to interact with parallel and distributed storage.
4) *An implementation of a real-world use case from the hydrogeology domain* enriched with features enabled by our architecture, such as cloud and streaming support for delocalization and data assimilation, respectively.

The contributions of this paper are an extension of our previous work [6], where we introduced the basic ideas behind Spark-DIY and provided a preliminary implementation. Specifically, we focus on three new directions: (1) we study the big data-HPC convergence problem in a more comprehensive manner, with a thorough review of the existing body of literature; (2) we propose new technical content related to the general model of the proposed architecture; and (3) we introduce several optimizations in our reference implementation and expand the scale and scope of our evaluations to study its effectiveness.

The rest of this paper is organized as follows: Sections II and III introduce the BDA and HPC ecosystems, respectively, and develop on their current state; Section IV presents relevant works related to the HPC-BDA convergence problem; Section V analyzes the challenges and opportunities of the convergence of such paradigms; Section VI details the proposal of an abstract architecture suitable for the interoperation of process- and data-centric platforms, which is later implemented in Section VII, using Apache Spark and a communication library built on MPI, and evaluated in Section VIII on a real use case from the hydrogeology domain; and Section IX summarizes this work, its applications, and directions for future research.

II. BIG DATA ANALYTICS ECOSYSTEM

Big data affects many different ecosystems and areas of research and business; thus it has no unique definition, and its scope is still a controversial topic in these communities. From the data analysis perspective, the *multi-V* model reflects a way to define big data by describing several of its features, and it keeps evolving over time adding more attributes as needed [7]. The core characteristics included in this model are as follows:

- *Volume of data.* Volume is necessary in order to get valuable insight from analytics tools. Usually one finds volumes on the order of petabytes or terabytes at the enterprise level. These volumes can also be quantified on the order of billions of records, tables, files, or transactions depending on the data structure required by the underlying storage system. In order to provide sufficient

quality of service, big data systems and applications must be designed to handle such large data volumes efficiently and reliably.

- *Velocity of data production and processing.* Data can be produced and consumed at different rates. Big data systems can even incorporate diverse source frequencies and processing speeds, including batch processing, streaming, and near and real-time speed.
- *Variety of data types.* Nowadays a data source can be anything (sensors, web applications, mobile devices, etc.). Hence, data can be highly heterogeneous and may be unstructured. In addition, data types depend greatly on the application and its domain: we find structured statistical data in business intelligence, time series, and geospatial data in the Internet-of-Things (IoT), and media, text and graph data in social environments. Platforms must be able to understand and integrate this diverse data to aggregate the knowledge from different sources.

In addition, from the business perspective there are other key features, such as the following [8]:

- *Veracity of data.* The volume of data is key to obtaining knowledge, but the derived information would be flawed if the quality of data is low. High-quality big data must be reliable in terms of trust and integrity, in order to attain acceptable veracity.
- *Value in business terms.* The model or analysis that results from processing big data must provide enterprise value to make up for the investment expenses necessary to collect and analyze data.

These definitions have a key aspect in common: big data focuses on data that is perceived as large in volume. This paradigm shift has affected all areas of computing from data acquisition transfer and storage to data analysis and visualization. This shift was reflected in traditional areas of business and science, such as genomics, climatology, finance, and business intelligence, which were able to obtain better knowledge with existing methods but also promoted novel areas of research to exploit the intrinsic value of data and improve the system's capability to cope with the requirements of data processing and storage. Areas such as IoT and BDA developed greatly thanks to the advances in big data.

Big data analytics is one of the best examples of how big data disrupted an established area such as business intelligence, exploiting advanced analytics techniques operating on big data to evolve from descriptive tools to predictive and prescriptive models. Today, enterprises are exploring big data to incorporate knowledge discovery into their business in order to detect interrelations among apparently unrelated attributes of datasets [9]. Enterprises can now understand the current state of the business and customer behavior through complex techniques such as predictive analytics, data mining, statistical analysis, data visualization, artificial intelligence, and natural language processing, paired with support platforms such as Map-Reduce, in-database analytics, in-memory databases, and columnar data stores. Some of these techniques have been

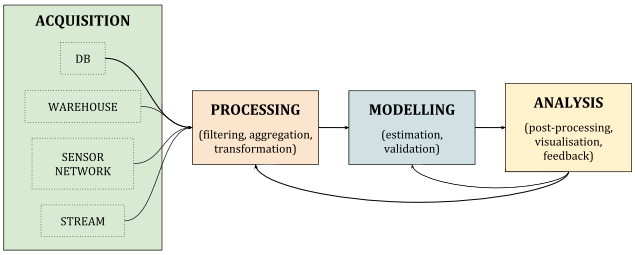


FIGURE 1. Typical workflow of a BDA application.

around for years, and they have been revamped because of their good adaptability to very large datasets with minimal data preparation. In addition, infrastructures such as cloud computing offer the possibility to lower the economical costs of deploying BDA and building analytics workflows at different levels of abstractions.

Figure 1 represents the traditional knowledge discovery workflow for BDA, which includes dealing with data acquisition from diverse sources, processing and combining data in many ways in order to build a model that can be used for analysis and visualization, and incorporating feedback mechanisms to refine data processing and modeling stages. This workflow has been usually combined with the lambda architecture [10] to provide scalable integration and interoperability across different datasets through real-time analytics. This architecture was proposed with the goal of providing a generalist platform to serve different applications with diverse latency needs in a streaming environment. As shown in Figure 2, the lambda architecture includes a speed layer for pure stream processing in real time, a batch layer for storing raw data and processing higher quality views of long-term data, and a presentation layer that manages queries and output visualization.

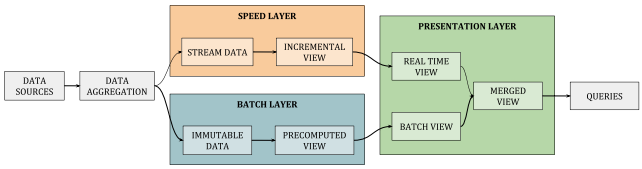


FIGURE 2. Representation of the lambda architecture.

Areas such as mobile technology, social media, IoT, and data-driven sciences are expected to generate data to a global total in the order of dozens of zettabytes [2]. Those data will yield valuable information for smart applications, science, and decision-making processes in business.

A. CLOUD COMPUTING AND BEYOND

BDA faces the challenge of continuously adapting to increasing data volume and complexity. This translated to a continuous need to scale out reliably when scale up becomes infeasible [11]. In this context, cloud computing became a widely adopted infrastructure for BDA [12].

Cloud computing is a popular paradigm that relies on resource sharing and virtualization to provide the end user

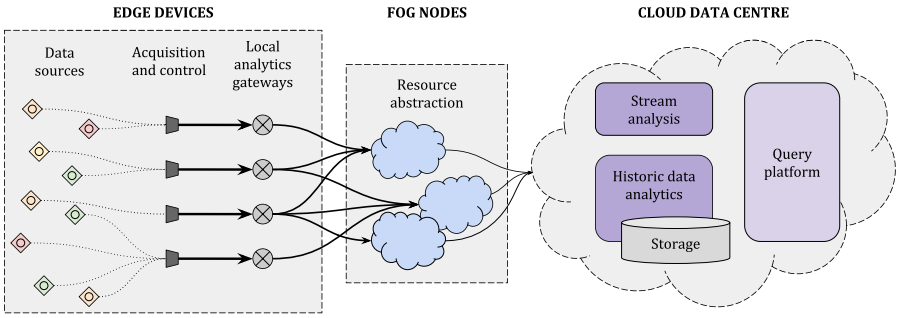


FIGURE 3. Architecture of a massively distributed infrastructure comprising edge devices, intermediate fog nodes, and core cloud datacenters.

with a transparent, scalable and elastic system that can be expanded or reduced on the fly. It emerged with the idea of virtually unlimited resources obtainable on demand [13], and its popularity is a consequence of some of the core features of cloud service models, such as the following:

- Minimal management effort, since the infrastructure is maintained and administrated by a third-party and system deployment can be eased by relying on high-level service models from Platform-as-a-Service up to Analytics-as-a-Service [14].
- Automatic or manual scale up or down according to utilization, thus supporting elasticity.
- Potential to reduce economical costs, as it follows a pay-as-you-go model.
- Flexible data sharing and platform integration for heterogeneous analytics workloads.

Given these benefits, enterprises and scientific institutions have been making efforts to make their applications cloud-ready [15]. Nevertheless, cloud computing presents challenges related to the lack of control of the underlying hardware infrastructure, the privacy concerns that arise from hosting datasets on third-party servers, and the transfer time and cost required to upload and download large quantities of data [16]. For some applications relying on many data sources generating large volumes of data at high velocities, centralizing all data to a very limited number of data centers is no longer viable, especially if low latency is required by the end users.

These limitations led to models that evolved cloud architectures aiming to alleviate the data centralization problem by combining processing, storage, and communication in distributed services that run closer to the data production environment in a hierarchical multitiered manner. These paradigms include mobile cloud computing [17], edge computing [18], [19] and fog computing [20]. Figure 3 shows how edge devices interact with intermediate aggregation and processing components to derive local analytics and reduce the volume of data to be transferred to higher-level layers. Fog data centers orchestrate and abstract their network and computing resources in order to relay aggregated data to the final cloud, where data are finally stored for archival purposes, and broad analysis is conducted. Upcoming scenarios might provide terabytes of data per hour, making efficient real-time operations critical for monitoring, decision-making, and digital twin coordination. In addition to highly distributed platforms, high-performance computing infrastructures and methods are expected to improve the processing capabilities of cloud providers to cope with these extreme data and computation requirements [2]. Of particular importance in this context is the concept of storage elasticity, that is the ability to dynamically adapt storage services to the data needs, in terms of both sustained I/O throughput [21] and storage space [22]; otherwise the cost of operating with data on the cloud becomes prohibitively expensive. This elasticity dimension has important consequences in the design of converged high-performance computing and big data analytics frameworks: they need to be able to adapt to sudden changes in the data layout and requirements on the fly, hiding the details of how the computation needs to be reorganized in response to these changes, while maintaining high performance and scalability.

B. DATA-CENTRIC BATCH AND STREAM PROCESSING

Minimizing data movements is important for the final performance. At the application development stage, working with programming models that provide an abstraction of resource allocation, data management, and task execution via the data-processing layer can result in an improvement of performance and locality.

The Map-Reduce [23] data-processing model was the most relevant data-centric model when BDA research took off, since it enables analytics on big datasets by parallelizing computations for HPC and multicore environments [24]. A Map-Reduce-based algorithm consists of a two-phase algorithm that takes as input a set of key-value pairs retrieved from the input files. The input is split across a group of homogeneous *map* functions, which process the data and forward the result to the *reduce* tasks in order to aggregate and write the final result. The original Map-Reduce implementation by Google relies on the Google File System (GFS) [25] to achieve locality by block replication and considers data-aware task scheduling. A similar approach is followed by the open-source Map-Reduce implementation, Hadoop [26], and its partner file system Hadoop Distributed File

System (HDFS) [27]. Map-Reduce applications work with many large files and need to execute fast transfers and operations on a wide and diverse dataset. It saw wide adoption in dedicated data analytics clusters and cloud computing. Recent advances extend the capability of resource-limited infrastructure (e.g. on-premise clusters, edge devices) with ephemeral, short-lived resources from public clouds to provide a boost during peak utilization [28]. Besides the numerous works that took advantage of the multitude of runtime implementations and optimizations in a variety of scenarios, Map-Reduce also had a major impact on subsequent models that were inspired by its paradigm.

One of the models that emerged from Map-Reduce is Map-Reduce-merge [29], a model that adds a *merge* phase that can efficiently aggregate the data already partitioned and sorted by the map and reduce modules. Map-Reduce does not directly support processing multiple related heterogeneous datasets, a limitation that causes efficiency issues when Map-Reduce is applied in relational operations such as joins. The Map-Reduce-merge model can, on the other hand, express relational algebra operators and implement several join algorithms.

Map-iterative-reduce [30] is an alternative model that extends Map-Reduce to better support reduce-intensive applications, while substantially improving its efficiency by eliminating the implicit synchronization barrier between the map and the reduce phases. Among implementations of map-iterative-reduce we can find Twister [31], Haloop [32] and Twister4Azure [33]. The work in [31] suggests that iterative and interactive applications are the ones that could take the highest advantage of in-memory data storage for fast reuse.

The Spark [34] programming model supports a wide range of functionalities that enable the development of applications that do not fit nicely to the Map-Reduce paradigm, such as many iterative machine learning algorithms and interactive data analysis tools. The Spark framework relies heavily on the concept of *resilient distributed dataset* (RDD) [35] to provide this functionality. RDDs are in-memory collections of data, and the operations on them are tracked in order to provide fault tolerance. According to its authors, the system has proven to be highly scalable and fault tolerant. However, in most Java-based Map-Reduce platforms [36] the deep component stack and its dependence on the JVM entails a significant memory consumption that also affects execution time because of frequent garbage collection operations [37], [38], and serialization if bindings to other languages are used [39]. A performance comparison between Hadoop and Spark frameworks in terms of CPU, memory and I/O usage is presented in [40].

Map-Reduce-based programming models have also evolved into language frameworks that provide a data access layer through a set of APIs, thus eliminating the need to reimplement repetitive tasks by working on top of the processing layer [41]–[45]. For example, Spark has inspired subsequent works such as GraphX [41], which extends the framework to support graph parallel computing. Working with graphs has, as indicated by the authors, specific challenges and requirements that were not fully addressed by previous works. In a similar trend, several frameworks have explored the possibility of building rich data SQL-like abstractions for database processing. For example, Spark SQL [42] extends Spark to integrate database processing into the framework. Pig Latin [43], HiveQL [44] and REX [45] rely on high-level data-flow languages, and execution frameworks whose compilers produce sequences of batch-processing Map-Reduce programs.

Moreover, some models evolved into workflow frameworks to support the composition of heterogeneous and coupled components to simulate different aspects of an application model [46]. As these modules interact and exchange significant volumes of data at runtime, minimizing these transfers and making them efficient have a major impact on the overall performance [47]. Consequently, data locality enforcement has been studied in several works tackling task and job scheduling [48], data-flow optimization [49], and resource allocation [50].

In-memory computing has also affected database-oriented platforms with approaches such as Phoenix [51] for shared-and-distributed-memory machines. Shark [52], which supports the Hive warehousing system [53] on Spark, is a popular similar approach but oriented toward SQL-based data analytics by means of machine learning. These algorithms are typically iterative; thus, in-memory computing suits well the need for cached data to be reused. Similarly, pure Map-Reduce paradigms have benefited from in-memory trends, resulting in platforms for memory-intensive workloads such as Mammoth [36], Piccolo [54], Main-Memory Map-Reduce (M3R) [55], and Hyracks [56].

Some of the former works indicate that in-memory databases and computing are able to scale to petascale systems [36], [57]. No further work has found indication whether this could hold for exascale systems, however. New technologies based on multicore processors can improve the performance of applications by favoring locality through intranode data sharing, which minimizes data exchanges across compute nodes [47], [58]. The prospective usage of Map-Reduce-based models at different levels of parallelism within the computing infrastructure, as typically done in HPC systems, might provide a shared-space programming abstraction that replaces existing parallel programming models such as message passing.

III. HIGH-PERFORMANCE COMPUTING ECOSYSTEM

High-performance computing refers to the usage of aggregated computing power in order to run complex parallel programs efficiently and as fast as possible. This term is tightly related to the concept of *supercomputing*, which pushes HPC to the highest operational rate of the available technology. Nowadays, top modern supercomputers perform on the order of 100 petaflops, and a machine capable of delivering one exaflop is expected to appear around 2020 [59].

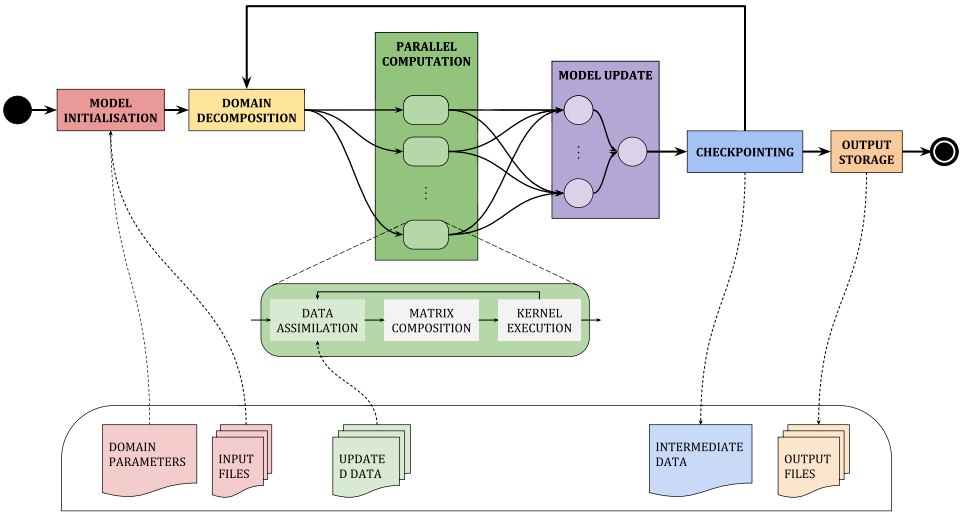


FIGURE 4. Application model for the typical HPC scientific application.

All this computational power and sophisticated infrastructures involve massive investment in hardware development, runtime design, and daily operational costs. Naturally, these means have been put to the service of strategic areas of science and industry that rely on complex numerical applications that cannot be run on commodity machines because of their performance requirements. Those include sectors such as aviation, energy, pharmaceutical, oil and gas, and automotive, and high-end scientific research on climate, medicine, bioinformatics, and physics.

To exploit the scalability and performance of supercomputers, HPC applications rely heavily on parallelism techniques to maximize the usage of resources. Supported by advanced execution engines, these applications coordinate parallel processing on many cores and nodes with network-intensive data transfers between compute and storage nodes. In addition, some applications need to iterate to refine their results, modify the underlying model, or incorporate new data. Figure 4 depicts these relationships, which form the structure of many HPC applications. The core simulated models are typically initialized with a combination of input data and base environmental conditions as parameters, and the simulation domain is distributed so that kernel computations can be conducted in parallel. Ideally, these simulations are pleasingly parallel, and computations can be executed independently while incorporating partial new data. Once kernels converge, the resulting data are merged with the results coming from the other processing units in order to update the model, typically leading to a communication-intensive process that results in the input that will be fed to the following step. As a fault tolerance measure, most simulations include checkpointing procedures to store intermediate models and restore the simulation from them in case of failure. Simulation results are then written to storage.

HPC has been also affected by current data-centric trends, and scientists are already tackling how HPC can benefit from the availability of big data and analytics techniques. High-performance data analytics, data-intensive scientific computing, visualization and machine learning are areas of research that currently inherit the performance and scalability aspirations of traditional HPC, while incorporating new challenges that affect how data are managed and transmitted at all levels of the system and software stack.

A. SUPERCOMPUTERS AND DATA-INTENSIVE CLUSTERS

Large scale HPC infrastructures (such as supercomputers, grids, clouds, and clusters) have been widely developed with the objective of providing a suitable platform for high-performance and high-throughput computing. Since these paradigms typically require massive hardware resources and dedicated middleware, large scale computing holds specific challenges in order to achieve sufficient efficiency in terms of memory, CPU, I/O, network latencies, and power consumption, to name a few. These systems are oriented toward supporting resource-demanding and complex applications with heavy resource requirements; thus they need dedicated platforms that orchestrate tasks and manage resources in order to behave in a coordinated manner. These pieces of software constitute the middleware that permits node intercommunication, data transmission, load balancing, task assignment and fault tolerance.

Traditional HPC infrastructures [60]–[63] are built in such a way that storage and computation are not located in the same nodes, following the schema depicted in Figure 5. Networks are also isolated to avoid the interference of I/O operations to the parallel file system with computation communications. Parallel file systems maintain a logical space view and provide an efficient access to data, which can be distributed through several sites and among multiple I/O servers and disks to deliver a higher degree of parallelism.

Several issues are still not solved with regard to these infrastructures. In particular, computer scientists have

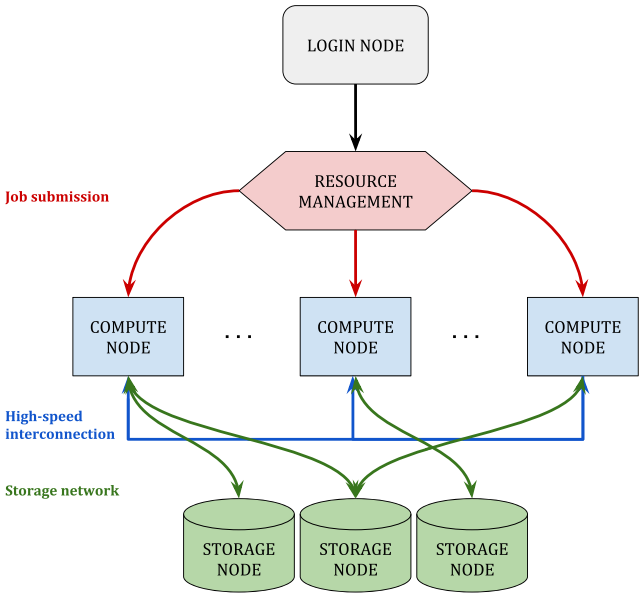


FIGURE 5. Traditional architecture of an HPC infrastructure, with isolated storage and computation networks.

realized that, as problems become larger and more complex, a powerful infrastructure is not sufficient to achieve proper scalability in terms of overall performance, resource utilization, and power efficiency. With the advent of data-centric trends, recent works have suggested that improving data locality across all layers of the system stack is key to move toward exascale infrastructures efficiently [64].

Some authors claim that the current architecture of high-end computing systems is inefficient because storage is completely segregated from the compute resources. Thus, further network interconnections are needed to access storage [65]. Storage systems constitute one of the greatest bottlenecks when dealing with data-intensive computations [66]. Therefore, data awareness in file systems and storage infrastructures can significantly improve the system's overall locality, as other layers can benefit from the system's knowledge of data placement. In order to avoid the drawbacks of traditional parallel file systems, a new generation of distributed file systems has emerged as support layers for data-centric frameworks such as Map-Reduce. The Hadoop File System (HDFS) [67] and the Google File System (GFS) [25] are relevant examples of such file systems portraying a focus on data locality. Work in this area has also been conducted to improve locality by moving data to the node's memory to minimize interaction with storage nodes. This resulted in new infrastructure architectures that incorporate deeper memory hierarchies and local storage in compute nodes, following the model of cloud-oriented data-centers [68].

The influence of big data and analytics in supercomputing is also reflected in the incorporation of new hardware architectures tailored for deep learning and data-intensive computing [3], resulting in dedicated accelerators such as vector processors, tensor processing units (TPUs), general-purpose graphical processing units (GPGPUs), and field-programmable gate arrays (FPGAs). These new technologies provide further computational power for applications, but it is still unclear which areas will require the full Exascale power that will be provided by impending heterogeneous infrastructures, since the bottleneck might remain at upper layers of the software stack such as monitoring, resource management, data management, and communications [69]. In addition, applications are also evolving toward complex workflows involving iterative analytics, data-intensive operations, and compute-intensive computations. Making an efficient usage of supercomputers in this landscape will require algorithm, execution model, and data management refinements to support applications with mixed requirements, without diminishing usability.

B. PARALLEL PROGRAMMING MODELS AND EXECUTION MODELS

HPC applications aim to run at the maximum level of parallelism provided by supercomputers in order to reduce execution time and increase scalability. On submission, applications are provided with a set of allocated processing units distributed across several nodes, and optionally different types of accelerators might be assigned if present in the infrastructure. Figure 6 represents these diverse processing units.

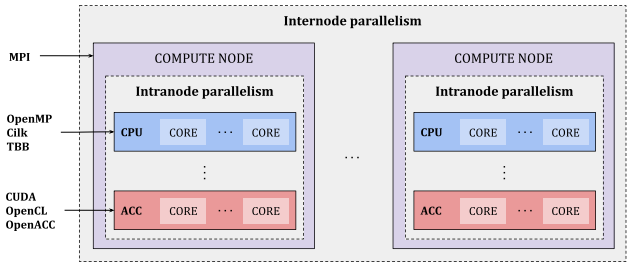


FIGURE 6. Parallelism layers in HPC programming models, including inter and intranode parallelism.

The Message Passing Interface (MPI) standard [70] is the most common procedure to exploit internode parallelism in HPC environments and is the basis for numerous runtimes and workflows for scientific computing. The implementations of MPI allow the execution of standard operations comprising multiple processes on distributed-memory platforms, which provides coarse-grained parallelism sufficient for petascale applications.

Thread-level parallelism is the basis for fine-grained intranode parallelism for multicore CPUs. Developers can choose from a wide range of threading libraries such as POSIX threads, Intel's Threading Building Blocks (TBB) [71], and Microsoft's Parallel Patterns Library [72]. Nowadays, the open specification for multiprocessing (OpenMP) [73] is still one of the most-used tools for parallelization, mostly because its annotation-based nature minimizes the impact on sequential code. As machines reached petascale, combining

MPI and OpenMP became a common procedure to reach massive parallelism on machines supporting distributed and shared-memory [74]–[76].

Current HPC infrastructures have incorporated different types of accelerators to enhance the performance of specific applications. Programming models adapted accordingly to ease the access to further finer-grained intranode parallelism. GPGPUs are the most widely adopted accelerator in current HPC machines, given their many-core architecture, which pushes forward massive parallelism to the order of thousands of cores on a single chip. Several libraries enable the interaction with GPGPUs, such as OpenCL [77], Nvidia CUDA [78], and OpenACC [79], supporting data offload to the accelerators, kernel operator definition, direct execution of such code on the device, and result retrieval back to the host CPU. Accelerator runtimes have also been integrated with intranode parallelism through OpenMP [80], and internode parallelism via MPI [81], [82]. The mechanisms to build hybrid runtimes exploiting both intra and internode parallelism had a major influence on subsequent advances in further parallelism integration, and they are expected to be present in future exascale systems to cope with the need for adaptive hybrid programming models for heterogeneous extreme-scale machines [83].

IV. CURRENT TRENDS IN HPC AND BDA CONVERGENCE

In the literature we can find many attempts to incorporate the beneficial features of HPC and BDA into their corresponding areas. Section IV-A presents relevant works trying to accelerate BDA by means of HPC computing models (mainly MPI) and advanced techniques to interact with the underlying network and accelerators. Correspondingly, Section IV-B presents how data-centric paradigms and BDA infrastructures (primarily clouds) have been exploited to enhance HPC.

Finally, Section IV-C analyzes the most relevant endeavors toward HPC and BDA interoperability for hybrid applications, not necessarily attempting to improve one model or the other but focusing on the goal of both paradigms coexistence in a single application. These works are scarcer than others that focus on improving a single ecosystem, but they are highly relevant since they are oriented toward avoiding reprogramming applications and exploiting the advantages of both worlds. This is the primary approach in our work.

A. USAGE OF HPC TO ENHANCE BDA

The focus on performance of HPC is attractive for BDA users who must deal with increasing number of problems but have limited processing time. We hereby introduce how previous works accelerated BDA by exploiting the high-performance and scalability of computing models such as MPI, and specific architectural features of HPC infrastructures.

1) PROCESS-CENTRIC COMPUTING MODELS: MPI AND OPENMP

Implementations of traditionally data-centric frameworks such as Map-Reduce have been developed by using MPI. 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 and introduce overheads. One such framework was proposed in [84], in the form of a parallel library that allows algorithms to be expressed by using the Map-Reduce paradigm, simplifying programming by using map and reduce operations callable from C++, C, FORTRAN, or scripting languages such as Python. Another related work is Smart [85], a framework that mimics Map-Reduce to execute data analytics algorithms alongside computational simulations in time-sharing or space-sharing modes, in a process known as *in-situ analytics*. The framework uses both MPI and OpenMP to parallelize tasks over distributed and shared-memory. A more recent Map-Reduce framework over MPI is Mimir [86]. It includes a redesign of the execution model with optimization techniques to increase performance, reduce memory usage, and improve scalability. Another variant is FT-MRMPI [87], an extension that provides a fault-tolerant Map-Reduce framework on MPI for HPC clusters.

Other works attempted to develop novel approaches to data-centric programming. For example, in [88] the authors proposed an event-driven pipeline and in-memory shuffle using DataMPI-Iteration, which provided overlapping of computation and communication for iterative BDA computing and showed a speedup of $9\times - 21\times$ over Apache Hadoop and $2\times - 3\times$ over Apache Spark for PageRank and K-means. Another approach for running data-centric applications on MPI beyond the Map-Reduce model was proposed in [89], where the authors presented a set of building blocks that provide scalable data movement capability to computational scientists and visualization researchers for writing their own parallel analysis. This work is the origin of the Do-It-Yourself parallel runtime (DIY) [90], a full data-driven execution engine usable for any topology defined by the user.

2) INFRASTRUCTURE: NETWORKING AND ACCELERATORS

Optimizing data-centric platforms for specific heterogeneous architectures is a popular direction to accelerate BDA. MrPhi [91] targets Intel Xeon Phi coprocessors. A solution for hybrid cloud bursting is discussed in [92] and extended with a detailed performance model [28]. A similar solution for Spark on GPUs was IBMSparkGPU [93], but it is valid for local tasks only. Trace [94], 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. The authors also studied in [95] various frameworks for deep learning networks that can scale across multiple machines with full parallel support and distributed execution, such as TensorFlow, CNTK, Deeplearning4j, MXNet, H2O, Caffe, Theano, and Torch.

Some solutions have been proposed to accelerate big data processes using FPGAs. Ghasemi *et al.* [96] created a custom Map-Reduce framework that is capable of combining

Map-Reduce FPGA kernels with a template interface. Segal *et al.* [97] presented SparkCL, and automatically generated OpenCL kernel for Altera FPGAs. However, they identified that, because of architectural limitations, data transfer overhead suggests that only highly compute-intensive tasks should be offloaded [98]. In [99] the authors propose a solution to implement Map-Reduce entirely in an FPGA. In order to overcome former problems, data are read entirely from Java to the FPGAs and the final results are copied back into Spark using memory mapped byte buffers, which avoid straining FPGA resources.

Networking and storage are closely related and have been exploited to improve BDA platforms. A first attempt to optimize Map-Reduce storage on HPC clusters by utilizing Lustre as the storage provider for Remote Direct Memory Access (RDMA) intermediate data was presented in [100]. Other works tried to adapt Map-Reduce and its underlying HDFS to use GPFS [101]–[103] or provide subfiles and locality [104]. Results indicated that BDA platforms still suffered from the reduced locality offered by such a setting. Consequently, the authors in [105] proposed a two-layer storage system that exploits PFS performance but incorporates an intermediate in-memory storage system, with good results.

A proposal to accelerate Spark communication was presented in [106], which used a high-performance RDMA-accelerated data shuffle in the Spark framework on high-performance networks and provided a performance improvement of 80%. We also can see interest in the usage of HPC systems for BDA in the commercial sector. For example, PayPal has shown how the high concurrency and low latency of HPC systems can be used for fraud detection [107].

B. USAGE OF BDA TO ENHANCE HPC

Evidence of convergence in the opposite direction also appears, especially in works aiming to incorporate data-centric computing models, such as Map-Reduce in HPC applications, and in efforts to exploit BDA computing facilities, such as clouds to scale scientific computing.

1) DATA-CENTRIC COMPUTING MODELS: MAP-REDUCE

Scientific applications and their adaptability to new computing paradigms have elicited increasing attention from the scientific community. The applicability of the Map-Reduce scheme for scientific analysis has been notably studied, especially for data-intensive applications resulting in an overall increased scalability for large datasets, even for tightly coupled applications [108].

Several works have analyzed how current HPC applications could be adapted to Map-Reduce models. In [109], Srirama *et al.* studied how some scientific algorithms could be adapted to the Hadoop Map-Reduce framework. They established a classification of algorithms according to the structure of the Map-Reduce schema these would be transformed to. They suggested that not all of those algorithms would be optimally adapted by their selected Map-Reduce implementation, yet they would suit other similar platforms such as Twister or Spark. They focused on the transformation of particular algorithms to Map-Reduce by redesigning the algorithms themselves. A similar approach is HAMA [110], a framework that provides matrix and graph computation primitives on top of Map-Reduce. An advantage of this framework over traditional MPI approaches to matrix computations is the fault tolerance provided by the underlying Hadoop framework. An approach for using Hadoop Map-Reduce in scientific workflows was explained in [111], whose authors proposed a new architecture named SciFlow. This architecture consisted of a new layer added on top of Hadoop, enhancing the patterns exposed by the framework with new operations (join, merge, etc.). Scientific workflows are represented as a DAG composed of these operations. A theoretical analysis of migrating common HPC-oriented workflows to a BDA processing platform (i.e., Apache Hadoop) was made in [112]. The authors implemented six representatives of common scientific workflow patterns in an Apache Hadoop environment and discussed implementation challenges as well as Hadoop environment applicability for each of the basic patterns.

Other works attempting to tailor Map-Reduce and data analytics frameworks have been developed for HPC. These environments target a particular family of applications or processor architecture, but they are not generalized for reuse in other contexts. A preliminary work was ROOT [113], an object-oriented C++ high-energy physics (HEP) framework designed for storing and analyzing petabytes of data efficiently by using an object container optimized for statistical data analysis over very large datasets. Another attempt is an extension of Map-Reduce with access patterns (MRAP) [114], which targets HPC analytics with a focus on data locality.

BDA analytics tools, such as Hadoop and Spark, are being explored to provide straightforward data distribution and caching mechanisms in data-intensive HPC applications. Their data-centric nature permits reasoning about tasks over distributed data abstractions without worrying about task scheduling, which is managed by the middleware to enforce data locality and minimize transfers. The inherent parallelism of these tools has resulted in positive experimental results showing their suitability for massively parallel workloads such as MTC-like workflows [115]. Other works explored the usage of high-level machine learning libraries for HPC typographic reconstruction [116] with good results. Nevertheless, challenges remain with respect to workflows built with a pure HPC focus, which rely on MPI and traditional storage infrastructures [117].

Because Spark underlies many BDA tools, the performance of Spark for scientific computing has been studied in several works. Sherish *et al.* recently showed in [118] how BDA tools can be used for HEP data analysis because extremely large HEP datasets can be represented and held in memory across the system and accessed interactively by encoding an analysis using the high-level programming abstractions in Spark. Kira [115], a flexible and distributed

astronomy image processing toolkit using Apache Spark, was used to implement a source extractor application called Kira SE for astronomy images. The study shows that Spark may be an alternative to an equivalent C program for many-task applications. Another interesting study was presented in [119], where the performance of a Spark implementation of a classification algorithm in the domain of high-energy physics (HEP) was evaluated. The results showed that the implementation scaled well but the performance was poor compared with the results of an untuned MPI implementation of the same algorithm.

2) INFRASTRUCTURE: DISTRIBUTED STORAGE AND CLOUD COMPUTING

Scientific workflows are composed of heterogeneous and coupled components that interact and exchange significant volumes of data at runtime. Making these transfers efficient has a potential major impact on the overall performance of the resulting application [120]. As a consequence, both the storage infrastructure and the logical file system abstractions could affect performance and scalability, thus making data management a key aspect in workflow design and implementation [121]. In order to support the degree of scalability and performance required by modern simulators, one of the key elements to take into consideration is the avoidance of I/O bottlenecks [122]. Given the workflow nature of many state-of-the-art simulators for scientific computing, Srirama *et al.* [123] proposed a workflow-partitioning strategy to reduce the data communication in the resulting deployment. Matri *et al.* [124], [125] analyzed the applicability of binary large objects (known as *blobs*) and object storage systems to solve the problems with POSIX-IO-compliant file systems and as a mechanism to replace distributed file systems for BDA analytics.

Several works have addressed the opportunities of shifting scientific workflows from traditional HPC and HTC infrastructures to BDA computing infrastructures such as clouds. In particular, authors have focused on exploring data-intensive workflows, since they are the most tightly related to conventional BDA applications in terms of data volumes [126], [127]. Experimentation with well known workflows shows that running costs could be significantly decreased with BDA infrastructures, but performance would suffer from virtualization and latency overheads [128]–[130]. The relationship between Map-Reduce and the cloud for scientific applications has also been tackled in [131], which establishes that performance and scalability tests results are similar between traditional clusters and virtualized infrastructures. Nonetheless, these results for Map-Reduce workflows are not generalizable to other application models found in HPC, since the performance of network in cloud is worse than that of HPC by one to two orders of magnitude [132], [133]. Other authors indicate, however, that the low maintenance and economical cost of clouds made it a viable option for small scale clusters with a tolerable performance loss [134], [135]. Consequently, cloud computing has been proved as a good solution for scientists who need resources instantly and temporarily for fulfilling their computing needs [136].

In this context, trends evolved toward migrating scientific applications to the cloud by means of several techniques. D'Angelo [137] described a Simulation-as-a-Service schema in which parallel and distributed simulations could be executed transparently, which requires dealing with model partitioning, data distribution and synchronization. He concludes that the potential challenges concerning hardware, performance, usability and cost that could arise could be overcome and optimized with the proper simulation model partitioning. Following a similar approach, Yu *et al.* [138] proposed an application adaptation middleware to allow legacy code migration to the cloud. In this work, a virtualization architecture is implemented by means of a web interface and a Software-as-a-Service market and development platform. Similarly, [139] proposes moving desktop simulation applications to the cloud via virtualized bundled images. These are generalist approaches that do not take into consideration the internal structure of the HPC applications, thus might not suffice for the resource-intensive computations required by HPC simulations.

C. ENDEAVORS TOWARD HPC AND BDA INTEROPERABILITY

The scientific community is aware that tools such as Apache Spark provide an interesting baseline for integration of scientific simulations in BDA environments. However, the data abstraction and application model of Spark are not easily supported using MPI, which is the main programming model in HPC [140]. Using Spark for HPC applications, while appealing, poses important convergence challenges.

The work in [141] introduced a methodology for graph processing to bridge the gap between Spark-based graph computing and HPC. Evaluations made on the Blue Waters supercomputer showed poor scalability of Spark vs. MPI+OpenMP for graph operations. In an effort to progress, Fox *et al.* presented in [142] a framework, named HPC-ABDS, that detected points for possible integration but also identified problems with workflow systems, data transport, and file management layers. Gittens *et al.* explored [143] the tradeoffs of performing linear algebra using Apache Spark compared with traditional C and MPI implementations on HPC platforms. The results showed a poor performance of Spark vs. MPI for matrix multiplications: from 2× to 25× performance gap. However, the authors highlighted the potential of incorporating MPI-based execution models to Spark, indicating that overheads might be tolerable. 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 but desired by the scientific community.

This paper presents an abstract architecture for execution model interoperability that, to the best of our knowledge, has not been introduced before in the literature. In addition,

TABLE 1. Summary of the main features of BDA and the HPC ecosystems.

	BIG DATA ANALYTICS ECOSYSTEM	
	DATA-CENTRIC PLATFORMS	**CLOUD, FOG**
Pros	Fault-tolerance by design Transparent data locality Productive programming interface Synergetic pre-built tools for composite jobs	Flexibility through virtualization Diverse local storage (NVRAM, SSD, scratch) Elasticity Massive geographic distribution
Cons	Low resource management control Significant memory overhead Poor support of binary input Deep software and communication stack Poor integration with simulation kernels	Resource sharing High latency Enterprise hardware Privacy concerns
	HIGH PERFORMANCE COMPUTING ECOSYSTEM	
	PARALLEL PROGRAMMING PLATFORMS	**SUPERCOMPUTER**
Pros	Exploit maximum parallelism Low overhead Generalist interface Bare-metal access	Top-tier hardware including accelerators Centralised Fast interconnections
Cons	Limited data abstractions Steep learning curve No native provenance nor replication Low portability	Decoupled storage Limited availability

we provide an implementation that allows users to benefit from efficient MPI libraries accessible from Spark with little effort on their side. As a result, we have developed a platform, called *Spark-DIY*, that provides advanced capabilities compared with other related solutions in the literature. For example, compared with [106], we provide compatible block management between the native side and Spark by using Java Native Interface (JNI). Compared with [100], our solution provides not only powerful I/O through MPI but also computing scalability. Moreover, our implementation constitutes a general solution that is not specific to any domain of data type, unlike the work presented in [118]. Our approach is more similar to the solution proposed in [144], but Anderson *et al.* used HDFS to exchange data among Spark and MPI, while we use memory directly for increasing efficiency. Moreover, we rely on an intermediate library based on MPI that manages the block communication graph, which avoids the burden of direct MPI usage.

Besides the former works, two platforms are very close to Spark-DIY in terms of aim and functionality. Spark-DIY is similar to Spark-MPI [145], a solution that extends the Spark ecosystem with the MPI applications using the Process Management Interface (PMI) to allow the creation of MPI processes from Spark. We relied on Spark-MPI to inspire the deployment mechanism of Spark-DIY, and we incorporated significant architectural and implementation features that make Spark-DIY much more complete and general. Alchemist [146] is another effort in this direction, focusing on the ability to call MPI-based libraries from Spark. Using Alchemist with Spark helps accelerate HPC computations, while still retaining the benefits of working within the Spark environment. The differences between Spark-DIYand Alchemist are mainly in terms of internal implementation.

V. CONVERGENCE CHALLENGES AND OPPORTUNITIES

To study the challenges and opportunities for convergence, we have summarized the main features of both ecosystems in Table 1. We now analyze the challenges and opportunities they yield for future convergence.

From the domain perspective, the iterative nature of the simulation algorithms clearly yields collective operations that do not fit nicely into the typical BDA paradigms. Therefore, significant efforts must be conducted to converge simulations and BDA algorithms [147].

Regarding workflow development and deployment, we conclude that a promising research line for large-scale scientific workflows would be working toward a hybrid approach between MPI and BDA-oriented data abstractions. Such a model would blend the slim MPI processes and their generalist nature with the ability to reason about data processing without explicitly implementing data parallelism that BDA platforms provide. The former features are highly desired by scientists who want to focus on their problem, rather than on the computational elements of their work.

This would result in a highly productive and efficient mechanism to build and deploy both scientific workflows and BDA applications, which is currently desired by the exascale community [147], [148].

Another major point arising related to data management in BDA solutions is the lack of flexibility for programmers to express complex data structures. This approach does not fit the complexity of data in HPC applications that need to show complex views of data to the users and the underlying system software. The data requirements of scientific applications are expected to become larger in the next few years, increasing the pressure on the parallel file systems, which are currently seen as a serious performance bottleneck. It becomes increasingly important to better understand application data models and to be able to efficiently map them on the underlying storage through novel techniques.

In addition, upcoming platforms will take into consideration other middleware aspects that made BDA platforms so successful, such as transparent fault tolerance. As a consequence, there is a need to integrate the fault tolerance techniques found in HPC, mostly oriented toward batch and iterative workloads (e.g., multilevel checkpointing), with the methods from the BDA side that tackle large volumes of tasks (e.g., data replication and provenance). This also has an effect on locality, and tradeoffs between these techniques must be addressed.

From the infrastructure side, we have seen that memory has become the limiting factor for new BDA platforms. We also note that emerging MTC scientific workflows require significant amount of memory for processing, caching, and exploiting in-memory solutions for enhanced performance. As a consequence, instead of tailoring the hardware to the execution of many small tasks, upcoming data-intensive infrastructures should heavily invest in both volatile and nonvolatile memory and deepen the storage stack. Hence, increasing memory in commodity clusters and clouds is key to supporting the upcoming execution platforms. These additional resources could mitigate the requirements of new workloads, and they would help to support the emerging in-memory and caching mechanisms coming from data-aware computing. In addition, this integration with more BDA-oriented infrastructures is expected to benefit pure HPC workloads in the near future [149].

To summarize, the desired confluence of BDA and HPC raises a number of challenges: overcoming the differences in cultures and tools; adopting new infrastructure architectures; ensuring the coexistence of stream and batch models; and coordinating resource allocation efficiently in virtualized and shared environments. Software libraries for common intermediate processing tasks need to be promoted, and a complete software ecosystem for application development is needed. The divergence of programming models and languages poses a convergence issue with regard not only to interoperability of the applications but also to the interoperability between data formats from different programming languages.

VI. CONVERGENCE ARCHITECTURE FOR HPC-BDA APPLICATIONS

Our approach is to integrate the data-centric and process-centric execution models 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 these execution models:

1) BDA users can rely on process-centric execution models to accelerate and scale their workloads.
2) HPC users gain access to high-level BDA libraries and increase their productivity.
3) Both types of users benefit from the flexibility to select the paradigm that matches their infrastructure, whether it is a cloud (BDA-oriented, and suitable for data-centric computing) or a supercomputer (HPC-oriented, and tailored for maximum communication and processing performance.) Furthermore, they could incorporate operations not typically available in their native settings.

Guided by our objective to offer the user the best features from each ecosystem, we formulate the following design goals for the integrated architecture:

D1 *Interoperability.* Process- and data-centric platforms target different canonical problems; therefore adapting a problem from one to the other should be explicit. To make the user aware of which model is currently active, we must keep both platforms separated but unified in terms of programmability and interoperable through explicit conversions.

D2 *Production readiness.* We believe that the viability of our solution will depend on being able to use standard versions of the underlying execution models without any changes. Thus, interoperability must be enabled through a middleware layer transparently to the users, so that applications built for pure platforms could run almost out-of-the-box.

D3 *Usability.* Although the user must be aware of the explicit interoperability, including overheads associated with switching contexts, the knowledge of the underlying data model and interoperation mechanisms should be minimal to preserve the nature of the programming and data interface. This would reduce the learning curve and minimize the impact on existing code.

D4 *Flexibility.* We want to support multiple data types and provide flexibility for different datasets to coexist in the same application. This includes the need to support stateful and stateless datasets.

D5 *Performance.* The data locality capability of data-centric execution models is a key feature and must be enforced as much as possible. On the other hand, the efficiency and scalability of process-centric execution models should be exploited whenever possible

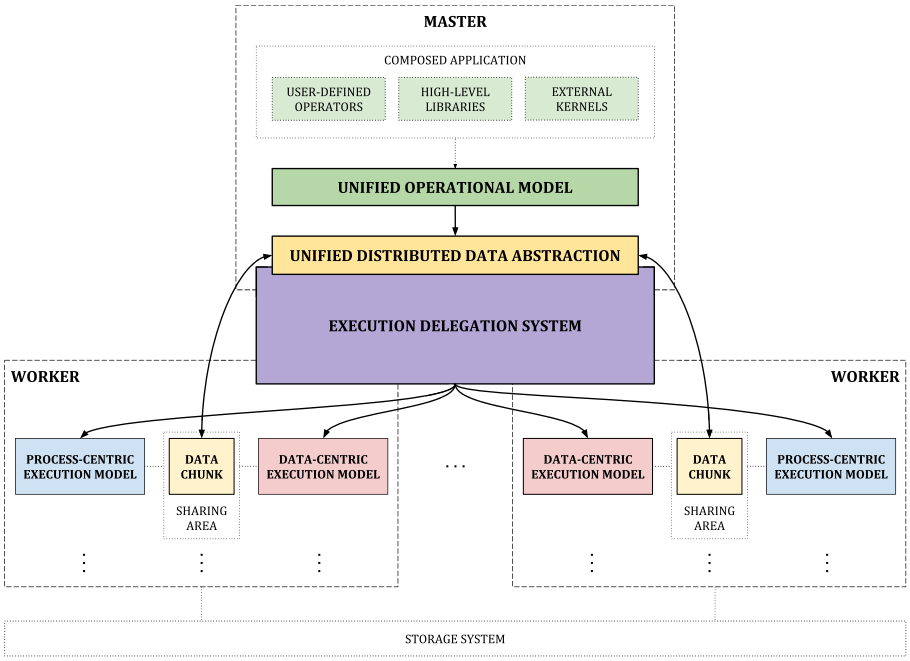


FIGURE 7. Overview of the abstract generalist architecture for HPC-BDA.

to accelerate communication- and compute-intensive operations.

These design goals are embodied in Figure 7, which depicts the interactions between the main components of our envisioned architecture:

- A *unified distributed data abstraction* for generic data types (D1, D3, D4).
- An associated *unified operational model* to interact with said data abstraction (D1, D3).
- An *execution delegation system* capable of selecting the appropriate execution model for each step in the application (D1, D2, D5).

These elements are detailed in the following sections.

A. UNIFIED DISTRIBUTED DATA ABSTRACTION

BDA data abstractions rely heavily on the concept of partition, block or chunk to manipulate large collections of records in a single-program-multiple-data (SPMD) manner. By distributing the overall data volume in chunks, data-centric platforms naturally abstract parallelism from the application developer. Moreover, since data drives the computation, and workload balance is planned on the fly based on results from intermediate steps, such abstractions facilitate the efficient allocation of computing resources with minimal intercommunication. In addition, these abstractions are typically immutable and stateless, in the sense that operations on these datasets result in a new dataset containing the updated records.

On the other hand, HPC applications are not enforced to use any specific abstraction, given their process-centric nature, in which computation and data parallelization are explicitly managed by the programmer. Nevertheless, such applications are usually built for primitive data types, since input and output data are normally stored in binary files and most operations are numerical. As opposed to the BDA abstractions, a key feature of HPC datasets is their need for statefulness, required to preserve the results from previous operations since data structures are reused.

Any architecture that aims to interoperate process- and data-centric execution models must be able to cope with the core characteristics of their respective data abstractions. In our design, we propose having a unified distributed data abstraction (UDDA) inspired by the data-awareness and task-based parallelism of data-centric abstractions (D3) but with the possibility to preserve state as required by HPC applications (D4). As shown in Figure 7, this abstraction represents a distributed collection of data organized in chunks, which can be locally accessible by both process- and data-centric computing units (D1).

Our analysis of features and requirements for HPC and BDA applications suggests that in order to implement this data abstraction the following properties should be enforced:

- Internal data types contained in the distributed data abstraction should not be limited. Collections of user-defined data types should be possible to preserve the semantic richness of BDA abstractions (D4).
- In order to ensure that the BDA SPMD operations and HPC process-centric computations hold simultaneously given a UDDA, the records contained in the collection must all belong to the same data type.
- The number of data chunks in a UDDA should be set automatically for the users' convenience, following the

trend in BDA platforms. However, HPC users sometimes need to impose a specific number of chunks to meet application domain limitations (e.g., when each data chunk represents an individual parametrisation of a domain). Therefore, data redistribution should be made available to support interoperability between datasets representing different domain topologies.
- Location of data chunks should be transparent to the users and respected as much as possible by the execution engines. In addition, users should not be aware of the underlying topological relationships between data chunks, neither for interacting with individual records nor for defining new operations on the overall dataset (D3). Nevertheless, implementations of a UDDA will have to rely on locality information to track chunks.

B. UNIFIED OPERATIONAL MODEL

BDA programming models are typically based on data flow, assuming that operations manipulate distributed datasets, and generate new datasets as result. They are massively inspired by functional programming and tend to avoid state changes, mutable data, and dependencies to global or local state. This has the benefit of providing identical results each time a function is called with the same input, regardless of previous operations. If there is no data dependency between such expressions, their order can be reversed, or they can be performed in parallel and will not interfere with one another. These features made these paradigms popular because it is easy to reason about data in this way, and building parallel program becomes less error-prone if the user does not have to take state into consideration. Statefulness also assists provenance, since operations can be reexecuted in case of failure.

In contrast, HPC programming interfaces rely heavily on in-place stateful paradigms, in which computation, communication, and data updates occur under the same programming scope. Intra and internode-level parallelism occurs at different levels, and the memory model is key to build an application since operations on data remain stateful and affect subsequent control flow and output results. Consequently, HPC applications are complex to design and code, and users are required to be much more aware of the implications of every change they conduct on the dataset.

Consequently, the flexibility of HPC programming paradigms can be sometimes overwhelming, while semantically rich paradigms are usually favored by end users because of higher productivity and smoother learning curve. Nonetheless, the declarative nature of BDA approaches excels in usability and adoption but lacks the capability to express the stateful procedural methods required in HPC.

Keeping hybrid applications in mind, which are composed of interleaving BDA and HPC stages, a unified architecture for BDA and HPC must clearly support traditional data-centric operations and incorporate HPC-oriented functionality (D1). In addition, as shown in Figure 7, it must also support the definition of operators for lambda expressions, while remaining compatible with existing implementations of high-level libraries (e.g., for machine learning or graph processing) and computing kernels (D3). The UOM integrates these needs into an abstract function space that aggregates all potential function definitions between two UDDAs. Our analysis indicates that certain specializations are necessary in order to expose a set of operations capable of meeting the requirements of composite applications, such as stateful functions, cardinality, and type-preserving functions.

C. EXECUTION DELEGATION SYSTEM

As depicted in Figure 7, the proposed architecture follows a master-worker scheme. With this structure, the definition of the application and the execution model-dependent parallel execution can be isolated, thus making clear for the user whether a task will be conducted locally or in a distributed manner (D1). The master entity holds the application, which defines the required UDDAs, and relies on the implementation of the UOM to interact with their content and to describe the steps it is composed of. On execution, parallel steps will be delegated to the worker entities through the execution delegation system (EDS), which will interpret the requested operations and select the appropriate execution model (D5). For increased adoption and simplicity to the end user, the EDS constitutes an entity that is independent of the underlying execution models and acts only as in intermediary without disrupting them (D2).

Given a UOM function space, a subset of functions will map to the data-centric execution model, and another subset will map to the process-centric execution model. The objective of the EDS is to enable the delegation of those operations to the appropriate execution model, further defining sets such subsets in relation to the execution models it must interact with.

VII. IMPLEMENTATION OF THE ARCHITECTURE: THE SPARK-DIY PLATFORM

In this section, we introduce an implementation of our generalist architecture for BDA and HPC, named *Spark-DIY*, based on Apache Spark and the highly scalable data-intensive communication pattern library DIY (*Do-It-Yourself block parallelism*) [150]. As a result, Spark-DIY is able to run Spark ultimately on top of MPI to enable the efficient execution of HPC operations on a supercomputer, to assist in the integration of existing scientific codes into a BDA environment, and to preserve the usability and flexibility of BDA tools.

Figure 8 shows the interactions between the main components of the proposed implementation in relation to the abstract entities described in the generalist architecture. The following sections explain their role from the end users' perspective and the accompanying internal behavior of the system.

A. SELECTED EXECUTION PLATFORMS

All implementations of the generalist HPC-BDA architecture must build upon existing execution platforms as building blocks. Their data abstractions, programming interfaces, and

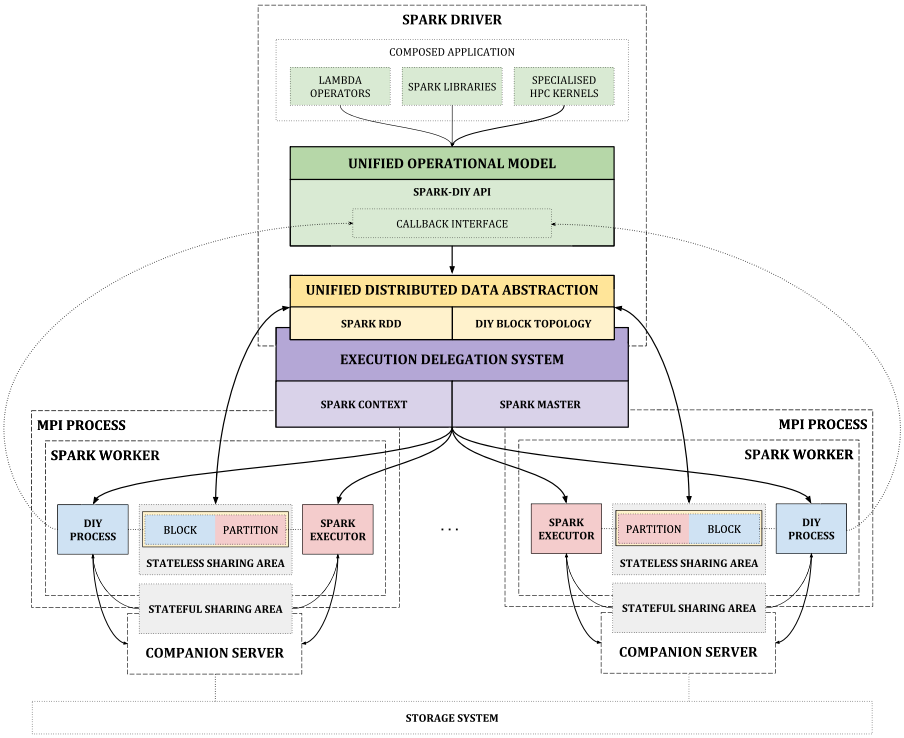


FIGURE 8. Implementation of Spark-DIY, including optimizations and enhancements.

execution models will impose technical limitations on the necessary interoperation mechanisms for each element in the architecture. Therefore, it is necessary to analyze in depth each execution model to find the key features that will enable the implementation of the architecture.

Below we describe major features of the execution platforms selected for this implementation.

1) DATA-CENTRIC EXECUTION PLATFORM: APACHE SPARK

Spark is arguably the most popular BDA processing framework, 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) [35]. A RDD is a read-only, resilient collection of objects partitioned across multiple nodes that hold 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 or moves them to local or distributed storage.

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*.

2) PROCESS-CENTRIC EXECUTION PLATFORMS: DIY

DIY is an C++ and MPI 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 transparently to the user.

Given a block decomposition and assignment to MPI processes, the user is able to run reusable communication patterns between the blocks in a neighborhood and global operations over all blocks, such as reductions. Therefore, DIY users can execute common communication patterns just by defining the block type and domain topology, without knowledge of the underlying communication details. Consequently, a problem can be decomposed into a large number of data-parallel subproblems, and data can be efficiently exchanged among them by using regular local and global communication patterns whose implementation has been tuned for HPC. DIY has been applied in a diverse array of science and analysis codes [151]–[155] and has demonstrated efficient scaling on leadership-class supercomputers.

B. INTEROPERATION MECHANISMS

The similarity between Spark RDDs and DIY block parallelism and the resemblance between Spark Map-Reduce and DIY merge-reduce communication patterns are the basis for our integration of these two models. We will emphasize the data and programming interfaces exposed by Spark as much as possible to preserve its compatibility with other tools, libraries, and platforms in the BDA ecosystem. Moreover, we will incorporate HPC features through the careful inclusion of DIY.

Given the design of the generalist architecture, three aspects of Spark and DIY need to be connected: their data abstractions, to implement the UDDA; their programming models, to implement the UOM; and their execution models, to implement the EDS.

The implementation details needed to connect each of these components between the two execution models are summarized in Figure 9 and detailed in the following sections.

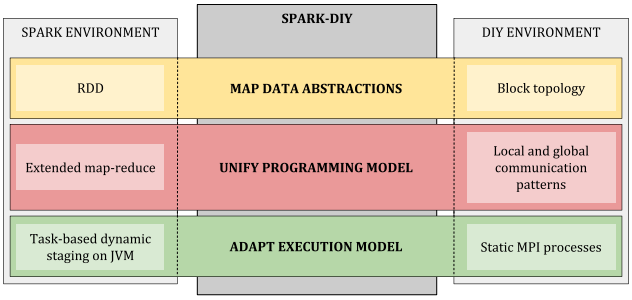


FIGURE 9. Interoperation mechanisms between Spark and DIY in the Spark-DIY platform.

1) IMPLEMENTATION OF THE UNIFIED DISTRIBUTED DATA ABSTRACTION

In the case of both Spark and DIY, the way data are arranged determines the development of algorithms and the behavior of the execution model. This also happens with respect to the UDDA defined in the architecture. Consequently, the first aspect that must be aligned is the way in which both frameworks represent their data abstractions, conforming to the definition of the UDDA. Briefing, UDDAs rely on a set of chunks and their associated state. Both Spark and DIY build upon the concept of partitioned datasets (RDDs and DIY block topologies, respectively), so first we need to establish a mapping between these two data abstractions.

If we think of the RDD as the equivalent of the distributed domain represented by a DIY block decomposition, each data partition in an RDD maps directly to a data block in DIY. In this context, the RDD dataset is partitioned into independent DIY blocks, where each partition P_i maps to a corresponding block B_i, preserving the same data elements inside the partition and respecting locality and order relationships, since no data transfers occur to build this mapping. As a consequence, the resulting dataset constitutes a distributed collection that reflects the inner structure of a RDD. DIY blocks are ordered in a 1-D ordering based on their global block ID, the subscript i in B_i. No other spatial or abstract ordering is assumed. While for a particular problem, a certain block topology (eg, a 3-D lattice) may reduce communication distance, the Spark-DIY framework is generic and does not know this. Nor do we provide any hints as to inform the framework, favoring flexibility and ease of use over potential performance gains in this case.

This is the basis for the implementation of the UDDA, since at this point we already have two mechanisms to handle distributed collections of data, and a suitable mapping between the data chunks. An additional benefit of relying heavily on RDDs to implement the UDDA is the fact that we can exploit the Spark framework to control data partitioning and enforce locality. These two features are closely related to the locality and cardinality properties of UDDAs. Nevertheless, RDDs are stateless, and UDDAs require the possibility to include statefulness in their definition. This is a complex technical challenge we tackle in Sec. VII-C.

Another aspect the UDDA must address is resilience. It should come as no surprise that the underlying data models may handle resilience differently; and in fact this is the case with RDDs and DIY blocks. RDDs are fundamentally resilient, whereas DIY blocks, usually maintained only in main memory of an underlying MPI program, can be lost in the case of unexpected termination. DIY offers an out-of-core mode where blocks are cached to disk, and it also has parallel I/O functionality to read/write all blocks from/to a DIY file in a parallel file system. In our current implementation, we use only the main memory features of DIY. The UDDA therefore changes its resilience characteristics depending on whether the UDDA currently represents a Spark RDD or a DIY block. Although the program is temporarily non-resilient while in a DIY section, the resilience properties of the UDDA are restored when switching back to a Spark section. Furthermore, if a DIY section is long-running and resilience is a concern, then checkpoint-restart can be used to address it.

In this regard, our approach can take advantage of VeloC [156], an exascale-ready checkpointing system that leverages heterogeneous storage hierarchies to implement multilevel resilience strategies. Two key features of VeloC are particularly interesting in this context: (1) it exposes a memory-based API that is well suited to protect the critical data structures stored in main memory by DIY; and (2) it implements an asynchronous mechanism that hides the overhead of the resilience strategies in the background, while DIY continues running. Therefore, with this approach, DIY sections can be made resilient with minimal overhead. Note that Spark also makes use of checkpointing to prune the lineage (needed to recover lost RDD partitions) or provide stateful transformations. Thus, by delegating checkpointing to VeloC instead of the native Spark implementation, a unified checkpointing mechanism can be implemented that exploits synergies between RDDs and DYI blocks to further reduce the overhead of resilience.

2) IMPLEMENTATION OF THE UNIFIED OPERATIONAL MODEL

Spark actions and transformations constitute a complete interface that covers many functions in the UOM, with the particularity that all of them are stateless. On the other hand, DIY offers further flexibility to incorporate its communication patterns and stateful operations, while providing support to interact with native code, existing simulation kernels, and parallel file systems. Since the Spark API already offers a comprehensible programming interface that is easily expandable, we preserve it in our implementation of the Spark-DIY API with the addition of new operations. Ultimately, users would write a Spark program that can be enriched with this new functionality.

Spark-DIY operations on partitions are triggered by the internal algorithms in DIY but expressed as user-defined callbacks written by the user in Scala as part of the driver code, who also defines the data type of the records and the supported operators (e.g., *unary* for independent transformations, *binary* for reductions, *hash* for partitioning, and *kernel* for invoking native code). Moreover, high-level libraries remain available through the usual Spark API.

3) IMPLEMENTATION OF THE EXECUTION DELEGATION SYSTEM

Figure 8 shows the deployment of Spark-DIY and the interaction of the Spark and DIY components that constitute this implementation of the EDS. Starting from the Spark driver, which is the component that guides the entire execution, tasks will be executed either as executors spawned inside the Spark workers or as DIY processes. Spark workers are deployed as an MPI application so that a valid communicator exists for DIY operations before their execution. This assists the adaptation of the dynamic task-based execution model from the Spark framework to the static set of MPI processes used by DIY.

The EDS relies on the Spark context for data partitioning and the Spark master for task scheduling and serialization. Moreover, a middlelayer handles task delegation to DIY processes for specific Spark-DIY functions and the implementation of the data mapping for the UDDA.

Given the previous implementation of the UOM, pure Spark operations will be delegated to the data-centric execution model, namely, Spark executors. The remaining computing-intensive operations will be delegated to the DIY execution model, thus being executed in MPI processes. Ultimately, all Spark-DIY operations start by spawning Spark executors, which will then delegate the operation to DIY code. Upon invoking a function that is delegated to DIY, several tasks are conducted internally:

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 operator creates executors that live in the MPI environment and contain the data of the corresponding partition, which enforces locality.
2) *Map RDD partitions to DIY blocks.* 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. In order to achieve this, in most cases data needs to be copied from the Java to the native side.
3) *Delegate operation 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.

C. OPTIMIZATIONS AND ENHANCEMENTS

In the former implementation the specific features of each execution model sometimes limit the potential performance and functionality that can be attained. For example, dealing with generic user-defined data types on the DIY side involves conducting more serialization steps than would be required for primitive data types. Another issue is that stateful functions cannot be supported without external assistance. This section describes the optimizations and extensions we incorporated into this implementation to fully support the generalized architecture, and extend the functionality enabled by the selected execution platforms.

We have conducted optimizations to solve two kinds of issues: limitations to the implementation the UDDA and UOM and performance problems related to the way in which execution platforms interoperate. For the first case, we have to find a way to support stateful functions and persistent datasets, which cannot be done out of the box since we ultimately rely on Spark RDDs and executors, which do not preserve state. For the second case, the need for generality in the data abstractions adds significant overhead in terms of memory and execution time due to the need for additional serialization. Moreover, HPC-oriented storage is currently not supported because all data I/O is supposed to be handled through the Spark context, thus forcing applications to conduct additional stages to make input data suitable for subsequent process-centric operations. These circumstances add significant overhead and limit the performance and scalability of applications built with this platform.

The following paragraphs describe the elements in Figure 8 that are related to these improvements. After these enhancements, Spark-DIY is capable of supporting the three base application models depicted in Figure 10: applications requiring DIY transformations with RDDs as input and output; data ingestion from a Spark RDD into a process-centric application; and iterative analysis of data resulting from a process-centric operator. These base models can be combined

to design more complex applications, which makes Spark-DIY much more flexible and general than other solutions such as Alchemist [146].

1) SHARED-MEMORY REGIONS FOR STATEFUL OPERATIONS

In order to integrate RDDs and stateful functions, we introduce a new architectonic element capable of preserving state after an RDD operation, which means the dataset is reused and updated with the results of such operation. This entity is depicted in Figure 8 as a *companion server* that is intrinsically associated with a specific Spark worker in a peer-to-peer manner and can communicate with the executors spawned in it.

The companion server is responsible for managing a shared-memory region that is allocated, attached, and maintained during the whole execution of this entity. Therefore, it can hold a data chunk and maintain it even after the task that runs an operation finishes and the corresponding executor dies. This allows functions to update the values in the data chunk and preserve the results without returning a new dataset, effectively meeting the statefulness requirement.

As a result, this implementation splits the data-sharing area defined in the generalist architecture into two regions: a stateful data-sharing area maintained by the companion server and a stateless data-sharing area used as intermediate in-memory storage for communicating the Spark and DIY execution platforms.

2) DATA SERIALIZATION MINIMIZATION

The Spark-DIY API exposes operations on data abstractions that are generic and can be tailored for user-defined data types. Doing so, however, has significant performance implications since the internal memory management involves several serialization and deserialization steps, not just on the Spark side, but also on the DIY side and the code that bridges them.

Nevertheless, optimizations can be conducted if collections are limited to native data types. Spark-DIY offers an interface for these types with reduced overhead, since serialization between Spark and DIY is not required. In addition, for datasets containing native data types, data can be shared directly between the RDD partition and the DIY block, which reduces the number of copies conducted during the delegation process.

The impact of these optimizations can lead to a significant reduction in the data manipulation overhead introduced by Spark-DIY each time a partition is processed in DIY, as indicated in Table 2. This is especially useful for scientific tasks, since most of their data are numeric.

TABLE 2. Comparison of the data manipulation time for a collection of integers in the initial approach to Spark-DIY, and its optimization for primitive data types.

Block Size	Initial	Optimized	Improvement
0.25M	2.67s	0.05s	98.12%
1M	11.44s	0.07s	99.40%
2M	21.60s	1.08s	95.00%

3) EXTERNAL DATA MANAGEMENT VIA MPI I/O

The current Spark-DIY implementation relies on the Spark context to interact with storage, thus limiting the I/O possibilities for HPC-oriented stages and forcing the addition of auxiliary operations to make input data suitable for subsequent process-centric operations.

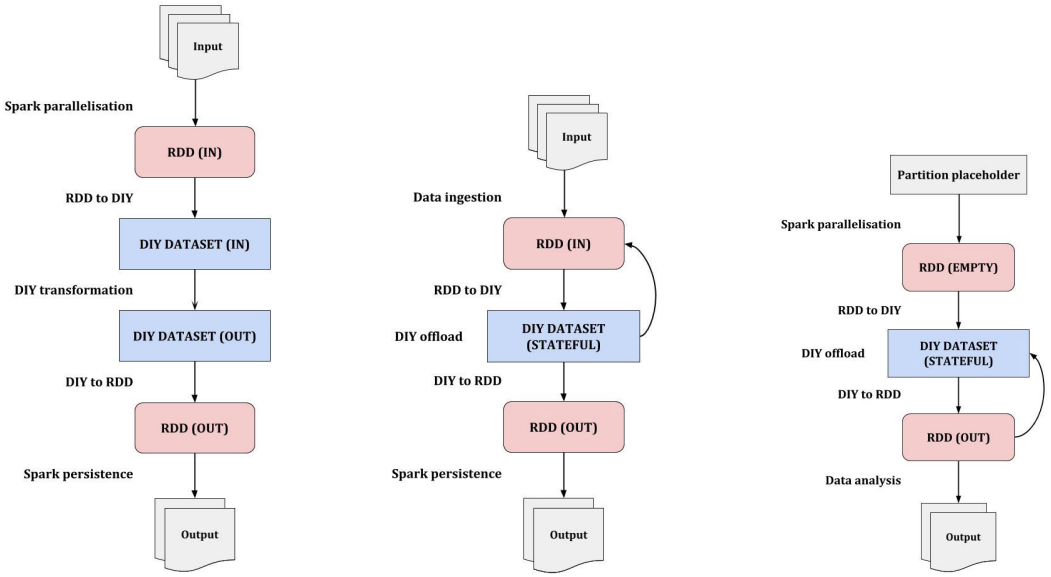
To overcome this issue, we use once again the companion server entity to act as an intermediary proxy to the storage system, which can now also be a HPC-oriented parallel file system. The companion server is implemented as an MPI algorithm capable of conducting collective and parallel operations on input data, which is now placed in the stateful shared data region for subsequent usage from either the Spark or the DIY operators. This enables all the potential of MPI I/O to benefit from the highly optimized parallel I/O in HPC systems as an alternative to current big data storage systems such as HDFS.

D. USAGE

Basically, in Spark-DIY, end users can execute their applications without major modifications. The aaplications are exposed to a limited number of additional elements of the interoperation layer in addition to the basic Spark API, but they are supplied as a driver by the Spark-DIY system. 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.* A catalog is offered where users can select a prebuilt 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 catalog, we have also developed the internals of Spark-DIY in a generic manner. New data types can be defined in a helper file that is later used by the proxy code generation utility. New data types must define serialization and deserialization functions, since both RDD and DIY block elements need to be serializable.
2) *Define the callback operators for the record.* Similarly as in Spark, the operations to be conducted on records must be defined. To access these operators from DIY, users must implement the proper method as an object that extends the Spark-DIY callback interface.
3) *Delegate execution on a DIY dataset.* Once an RDD is created, along with its operators, we can run the desired operation. Its result is a new RDD that can be further used in the driver with subsequent combinations of Spark functions or DIY algorithms.

Listing 1 shows a simple word count application written in Spark-DIY, which maps to the first application model in Figure 10. Lines 11-14 conduct a pure Spark map but



(a) DIY transformations. (b) Data ingestion. (c) Iterative analysis.

FIGURE 10. Base application models supported by Spark-DIY.

```
1  class WordcountCallback extends PairDIYCallback {
2   override def binary(x:PairRecord,
3                       y:PairRecord): PairRecord = {
4    new PairRecord(x.first, x.second + y.second)
5   }
6   override def key_hash(x:PairRecord): Long = x.first.##
7  }
8
9  def main() {
10  // ...
11  val mapRDD = sparkContext
12               .textFile(file)
13               .flatMap(_.split(" "))
14               .map(x => new PairRecord(x, 1))
15
16  val mapDDD = new PairDDD(mapRDD, numBlocks,
        sparkContext, statefulness)
17  val outRDD = mapDDD.DIYreduceByKey(new
        WordcountCallback())
18  // ...
19 }
```

Listing 1. Example of a word count application written with the Spark-DIY *DIYreduceByKey* operator.

```
1  class KernelCallback extends OffloadDIYCallback {
2   override def kernel() = {
3    // Call kernel interface
4   }
5   override def input(ptr:Long) = {
6    // Arrange data in memory
7   }
8   override def output(ptr:Long) = {
9    // Retrieve data from memory
10  }
11 }
12
13 def main() {
14  // ...
15  for (i <- 0 to iterations)
16   var mapRDD = streamContext
17                .socketTextStream(host, port)
18                .flatMap(_.split(" "))
19                .map(x => x.toDouble)
20
21   var offDDD = new OffloadDDD(mapRDD, numBlocks,
         sparkContext, stateful)
22   var outRDD = offDDD.DIYoffload(new KernelCallback())
23  }
24  // ...
25 }
```

Listing 2. Example of a data ingestion application written with the Spark-DIY *DIYoffload* operator.

create an RDD of the DIY data type *PairRecord*. This RDD is used as input to generate a DIY dataset in line 16, which is the input for the DIY reduction in line 17. Notice that the creation of such a dataset involves the specification of the base RDD, the final cardinality of the dataset (variable *numBlocks*), the Spark context that will assist task management, and whether the dataset is stateful or not. The output dataset is an RDD of the same data type that can be used in subsequent Spark operations. Lines 1 to 6 contain the definition of the callbacks required for the reduction and the key-based partitioning of data records. These functions will be invoked by the underlying DIY algorithms, so they must comply with the callback interface exposed by Spark-DIY.

Listing 2 shows a simple iterative data ingestion application written in Spark-DIY (corresponding to the second application model in Figure 10). Inside the loop, first we get the input data from a stream as a collection of primitive doubles (lines 16-19). Such data are offloaded to a DIY operator (line 21) that links to a simulation or processing kernel defined in the kernel interface, that describes how the kernel is called, and how input and output data are arranged (lines 1-10). The requested dataset will be stateful in this case, because we will update its contents with fresh data obtained in the following iteration.

VIII. USE CASE: THE ENKF-HGS HYDROGEOLOGIC DATA ASSIMILATION WORKFLOW

Having good-quality predictions of the behavior of hydrological environmental systems is key for water management. These systems are represented by complex mathematical models that incorporate numerous elements and rely on

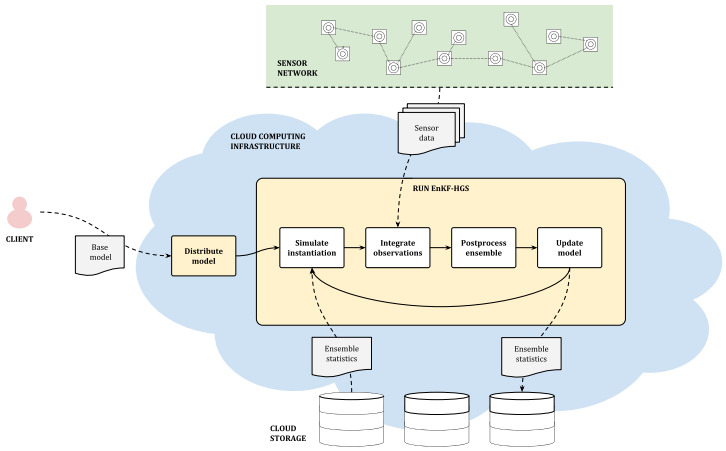


FIGURE 11. Interoperation of EnKF-HGS with the data assimilation sensor network and its supporting cloud infrastructure.

multiscale non linear processes and matrix operations. The inherent computational complexity of these computations has led scientists to implement parallel versions of these models in the form of tailored simulators for multicore environments. EnKF-HGS is one of the state of the art simulators in the hydrology domain to provide functionality for real-time stochastic simulations of the groundwater and surface water profiles.

EnKF-HGS is a representative use case of many applications currently found in scientific computing [157]–[160] that can be solved by the Monte Carlo method, which involves a large random sampling to determine the properties of some phenomenon or system behavior. Each simulation in the ensemble of realizations represents a long-running compute- and I/O-intensive process, which comprises the sequential execution of two proprietary simulation kernels: an input data filter, named GROK, and HydroGeoSphere (HGS) [161]. Each model realization represents an instantiation of the numerical model provided with a different combination of input parameters and system conditions. In order to update the model with the assimilated data, a communication-intensive stage must be conducted afterwards, which is implemented by using MPI in the original version of EnKF-HGS.

Recent technological and mathematical advances allowed significant improvements in the precision of the simulations by integrating data acquisition techniques with the modeling process [162]. This allows the aggregation of data from different sources and presenting them to the user in a single format for further analysis, by using a cloud-based data integration system to store and explore data [163]. A similar approach was proposed for EnKF-HGS, including an architecture for a system combining a wireless environmental monitoring module as data source and a cloud-based computing service to perform environmental simulations [164]. The tool is also meant to incorporate real-time sensor data to refine its predictions. This motivates the need to develop a version of the simulator able to exploit BDA features such as cloud and streaming support, which is currently not easily available for MPI environments. Figure 11 depicts the elements involved in EnKF-HGS operations with real-time data assimilation in the cloud: 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 deployed in the Swiss valleys; after each step, results are stored in a distributed manner in cloud storage for subsequent iterations. The combination of these performance and infrastructure requirements makes a case for the need for scalable HPC-BDA convergence in EnKF-HGS. Moreover, this would be beneficial for other applications that show similar needs to fuse sensor and simulation data, including weather forecasting [165] and carbon cycle [166] studies.

In [167], we reported our experience combining traditional HPC with BDA-inspired paradigms and platforms, in the context of scientific ensemble workflows such as EnKF-HGS. Our goal was to provide a suitable environment that combined the HPC and BDA 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 because of the large memory requirements, and it generated errors and did not scale well for the Kalman filter cooperative processes, mainly due to the shuffle phase in large-scale reductions, combined with the overhead of the platform. Limitations of the shuffle phase have been reported by others [168], [169] as well. They can be traced back to multiple causes: high memory utilization for buffering of shuffle blocks, load imbalance, explosion of files, high I/O contention, etc. The following sections describe how this application can be enhanced to cope with its scalability and interoperability requirements through Spark-DIY and its composition capabilities. The results of its optimizations as scale increases are also presented.

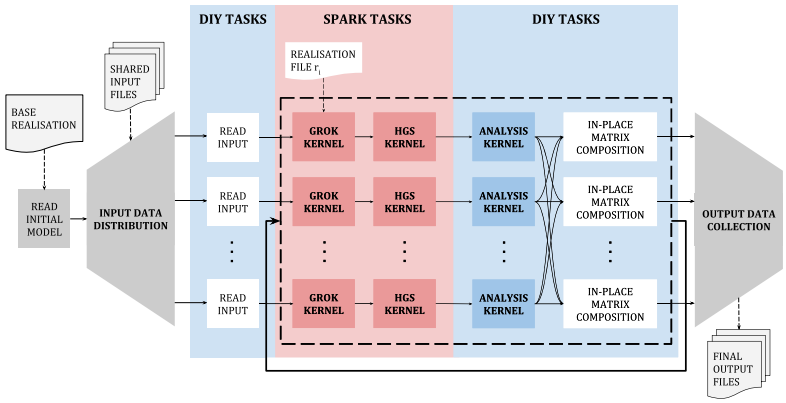


FIGURE 12. Implementation of data-centric EnKF-HGS on Spark-DIY. The data assimilation is mapped to Spark tasks, and the model update region is delegated to DIY processes. Input data are read via MPI I/O through DIY tasks in collaboration with the companion servers.

A. BUILDING ENKF-HGS WITH SPARK-DIY

EnKF-HGS is based on the data ingestion application model, because it incorporates stream data into the simulation stage in an iterative manner. The result the implementation of EnKF-HGS in Spark-DIY was an application with a parallel region in charge of executing the GROK and HGS kernels in Spark tasks for data assimilation, and a model update region that delegates matrix analysis to DIY as an MPI kernel through the Spark-DIY *offload* operator. This structure is depicted in Figure 12.

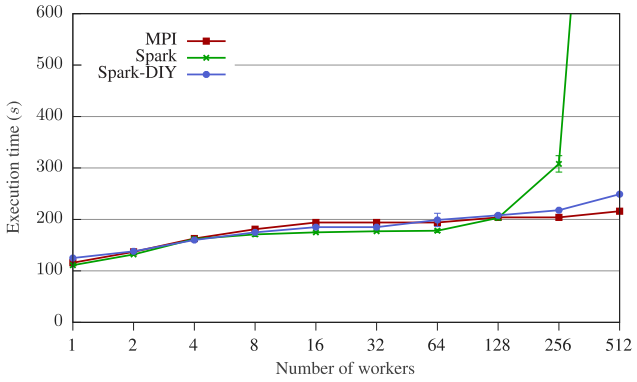
This implementation of EnKF-HGS combines the shared-memory enhancements of Spark-DIY to enable statefulness in the model analysis region of the application. Delegated DIY tasks are also used to incorporate the base input data in the appropriate columnar view, reading through the companion servers via MPI I/O, and getting the result into Spark by means of DIY operations. The legacy kernels can be executed as usual afterwards, and the results are fed to the MPI analysis kernel invoked by DIY, which will execute the filtering stage and update the model inplace without needing any data collection in the driver process. Subsequent iterations will continue to operate in a distributed manner until the final output is stored in chunks. An additional benefit of using Spark-DIY in this case is that we can reuse the original model update kernel implemented in MPI, which drastically reduces the development effort to obtain a full implementation of a BDA-capable EnKF-HGS. Moreover, we can exploit some features specific to Spark-DIY to our advantage:

- The shared-memory regions that support stateful operations have a main role in this implementation because the in-place update of the model is necessary to eliminate the need to collect the matrices in the driver process. This is because of the analysis kernel being inherently stateful. Furthermore, these regions minimize the serializations between iterations because they remain active after they are created by the companion servers. This has a positive effect on the reduction of data management overhead.
- The optimizations introduced in Spark-DIY tackling primitive data types also contribute to reducing the negative effects of serialization on performance. Since EnKF-HGS relies on numeric algorithms, all of its data are handled in terms of floats or integers, thus simplifying the management of shared buffers.
- The MPI I/O external data management represents a major benefit versus a pure Spark implementation because of the way in which input data is organized. Input files can be read in parallel to retrieve the particular data required in each process, exactly in the way it is needed (i.e., in columns). This is advantageous because MPI I/O has good performance and scalability, and enables future optimizations for data management. Moreover, data are read directly into the shared-memory region of the companion server, reducing the overhead of passing data from Spark to DIY. Although this has the drawback of adding some overhead due to the need to involve the companion servers, it effectively removes any input bottlenecks in the driver. A similar argument could be provided in favor of storing the output directly through this mechanism.

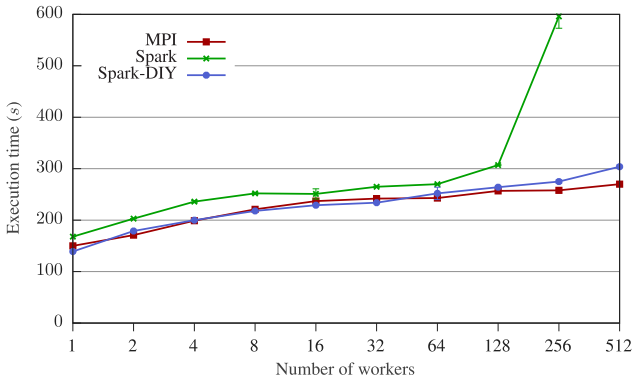
B. EXPERIMENTAL SETUP

For experimental evaluation, we have analyzed the behavior of EnKF-HGS implemented in Spark-DIY with the original MPI implementation and a version written in pure Spark, as presented in [170].

The experiments were run on 44 bare metal nodes of the Chameleon cloud at the University of Chicago running Apache Spark version 2.2.0. Each node has an Intel Xeon CPU E5-2670v3@2.30GH (Haswell) processor with 12 physical cores and 135 GB 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 an MPI process. Each computing unit (i.e., executor) was given 3 GB of memory.



(a) Execution time for one iteration.



(b) Execution time for two iterations.

FIGURE 13. Evaluation results of EnKF-HGS on Spark-DIY measured in execution time (in seconds) for one (a) and two (b) iterations.

The behavior was measured as the number of realizations increases and with a constant workload per realization. Two situations were analyzed: running only one iteration and running several iterations. However, for this paper we report results for one and two iterations of the Kalman filter, because succeeding iterations incorporate further randomness that could lead to inadequate comparisons. Both cases cover the entire application workflow.

C. PERFORMANCE RESULTS

Figure 13 shows the average results of the evaluation of EnKF-HGS on Spark-DIY to compare its performance, measured in execution time, with a pure Spark implementation of EnKF-HGS, and its original MPI implementation taken as baseline. The three implementations were evaluated on the Chameleon testbed with real initial input data and synthetic data for assimilation to limit the heavy stochastic nature of the simulation kernels. The experiments were run five times, resulting in standard deviations lower than 1% of the total execution time in 77% of the cases. The remaining cases always show standard deviations lower than 6% of the total execution time.

The results for one iteration show that Spark-DIY performs similarly to Spark, which is positive because this means the delegation layer is not introducing significant overheads. In addition, both implementations show good results against MPI when more than one node is involved in the computation.

In the case of more iterations, the Spark implementation is outperformed by MPI and also by Spark-DIY an average of 14% for 1 to 128 workers. This is related to the need for data collection for the analysis stage that must be conducted to prepare the model for following iterations after the parallel region. This step is necessary in the Spark implementation because we cannot conduct in-place computations, but it is not the case for the current implementation of Spark-DIY, which shares the analysis kernel with the MPI implementation. The most remarkable result is, however, that the Spark implementation fails to scale properly beyond 128 workers. We tracked this issue to a series of structural bottlenecks in the driver process that were related to I/O management of the initial dataset and the final output. In addition, Spark does not support stateful operations, which implies that data resulting from the parallel region must be collected to conduct the model update, creating a bottleneck that limits scalability beyond 128 realizations.

As a consequence of the I/O and shared-memory management capabilities of Spark-DIY, the Spark-DIY implementation is consistently competitive with MPI as the number of realizations increases and achieves comparable scalability. The reason for this is the key effect of statefulness in the overall application: it not only improves performance by eliminating overheads related to data being serialized and moved around as the procedure advances but also eliminates the need for collecting data to conduct the update of the model. These memory regions allow the analysis kernels to conduct the model update in-place, without involving the driver process at all. Furthermore, initial input data can be read in parallel and placed directly in the corresponding memory region, eliminating the input data processing bottleneck.

Nonetheless, the detrimental effects of Spark task generation, scheduling, and management are visible when the number of workers becomes very high. For example, Spark-DIY takes 16% more execution time than MPI for 512 workers, which highlights the slim nature of the MPI environment. We believe this is a reasonable tradeoff for the flexibility of incorporating the elements of a whole new ecosystem into an HPC application, and further optimizations might alleviate this issue in future works.

IX. CONCLUSION

This paper has presented an architectural framework to facilitate the convergence of the HPC and big data worlds. This framework enables the interoperability of established BDA and HPC execution models and allows the users to create and deploy hybrid applications including big data and HPC components without forcing the users to rewrite one or the other. This way, users can exploit the best tools in both worlds into the same application.

The paper has also presented a generalist execution model interoperability architecture for HPC-BDA applications relying on a formal definition of a generic unified distributed data

abstraction (UDDA) and its associated unified operational model (UOM), which sets the foundation of a theoretical frame for the analysis and definition of composite HPC-BDA applications.

To show its feasibility, we have implemented Spark-DIY, an instantiation of the former architecture based on Spark and MPI execution platforms. We evaluated it in a real-world use case from the hydrogeology domain enriched with features enabled by our architecture. The evaluation shows good performance and scalability for communication-intensive operations in comparison with Spark and enables the integration of elements from both the BDA and HPC ecosystems for applications with diverse requirements.

Future work could enhance the architecture to support heterogeneity, which would be beneficial for users relying on these hardware elements for the scalability of their HPC applications (e.g., image processing) or aiming to accelerate specific portions of their BDA workloads (e.g. deep learning). Future advancements should also aim to reincorporate some features that are desired by BDA users in production environments yet are left behind by the architecture in its current state, such as multitenancy, fault tolerance, and elasticity. In addition, usability and productivity could be addressed with formal studies by the BDA and HPC user communities.

Further real-world use cases could be built by using Spark-DIY to incorporate the potential of higher-level libraries, so that the HPC community can benefit from the myriad libraries and platforms built on top of Spark without giving away scalability. Spark's ease of use is lacking in the HPC software stack, and Spark-DIY provides HPC practitioners of such characteristics that are commonplace in the BDA world. Exemplars from the BDA side, such as HPDA or IoT applications, would be particularly interesting. We will also look for hybrid use cases that require Spark computation on RDDs interleaved with HPC computation.

We will also conduct further experiments on the use case we presented. In particular, we will experiment with other computing platforms (HPC, cloud) to evaluate the portability, usability, and flexibility of our approach. This will also require an analysis of the state of the system in terms of disk and network usage. We will also evaluate different configurations of Spark with multicore workers and compare performance with Spark-DIY, which also supports multiple blocks per MPI rank. Future work also includes keeping Spark-DIY up to date with the latest versions of Spark and DIY, for example, Spark's recent move from RDDs to DataFrames.

Results presented in this paper were obtained by 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, and A. Pflieger, "ETP4HPC strategic research agenda achieving HPC leadership in Europe," European Technology Platform (ETP) for High-Performance Computing (HPC), European Union, Tech. Rep. SRA 3, 2017.

[2] "European big data value strategic research and innovation agenda," Big Data Value Assoc., European Union, Tech. Rep. SRIA 3, Oct. 2017.

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

[4] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, and B. de Supinski, "Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry," *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 4, pp. 435–479, 2018.

[5] BDVA-ETP4HPC, "The technology stacks of high performance computing and big data computing: What they can learn from each other," Big Data Value Assoc., Eur. Technol. Platform High-Perform. Comput., European Union, Tech. Rep., Nov. 2018.

[6] S. Caíno-Lores, J. Carretero, B. Nicolae, O. Yildiz, and T. Peterka, "Spark-DIY: A framework for interoperable spark operations with high performance block-based data models," in *Proc. IEEE/ACM 5th Int. Conf. Big Data Comput. Appl. Technol. (BDCAT)*, Dec. 2018, pp. 1–10.

[7] D. Laney, "3D data management: Controlling data volume, velocity, and variety," *META Group Res. Note*, vol. 6, no. 70, p. 1, 2001.

[8] P. Zikopoulos and C. Eaton, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. New York, NY, USA: McGraw-Hill, 2011.

[9] P. Russom, "Big data analytics," *TDWI Best Practices Rep.*, vol. 19, no. 4, pp. 1–34, 2011.

[10] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*. New York, NY, USA: Manning Publications, 2015.

[11] C.-W. Tsai, C.-F. Lai, H.-C. Chao, and A. V. Vasilakos, "Big data analytics: A survey," *J. Big Data*, vol. 2, no. 1, p. 21, Oct. 2015, doi: 10.1186/s40537-015-0030-3.

[12] D. Talia, P. Trunfio, and F. Marozzo, *Data Analysis in the Cloud: Models, Techniques and Applications*. Amsterdam, The Netherlands: Elsevier, 2015.

[13] P. Mell and T. Grance, "Effectively and securely using the cloud computing paradigm," in *Proc. Inf. Technol. Lab. (NIST)*, 2009, pp. 304–311.

[14] F. Zulkernine, P. Martin, Y. Zou, M. Bauer, F. Gwadry-Sridhar, and A. Aboulnaga, "Towards cloud-based analytics-as-a-service (CLAaaS) for big data analytics in the cloud," in *Proc. IEEE Int. Congr. Big Data*, Jun. 2013, pp. 62–69.

[15] P. Raj, A. Raman, D. Nagaraj, and S. Duggirala, *High-Performance Big-Data Analytics Computing Systems and Approaches*. Cham, Switzerland: Springer, 2015.

[16] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan, "Computational solutions to large-scale data management and analysis," *Nature Rev. Genet.*, vol. 11, no. 9, pp. 647–657, Sep. 2010. [Online]. Available: https://www.nature.com/articles/nrg2857

[17] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Commun. Mobile Comput.*, vol. 13, no. 18, pp. 1587–1611, Dec. 2013.

[18] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[19] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5. New York, NY, USA: ACM, 2015, pp. 37–42.

[20] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, Oct. 2014.

[21] B. Nicolae, P. Riteau, and K. Keahey, "Towards transparent throughput elasticity for IaaS cloud storage: Exploring the benefits of adaptive block-level caching," *Int. J. Distrib. Syst. Technol.*, vol. 6, no. 4, pp. 21–44, 2015.

[22] B. Nicolae, P. Riteau, and K. Keahey, "Bursting the cloud data bubble: Towards transparent storage elasticity in IaaS clouds," in *Proc. 28th IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Phoenix, AZ, USA, May 2014, pp. 135–144.

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

[24] J. Fritsch and C. Walker, "The problem with data," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput. (UCC)*, Jun. 2014, pp. 708–713.

[25] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[26] T. White, *Hadoop: The Definitive Guide*. Newton, MA, USA: O'Reilly Media, 2009.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. (MSST)*, May 2010, pp. 1–10.
[28] F. J. Clemente-Castelló, B. Nicolae, R. Mayo, and J. C. Fernández, "Performance model of MapReduce iterative applications for hybrid cloud bursting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1794–1807, Aug. 2018.
[29] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified relational data processing on large clusters," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 1029–1040.
[30] R. Tudoran, A. Costan, and G. Antoniu, "MapIterativeReduce: A framework for reduction-intensive data processing on azure clouds," in *Proc. 3rd Int. Workshop MapReduce Appl. Date*, 2012, pp. 9–16.
[31] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 810–818, doi: 10.1145/1851476.1851593.
[32] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 285–296, 2010.
[33] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, "Scalable parallel computing on clouds using twister4azure iterative mapreduce," *Future Generat. Comput. Syst.*, vol. 29, no. 4, pp. 1035–1048, 2013.
[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput. (HotCloud)*, Berkeley, CA, USA, 2010, p. 10.
[35] 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 *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, 2012, p. 2.
[36] X. Shi, M. Chen, L. He, X. Xie, L. Lu, H. Jin, Y. Chen, and S. Wu, "Mammoth: Gearing Hadoop towards memory-intensive MapReduce applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2300–2315, Aug. 2015.
[37] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *Proc. 15th Workshop Hot Topics Oper. Syst. (HotOS XV)*, May 2015.
[38] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, *How Data Volume Affects Spark Based Data Analytics on a Scale-Up Server*. Cham: Springer, 2016, pp. 81–92, doi: 10.1007/978-3-319-29006-5_7.
[39] L. Salucci, D. Bonetta, and W. Binder, "Lightweight multi-language bindings for apache spark," in *Proc. 22nd Int. Conf. Euro-Par Parallel Process.*, vol. 9833. New York, NY, USA: Springer-Verlag, 2016, pp. 281–292, doi: 10.1007/978-3-319-43659-3_21.
[40] A. Singh, M. Mittal, and N. Kapoor, "Data processing framework using apache and spark technologies in big data," in *Big Data Processing Using Spark in Cloud*. Singapore: Springer, 2019, pp. 107–122.
[41] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *Proc. 1st Int. Workshop Graph Data Manage. Exper. Syst. (GRADES)*, New York, NY, USA, 2013, pp. 2:1–2:6.
[42] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, and A. Ghodsi, "Spark SQL: Relational data processing in Spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[43] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.
[44] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive—A petabyte scale data warehouse using hadoop," in *Proc. IEEE 26th Int. Conf. Data Eng. (ICDE)*, Mar. 2010, pp. 996–1005.
[45] S. R. Mihaylov, Z. G. Ives, and S. Guha, "REX: Recursive, delta-based data-centric computation," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1280–1291, 2012.
[46] C. Dobre and F. Xhafa, "Parallel programming paradigms and frameworks in big data era," *Int. J. Parallel Program.*, vol. 42, no. 5, pp. 710–738, 2014.
[47] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling *in-situ* execution of coupled scientific workflow on multi-core platform," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2012, pp. 1352–1363.
[48] F. Zhang, Q. M. Malluhi, T. Elsayed, S. U. Khan, K. Li, and A. Y. Zomaya, "CloudFlow: A data-aware programming model for cloud workflow applications on modern HPC systems," *Future Gener. Comput. Syst.*, vol. 51, pp. 98–110, Oct. 2014.
[49] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.
[50] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in *Proc. 2nd Workshop Many-Task Comput. Grids Supercomput.*, 2009, p. 8.
[51] A. Al-Badarneh, H. Najadat, M. Al-Soud, and R. Mosaid, "Phoenix: A MapReduce implementation with new enhancements," in *Proc. 7th Int. Conf. Comput. Sci. Inf. Technol. (CSIT)*, Jul. 2016, pp. 1–5.
[52] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Fast data analysis using coarse-grained distributed memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2012, pp. 689–692.
[53] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a MapReduce framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.
[54] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. OSDI*, vol. 10, 2010, pp. 1–14.
[55] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat, "M3R: Increased performance for in-memory Hadoop jobs," in *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, Aug. 2012, doi: 10.14778/2367502.2367513.
[56] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Proc. IEEE 27th Int. Conf. Data Eng. (ICDE)*, Washington, DC, USA, Apr. 2011, pp. 1151–1162, doi: 10.1109/ICDE.2011.5767921.
[57] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA—Preparatory data analytics on peta-scale machines," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, pp. 1–12.
[58] A. J. Awan, V. Vlassov, M. Brorsson, and E. Ayguade, "Node architecture implications for in-memory data analytics on scale-in clusters," in *Proc. 3rd IEEE/ACM Int. Conf. Big Data Comput., Appl. Technol.*, Dec. 2016, pp. 237–246.
[59] L. Hluchý *et al.*, *Heterogeneous Exascale Computing*, L. Kovács, T. Haidegger, and A. Szakál, Eds. Cham, Switzerland: Springer, 2020, doi: 10.1007/978-3-030-14350-3_5.
[60] Los Alamos National Laboratory. *Trinity Project*. Accessed: May 8, 2019. [Online]. Available: http://www.lanl.gov/projects/trinity/index.php/
[61] National Center for Supercomputing Applications. *Bluewaters Project*. [Online]. Available: http://www.ncsa.illinois.edu/BlueWaters/
[62] The National Institute for Computational Sciences. *Kraken Cray XT5 System*. [Online]. Available: http://www.nics.tennessee.edu/computing-resources/kraken/
[63] Argonne National Laboratory. *Intrepid—Blue Gene/P Solution*. [Online]. Available: https://www.alcf.anl.gov/intrepid
[64] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon, "The magellan report on cloud computing for science," U.S. Dept. Energy, Washington, DC, USA, Tech. Rep. 3, 2011.
[65] I. Raicu, I. T. Foster, and P. Beckman, "Making a case for distributed file systems at exascale," in *Proc. 3rd Int. Workshop Large-Scale Syst. Appl. Perform.*, 2011, pp. 11–18.
[66] J. Lofstead and R. Ross, "Insights for exascale IO APIs from building a petascale IO API," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2013, pp. 1–12.
[67] D. Borthakur, "HDFS architecture guide," Hadoop Apache Project, Apache Softw. Found., Appl. Note, 2008, p. 53.
[68] S. M. Strande, P. Cicotti, R. S. Sinkovits, W. S. Young, R. Wagner, M. Tatineni, E. Hocks, A. Snavely, and M. Norman, "Gordon: Design, performance, and experiences deploying and supporting a data intensive supercomputer," in *Proc. 1st Conf. Extreme Sci. Eng. Discovery Environ., Bridging eXtreme Campus Beyond (XSEDE)*, New York, NY, USA, 2012, pp. 3:1–3:8, doi: 10.1145/2335755.2335789.
[69] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.* Berlin, Germany: Springer, 2010, pp. 1–25.
[70] MPI Forum. (Jun. 2015). *MPI: A Message-Passing Interface Standard, Version 3.1*. [Online]. Available: https://www.MPI-forum.org/docs/MPI-3.1/MPI31-report-book.pdf

[71] T. Willhalm and N. Popovici, "Putting Intel threading building blocks to work," in *Proc. 1st Int. Workshop Multicore Softw. Eng. (IWMSE)*, New York, NY, USA, 2008, pp. 3–4, doi: 10.1145/1370082.1370085.

[72] C. Campbell and A. Miller, *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*, 1st ed. Redmond, WA, USA: Microsoft Press, 2011.

[73] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan./Mar. 1998.

[74] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proc. 17th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, Feb. 2009, pp. 427–436.

[75] P. D. Mininni, D. Rosenberg, R. Reddy, and A. Pouquet, "A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence," *Parallel Comput.*, vol. 37, no. 6, pp. 316–326, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000512

[76] S. Dong and G. E. Karniadakis, "Dual-level parallelism for high-order CFD methods," *Parallel Comput.*, vol. 30, no. 1, pp. 1–20, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016781910300173X

[77] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, p. 66, 2010.

[78] D. Kirk, "Nvidia CUDA software and GPU parallel computing architecture," in *Proc. ISMM*, vol. 7, 2007, pp. 103–104.

[79] R. Farber, *Parallel Programming With OpenACC*. Newnes, 2016.

[80] J. Guan, S. Yan, and J.-M. Jin, "An openMP-CUDA implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-GPU computing systems," *IEEE Trans. Antennas Propag.*, vol. 61, no. 7, pp. 3607–3616, Jul. 2013.

[81] P. Rakic, D. Milawinovic, Z. Zivanov, Z. Suvajdzin, M. Nikolic, and M. Hajdukovic, "MPI-CUDA parallelization of a finite-strip program for geometric nonlinear analysis: A hybrid approach," *Adv. Eng. Softw.*, vol. 42, no. 5, pp. 273–285, 2011, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997810001286

[82] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige, "Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 23–29, Mar. 2011, doi: 10.1145/1964218.1964223.

[83] M. U. Ashraf, F. A. Eassa, A. A. Albeshri, and A. Algarni, "Performance and power efficient massive parallel computational model for HPC heterogeneous exascale systems," *IEEE Access*, vol. 6, pp. 23095–23107, 2018.

[84] 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.

[85] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: A MapReduce-like framework for *in-situ* scientific analytics," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, p. 51.

[86] 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 *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 1098–1108.

[87] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault tolerant MapReduce-MPI for HPC clusters," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, New York, NY, USA, 2015, pp. 34:1–34:12, doi: 10.1145/2807591.2807617.

[88] F. Liang and X. Lu, "Accelerating iterative big data computing through MPI," *J. Comput. Sci. Technol.*, vol. 30, no. 2, pp. 283–294, 2015.

[89] T. Peterka, R. Ross, A. Gyulassy, V. Pascucci, W. Kendall, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri, "Scalable parallel building blocks for custom data analysis," in *Proc. IEEE Symp. Large Data Anal. Vis.*, Oct. 2011, pp. 105–112.

[90] T. Peterka, D. Morozov, and C. Phillips, "High-performance computation of distributed-memory parallel 3D Voronoi and Delaunay tessellation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 997–1007.

[91] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh, "MrPhi: An optimized MapReduce framework on Intel Xeon phi coprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3066–3078, Nov. 2015.

[92] F. J. Clemente-Castelló, B. Nicolae, R. Mayo, J. C. Fernández, and M. M. Rafique, "On exploiting data locality for iterative MapReduce applications in hybrid clouds," in *Proc. 3rd IEEE/ACM Int. Conf. Big Data Comput., Appl. Technol. (BDCAT)*, Shanghai, China, Dec. 2016, pp. 118–122.

[93] (2017). *GPU Enabler for Spark*. [Online]. Available: https://github.com/IBMSparkGPU/GPUEnabler

[94] T. Bicer, D. Gürsoy, V. De Andrade, R. Kettimuthu, W. Scullin, F. De Carlo, and I. T. Foster, "Trace: A high-throughput tomographic reconstruction engine for large-scale datasets," *Adv. Struct. Chem. Imag.*, vol. 3, no. 1, p. 6, Jan. 2017, doi: 10.1186/s40679-017-0040-7.

[95] J. Fox, Y. Zou, and J. Qiu, "Software frameworks for deep learning at scale," Internal Indiana Univ., Indianapolis, IN, USA, Tech. Rep., 2016.

[96] E. Ghasemi and P. Chow, "Accelerating Apache Spark with FPGAs," *Concurrency Comput., Pract. Exper.*, vol. 31, no. 2, p. e4222, 2019.

[97] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, "SparkCL: A unified programming framework for accelerators on heterogeneous clusters," 2015, *arXiv:1505.01120*. [Online]. Available: https://arxiv.org/abs/1505.01120

[98] O. Segal, M. Margala, S. R. Chalamalasetti, and M. Wright, "High level programming framework for FPGAs in the data center," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, 2014, pp. 1–4.

[99] A. J. Awan, "Performance characterization and optimization of in-memory data analytics on a scale-up server," Ph.D. dissertation, School Inf. Commun. Technol., KTH, Stockholm, Sweden, 2017. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-217910

[100] M. Wasi-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 *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 291–300.

[101] Y. Wang, R. Goldstone, W. Yu, and T. Wang, "Characterization and optimization of memory-resident MapReduce on HPC systems," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 799–808.

[102] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, and S. Weil, "Ceph as a scalable alternative to the Hadoop distributed file system," *Login, USENIX Mag.*, vol. 35, no. 4, pp. 38–49, 2010.

[103] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud analytics: Do *we really* need to reinvent the storage stack?" in *Proc. HotCloud*, 2009, pp. 1–5.

[104] F. G. Carballeira, A. Calderón, J. Carretero, J. Fernández, and J. M. Pérez, "The design of the expand parallel file system," *Int. J. High Perform. Comput. Appl.*, vol. 17, no. 1, pp. 21–37, 2003.

[105] P. Xuan, J. Denton, P. K. Srimani, R. Ge, and F. Luo, "Big data analytics on traditional HPC infrastructure using two-level storage," in *Proc. Int. Workshop Data-Intensive Scalable Comput. Syst. (DISCS)*, New York, NY, USA, 2015, pp. 4:1–4:8, doi: 10.1145/2831244.2831253.

[106] X. X. Lu, M. W. Ur Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating Spark with RDMA for big data processing: Early experiences," in *Proc. IEEE 22nd Annu. Symp. High-Perform. Interconnects (HOTI)*, Aug. 2014, pp. 9–16.

[107] I. Lopez. (2013). *IDC Talks Convergence in High Performance Data Analysis, 2013*. Accessed: Feb. 14, 2016. [Online]. Available: http://www.datanami.com/2013/06/19/idc_talks_convergence_in_high_performance_data_analysis/

[108] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analyses," in *Proc. IEEE 4th Int. Conf. (eScience)*, Dec. 2008, pp. 277–284.

[109] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to clouds using MapReduce," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 184–192, 2012.

[110] S. Seo, E. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the MapReduce framework," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Nov. 2010, pp. 721–726.

[111] P. Xuan, Y. Zheng, S. Sarupria, and A. Apon, "SciFlow: A dataflow-driven model architecture for scientific computing using hadoop," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 36–44.

[112] E. Dede, M. Govindaraju, D. Gunter, and L. Ramakrishnan, "Riding the elephant: Managing ensembles with Hadoop," in *Proc. ACM Int. Workshop Many Task Comput. Grids Supercomputers (MTAGS)*, New York, NY, USA, 2011, pp. 49–58.

[113] I. Antcheva *et al.*, "ROOT—A C++ framework for petabyte data storage, statistical analysis and visualization," *Comput. Phys. Commun.*, vol. 180, no. 12, pp. 2499–2512, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465511000701

[114] S. Sehrish, G. Mackey, P. Shang, J. Wang, and J. Bent, "Supporting hpc analytics applications with access patterns using data restructuring and data-centric scheduling techniques in MapReduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 158–169, Jan. 2013.

[115] 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 *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2015, pp. 918–927.

[116] Y. S. Nashed, D. J. Vine, T. Peterka, J. Deng, R. Ross, and C. Jacobsen, "Parallel ptychographic reconstruction," *Opt. Express*, vol. 22, no. 26, pp. 32082–32097, 2014.

[117] A. Luckow, P. Mantha, and S. Jha, "Pilot-abstraction: A valid abstraction for data-intensive applications on hpc, Hadoop and cloud infrastructures?" 2015, *arXiv:1501.05041*. [Online]. Available: https://arxiv.org/abs/1501.05041

[118] S. Sehrish, J. Kowalkowski, and M. Paterno, "Spark and HPC for high energy physics data analyses," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May/Jun. 2017, pp. 1048–1057.

[119] S. Sehrish, J. Kowalkowski, and M. Paterno, "Exploring the performance of Spark for a scientific use case," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2016, pp. 1653–1659.

[120] Z. Zhang, "Processing data-intensive work OWS in the cloud," Tech. Rep., 2012.

[121] K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman, "Rethinking data management for big data scientific workflows," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 27–35.

[122] V. Nuthula and N. R. Challa, "Cloudifying apps—A study of design and architectural considerations for developing cloudenabled applications with case study," in *Proc. IEEE Int. Conf. Cloud Comput. Emerg. Markets (CCEM)*, Oct. 2014, pp. 1–7.

[123] S. N. Srirama and J. Viil, "Migrating scientific workflows to the cloud: Through graph-partitioning, scheduling and peer-to-peer data sharing," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Aug. 2014, pp. 1105–1112.

[124] P. Matri, Y. Alforov, A. Brandon, M. Kuhn, P. Carns, and T. Ludwig, "Could blobs fuel storage-based convergence between HPC and big data?" in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2017, pp. 81–86.

[125] P. Matri, Y. Alforov, Á. Brandon. M. S. Pérez, A. Costan, G. Antoniu, M. Kuhn, P. Carns, and T. Ludwig, "Mission possible: Unify HPC and big data stacks towards application-defined blobs at the storage layer," *Future Gener. Comput. Syst.*, to be published. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17330583

[126] Y. Zhao, X. Fei, I. Raicu, and S. Lu, "Opportunities and challenges in running scientific workflows on the cloud," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discovery (CyberC)*, Oct. 2011, pp. 455–462.

[127] G. Lin, B. Han, J. Yin, and I. Gorton, "Exploring cloud computing for large-scale scientific applications," in *Proc. IEEE 9th World Congr. Services*, Jun. 2013, pp. 37–43.

[128] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Piscataway, NJ, USA, Nov. 2008, pp. 50:1–50:12.

[129] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *Proc. IEEE 4th Int. Conf. eScience (eSci)*, Dec. 2008, pp. 640–645.

[130] G. B. Berriman, E. Deelman, G. Juve, M. Rynge, and J.-S. Vöckler, "The application of cloud computing to scientific workflows: A study of cost and performance," *Phil. Trans. Roy. Soc. London A, Math., Phys. Eng. Sci.*, vol. 371, no. 1983, 2012, Art. no. 20120066, doi: 10.1098/rsta.2012.0066.

[131] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "MapReduce in the clouds for science," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Nov. 2010, pp. 565–572.

[132] C. Evangelinos and C. Hill, "Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2," *Ratio*, vol. 2, no. 2, pp. 2–34, 2008.

[133] A. Gupta and D. Milojicic, "Evaluation of HPC applications on cloud," in *Proc. 6th Open Cirrus Summit*, Oct. 2011, pp. 22–26.

[134] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. Berman, and P. Maechling, "Scientific workflow applications on Amazon EC2," in *Proc. 5th IEEE Int. Conf. E-Sci. Workshops*, Dec. 2009, pp. 59–66.

[135] Z. Hill and M. Humphrey, "A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster?" in *Proc. 10th IEEE/ACM Int. Conf. Grid Comput.*, Oct. 2009, pp. 26–33.

[136] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[137] G. D'Angelo, "Parallel and distributed simulation from many cores to the public cloud," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2011, pp. 14–23.

[138] D. Yu, J. Wang, B. Hu, J. Liu, X. Zhang, K. He, and L.-J. Zhang, "A practical architecture of cloudification of legacy applications," in *Proc. IEEE World Congr. Services (Services)*, Jul. 2011, pp. 17–24.

[139] S. Srirama, V. Ivanistsev, P. Jakovits, and C. Willmore, "Direct migration of scientific computing experiments to the cloud," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2013, pp. 27–34.

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

[141] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 293–302.

[142] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow, "HPC-ABDS high performance computing enhanced apache big data stack," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2015, pp. 1057–1066.

[143] A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, P. Sharma, J. Yang, J. Demmel, J. Harrell, V. Krishnamurthy, M. W. Mahoney, and Prabhat, "Matrix factorizations at scale: A comparison of scientific data analytics in Spark and c+MPI using three case studies," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 204–213.

[144] 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," *Proc. VLDB Endowment*, vol. 10, no. 8, pp. 901–912, 2017.

[145] N. Malitsky, A. Chaudhary, S. Jourdain, M. Cowan, P. O'Leary, M. Hanwell, and K. K. V. Dam, "Building near-real-time processing pipelines with the Spark-MPI platform," in *Proc. New York Sci. Data Summit (NYSDS)*, Aug. 2017, pp. 1–8.

[146] A. Gittens, K. Rothauge, S. Wang, M. Mahoney, L. Gerhardt, Prabhat, J. Kottalam, M. Ringenburg, and K. Maschhoff, "Accelerating large-scale data analysis by offloading to high-performance computing libraries using alchemist," in *Proc. 24th ACM Int. Conf. Knowl. Discovery Data Mining (SIGKDD)*, London, U.K., 2018, pp. 293–301.

[147] ETP4HPC. European Technology Platform for High-Performance Computing. (2015). *Strategic Research Agenda 2015 Update*. [Online]. Available: http://www.etp4hpc.eu/pujades/files/ETP4HPC%20SRA%202%20Single%20Page.pdf

[148] (Apr. 2017). *Transition to Exascale Computing (H2020-FETHPC-2016-2017)*. [Online]. Available: http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/fethpc-02-2017.html

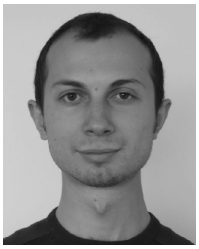
[149] A. Bilas, T. Cortes, D. Talia, M. S. Perez, J. Garcia-Blas, P. Gonzalez-Frez, A. Brinkmann, S. Anastasiadis, M. Muggeridge, C. Comito, S. Narasimhamurthy, A. Queralt, and F. Isaila, "Data storage for big data in the exascale era: Challenges and prospects," Univ. Carlos III Madrid, Getafe, Spain, Tech. Rep., Sep. 2015. [Online]. Available: https://www.dropbox.com/s/ws58kxm26j3o20a/nesus_report_WG4_sep2015.pdf

[150] D. Morozov and T. Peterka, "Block-parallel data analysis with DIY2," in *Proc. IEEE 6th Symp. Large Data Anal. Vis. (LDAV)*, Oct. 2016, pp. 29–36.
[151] K. Lu, H.-W. Shen, and T. Peterka, "Scalable computation of stream surfaces on large scale vector fields," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2014, pp. 1008–1019.
[152] Y. S. Nashed, D. J. Vine, T. Peterka, J. Deng, R. Ross, and C. Jacobsen, "Parallel ptychographic reconstruction," *Opt. Express*, vol. 22, no. 26, pp. 32082–32097, 2014.
[153] C. Sewell, J. Meredith, K. Moreland, T. Peterka, D. DeMarle, L.-T. Lo, J. Ahrens, R. Maynard, and B. Geveci, "The SDAV software frameworks for visualization and analysis on next-generation multi-core and many-core architectures," in *Proc. Companion High Perform. Comput., Netw. Storage Anal. (SC)*, 2012, pp. 206–214.
[154] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka, "Parallel particle advection and FTLE computation for time-varying flow fields," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, p. 61.
[155] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang, "A study of parallel particle tracing for steady-state and time-varying flow fields," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2011, pp. 580–591.
[156] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards high performance adaptive asynchronous checkpointing at large scale," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Rio de Janeiro, Brazil, May 2019, pp. 911–920.
[157] D. Ceperley, G. Chester, and M. Kalos, "Monte carlo simulation of a many-fermion study," *Phys. Rev. B, Condens. Matter*, vol. 16, no. 7, p. 3081, 1977.
[158] Z. Li and H. A. Scheraga, "Monte carlo-minimization approach to the multiple-minima problem in protein folding," *Proc. Nat. Acad. Sci. USA*, vol. 84, no. 19, pp. 6611–6615, 1987.
[159] L. Wang and S. L. Jacques, "Monte carlo modeling of light transport in multi-layered tissues in standard c," The Univ. Texas, MD Anderson Cancer Center, Houston, TX, USA, 1992, pp. 4–11.
[160] D. Perez, P. Rohlfshagen, and S. M. Lucas, "Monte–Carlo tree search for the physical travelling salesman problem," in *Proc. Eur. Conf. Appl. Evol. Comput.* Berlin, Germany: Springer, 2012, pp. 255–264.
[161] R. Therrien, R. G. McLaren, E. A. Sudicky, and S. M. Panday, "A three-dimensional numerical model describing fully-integrated subsurface and surface flow and solute transport," User Guide, 2010.
[162] G. Bauser, H.-J. Hendricks Franssen, S. Fritz, H.-P. Kaiser, U. Kuhlmann, and W. Kinzelbach, "A comparison study of two different control criteria for the real-time management of urban groundwater works," *J. Environ. Manage.*, vol. 105, pp. 21–29, Aug. 2012.
[163] M. P. McGuire, M. C. Roberge, and J. Lian, "Hydrocloud: A cloud-based system for hydrologic data integration and analysis," in *Proc. 5th Int. Conf. Comput. Geospatial Res. Appl. (COM. Geo)*, Aug. 2014, pp. 9–16.
[164] A. Lapin, E. Schiller, P. Kropf, O. Schilling, P. Brunner, A. J. Kapic, T. Braun, and S. Maffioletti, "Real-time environmental monitoring for cloud-based hydrogeological modeling with hydrogeosphere," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Aug. 2014, pp. 959–965.
[165] 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 *Proc. EGU Gen. Assembly Conf. Abstracts*, vol. 19, 2017, p. 2170.
[166] 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 Biol.*, vol. 11, no. 1, pp. 89–105, 2005.
[167] S. Caíno-Lores, A. Lapin, J. Carretero, and P. Kropf, "Applying big data paradigms to a large scale scientific workflow: Lessons learned and future directions," *Future Gener. Comput. Syst.*, to be published.
[168] A. Davidson and A. Or, "Optimizing shuffle performance in Spark," Dept. Electr. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep., 2013.
[169] 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 Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1663–1674, Jun. 2017.
[170] S. Caíno-Lores, A. Lapin, P. Kropf, and J. Carretero, "Methodological approach to data-centric cloudification of scientific iterative workflows," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.* Cham, Switzerland: Springer, 2016, pp. 469–482.

SILVINA CAÍNO-LORES received the Ph.D. degree in computer science and technology from the Carlos III University of Madrid, Spain, in 2019, under the supervision of Prof. J. C. Pérez. She is currently a Teaching and Research Assistant with the Computer Science Department, Carlos III University of Madrid, Spain. Her research interests include cloud computing, HPC scientific computing, and data-centric computing.

JESÚS CARRETERO has been a Full Professor of computer architecture and technology with the Universidad Carlos III de Madrid, Spain, since 2002. His research interests include high-performance computing systems, large-scale distributed systems, and real-time systems. He is currently a Senior Member of the IEEE Computer Society. He was the Action Chair of the IC1305 COST Action *Network for Sustainable Ultrascale Computing Systems* (NESUS). He is also involved in three other EU Projects. He has been the General Chair of HPCC 2011, MUE 2012, ISPA 2016, and CCGRID 2017. He is currently an Associated Editor of *Future Generation Computing Systems* journal.

BOGDAN NICOLAE received the Ph.D. degree from the University of Rennes 1, France, and the Dipl.Eng. degree from the Politehnica University of Bucharest, Romania. He is currently a Computer Scientist with the Argonne National Laboratory and a Scientist with the University of Chicago Consortium for Advanced Science and Engineering (CASE). He specializes in scalable storage, data management, and fault tolerance for large-scale distributed systems, with a focus on cloud computing and high-performance architectures. He has (co)authored numerous articles in the areas of scalable I/O, storage elasticity and virtualization, data and metadata decentralization and availability, multiversioning, checkpoint-restart, and live migration.



ORCUN YILDIZ received the Ph.D. degree in computer science from the Ecole Normale Superieure de Rennes, France, in December 2017. He is currently a Postdoctoral Researcher with the Mathematics and Computer Science Division, Argonne National Laboratory. His research interests include scientific workflows, big data processing, I/O management, and high-performance computing.

TOM PETERKA received the Ph.D. degree in computer science from the University of Illinois at Chicago, in 2007. He is currently a Computer Scientist with the Argonne National Laboratory, a Scientist with the University of Chicago Consortium for Advanced Science and Engineering (CASE), an Adjunct Assistant Professor with the University of Illinois at Chicago, and a Fellow of the Northwestern Argonne Institute for Science and Engineering (NAISE). His research interest includes large-scale parallel in situ analysis of scientific data. He was a recipient of the 2017 DOE Early Career Award and three best paper awards. He has published in ACM SIGGRAPH, IEEE VR, IEEE TVCG, and ACM/IEEE SC, among other top conferences and journals. He also works actively in several DOE-and NSF-funded projects.