On the Convergence of Big Data Analytics and High-Performance Computing: A Novel Approach for Runtime Interoperability

by

Silvina Caíno-Lores

in partial fulfillment of the requirements for the degree of

PhD in Computer Science and Technology

Universidad Carlos III de Madrid

Advisor:

Prof. PhD Jesús Carretero Pérez

Tutor:

Prof. PhD Jesús Carretero Pérez

Leganés, May, 2019

Tesis Doctoral

On the Convergence of Big Data Analytics and High-Performance Computing: A Novel Approach for Runtime Interoperability

AUTOR: Silvina Caíno Lores

TUTOR DIRECTOR: Prof. Dr. Jesús Carretero Pérez

Firmas del Tribunal Calificador

	Nombre y apellidos	Firma
Presidente:		
Secretario:		
Vocal:		

En Leganés, a de de 2019

Declaration of Authorship

I, Silvina Caíno-Lores, declare that this thesis titled, *'On the Convergence of Big Data Analytics and High-Performance Computing: A Novel Approach for Runtime Interoperability'* and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Published Contents

This thesis includes published content not singled out with typographic means and references. The inclusion of material from these sources is explicitly specified in each chapter where an inclusion occurs, and listed below.

- 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 *5th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT 2018)*, Zurich, Switzerland, December 2018, DOI: 10.1109/BDCAT.2018.00010
  - Partial content of this publication is included in Chapters 2 and 7.
- 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", in *Future Generation Computer Systems*, April 2018, DOI: 10.1016/j.future.2018.04.014
  - Partial content of this publication is included in Chapters 4 and 5.
  - This source was also partially included in *Approaches for Cloudification of Complex High Performance Simulation Systems*, submitted by Andrei Lapin in partial fulfilment for the degree of PhD in Computer Science at the University of Neuchâtel in 2017.
- S. Caíno-Lores, F. Isaila, and J. Carretero, "Data-Aware Support for Hybrid HPC and Big Data Applications", in *Doctoral Symposium at the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid*

Computing (CCGRID 2017), Madrid, Spain, May 2017, DOI: 10.1109/CCGRID.2017.55

  - Partial content of this publication is included in Chapter 3.

- S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, "Efficient design assessment in the railway electric infrastructure domain using cloud computing", in *Integrated Computer-Aided Engineering*, vol. 24, pp. 57–72, December 2016, DOI: 10.3233/ICA-160532

  - Partial content of this publication is included in Chapter 5.

- S. Caíno-Lores, A. Lapin, P. Kropf, and J. Carretero, "Methodological Approach to Data-Centric Cloudification of Scientific Iterative Workflows", in *16th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2016)*, Granada, Spain, December 2016, DOI: 10.1007/978-3-319-49583-5_36

  - Partial content of this publication is included in Chapters 4 and 5.
  - This source was also partially included in *Approaches for Cloudification of Complex High Performance Simulation Systems*, submitted by Andrei Lapin in partial fulfilment for the degree of PhD in Computer Science at the University of Neuchâtel in 2017.

- S. Caíno-Lores, A. Lapin, P. Kropf, J. Carretero, "Lessons Learned from Applying Big Data Paradigms to a Large Scale Scientific Workflow", in *11th Workshop on Workflows in Support of Large-Scale Science (WORKS 2016)*, in conjunction with *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*, Salt Lake City, Utah, USA, November 2016, DOI: 10.1016/j.future.2018.04.014

  - Partial content of this publication is included in Chapter 5.
  - This source was also partially included in *Approaches for Cloudification of Complex High Performance Simulation Systems*, submitted by Andrei Lapin in partial fulfilment for the degree of PhD in Computer Science at the University of Neuchâtel in 2017.

- S. Caíno-Lores, J. Carretero, "A Survey on Data-Centric and Data-Aware Techniques for Large Scale Infrastructures", in *18th International Conference on Computer and Information Sciences (ICCIS 2016)*, Dubai, UAE, March 2016, Available at `https://pdfs.semanticscholar.org/6abe/7355ba8d703eca70576b21adba4a64ba6516.pdf`

  - Partial content of this publication is included in Chapter 2.

- A. García, S. Caíno-Lores, F. García-Carballeira, and J. Carretero, "A multi-objective simulator for optimal power dimensioning on electric railways using cloud computing", in *5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2015)*, Kolmar, France, July 2015, DOI: `10.5220/0005573404280438`

  - Partial content of this publication is included in Chapter 5.
  - This source was also partially included in *A Cloudification Methodology for High Performance Simulations*, submitted by Alberto García in partial fulfilment for the degree of PhD in Computer Science and Technology at the University Carlos III of Madrid in 2016; and *Enabling Data Locality in Multidimensional Scientific Applications with Many-Task Computing and Map-Reduce*, submitted by Silvina Caíno-Lores in partial fulfilment for the degree of

MSc in Computer Science and Technology at the University Carlos III of Madrid in 2015.

- S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, "A cloudification methodology for multidimensional analysis: Implementation and application to a railway power simulator", in *Simulation Modelling Practice and Theory*, vol. 55, pp. 46–62, June 2015, DOI: 10.1016/j.simpat.2015.04.002

  – Partial content of this publication is included in Chapter 5.

  – This source was also partially included in *A Cloudification Methodology for High Performance Simulations*, submitted by Alberto García in partial fulfilment for the degree of PhD in Computer Science and Technology at the University Carlos III of Madrid in 2016; and *Enabling Data Locality in Multidimensional Scientific Applications with Many-Task Computing and Map-Reduce*, submitted by Silvina Caíno-Lores in partial fulfilment for the degree of MSc in Computer Science and Technology at the University Carlos III of Madrid in 2015.

Acknowledgements

This work is the result of years of hard work and struggle. None of those efforts would have led to a successful result without the guidance of my supervisor, Prof. Jesús Carretero, whom I can gladly consider a mentor. I would also like to acknowledge Prof. Florin Isaila, who encouraged me to believe in my own work and defend it. I hope this thesis does honour to his memory.

I was fortunate to work with excellent collaborators. Special thanks to Prof. Peter Kropf and his team in the University of Neuchâtel, and Dr. Tom Peterka and the Mathematics and Computer Science division staff at the Argonne National Laboratory for their valuable lessons on teamwork and scientific excellence.

In-house support was also critical for the success of this thesis. I would like to appreciate the daily support by the ARCOS team. You have all provided me with great knowledge and thoughtful feedback to improve this work. On a more personal note, I would like to express my gratefulness to the colleagues that became friends during these years, who gave me the comfort I needed during lonely, frustrating and difficult times. Many thanks to Estefanía and Garci for being my everyday rock and encouraging me to endure.

None of this would have been possible without the daily support of my family and friends. I would like to show my gratitude to my partner Alfredo for bearing with me and being my safety net; my cousins and their families for their cheerful attitude, which was a major refuge; my masters, Mercedes and Marco, who kept my mind and body healthy during this process; and my kitties for their warm affection during stressful days. Finally, I need to express my deepest gratitude to my mother Silvia and my adopted mum Silvia. They are my best role models, for their strength and wisdom are exemplar.

“Sean los orientales tan ilustrados como valientes"

José Gervasio Artigas

All that is gold does not glitter,
Not all those who wander are lost;
The old that is strong does not wither,
Deep roots are not reached by the frost.

From the ashes, a fire shall be woken,
A light from the shadows shall spring;
Renewed shall be blade that was broken,
The crownless again shall be king.

The Riddle of Strider by J. R. R. Tolkien

UNIVERSITY CARLOS III OF MADRID

Abstract

School of Engineering
Computer Science and Engineering Department

Ph.D. in Computer Science and Technology

On the Convergence of Big Data Analytics and High-Performance Computing: A Novel Approach for Runtime Interoperability

by Silvina Caíno-Lores

Convergence between high-performance computing (HPC) and Big Data analytics (BDA) is currently an established research area that spawned new opportunities for unifying the platform layer and data abstractions in these ecosystems. This thesis builds on the hypothesis that HPC-BDA convergence at platform level can be attained by enabling runtime interoperability in a way that preserves BDA platform usability and productivity, exploits HPC scalability and performance, and expands both BDA and HPC capabilities to cope with prospect hybrid application models. The goal is to architect an abstract system that enables the interoperability of established BDA and HPC runtimes.

In order to exploit the benefits of BDA data-centric paradigms, this thesis presents a data-centric transformation methodology to allow process-centric workloads the interaction with BDA platforms and storage infrastructures. Furthermore, an architecture to achieve full runtime interoperability is proposed. It reflects the key design features that interest both the HPC and BDA communities, and includes an abstract data collection and operational model that generates a unified interface for hybrid applications. It also incorporates

a mechanism to transfer each stage of the application to the appropriate runtime.

This architecture can be implemented in different ways depending on the process- and data-centric runtimes of choice, and the mechanisms put in place to effectively meet the requirements of the architecture. The Spark-DIY platform is introduced as a possible implementation. It preserves the interfaces and execution environment of the popular BDA platform Apache Spark –thus 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.

Finally, these solutions are analysed in terms of performance by applying them to a representative use case, EnKF-HGS. This application is a clear example of how current HPC simulations are evolving towards hybrid HPC-BDA applications, integrating HPC simulations within a BDA environment. Other auxiliary use cases –like an application from the railway domain and a BDA benchmark operator– are also introduced to support other specific contributions of this thesis.

UNIVERSIDAD CARLOS III DE MADRID

Resumen

Escuela Politécnica Superior
Departamento de Informática

Doctorado en Ciencia y Tecnología Informática

Sobre la Convergencia del Análisis de Macrodatos y la Computación de Altas Prestaciones: Un Nuevo Enfoque para la Interoperabilidad entre Entornos de Ejecución

por Silvina Caíno-Lores

La convergencia entre la computación de altas prestaciones (HPC) y el análisis de macrodatos (BDA) es actualmente un área de investigación establecida que ha generado nuevas oportunidades para la unificación de la capa de plataforma y las abstracciones de datos en estos ecosistemas. Esta tesis desarrolla la hipótesis de que la convergencia HPC-BDA a nivel de plataforma puede ser obtenida con la habilitación de mecanismos de interoperabilidad entre entornos de ejecución, de modo que se preserve la usabilidad y productividad de las plataformas BDA, se explote la escalabilidad y rendimiento de HPC, y se expandan las capacidades de HPC y BDA para tratar futuros modelos híbridos de aplicación. El objetivo es desarrollar un sistema abstracto que permita la interoperabilidad de entornos de ejecución ya establecidos en los ecosistemas BDA y HPC.

Con el fin de explotar los beneficios de los paradigmas orientados a datos en BDA, esta tesis presenta una metodología de transformación también orientada a datos que permite a las aplicaciones orientadas a proceso interactuar con plataformas BDA y sus correspondientes infraestructuras de

almacenamiento. Además, se propone una arquitectura para obtener interoperabilidad total entre entornos de ejecución. Ésta refleja las características de diseño clave que interesan a las comunidades BDA y HPC, e incluye una abstracción de colección de datos y modelo operacional que genera una interfaz unificada para aplicaciones híbridas. Además, incorpora un mecanismo para transferir cada etapa de la aplicación al entorno de ejecución adecuado.

Esta arquitectura puede ser implementada de distintas maneras dependiendo de los entornos de ejecución orientados a datos y proceso seleccionados, y las tćnicas utilizadas para cumplir de manera efectiva con los requisitos de la arquitectura. La plataforma Spark-DIY se introduce como posible implementación. Preserva las interfaces y entorno de ejecución de la popular plataforma BDA Apache Spark –haciéndola compatible con cualquier aplicación o herramienta basada en Spark–, mientras provee comunicación y ejecución eficiente de núcleos de simulación y análisis a través de DIY, una potente biblioteca de patrones de comunicación construida sobre MPI.

Finalmente, estas soluciones son analizadas en términos de rendimiento al aplicarlas a un caso de uso representativo, EnKF-HGS. Esta aplicación es un ejemplo claro de cómo las simulaciones HPC están evolucionando hacia aplicaciones HPC-BDA híbridas, integrando simulaciones HPC dentro de un entorno BDA. Otros casos de uso auxiliares –como una aplicación del ámbito ferroviario y un operador referente de BDA– son introducidos para apoyar otras contribuciones específicas de esta tesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

API	Application Programming Interface
BD	Big Data
BDA	Big Data Analytics
CPU	Central Processing Unit
DIY	Do-It-Yourself Block Parallelism
DL	Deep Learning
EC2	Elastic Compute Cloud
EFS	Elastic File System
EnKF	Ensemble Kalman Filter
FPGA	Field-Programmable Gate Arrays
GFS	Google File System
GPFS	IBM General Parallel File System
GPGPU	General Purpose Graphic Processing Unit
GPU	Graphic Processing Unit
HDFS	Hadoop Distributed File System
HGS	HydroGeoSphere
HPC	High-Performance Computing
HPDA	High-Performance Data Analytics
HTC	High-Throughput Computing
I/O, IO	Input/Output
IoT	Internet of Things
JNI	Java Native Interface
JVM	Java Virtual Machine
ML	Machine Learning
MNA	Modified Nodal Analysis
MPI	Message Passing Interface
MPI-IO	Message Passing Interface Input/Output
MR	Map-Reduce

MTC	Many-Task Computing
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PFS	Parallel File System
PPL	Parallel Patterns Library
RDD	Resilient Distributed Dataset
RDMA	Remote-Direct Memory Access
RDS	Runtime Delegation System
RPC	Remote Procedure Call
RPCS	Railway Power Consumption Simulator
QoS	Quality of Service
SPMD	Single Program, Multiple Data
SQL	Structured Query Language
TBB	Threading Building Blocks
TPU	Tensor Processing Unit
UC3M	University Carlos III of Madrid
UDDA	Unified Distributed Data Abstraction
UOM	Unified Operational Model
VM	Virtual Machine

CHAPTER 1

INTRODUCTION

The information technology ecosystem is currently in transition to a new generation of applications requiring intensive data acquisition, processing and storage. As a result of this shift towards data-intensive computing, there is a growing confluence between high-performance computing (HPC) and Big Data analytics (BDA), given that many HPC applications produce Big Data to be manipulated with analytics techniques, while BDA is a growing consumer of HPC capabilities.

More precisely, HPC scientific applications are key tools in many research areas that rely on multiple, diverse, and distributed operations over various datasets, usually yielding significant computational complexity and data dependencies. Nowadays, HPC applications are increasingly demanding data analysis and visualisation over major datasets, which is shifting these originally computationally intensive systems towards parallel data-intensive problems. On the other hand, BDA applications are demanding the performance level of the supercomputing ecosystem, thus requiring acceleration and increased scalability. As a result, this general trend is leading to greater confluence between the HPC and BDA paradigms.

Nevertheless, HPC and BDA systems have been traditionally built to solve different problems: HPC focuses on computationally-intensive tightly-coupled applications, and BDA tackles large volumes of loosely-coupled tasks. These objectives have determined the underlying architectures of HPC and BDA infrastructures. In a typical HPC infrastructure, compute and data subsystems are totally decoupled, using parallel file systems for data storage, but connected through high-speed interconnections, as in grids or clusters. On the

other hand, BDA systems co-locate computation and data on the same node and focus on elasticity, thus clouds become their preferred infrastructure [1].

The tools and cultures of HPC and BDA have also diverged to solve their canonical problems. However, between both worlds there are different degrees of intermediate HPC-BDA applications that portray mixed requirements. These applications could be executed on both platforms, but none of them are fully ideal in their current state, mainly due to requirements such as scalability, performance, and resource efficiency.

In this scenario, upcoming applications will suffer the lack of an ideal environment able to cope with their computing and data requirements. Recent works have suggested the opportunity of combining the HPC and BDA approaches to alleviate this issue [2]. For example, typical BDA programming models have been considered to substitute MPI parallelism induction mechanisms, following a data-centric approach. In addition, we can also see this opportunity affecting the underlying computing infrastructures. Indeed, typical BDA infrastructures like clouds could inspire hybrid platforms for exascale scientific workflows [3].

Applying some of these BDA mechanisms can improve scalability in parameter-based HPC applications relying on a large pool of loosely-coupled tasks. However, other types of applications were not able to benefit from this, as they did not fit the prototypical structural model of BDA platforms. Due to the former reasons, there is currently an increasing agreement on the need for those ecosystems to converge to produce environments that have the performance of HPC and the usability and flexibility of the BDA stack. As a consequence, our research question is *how can we build a platform able to manage applications built for computationally-intensive simulations, data-intensive analysis, or both, without hurting performance and data-awareness?*.

To answer this question, this thesis explores the key features of BDA and HPC ecosystems, with a focus on the platform layer and the core runtimes

that support BDA and HPC processing frameworks, which we will refer to in this document as *data-centric* and *process-centric* runtimes, respectively. We will take this knowledge as baseline to elaborate a theoretical frame for the development of generalist solutions for runtime convergence of BDA and HPC platforms.

1.1 Objectives

Current trends in scientific computing highlight that interoperability and scaling convergence of HPC and BDA runtimes is crucial to the future, and unification is essential to address a spectrum of major research domains. This is especially true when targeting the scalability of data-intensive applications, making massive data transmissions, applying complex analysis on data, or storing large amounts of data.

The main goal of this thesis is **to research new approaches to facilitate the convergence of HPC and BDA paradigms by providing common abstractions and mechanisms for improving scalability, data locality exploitation, and execution adaptivity on large scale systems**, while preserving the most relevant features for their corresponding communities, in order to provide a system suitable for the composition of applications with mixed BDA and HPC stages.

To achieve this goal, this thesis aims to accomplish the following specific objectives:

O1 Analyse the key features that characterise HPC and BDA ecosystems.

O2 Provide a mechanism to reshape HPC-oriented workflows in order to adapt them to data-centric environments.

O3 Design and develop an architecture that offers runtime interoperability for hybrid HPC-BDA applications, incorporating unified operational

and data models that support high-level analytics methods and high-performance kernels for composite applications.

O4 Evaluate on meaningful use cases, representative of target hybrid applications.

1.2 Contributions

The main contributions of this thesis are:

C1 A methodology to adapt iterative scientific applications to a data-centric paradigm.

C2 A formal definition of a generic unified distributed data abstraction and unified operational model, which sets the foundation of a theoretical frame for the analysis and definition of composite HPC-BDA applications.

C3 A generalist runtime interoperability architecture for HPC-BDA applications, which includes a delegation mechanism to select the appropriate runtime (process- or data-centric) for each stage of the composite application.

C4 An implementation of the former architecture based on Spark and MPI.

C5 An implementation of a real-world use case from the hydrogeology domain, enriched with features enabled by our architecture like cloud and streaming support for delocalisation and data assimilation.

1.3 Structure and Contents

This document details the work conducted through the development of this thesis, and it is structured as follows:

- Chapter 1, *Introduction*, has briefly presented the scope, motivation and objectives of this thesis in the context of HPC and BDA convergence.
- Chapter 2, *State of the Art*, establishes the foundation for the contributions of this thesis, depicting the current state of both BDA and HPC ecosystems in terms of their infrastructures, platforms and applications. This chapter also compares their features, communities, and prospective evolution in order to derive the characteristics that will be beneficial in a future hybrid setting, and presents relevant advances on convergence found in the literature.
- Chapter 3, *Problem Statement*, deepens the motivation of this work in light of the challenges found for each ecosystem, and presents the hypothesis and top-level approach for the rest of the thesis.
- Chapter 4, *Data-Centric Transformation Methodology for HPC Process-Centric Applications*, introduces a data-centric platform enablement methodology, which allows HPC applications to exploit the benefits of data-centric computing paradigms and resources.
- Chapter 5, *Data-Centric Transformation of HPC Scientific Applications*, applies the former methodology to representative use cases and presents evaluation results.
- Chapter 6, *Generalist Interoperability Architecture for Hybrid HPC-BDA Applications*, depicts the proposed global architecture for runtime interoperability. This chapter also discusses the design and implementation of a prototype platform to enable the composition of applications with both HPC and BDA stages.
- Chapter 7, *Evaluation of Spark-DIY with HPC and BDA Applications*, shows how hybrid HPC-BDA applications can be built using an implementation of the former architecture, and presents evaluation results to support the viability of the interoperability model.

- Chapter 8, *Conclusions*, summarises this thesis and its objectives, detailing its contributions and results, while discussing potential directions for future research enabled by this work.

CHAPTER 2

STATE OF THE ART

This chapter includes an overview of the different fields related to this thesis. A literature review of these topics is provided, with special emphasis in high-performance computing (HPC) ecosystem, Big Data analytics (BDA) ecosystem, and current trends in HPC and BDA convergence.

2.1 Big Data Analytics Ecosystem

Big Data affects many different ecosystems and areas of research and business, thus there is no unique definition for it 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 [4]. The core characteristics included in this model are:

- **Volume of data.** Volume is necessary in order to get valuable insight from analytics tools. It is usual to find volumes in the order of peta or terabytes at the enterprise level. These volumes can also be quantified in 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, 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 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 the following features are also key [5]:

- **Veracity of data.** The volume of data is key to obtain 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 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 analyse data.

These definitions have a key aspect in common: Big Data focuses on data, in particular, on data that is perceived as large in volume. This paradigm-shift centred towards data has affected all areas of computing from data acquisition, transfer and storage; to data analysis and visualisation. This reflected on traditional areas of business and science –like genomics, climatology, finance, and business intelligence– that 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 like Internet-of-things (IoT) and Big Data analytics (BDA) developed greatly thanks to the advances in Big Data.

Big Data analytics (BDA) is one of the best examples of how Big Data disrupted an established area like 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 to detect interrelations among apparently unrelated attributes of datasets [6]. Enterprises can now understand the current state of the business and customer behaviour through complex techniques like predictive analytics, data mining, statistical analysis, data visualisation, 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 around for years and they have been revamped due to their good adaptability to very large data sets with minimal data preparation. In addition, infrastructures like cloud computing offer the possibility to lower the economical costs of deploying BDA, and building analytics workflows at different levels of abstractions.

Figure 2.1 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 visualisation, finally incorporating feedback mechanisms to refine data processing and modelling stages. This workflow has been usually combined with the lambda architecture [7] to provide scalable integration and interoperability across different datasets trough 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 Fig. 2.2, the lambda architecture includes a speed layer

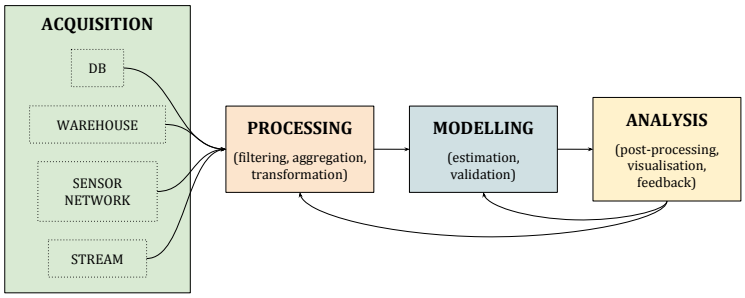


FIGURE 2.1: Typical workflow of a BDA application.

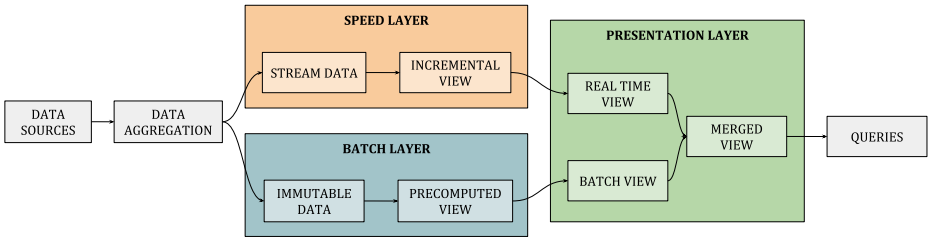


FIGURE 2.2: Representation of the lambda architecture.

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 visualisation.

Looking ahead, it is expected that areas such as mobile technology, social media, IoT and data-driven sciences will generate data to a global total in the order of dozens of zettabytes [8]. This data will yield valuable information for smart applications, science and decision making processes in business.

2.1.1 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 [9]. In this context, cloud computing became a widely adopted infrastructure for BDA [10].

Cloud computing is a popular paradigm that relies on resource sharing and virtualisation to provide the end-user 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 [11], and its popularity is a consequence of some of the core features of cloud service models, such as:

- Minimal management effort, as 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 [12].
- Automatic or manual scale up or down according to utilisation, 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 [13]. Nevertheless, cloud computing presents challenges related to the lack of control of the underlying hardware infrastructure, the privacy concerns that arise from hosting data sets on third-party servers, and the transfer time and cost required to upload and download large quantities of data [14]. For some applications relying on many data sources generating large volumes of data at high velocities, centralising all data to a very limited number of data centres 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 centralisation problem by combining processing, storage

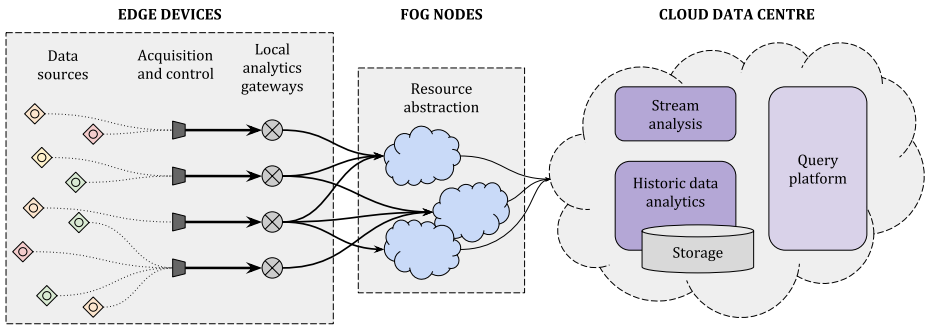


FIGURE 2.3: Architecture of a massively distributed infrastructure comprising edge devices, intermediate fog nodes, and core cloud datacentres.

and communication in distributed services that run closer to the data production environment in a hierarchical multi-tiered manner. These paradigms include mobile cloud computing [15], edge computing [16, 17] and fog computing [18]. Figure 2.3 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 centres 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 [8].

2.1.2 Data-Centric Batch and Stream Processing

Minimising data movements is very important for the final performance. At the application development stage, working with programming models that

provide a data processing layer able to abstract resource allocation, data management and task execution can result in an improvement of performance and locality.

The map-reduce [19] data processing model was the most relevant data-centric model when BDA research took off, as it enables analytics on big datasets by parallelising computations for HPC and multi-core environments [20]. 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) [21] to achieve locality by block replication, and considers data-aware task scheduling. A similar approach is followed by the open map-reduce implementation, Hadoop [22], and its partner file system Hadoop Distributed File System (HDFS) [23]. Map-reduce applications work with many large files and need to execute fast transfers and operations on a wide and diverse dataset. Besides the numerous works that took advantage of it to improve performance of a wide range of applications, it had a major impact in subsequent map-reduce-inspired models.

One of the models that emerged from map-reduce is map-reduce-merge [24], 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, limitation that causes efficiency issues when map-reduce is applied in relational operations like joins. The map-reduce-merge model can, on the other hand, express relational algebra operators and implement several join algorithms.

Map-iterative-reduce [25] is an alternative model that extends map-reduce to better support reduce-intensive applications, while substantially improving its efficiency by eliminating the implicit synchronisation barrier between the

map and the reduce phases. Among implementations of map-iterative-reduce we can find Twister [26], Haloop [27] and Twister4Azure [28].

The work in [29] 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 [30] programming model supports a wide range of functionalities that enable the development of applications that do not fit nicely the map-reduce paradigm, such as many iterative machine learning algorithms and interactive data analysis tools. The Spark framework relies heavily in the concept of *resilient distributed dataset* (RDD) [31] to provide this functionality. RDDs are in-memory collections of data, and the operations on them are tracked in order to provide significant 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 [32] the deep component stack and its dependence on the JVM yield a significant memory consumption that also affects execution time due to frequent garbage collection operations [33] and serialisation if bindings to other languages are used [34].

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 re-implement repetitive tasks by working on top of the processing layer [35]. For example, Spark has inspired subsequent works like GraphX [36], 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, Pig Latin [37], HiveQL [38] and REX [39] 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 [40]. As these modules interact and exchange significant volumes of data at runtime, minimising these transfers and making them efficient has a major impact in the overall performance [41]. Consequently, data locality enforcement has been studied in several works tackling task and job scheduling [42], data-flow optimisation [43], and resource allocation [44].

In-memory computing has also affected database oriented platforms with approaches like Phoenix [45] for shared and distributed memory machines. Shark [46], which supports the Hive warehousing system [47] on Spark, is a popular similar approach, but oriented towards 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 [32], Piccolo [48], Main-Memory Map-Reduce (M3R) [49], and Hyracks [50].

Some of the former works indicate that in-memory databases and computing are able to scale to petascale systems. No further work has found indicating whether this could hold for exascale systems though. New technologies based in multi-core processors can improve the performance of applications by favouring locality through intra-node data sharing, which minimises data exchanges across compute nodes [41]. 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.

2.2 High-Performance Computing Ecosystem

High-performance computing (HPC) 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 in the order of one hundred petaflops, and a machine capable of delivering one exaflop is expected to appear around 2020 [51].

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 due to their performance requirements. This includes sectors like 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 maximise the usage of resources. Supported by advanced runtimes, 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 2.4 depicts these relationships which form the structure of many HPC applications. The core simulated models are typically initialised 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

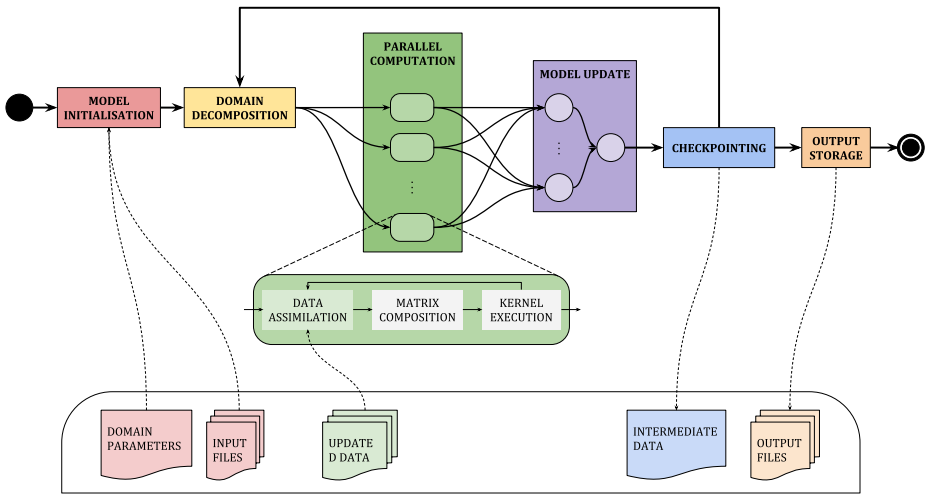


FIGURE 2.4: Application model for the typical HPC scientific application.

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 check-pointing procedures to store intermediate models and restore the simulation from them in case of failure. Finally, simulation results are written to storage.

HPC is not unaffected 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, visualisation 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.

2.2.1 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. As 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 towards 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 are built in such way that storage and computation are not located in the same nodes, following the schema depicted in Fig. 2.5. 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 higher degree of parallelism.

There are several issues that are still not solved by the academia with regard to these infrastructures. In particular, computer scientists have realised that, as problems become larger and more complex, a powerful infrastructure is not sufficient to achieve proper scalability, both in terms of overall performance, resource utilisation, 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 towards exascale infrastructures efficiently [52].

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 [53]. Storage systems constitute one of the greatest bottlenecks when dealing with data-intensive computations. Therefore, data awareness in file

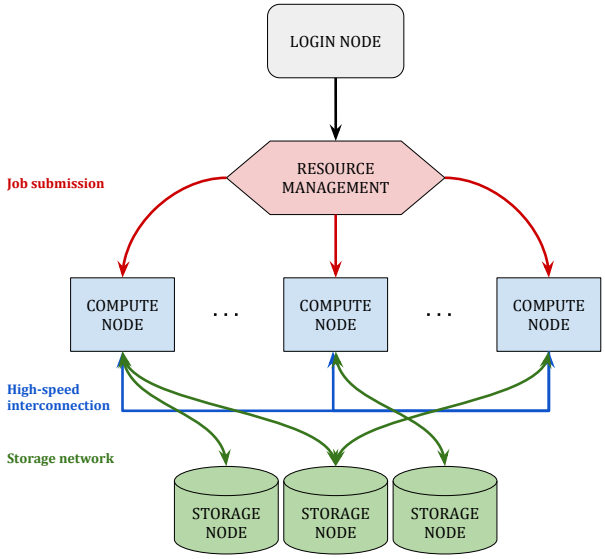


FIGURE 2.5: Traditional architecture of a HPC infrastructure, with isolated storage and computation networks.

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. 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 like map-reduce. The Hadoop File System (HDFS) [54] and the Google File System (GFS) [21] 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 minimise 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-centres [55].

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 like vector processors, tensor processing units (TPUs), general purpose graphical processing units (GPGPUs), and field programmable gate arrays (FPGAs).

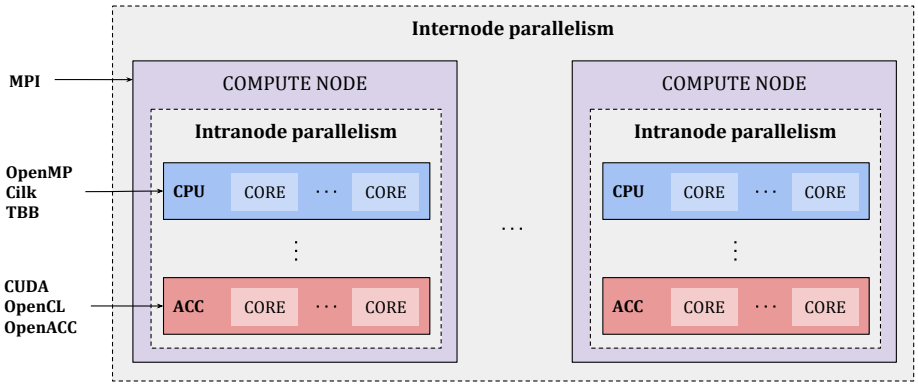


FIGURE 2.6: Parallelism layers in HPC programming models, including inter- and intra-node parallelism.

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, as the bottleneck might remain at upper layers of the software stack like monitoring, resource management, data management, and communications [56]. In addition, applications are also evolving towards complex workflows involving iterative analytics, data-intensive operations and compute-intensive computations. Making an efficient usage of supercomputers in this landscape will require algorithm, runtime and data management refinements to support applications with mixed requirements, without diminishing usability.

2.2.2 Parallel Programming Models and Runtimes

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 2.6 represents these diverse processing units.

The message passing interface (MPI) standard is the most common procedure to exploit inter-node 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 terascale applications.

Thread-level parallelism is the basis for fine-grained intra-node parallelism for multi-core CPUs. Developers can choose from a wide range of threading libraries like POSIX threads, Intel's threading building blocks (TBB) [57], and Microsoft's parallel patterns library (PPL) [58]. Nowadays, the open specification for multi-processing (OpenMP) [59] is still one of the most used tools for parallelisation, mostly because its annotation-based nature minimises 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 [60–62].

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-grain intra-node parallelism. GPGPUs are the most widely adopted accelerator in current HPC machines given they power efficiency and their many-core architecture, which pushes forward massive parallelism to the order of thousands of cores in a single chip. There are several libraries that enable the interaction with GPGPUs, such as OpenCL [63], Nvidia CUDA [64], and OpenACC [65], 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 intra-node parallelism through OpenMP [66], and inter-node parallelism via MPI [67, 68]. The mechanisms to build hybrid runtimes exploting both intra- and inter-node parallelism had major influence in subsequence 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 [69].

2.3 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 2.3.1 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, Sec. 2.3.2 presents how data-centric paradigms and BDA infrastructures (primarly clouds) have been exploited to enhance HPC.

Finally, Sec. 2.3.3 analyses the most relevant endeavours towards HPC and BDA interoperability for hybrid applications, not necessarily attempting to improve one model or the other, but focusing on the goal of coexisting both paradigms in a single application. These works are scarcer than others that focus on improving a single ecosystem, but are highly relevant since they share the same scope than this thesis.

2.3.1 Usage of HPC to Enhance BDA

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

Process-Centric Computing Models: MPI and OpenMP

Implementations of traditionally data-centric frameworks like map-reduce have been developed 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 of such frameworks was proposed in [70], which is a parallel library that allows algorithms to be expressed in 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 [71], a framework that mimics map-reduce to execute data analytics algorithms alongside computational simulations –in a process known as *in-situ analytics*– in time-sharing or space-sharing modes. The framework uses both MPI and OpenMP to parallelise tasks over distributed and shared memory. A more recent map-reduce framework over MPI is Mimir [72]. It includes a redesign of the execution model with optimisation techniques to increase performance, reduce memory usage, and improve scalability. Another variant is FT-MRMPI [73], 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 [74] 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 9x-21x over Apache Hadoop, and 2x-3x 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 [75], where the authors presented a set of building blocks that provide scalable data movement capability to computational scientists and visualisation researchers for writing their own parallel analysis. This work is the origin of the Do-It-Yourself parallel runtime (DIY) [76], a full data-driven runtime usable for any topology defined by the user.

Infrastructure: Networking and Accelerators

Optimising data-centric platforms for specific heterogeneous architectures is

a popular direction to accelerate BDA. MrPhi [77] targets Intel Xeon Phi coprocessors. A solution for hybrid cloud bursting is discussed in [78] and extended with a detailed performance model [79]. A similar solution for Spark on GPUs was IBMSparkGPU [80], but it is valid for local tasks only. Trace [81], is a high-throughput tomographic reconstruction engine for large-scale datasets using both (thread-level) shared memory and (process-level) distributed memory parallelisation using a special data structure called a replicated reconstruction object. The authors also studied in [82] various frameworks for deep learning networks that can scale across multiple machines with full parallel support and distributed execution, such us Tensorflow, CNTK, Deeplearning4j, MXNet, H2O, Caffe, Theano, and Torch.

Networking and storage are closely related and have been exploited to improve BDA platforms. A first attempt to optimise map-reduce storage on HPC clusters by utilising Lustre as the storage provider for RDMA intermediate data was presented in [83]. Other works tried to adapt map-reduce and its underlying HDFS to use GPFS [84–86]. Results indicated that BDA platforms still suffered from the reduced locality offered by such setting. Consequently, the authors in [87] proposed a two-layer storage system that exploits PFS performance but incorporated an intermediate in-memory storage system with good results.

A proposal to accelerate Spark communication was presented in [88], which used a high-performance RDMA-accelerated data shuffle in the Spark framework on high-performance networks and provided a performance improvement of 80%. Finally, we 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 [89].

2.3.2 Usage of BDA to Enhance HPC

Evidence of convergence in the opposite direction also appears, specially works aiming to incorporate data-centric computing models like map-reduce in HPC applications, and efforts to exploit BDA computing facilities like clouds to scale scientific computing.

Data-Centric Computing Models: Map-Reduce

Scientific applications and their adaptability to new computing paradigms have dragged increasing attention from the scientific community. The applicability of the map-reduce scheme for scientific analysis has been notably studied, specially for data-intensive applications, resulting in an overall increased scalability for large data sets, even for tightly coupled applications [90].

Several works have analysed how current HPC applications could be adapted to map-reduce models. In [91], Srirama, Jakovits and Vainikko study how some scientific algorithms could be adapted to the Hadoop map-reduce framework. They establish a classification of algorithms according to the structure of the map-reduce schema these would be transformed to. They suggest that not all of them would be optimally adapted by their selected map-reduce implementation, yet they would suit other similar platforms such as Twister or Spark. They focus on the transformation of particular algorithms to map-reduce by redesigning the algorithms themselves. A similar approach is HAMA [92], a framework which provides matrix and graph computation primitives on the 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. Finally, an approach for using Hadoop map-reduce in scientific workflows is that explained in [93], whose authors propose a new architecture named SciFlow. This architecture consists on a new layer added on the 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. Finally, a theoretical analysis of migrating common HPC-oriented workflows to a BDA processing platform (i.e., Apache Hadoop) was made in [94]. Authors implemented six representatives of common scientific workflow patterns in Apache Hadoop environment and discussed implementation challenges as well as Hadoop environment applicability for each of the basic patterns.

Other works have attempted 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 generalised for reuse in other contexts. A preliminary work was ROOT [95], an object-oriented C++ high-energy physics (HEP) framework designed for storing and analysing petabytes of data efficiently by using an object container optimised for statistical data analysis over very large data sets. Another attempt is an extension of map-reduce with access patterns (MRAP) [96], which targets HPC analytics with a focus on data locality.

BDA analytics tools –like 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 minimise transfers. The inherent parallelism of these tools has resulted in positive experimental results showing their suitability for massively parallel workloads like MTC-like workflows [97]. Other works explored the usage of high-level machine learning libraries for HPC ptychographic reconstruction [98] with good results. Nevertheless, challenges remain with respect to workflows built with a pure HPC focus, which rely on MPI and traditional storage infrastructures [99].

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 [100] 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 [101], 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 shown in [102], 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.

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 in the overall performance of the resulting application [103]. 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 [104]. 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 [105]. Given the workflow nature of many state-of-the-art simulators for scientific computing, Srirama et al. [106] proposed a workflow-partitioning strategy to reduce the data communication in the resulting deployment. Matri et al. [107, 108] analysed 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 like 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 [109, 110]. Experimentation with well known workflows shows that running costs could be significantly decreased with BDA infrastructures, but performance would suffer from virtualisation and latency overheads [111–113]. The relationship between map-reduce and the cloud for scientific applications has also been tackled in [114], which establishes that performance and scalability tests results are similar between traditional clusters and virtualised infrastructures. Nonetheless, these results for map-reduce workflows are not generalisable 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 [115, 116]. 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 [117, 118]. Consequently, cloud computing has been proved as a good solution for scientists who need resources instantly and temporarily for fulfilling their computing needs [119].

In this context, trends evolved to migrate scientific applications to the cloud by means of several techniques. D'Angelo [120] 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 synchronisation. He concludes that the potential challenges concerning hardware, performance, usability and cost that could arise could be overcome and optimised with the proper simulation model partitioning. Following a similar approach, Yu et al. [121] proposed an application adaptation middleware to allow legacy code migration to the cloud. In this work, a virtualisation architecture is implemented by means of a web interface and a Software-as-a-Service market and development platform. Similarly, [122] proposes moving desktop simulation applications to the cloud via virtualised

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.

2.3.3 Endeavours Towards HPC and BDA Interoperability

The scientific community is aware that tools like Apache Spark[1] 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 [123]. Using Spark for HPC applications, while appealing, poses important convergence challenges.

The work in [124] introduced a methodology for graph processing to bridge the gap between Spark-based graph computing and HPC. Evaluations made in 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 [125] a framework named HPC-ABDS, which detected points for possible integration, but also identified problems with workflow systems, data transport, and file management layers. Gittens et al. explored in [126] the trade-offs of performing linear algebra using Apache Spark, compared to traditional C and MPI implementations on HPC platforms. The results showed a poor performance of Spark vs. MPI for matrix multiplications: from 2x to 25x performance gap. However, the authors highlight the potential of incorporating MPI-based runtimes 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 would be desired by the scientific community.

[1] See https://spark.apache.org/

This thesis presents an abstract architecture for runtime interoperability that, to the best of our knowledge, has not been introduced before in the literature. In addition, we provide an implementation that allows users to benefit from efficient MPI libraries accessible from Spark with little effort on their parts. As a result, we achieved a platform (called Spark-DIY) that provides advanced capability compared with other related solutions in the literature. For example, compared with [88], we provide compatible block management between the native side and Spark by using JNI. Compared with [83], 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 [100]. Our approach is more similar to the solution proposed in [127], but Anderson et al. use HDFS to exchange data among Spark and MPI, while we use memory directly for increased 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, there are two platforms that are very close to Spark-DIY in terms of aim and functionality. Spark-DIY is similar to Spark-MPI [128], 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 [129] 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.

2.4 Summary

This chapter presented a depiction of the HPC and BDA ecosystems, diving into relevant works covering their convergence at different levels. Although there are numerous efforts related to the enhancement of one model with elements of the other, works covering true interoperability between platforms for hybrid applications are scarce and inconclusive, and are mainly focused on specific runtimes like MPI and Spark.

We could not find abstract architectures and theoretical work regarding HPC and BDA runtime interoperability, which is the main topic in this thesis. However, there are similar works related to our final implementation of a platform for HPC-BDA applications, but they lack the generality and flexibility of our solution.

With the objective of establishing the foundation of the work in this thesis, the following chapter analyses which are the most valuable features of the ecosystems hereby presented, describing exactly the scope of this work and the challenges that must be faced to achieve convergence in this context.

This chapter includes content published in:

- 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"* [130].
- S. Caíno-Lores, J. Carretero, *"A Survey on Data-Centric and Data-Aware Techniques for Large Scale Infrastructures"* [131].

CHAPTER 3

PROBLEM STATEMENT

HPC and BDA applications have conditioned their traditional solutions to the infrastructure and software architectures found in their respective ecosystems. Nowadays, with the advent of new problems requiring hybrid approaches, convergence became a critical priority for the industry and the academia, which brings new opportunities and vast challenges at all levels of the system stack.

This chapter presents a deep motivation of this thesis, comparing side by side both ecosystems, analysing why convergence is so difficult to attain out-of-the-box, and introducing the trade-offs that must be balanced depending on the final goal and target use case that are selected. In light of this information, this chapter also presents the specific problem covered by this thesis, stating the hypothesis and overall approach that will be developed in following chapters.

3.1 Problem Analysis

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 by various scientific communities to solve their own data analysis challenges [132]. HPC-BDA convergence became a hot-topic as applications and their associated data evolved outside from their original ecosystems. At that time, the problem for HPC was *how can we cope with increasing datasets?*, while BDA was wondering *how can be run analytics faster?*. Studying the existing

trade-offs, various experts considered there was a need for convergence of the classical HPC and BDA software stacks.

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. Major technical differences between HPC and BDA ecosystems include software development paradigms and tools, virtualisation and scheduling strategies, storage and networking models, resource allocation policies, and strategies for redundancy and fault tolerance [133]. 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. This situation led to the advent of specific research topics like high-performance data analytics and data-driven science.

Many challenges remain unsolved and this situation has been worsened by the appearance of new application domains that are completely hybrid in nature, like autonomous vehicles, surveillance, e-science with Big Data sources, monitoring of large scale infrastructures, and smart cities, to name a few. These domains have in common the need to support the simulation of very 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 behaviour. In this circumstances, borrowing features from the other paradigm proves insufficient, and deeper convergence becomes necessary 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 following paragraphs detail the causes identified in this thesis that affect further convergence.

3.1.1 Programming and Data Models

Operations in BDA and HPC are defined using different programming models, in particular there is a huge gap between functional and procedural programming, which yields complex variety for hybrid workloads in terms of their algorithmic structure [134]. In addition, it is sometimes required to incorporate legacy kernels and specialised components, making necessary to leverage HPC mathematical libraries for BDA, incorporate specialised numerical libraries for accelerators, and interoperate data formats.

As a result, programming models and software development tools in the BDA and HPC worlds are inconsistent [135], and trying to mix both models out-of-the-box generates memory overheads and poor scalability in a HPC environment [136]. In addition, the usage of merged BDA models presents limitations, such as high memory consumption and low efficiency in communication between cooperating processes [137].

Some BDA-oriented platforms also show drawbacks in terms of generality and versatility, since the offered functionality is limited to the common operations needed for data analysis. As a consequence, there is a need for a hybrid paradigm with coherent memory and a unified programming environment. Interoperability and data locality should be a priority, since data movement dominates performance and energy at scale.

3.1.2 Runtimes and Platforms

Middleware is built with different performance, multi-tenancy, and fault-tolerance expectations. Moreover, some of these requirements may not be

present in both models (i.e. MPI-based workloads implement fault-tolerance assuming check-pointing instead of transparent re-execution, as in BDA platforms). Thus, besides the importance of additional functionality, the core operational behaviour of these runtimes is currently tailored for diverse requirements in terms of size and volume of tasks, which is deeply related to the degree of parallelism and tenancy.

BDA platforms –like Hadoop and Spark– and infrastructures –such as clouds– could be used to improve the efficiency and scalability of some types of scientific applications with minor modifications aimed towards introducing the required degree of data locality. More specifically, simulators relying on parameter-sweep and partitionable domains, and kernel-based workflows comprising many loosely-coupled tasks, could greatly benefit from the massive parallelism of BDA paradigms. Another main benefit of having BDA frameworks as execution engines are their underlying resource manager and distributed file systems, which ease data distribution and task management. However, the disparity between collocated and distributed storage architectures in BDA and HPC systems, respectively, degrades performance when running BDA applications on HPC systems [138].

Finally, there is a general lack of performance metrics for hybrid applications, which are not purely compute-intensive any more, and may be borrowed from each ecosystem as required by each stage in the application.

3.1.3 Computing and Storage Infrastructures

To efficiently support new application domains, it is necessary to facilitate convergence below the upper system layers, exposing access to accelerators, local, distributed and parallel storage. In addition, building a solution for BDA environments like clouds expands the potential flexibility to configure the necessary hardware at each stage of the application.

Current application domains have shown a huge increase in the complexity of BDA applications, usually driven by the computation-intensive simulations, which are based on complex models and generate enormous amounts of output data. On the other hand, users need to apply advanced and tightly complex analytics and processing to this data to generate insights, which usually means that data analytic has to take place in-situ, using complex workflows and in synchrony with computing platforms. This requires novel BDA architectures, which will exploit the advantages of HPC infrastructure and distributed processing, and arises the challenges of maintaining efficient distributed data access and energy efficiency in such architectures.

Nowadays, innovative computing platforms are being proposed to cope with the requirements of modern applications. On one hand, new storage technologies like flash-based solid-state drives (SSDs) assisted supercomputers in their search for BDA support by providing deeper storage hierarchy that reduced the latency gap between main memory and the parallel file system [55, 139, 140]. On the other hand, clouds have revamped their underlying data centres to provide bare metal access to cutting-edge processors and accelerators on demand, both dedicated to mimic traditional clusters, and virtualised offering high-performance and heterogeneous cloud computing capabilities [141]. Both approaches are fusing into hybrid architecture models –like edge with supercomputing support– that bring many opportunities and challenges for software platforms [8].

3.2 Convergence Challenges and Opportunities

In order to study the convergence challenges and opportunities for convergence, we have summarised the main features of both ecosystems in Tab. 3.1. We now proceed to analyse the challenges and opportunities they yield for future convergence.

TABLE 3.1: 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 virtualisation 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

From the domain perspective, it is clear that the iterative nature of the simulation algorithms yield collective operations that do not fit nicely into the typical BDA paradigms. Therefore, significant efforts must be conducted to converge simulations and BDA algorithms [142].

Regarding workflow development and deployment, we conclude that a promising research line for large scale scientific workflows would be working towards an hybrid approach between MPI and BDA-oriented data abstractions. Such 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 the computational elements of their work. As we have seen, they come at the cost

of large amounts of memory overhead. 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 [142, 143].

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 shall 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 towards batch and iterative workloads (e.g. multi-level check-pointing), with the methods from the BDA side that tackle large volumes of tasks (e.g. data replication and provenance). This also has an effect in locality, and trade-off 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 factor that emerging MTC scientific workflows also require significant amounts 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 non-volatile memory and deepen the storage stack. Hence, increasing memory in commodity clusters and clouds is key to support the

upcoming execution platforms. These additional resources could mitigate the requirements of new workloads, and they would help the support of the emerging in-memory and caching mechanisms coming from data-aware computing. In addition, it is expected that this integration with more BDA-oriented infrastructures will benefit pure HPC workloads in the near future [144].

To summarise, this 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 implementing virtualisation for sharing, resource allocation, and efficiency. Software libraries for common intermediate processing tasks need to be promoted, and a complete software ecosystem for application development is needed. Finally, the divergence of programming models and languages poses a convergence issue, not only with regard to interoperability of the applications, but also to the interoperability between data formats from different programming languages.

3.3 Proposal

The former chapters and sections have provided a deep motivation and contextualisation of the topics covered in this thesis, and have detailed the current challenges and open problems in this field. At this point, we introduce the novel approach proposed in this work to tackle the problem of BDA and HPC convergence at the software level by defining, firstly, the guiding hypothesis, followed by the thesis and an overall view of our proposed approach.

3.3.1 Hypothesis

In the context of HPC-BDA convergence, we formulate that:

> *HPC-BDA convergence at the platform level can be attained by enabling interoperability of existing process- and data-centric runtimes.*

This must be done in a way that addresses a subset of the major challenges found in our analysis, namely preserving BDA platform usability and productivity, exploiting HPC scalability, and expanding both BDA and HPC capabilities to cope with prospect hybrid application models.

3.3.2 Thesis

The purpose of this thesis is *to research new approaches to facilitate the convergence of HPC and BDA paradigms* by providing common abstractions and mechanisms for improving scalability, data locality exploitation, and execution adaptivity on large scale computers.

This thesis shall outcome compromise solutions for generality, performance and efficiency, taking into consideration the many convergence challenges covered in the previous sections. More precisely, this thesis will focus on infrastructure independence, the need to conduct minimal changes to the platforms (none if possible), seamless integration, and the minimisation of interoperability overhead between runtimes.

3.3.3 Approach

To achieve the former goal, after analysing the key features that characterise HPC and BDA ecosystems (O1), we will design and develop an architecture that offers runtime interoperability for hybrid HPC-BDA applications (O3). Such an architecture will rely on an auxiliary mechanism to reshape HPC-oriented workflows in order to adapt them to data-centric environments (O2); a unified operational and data model of BDA and HPC; and support for

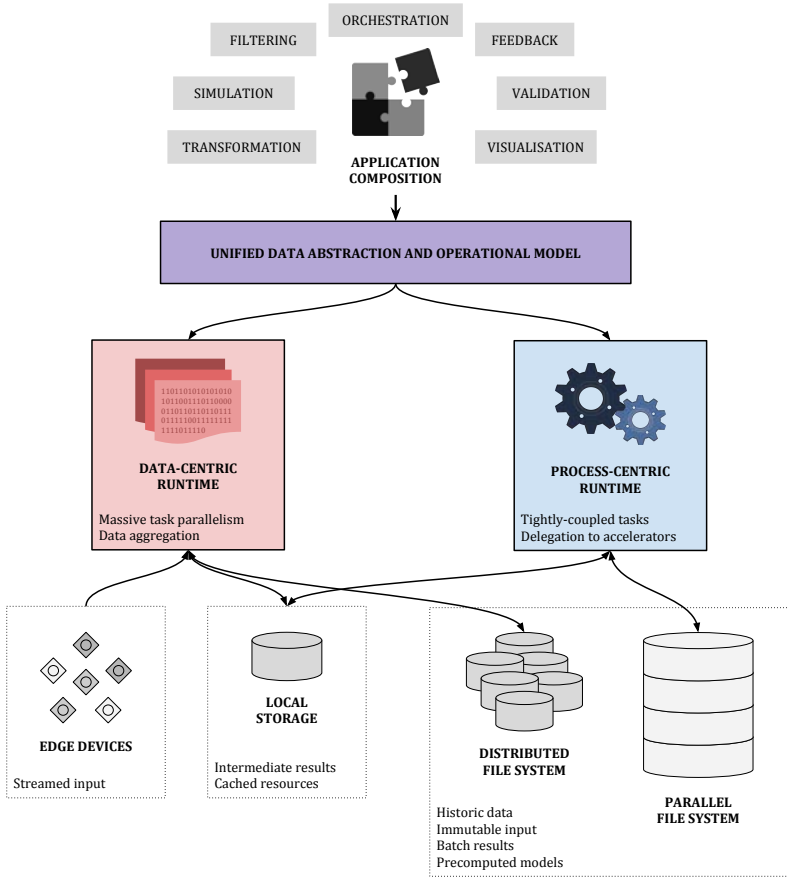


FIGURE 3.1: Overall thesis approach, depicting the core elements involved in the interoperation process.

high-level analytics methods and high-performance kernels for composite applications.

Given the context of this thesis and its main purpose, we propose to divide the architectural proposal in two core parts: one is related to the data-centric platforms that run beneath BDA workloads, and the other is related to the process-centric frameworks that support HPC applications. Figure 3.1 represents this duality and portrays the confluence of both runtimes.

In order to exploit the benefits of data-centric programming, task, and data

models, first of all we shall provide a data-centric transformation mechanism that allows process-centric workloads to interact with BDA platforms and storage infrastructures (O2), including local storage for cached resources, intermediate results, and auxiliary data; distributed storage for historic data, immutable input datasets, batch results, and precomputed models in the context of lambda models; and edge devices for stream-like input. This supports complex data aggregation and massive task-parallelism. There are plenty of benefits from adopting a data-centric view on computing problems, namely the side effect of cloud enablement, portability, and strong usability.

On the other hand, we need support for tightly-coupled tasks, which might also rely on acceleration kernels and data coming from isolated networks (e.g. parallel file system). The integration of process-centric runtimes enhances data-centric workloads with the possibility of exploiting upcoming architectures –such as edge with supercomputing assistance– and specialised hardware for common tasks –e.g. deep learning–.

These elements will be exposed through a common data abstraction and operational model that provides transparent access to the features desired at each stage of the workflow, which can be composed by typical HPC tasks like simulations, or operations that are usual in BDA settings, like visualisation and filtering. This approach offers enhanced usability for HPC users, extended functionality for BDA users, potential performance and scalability improvement in hybrid scenarios, and efficient workload integration through seamless runtime interoperation.

To asses the feasibility of the proposed architecture for HPC-BDA, we will explore the trade-offs between transparency, flexibility, and performance that appear in the former design by using synthetic benchmarks and real-worlds applications. Performance and scalability will be also assessed through evaluation on a meaningful use case, representative of target hybrid applications (O4).

3.4 Summary

Convergence between HPC and 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 virtualisation techniques and novel storage hierarchies. Therefore, the HPC [145] and BDA [8] communities have recognised new opportunities in unifying the platform layer and data abstractions for both HPC and BDA [3].

This thesis builds on the hypothesis that HPC-BDA convergence at platform level can be attained by enabling runtime interoperability in a way that preserves the beneficial features of BDA and HPC platforms, and expands both BDA and HPC capabilities to cope with prospect hybrid application models.

Our approach is to architect an abstract system that enables the interoperability of established BDA and HPC runtimes, in light of their canonical underlying infrastructures, and considering the requirements of hybrid applications, which we have analysed in depth. The following chapters develop this approach, detailing the architectural details of each component, while supporting its viability with evaluations conducted on synthetic and real use cases built for an implementation of this architecture.

This chapter includes content published in:

- S. Caíno-Lores, F. Isaila, and J. Carretero, *"Data-Aware Support for Hybrid HPC and Big Data Applications"* [146].
- 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"* [147].

- 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"* [130].

CHAPTER 4

DATA-CENTRIC TRANSFORMATION METHODOLOGY FOR HPC PROCESS-CENTRIC APPLICATIONS

Process-centric workloads, like HPC scientific applications, are key tools in many research areas that rely on multiple, diverse, and distributed operations over various datasets, usually yielding major computational complexity and data dependencies. While current process-centric workloads rely on hundreds of gigabytes of intermediate data [148], trends show that large scale scientific applications would have to address increasing data sizes, easily reaching petascale [149]. In this context, such applications face new performance and scalability challenges [150]

Recent works have suggested the opportunity of combining the traditional HPC approaches with BDA paradigms [2]. For example, typical BDA programming models –such as Apache Hadoop– have been considered to substitute MPI parallelism techniques, following a data-centric approach. In addition, we can also see this opportunity affecting the underlying computing infrastructures. Indeed, cloud computing –a key element in current BDA systems– could inspire hybrid platforms for exascale scientific workflows in which storage is not completely isolated from computing nodes [3, 151]. Applying some of these BDA techniques could improve scalability in certain types of scientific applications, especially those with many loosely-coupled tasks [152], or heterogeneous tasks with few interdependences [153].

In order to exploit the benefits of data-centric programming, task, and data models in a hybrid setting, first we must expose a mechanism to allow the interaction of process-centric workloads with BDA-oriented platforms and storage

facilities. In this chapter, we introduce a data-centric transformation methodology that enables complex data aggregation, massive task-parallelism, and infrastructure portability for existing process-centric applications. The following sections define the structural requirements of the applications that can be adapted with the guidelines provided by this technique, and describe the methodology itself.

4.1 Structure of HPC Process-Centric Applications

HPC scientific applications represent large-scale computational experiments in different domains. A scientific application aims to organise computational steps into a logical series in order to prove a scientific hypothesis based on a mathematical model. A good representative of such applications, which provides an execution environment and tools for data management, analysis, simulation, and visualisation, is a simulator. Simulators firstly emerged in meteorology and nuclear physics, then they became crucial in many other disciplines like economics, sociology, biology, geology, hydrology [154].

Simulators differ based on four main types of simulation problems defined in the literature, namely equation-based, agent-based, multi-scale, and Monte-Carlo simulations [155–157]. While equation-based and agent-based simulations are widely adopted and well-studied types, two other types –multi-scale and Monte Carlo simulations– were developed later due to the requirement of much greater computational power. Nevertheless, all of them share their iterative nature: a simulation relies upon a pre-defined algorithm that takes as an input a specific state of the system at a given time, and calculates its state at the next time slot under certain conditions.

This general structure of a HPC scientific application is thus depicted in Fig. 4.1, highlighting the key regions we will focus on:

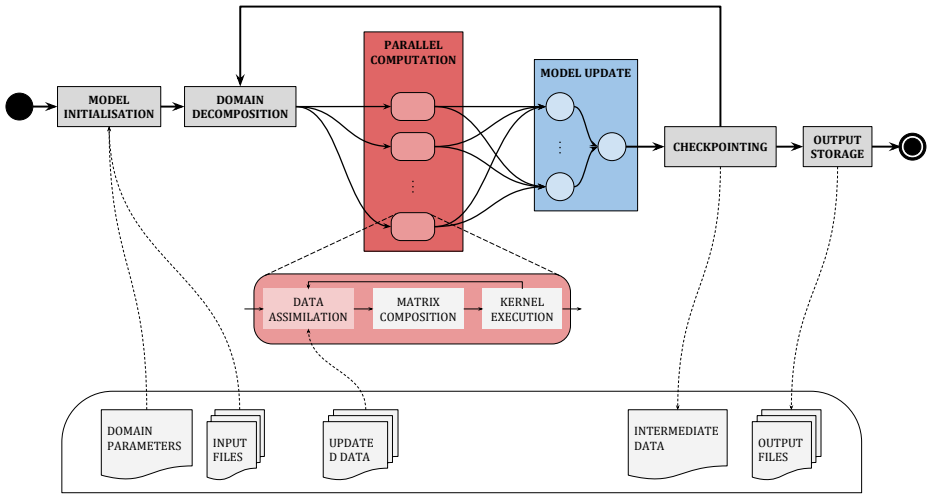


FIGURE 4.1: General structure of a HPC scientific application, with highlighted parallel and model update regions.

Parallel Computation Region. Most simulations start from a base model that is instanced according to variations on some of its parameters. If the simulation of these variations can be conducted independently, then we call this a parameter sweep process. In addition, each particular scenario may comprise further fine-grain levels of parallelism for concurrent computation of changes in portions of the domain, or in sections of the simulation algorithms that can be parallelised. These parts of the overall application conform its parallel computation region.

Model Update Region. The computations in the parallel region must be integrated somehow to reach consensus on how the application model will evolve for the following time step [94]. This is typically a communication-intensive process because data must be exchanged across all the processing units in execution. Furthermore, the results may be collected in a single processing unit, thus incorporating significant stress to memory management and usage. These procedures constitute the model update region of the application.

With the objective of making such existing HPC applications suitable for their integration to a hybrid HPC-BDA environment, we need to find a method to transform them with minimal impact on the original code and without disrupting the structure of their parallel and model update regions. However, the architectural differences between the analytics and scientific worlds require novel approaches to achieve satisfactory results in this transformation procedure. In the following section we provide a methodology to incorporate data-oriented mechanisms into production-ready scientific ensemble applications.

4.2 Transforming Iterative Process-Centric Workloads to a Data-Centric Model

The methodology we propose aims to (a) guide the transformation of an iterative parallel scientific application to a data-centric paradigm, while (b) maintaining a comparable level of performance against a traditional design. To achieve this, we have built this methodology in a data-aware manner, as data locality plays a major role in the final performance and scalability of these applications.

Inspired by iterative map-reduce schemes from the BDA ecosystem, the key to provide locality within our model is that independent simulation steps may rely on different node-local data, with no need for further communication. This perspective provides a high degree of parallelism that matches the parallel region of these applications, and also provides support for iterative applications, considering their need for a model update region.

The map-reduce paradigm that assists the methodology consists of two user-defined operations: *map* and *reduce*. The former takes the input and produces a set of intermediate (key, $value$) pairs that will be organised by key, so that every reducer gets a set of values that correspond to a particular key [19]. As a

data-centric paradigm, in which large amounts of information can be potentially processed, these operations run independently and only rely upon the input data they are fed with. Thus, several instances can run simultaneously with no further interdependence. Moreover, data can be spread across as many nodes as needed to deal with scalability issues.

4.2.1 Application Requirements

Our purpose is to divide the application into smaller simulations that can run with the same simulation kernel, but on a fragment of the full partitioned data set. As a consequence, this SPMD scheme matches the parallel region of the application. However, not every application can be transformed using this method, so first it is necessary to state which features must be found in suitable applications. First of all, we must analyse the original simulation domain in order to find an independent variable, k, that can act as index for the partitioned input data and the following procedures. This independent variable would be present either in the input data or the simulation parameters and it could represent, for example, independent time-domain steps, spatial divisions, or a range of simulation parameters. The existence of such independent variable is an absolute requirement of this methodology because we need a key for subsequent steps.

Once such index is found within the simulation domain, we must analyse the input data needed to run the simulation kernels. There are no specific requirements regarding the scope of input data, but it is necessary so establish, for each input element, to which of the following subsets it belongs to:

1. Data that is required for every value of k, this is shared data that needs to be made available to each independent execution.

2. Data that is only required for a specific value of k.

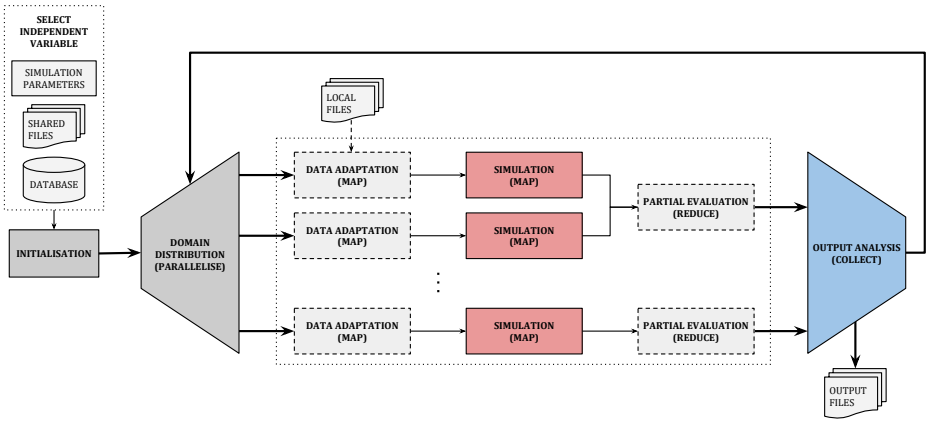


FIGURE 4.2: Overview of the data-centric transformation methodology. Dashed boxes indicate optional stages, which may not be necessary for every application. Red boxes highlight the parallel computation region, and the blue box represents the model update region.

Once the application is shown suitable for the process and all input data has been categorised, it can be transformed by conducting the steps indicated in the following section.

4.2.2 Transformation Process

The proposed methodology is depicted in Fig. 4.2. It consists of the definition of the following steps:

Key Selection. First, it is necessary to conduct an analysis of the original application in order to find a domain suitable for parallelisation, which is necessary to apply the methodology. As a result, we will find independent variables that constitute potential keys, and we must select one of them to act as the partitioning key, k, that will guide the domain distribution and the following stages.

Domain Distribution. Once the parallelisable domain is selected, we can model how the input will be distributed across the nodes to mimic the

behaviour of the parallel region of the application. This parallelisation stage distributes the proper portion of the input, for each value of k. This sets the fraction of the input data or model that will be processed for each instantiation of the guiding independent variable. This stage has critical effects in the final degree of data locality achievable by the simulation, since the proper preparation of the input data and parameters can save subsequent data transfers and communication between the simulations associated with each subdomain.

Data Adaptation. The input data for each partition of the domain might not be originally arranged in the way the simulations will expect. Therefore, in some cases one or more data adaptation stages would be required to aggregate all the necessary data for each value of k.

Simulation. One or more simulation stages wrap the kernels involved in the simulation workflow, in order to simulate each portion of the domain independently and autonomously. This yields the execution of not one, but many smaller simulations. The large number of simulations to be executed factors the inherent complexity of the simulation process, yet it can be massively distributed due to the independent nature of each simulation. Considering the previous domain partitioning stage, each simulation will be scheduled in the computing node that holds each domain partition as it would occur in a map-reduce application, so that no data transfers are required at this point. Therefore, we process the key-specific input in a way that exploits data locality, and minimises data transfers.

Partial Evaluation. Optionally, one or more reduction stages can be defined to filter or join partial outputs before the overall collection and evaluation of results to reduce contention in the synchronisation point for the model update region. This stage can also be used to aggregate node-local data to minimise transfer sizes for the following procedures.

Output Analysis. This constitutes an analysis stage for updating the model after the simulation, in which the processing methods in charge of creating the input for the following iteration take place. In this step, the output evaluation must be defined to reflect the end criteria, the generation of the following input, and the validity of the results per iteration. In the worst-case scenario, a collection point is typically needed to conduct such analysis. However, in some cases these procedures can be executed in a distributed manner, but this depends heavily in the use case and the selected implementation platform.

The objective of the former steps is to find a parallelisable simulation domain, in which we are able to select an independent variable to act as index for subsequent steps. This shall support the parallelisation of the domain in a key-value manner, so that further simulation pipes and optional partial evaluations can take place independently, as seen in massively-parallel data analytics frameworks. Of course, any partition-specific data will only concern the node that is going to process such domain partition, hence we can schedule the computation in the proper node to support data locality. This is particularly interesting if several simulation stages are involved, since they can be scheduled together to benefit from local intermediate files. After these procedures, partial results can be filtered and assessed in parallel as well, again following the initial domain distribution. Finally, these partial results can be analysed and processed to build the next iteration, but the effects of this synchronisation point can be alleviated by previous partial reductions.

4.3 Summary

This chapter described a data-centric methodological approach to enable the usage of BDA-oriented infrastructures and platforms in HPC scientific iterative workflows, which is one of the main contributions of this thesis. The main objective of this methodology is to define the common stages we can

find in these kinds of high-performance applications, and model them in such way we can maximise parallelism and data locality in the resulting workflow. This would allow scientists to benefit from elements in the BDA ecosystem with minimal development efforts, or to modernise their legacy applications systematically.

This data-centric methodology results in a redesign of the application that maximises simulation kernel reusage, because we only need to rearrange input and output data to reduce the scope of each problem. The way data is reorganised is, however, dependant on the particular simulation and the BDA platform used to implement the resulting design. The following chapter exemplifies how this methodology is applied to real use cases.

This chapter includes content published in:

- 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"* [158].
- S. Caíno-Lores, A. Lapin, P. Kropf, and J. Carretero, *"Methodological Approach to Data-Centric Cloudification of Scientific Iterative Workflows"* [159].
- S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, *"A Cloudification Methodology for Multidimensional Analysis: Implementation and Application to a Railway Power Simulator"* [137].

CHAPTER 5

DATA-CENTRIC TRANSFORMATION OF HPC SCIENTIFIC APPLICATIONS

In this chapter we show how to apply the proposed transformation methodology to two real-world use cases: one from the family of parameter sweep applications, and one representing iterative simulation ensembles. The first use case is a railway power consumption simulator (RPCS) that assists in the design, simulation and evaluation of railway electric infrastructures [160]. The second, EnKF-HGS [161], implements the ensemble Kalman filter (EnKF) technique for data assimilation into a hydrogeologic model (HGS) [162].

We have chosen RPCS and EnKF-HGS as case studies because they are a real-world applications in the process of being integrated with other BDA components, hence the need for them to be transformed to a data-centric form. Additionally, scalability is key in these scientific and industrial fields, as the resulting applications must be able to cope with larger experiments and new environmental models with increased complexity. Therefore, these use cases are representative of the simulations we tackle with our methodology, both in terms of complexity and resource consumption.

The following sections describe the applications in detail and elaborate on how the methodology is applied to attain a data-centric implementation. We will also analyse different metrics to evaluate the performance of the resulting transformed application.

5.1 The RPCS Railway Electric Infrastructure Simulation Tool

Power dimensioning and energy saving have been traditionally two main issues regarding the deployment of electrical grids. Railway electric lines, as a particular case of electric grids, are also concerned about these issues, trying to supply a steady flow of energy to the moving trains while saving as much energy as possible.

Simulators and expert systems are frequently used to design and test railway electric lines, prior to their installation, modelling the infrastructure and the train traffic in order to check the behaviour of the system. In particular, RPCS uses train movement information (i.e. train position and power consumption) to calculate the instantaneous power demand taking into account all railway elements such as tracks, overhead lines, and external consumers, which can be translated to an electric circuit as depicted in Fig. 5.1. As a result, the simulation indicates whether the power provisioned by power stations is enough or not. Simulator internals consist on composing the electric circuit on each instant, and solving that circuit using modified nodal analysis (MNA). RPCS is a multi-threading application that is memory bounded, strongly limited by the number of instants to be simulated simultaneously, and therefore by the number of available threads.

Nowadays, modern industrial expert systems are expected to go beyond behavioural simulation, assisting in the process of proposing new designs, taking into account all possible issues that may affect, or even determine, the final validity of a solution. In the context of RPCS, this means supporting the ability to generate different scenarios with variations in the characteristics of the infrastructure (e.g. the position of the electrical substations). Further assistance would involve searching for the best solutions across the problem domain, considering generation and evaluation heuristics extracted from expert's knowledge. This allows choosing the solutions that best fit to specific

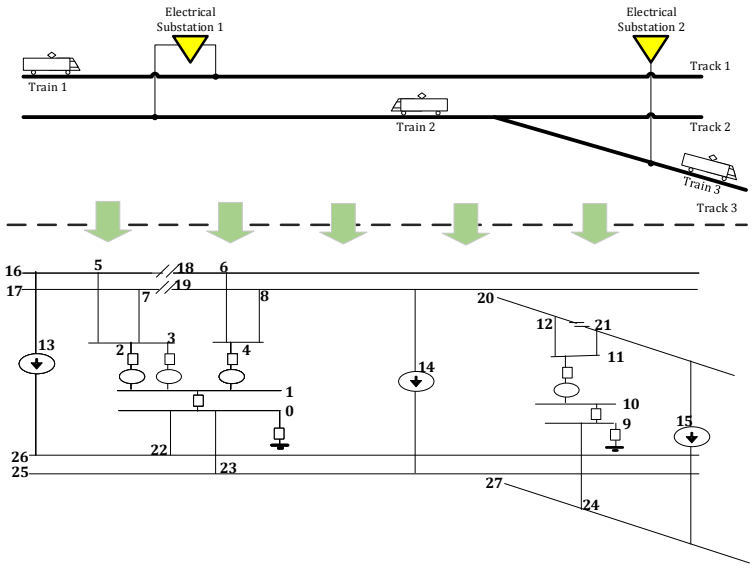


FIGURE 5.1: Railway infrastructure and its translation into an electric circuit.

criteria, and even evolve them in order to improve them following an iterative scheme.

Figure 5.2 represents the aforementioned process as a search engine that generates and evaluates solutions varying a set of parameters, performing the search across the solution space to meet user-defined restrictions and objectives. This engine would rely on RPCS for the simulation of each scenario through a common model ontology that translates the components of the infrastructure into elements of an electric circuit: voltage sources, branches, and consumers (current sources).

However, exploring the search space of a problem in several dimensions (e.g. sweeping different parameters of the simulation) may lead to perform hundreds or thousands of simulations, requiring thousands of hours of computing resources, limiting the number of experiments end-users are able to conduct. In this context, elastic BDA infrastructures like cloud computing raise as an option to scale to larger and more numerous experiments by requesting a

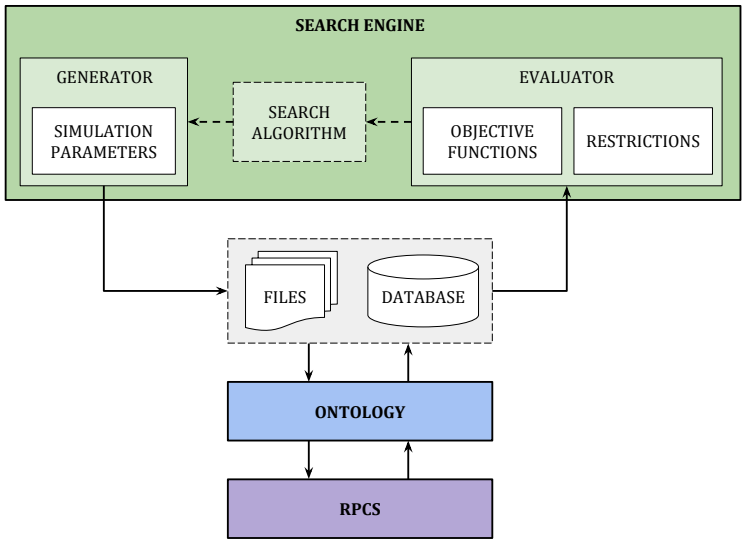


FIGURE 5.2: High-level view of the design generation process involving RPCS

.

distributed pool of virtual resources on-demand. The pay-per-use model of cloud computing frees the user from the burden of maintaining the infrastructure and brings the opportunity to tailor the hardware resources according to specific user needs or simulation characteristics.

In order to adapt the core simulation engine, RPCS, to said BDA environment, we first need to be able to distribute its workload across several nodes exploiting data locality as much as possible to reduce the effects of its memory-bound nature. This makes RPCS a suitable use case for our data-centric transformation methodology.

5.1.1 Simulator Description

The aim of RPCS is, provided a number of trains circulating across the lines, to calculate if the amount of power supplied by the electrical substations is

enough or not. RPCS depends on the definition of a infrastructure representing the railway installation, which contains a set of stations linked by tracks at a specific milemarker.

A number of trains circulate with a pre-defined profile along these stations, and through the tracks, according to their electric properties. This profile is constituted by a collection of records that relate the power consumption of the train with a specific instant and position, expressed as a milemarker of a track. More specifically, we define the profile of a specific train, $\mathscr{P}$, as a set of tuples (t, m, P) where t is the instant in the simulated time in which we know the position of the train on the track, m, and its instantaneous power consumption, P. The electric circuit formed by the trains and the infrastructure is thus composed and solved using MNA.

The overall structure of RPCS is shown in Fig. 5.3. It consists of a preparation phase in which all the required input data are read and partitioned to be executed in a predefined number of threads. Two classes of input files are handled:

- A common *infrastructure specification file* containing the initial and final time of the simulation, besides a wide range of domain-specific simulation parameters such as station and railway specifications and power supply definition.
- A set of train movement files, referred to as *circulation files*), structured in a time-based manner. Each line contains the calculation of speed and distance profiles for a particular train at a specific instant regarding the infrastructure constraints, with a one second interval.

Each of the threads then performs the actual simulation by means of an electric iterative algorithm, storing in shared memory the results that will be merged in the main thread to constitute the final output files.

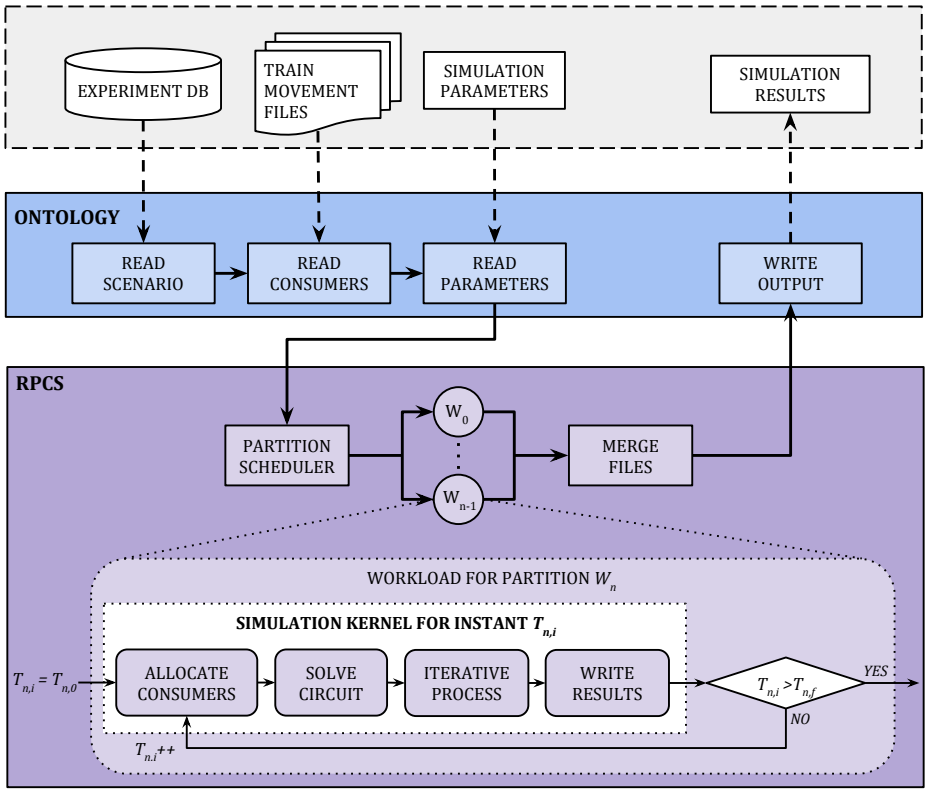


FIGURE 5.3: Detailed view of RPCS internal simulation structure.

The application is multi-threaded, so simulation workload is split among the available cores in the computer. Each thread simulates a different subset of the total simulated time. This split is performed as follows: let t_{ini} and t_{end} the initial and final simulated times defined in the input files, respectively, and let $T_0, T_1, ..., T_{n-1}$ the n threads of the application, the thread T_j simulates all $t_i \in [t_{ini_j}, t_{end_j})$ following Eq. 5.1.

$$\begin{aligned} t_{ini_j} &= j\frac{t_{end} - t_{ini}}{n} + t_{ini} \\ t_{end_j} &= (j+1)\frac{t_{end} - t_{ini}}{n} + t_{ini} \end{aligned} \tag{5.1}$$

5.1.2 Application of the Methodology

The key to transform RPCS to a data-centric model resides in its input circulation files, for they hold an indexed structure that stores in each line an $(instant, parameters)$ pair. As we said before, each simulated instant is independent from the others, because for each instant the circuit has to be composed, solved, and the results obtained, so we can divide the whole simulation period in multiple smaller simulations, each one of covering one second. Therefore, we can consider the temporal key as the independent variable. This covers the first application requirement to apply the methodology.

Regarding shared input data, each sub-simulation requires the overall infrastructure information, plus certain simulation parameters that are shared across all time steps. Independent simulations will also require the train circulation information at the specific time step, so the circulation files need to be partitioned and rearranged to aggregate all the necessary circulation data to simulate one single instant. This establishes the foundation for the distribution of the input data for the simulations.

At this point we can use the methodology to adapt RPCS. The transformed structure of RPCS is portrayed in Fig. 5.4, and represents the steps defined in Sec. 4.2:

Key Selection. As introduced before, the temporal domain of RPCS can be easily manipulated to distribute the problem. Consequently, the independent variable k will represent each of the time steps t_i to be simulated.

Domain Distribution. For each time step we need to conduct the usual RPCS simulation, as if the overall domain comprised a single instant. To achieve this, we need the information of the infrastructure, additional global parameters, and the circulation profiles of all the trains operating at instant t_i. Shared data can be replicated to each node for local access,

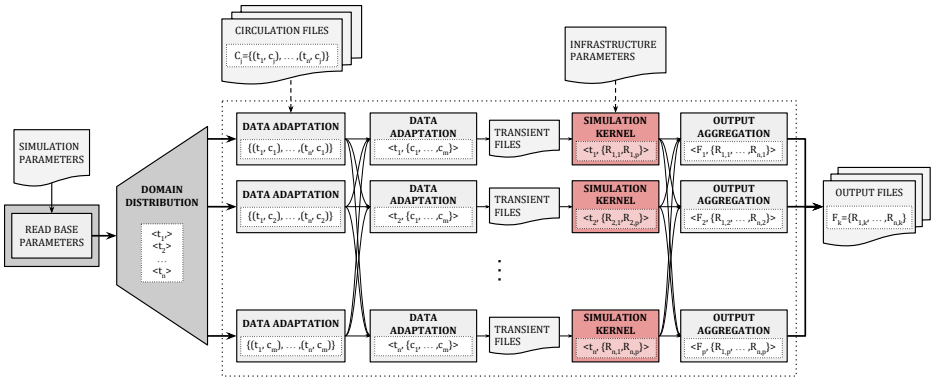


FIGURE 5.4: Structure of RPCS after applying the data-centric transformation methodology.

but circulation data needs to be rearranged, making a data adaptation phase necessary.

Data Adaptation. RPCS handles a circulation file per train in the railway system. Each file, C_j, contains the profile of a train with relevant information for the simulation at each instant t_i. The data adaptation phase rearranges this data to aggregate the circulation data of all trains operating at a given t_i.

Simulation. The previous output is used as input to the simulation stage. Then, the data for the instant being processed is passed to the electric algorithm itself along with the scenario information obtained from the infrastructure file that is also incorporated at this point. The output of the simulation represents different metrics, so for each of them the simulation will generate a specific record $R_{i,k}$.

Partial Evaluation. RPCS does not present a model update region. However, the resulting data needs to be aggregated to recreate each file F_k, conducting the opposite process of the data adaptation step.

Output Analysis. No analysis is required, the output is directly merged as in the original application, in which each output file contains the results for the whole temporal interval of the simulation.

5.1.3 Implementation on a BDA Platform

The previous design could be implemented in any of the available map-reduce frameworks. Among them, we selected Apache Hadoop 2.2.0 platform [22] given its popularity and community support. Its distributed file system is a great addition to the framework, since it allows automatic load balance. Moreover, it includes a distributed cache that supports auxiliary read-only file storage for tasks among all nodes, which suits neatly the needs of the shared infrastructure parameter file. Besides the former technical features, Hadoop has been adopted into many cloud environments resulting in reduced costs given its parallelism exploitation capabilities [163], which will help us scale the problem to a distributed environment.

We implemented this design via Hadoop Pipes API, since the original code was written in C++ and we wanted to maximise code reuse. Despite Pipes does not allow taking full advantage of Hadoop's potential given its limited functionality, it provided all the necessary tools to execute our framework, including *map* and *reduce* interfaces, basic data type support, and distributed cache access on job submission.

5.1.4 Evaluation

In order to asses the performance of the application, we compared its execution times on a cluster node against the original multi-thread implementation. In addition, we assessed the scalability of the distributed data-centric application on the public Amazon Elastic Compute Cloud (EC2). The following paragraphs describe the utilised resources and the obtained outcome.

Experimental Setup

First, we tested the performance of the original multi-thread application on a cluster node consisting of a 48 Xeon E7 cores and 110GB of RAM. This node was also used to test the resulting transformed application to avoid variations that may arise from heterogeneous configuration, resource differences, or network latency in case of the map-reduce application [117]. This isolation favours the multi-thread application, which is especially designed to perform in standalone environments. However, it allows to focus on the actual limiting factors that may affect scalability in large test cases like I/O, memory consumption and CPU usage.

Then, the Hadoop application was deployed on EC2 to assess scalability. The selected cloud infrastructure consisted of a general purpose *m1.medium* node as dedicated master and several memory optimised *m2.xlarge* machines as workers. Table 5.1 shows the main aspects of the selected instances, which were selected in order to maximise the number of cores for the data adaptation stage, while holding enough memory to execute as many containers as cores with sufficient memory for the simulation stage. The scalability tests varied the number of workers in order to check if scalability issues arise as the number of nodes increases.

Four test cases were considered with variations on the circuit size, simulation's initial and final time and, consequently, input data volume, execution time, and memory consumption. A description of these simulations is provided in Tab. 5.2. Cases I and II should not yield any significant load, yet simulation III is expected to reflect the system's behaviour under average problems. The biggest experiment, case IV, should reveal the platforms' actual limitations as simulations become larger, if any. These tests are meant to indicate the performance of the data-centric adaptation versus the original application under an increasing amount of input data and simulation time. All test cases are based on the same real case, a particular railway line at Madrid surroundings, with increasingly levels of detail and simulation periods. This line has

TABLE 5.1: EC2 instances used in the evaluation of RPCS.

Type	Role	Virtual CPUs	Memory (GB)	Local storage (GB)
m1.medium	master	1	3.75	410
m2.xlarge	worker	2	17.1	420

TABLE 5.2: Definition of test cases for RPCS.

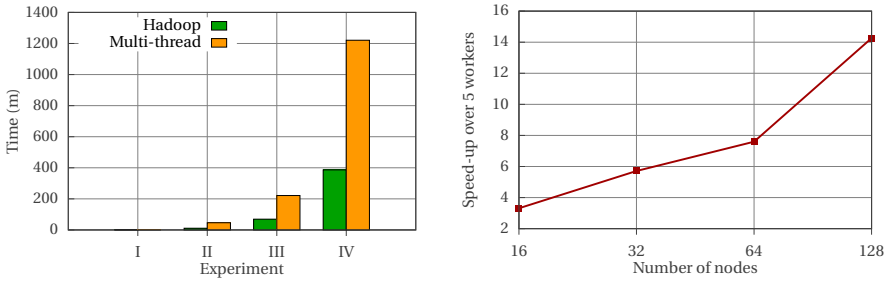
Experiment	Average elements per instant	Simulated time (h)	Input size (MB)
I	77	1	1.7
II	179	33	170
III	525	177	1228.8
IV	755	224	5324.8

been used before in other works [164] because it is a good example in size and complexity of a real railway project.

We will now analyse whether the transformed application behaves as expected in relation to performance and scalability by examining its execution times on two different environments. Figure 5.5 shows these results.

Programming Models: Hadoop vs. Multi-Threading

In Figure 5.5 (a) we observe the overall execution time for the transformed application including the map-reduce stages and input data upload. The latter has to be considered given that replication and balance must be achieved by the platform to distribute load evenly. The graph indicates that the performance obtained with map-reduce on Hadoop is remarkably better than the original multi-thread application, in particular, 68% less total simulation time for the largest experiment. The shared memory simulator's results might be caused by the bottleneck constituted by the physical memory and the disk. The latter is particularly critical, as all threads write their results to disk while they perform their computations in the original simulator.



(a) Execution time.

(b) Speed-up on EC2.

FIGURE 5.5: Evaluation results for the multi-threading and Hadoop implementations of RPCS.

The smallest experiment is an interesting exception, with execution times ten times greater than the original application. This reflects how the map-reduce framework's overhead significantly affects the time taken to complete such a small simulation compared to the original application benchmark.

Scaling-out to a Public Cloud

In Fig. 5.5 (b) we observe the speed-up obtained on EC2 when the number of workers is increased. The speed-up shown in the figure is related to the execution times obtained on a five-worker cluster as baseline, because these nodes represent similar resources than the local cluster node. As the figure indicates, increasing the number of workers decreases the total simulation time. However, the performance does not scale up linearly with the number of nodes: while with 16 nodes the speed-up is 3.3, with 64 nodes it is only 7.6. The reason behind this result is that the problem size becomes small for the cluster size as more nodes are added. Hence, less data is assigned to each worker and some resources become underutilised. Moreover, as we mentioned in the previous paragraph, in very small experiments the measured execution time is mostly spent in the platform's task preparation and scheduling, and not in the actual simulation. This results in degraded performance due to platform overhead. Therefore, it is necessary to increase the problem size as well as the number of worker nodes in order to achieve linear scalability.

5.2 The EnKF-HGS Hydrogeologic Data Assimilation Workflow

Having good quality predictions of the behaviour of hydrological environmental systems is key for water management. Such systems rely on complex multi-scale non-linear processes and matrix operations. The inherent computational complexity of these tasks have lead scientists to implement parallel versions of these models in the form of tailored simulators for multi-core environments. Some communities have also used grid-like HTC technologies to increase the scale, size and complexity of the addressable problems. Nevertheless, the ever-increasing datasets have shifted the interest towards more data-intensive infrastructures, like compute clouds, looking for flexibility, elasticity, and a satisfactory cost-performance trade-off.

Studies have shown the feasibility of running BDA-based frameworks for multi-scale data analysis, with complexities comparable to the hydrology domain [165, 166]. These works have shown that data and tool integration are easier for end-users in these BDA environments, while performance and storage capability remained comparable to grids. Other works approached the benefits of BDA infrastructures from a hybrid perspective, integrating data and computing infrastructures from grids with external cloud providers [167]. This work is particularly relevant, as it shows the feasibility of clouds for a wide range of hydrological problems, covering both computationally intensive HPC simulators, and MTC-like applications with multiple scenarios.

Recent technological and mathematical advances allowed to improve significantly the precision of the simulations by integrating data acquisition techniques with the modelling process [168]. HydroCloud [169] allows to aggregate data from different sources and present it to the user in a single format for further analysis, by using a cloud-based data integration system to store and explore data. A similar research line is followed by the team

involved in the development of EnKF-HGS, since they propose an architecture for a system combining a wireless environmental monitoring module as data source, and a cloud-based computing service to perform environmental simulations [170]. Even though the system was tested in a real-world deployment in the Emmental area in Switzerland, and proven to be operable, the performance of the core simulation procedures have not been built for the target infrastructure. This motivates the need to apply our transformation methodology, which will assist in the process of developing a version of the simulator able to exploit BDA features like cloud and streaming support with minimal development.

Using a classification of common scientific workflow patterns [94], the EnKF-HGS workflow represents a combination of four basic patterns: data-parallel, single input, and iteration patterns are used in the first phase of the hydrological workflow, while the multi-stage pattern is used in the second phase. These features make EnKF-HGS a representative use case of many applications currently found in scientific computing [171–174] that can be solved by the Monte Carlo method. Even though the simulators directed at solving these problems vary but they all share the same structure, which involves a large random sampling to determine the properties of some phenomenon or system behaviour.

Applications like EnKF-HGS perform reasonably well in regular HPC environments due to the availability of a high performance network storage and low-latency broadband network connection. However, when moved to a BDA environment, as required by this application, performance might drop drastically. In order to be able to benefit from the advantages of these BDA settings, while maintaining the performance at an acceptable level, we show in the following sections how this application could be shifted to a data-centric paradigm using the transformation methodology to guide its implementation.

5.2.1 Simulator Description

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, with an optional real-time control of water resource systems through a feedback mechanism.

The core of the simulator is the data assimilation process, which allows to incorporate observations of an actual environmental system into a numerical model of that system. This process allows to continuously improve the simulated model and to minimise the deviation of the model from the state of the actual system. In EnKF-HGS, data assimilation is implemented via the ensemble Kalman filter technique [162, 175], which approximates the uncertainty of model prediction through the forward simulation of an ensemble of model instantiations. These instantiations are called *realisations* in this application.

Each simulation in the ensemble of realisations represents a long-running I/O- and compute-intensive process, which comprises the sequential execution of two proprietary simulation kernels: HydroGeoSphere (HGS) and GROK [176]. Each model realisation represents an instantiation of the numerical model provided with a different combination of input parameters and system conditions. In our case the numerical model is the integrated hydrological modelling software HGS. HGS is able to perform dynamic stochastic simulations between water profiles composed of numerous elements, like the ones depicted in Fig. 5.6. It has been successfully used to simulate many complex problems involving surface water, groundwater and vegetation processes [177, 178]. The model is designed to take into account all key components of the hydrologic cycle using a rigorous conceptualisation of the hydrologic system [179, 180]. Based on these parameters, HGS allows the simulation of a wide range of physical characteristics and hydrological objects, such as wells, tile drains, and thermal energy transport. HGS mainly relies on the CPU to

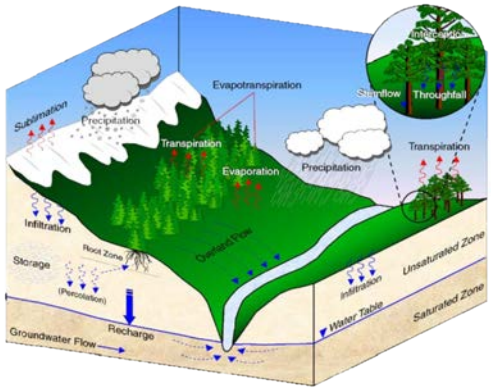


FIGURE 5.6: Typical surface water and groundwater processes in a pre-alpine type of valleys [181].

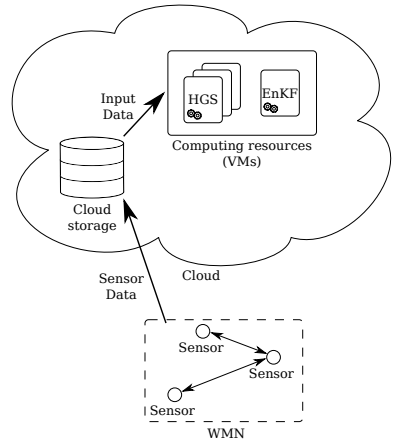


FIGURE 5.7: Architecture of the real-time environmental monitoring and hydrological modelling system depicted in [170].

solve complex differential equations and matrix operations, which makes it CPU-intensive. GROK is a preprocessor that prepares the input files for HGS, which makes GROK an I/O intensive application.

The ensemble Kalman filter is an implementation of the Monte Carlo method that consist of two distinct steps: the forward propagation of the ensemble of model realisations (i.e. forward propagation phase), and an update of the simulated model state with the measurements (i.e. filtering phase). The forward propagation phase comprises a large pool of independent model realisations, which introduces a tremendous demand for computing power, especially if combined with a complex numerical model such as HGS. The filtering phase is a relatively short-lasting but tightly-coupled process that performs a set of matrix operations and requires multiple data synchronisation points. Consequently, the first stage constitutes the parallel region of the application, and the second represents its model update region. On top of that, the iterative nature of the data assimilation process imposes that the two phases have to be repeated continuously, thus shifting the demand even further.

Even though this method results in higher quality model predictions than the conventional simulation methods, the high resource demand of the method remains an unsolved problem for many environmental scientists. The required amount of computations implies having a dedicated HPC infrastructure, which is not always available to the end-users. Moreover, data acquisition, integration and storage follows the scheme in Fig. 5.7, thus incorporates BDA elements like data streaming from sensors and cloud storage. Therefore, transforming EnKF-HGS to a data-centric application would not just help hydrogeologists to increase their addressable problem size without a heavy infrastructure investment, but would also ease its integration with this BDA ecosystem.

EnKF-HGS is originally implemented using MPI, which makes it a perfect representative of a complex and compute-intensive scientific application built on a process-centric runtime. In this case, MPI is used to distribute the simulations of the ensemble members over available CPUs, as each individual member represents a completely self-contained instantiation of the model. Figure 5.8 shows a simplified execution model of the MPI implementation of EnKF-HGS. The procedures executed in the MPI root process are identified using Roman numerals (from I to III), while numbers (1 to 3) refer to tasks that are computed distributively in MPI processes. There are three main stages in the workflow that comprise ensemble preparation, the iterative process of simulating the ensemble, and the final treatment of the results after the simulation. These stages are matched with the procedures in the figure as follows:

Ensemble Preparation Stage

I. Initialisation

At the beginning of the workflow, the root MPI process initialises global data structures and reads the provided model parameters from the input files.

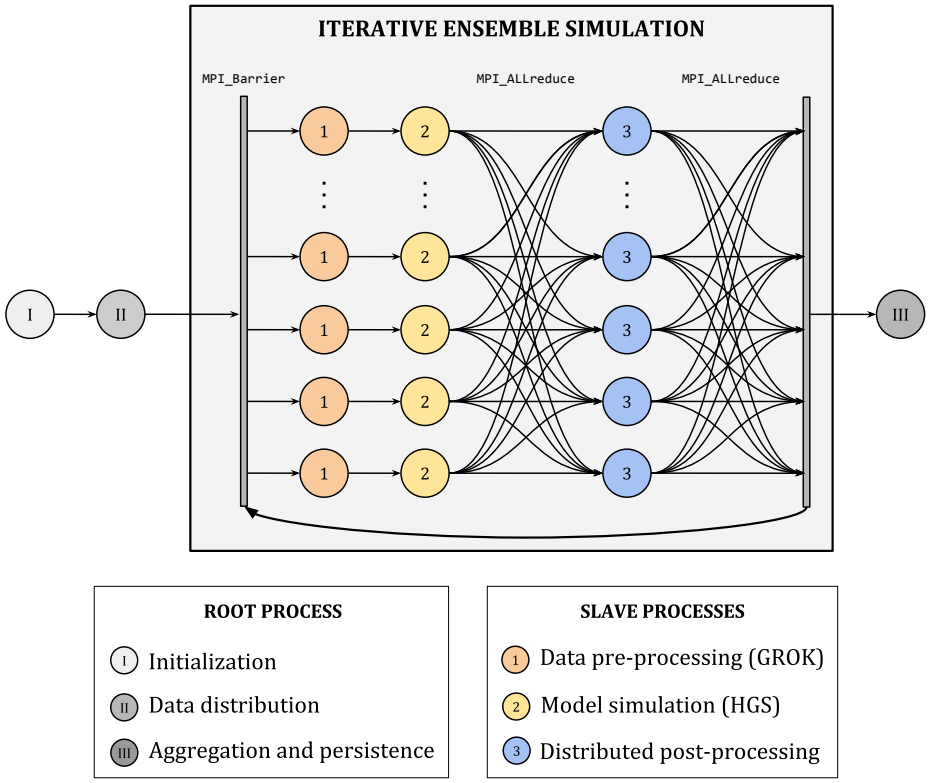


FIGURE 5.8: Original workflow of the MPI implementation of EnKF-HGS.

II. Data distribution

According to the initial model parameters, the root MPI process generates input files for each model realisation and stores the files in separate directories on the network storage.

Iterative Ensemble Simulation

1. Data pre-processing (GROK)

After all running MPI processes reach the first synchronisation point (an MPI barrier), each process execute the preprocessor GROK on the corresponding input directory in order to generate HGS-specific input files that will be written in the network file system.

2. **Model simulation (HGS)**

   HGS reads the output of GROK, runs the model realisation and writes the output to the network storage. At this point, all MPI processes synchronise for the second time using an MPI all-reduce directive, since further data updates require simulation results of all model realisations. This is a very compute-intensive and memory-consuming stage.

3. **Distributed post-processing**

   During the data update process, each model realisation is optimally weighted and updated with the most recent field measurements in order to reduce the simulation error. Each process is in charge of updating its block, and results are afterwards merged through another MPI all-reduce call.

Output Management

III. **Aggregation and persistence**

   After all iterations are completed and all MPI processes reach the barrier, the root MPI process aggregates the model simulation data from the realisation directories and updates the global data structures. Before the program terminates, the root MPI process writes the model simulation results to the output files.

In this description, stages (1) and (2) represent the parallel region of the application, and stage (3) constitutes the model update region. Finally, the repetition of stages (1), (2) and (3) shows the iterative structure of EnKF-HGS.

5.2.2 Application of the Methodology

As described, the original application consisted of an MPI implementation of an ensemble Kalman filter, which relied on two legacy binaries to execute the

simulation (GROK and HGS). EnKF-HGS operates with a set of realisations, which constitute independent instantiations of the model with different parameters, meeting the need for the existence of an independent variable in the application.

In terms of input, there are several files that describe the parameters of the hydrogeological model for the iterative process and the GROK kernel. All of these files are used by the ensemble as a whole, regardless of the realisation. On the other hand, HGS requires specific input for each realisation r_i, which can be found in two two-dimensional matrices M_1 and M_2 as columns $c_{1,i}$ and $c_{2,i}$, respectively. This data needs to be distributed to each simulation.

The transformation EnKF-HGS is shown in Fig. 5.9, and represents the following steps:

Key Selection. We selected the set of realisations as parallelisable domain. Hence, the realisation identifier, r_i, constitutes the key, and the collection of the data and parameters per realisation of the model, $c_{i,1}, c_{i,2}$, is the value.

Domain Distribution. The former key-value pairs are created after the main matrices M_1 and M_2 are generated, and are distributed afterwards.

Data Adaptation. Not required in this case, since realisation data is already shaped as needed.

Simulation. Realisations are executed independently across the nodes, as pipelines of the GROK and HGS kernels. The result of the HGS simulation corresponds to the updated columns for r_i, $\hat{c}_{1,i}$ and $\hat{c}_{2,i}$.

Partial Evaluation. All data needs to be gathered as it is generated by the simulation, thus this step is not needed.

Output Analysis. The simulation results are finally gathered to recreate the matrices for the model update region, which will result in new data for the following iteration.

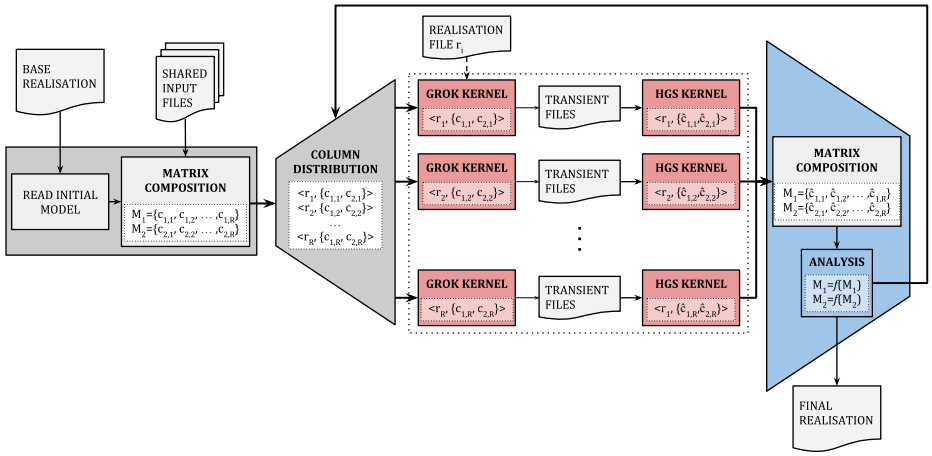


FIGURE 5.9: Structure of EnKF-HGS after applying the data-centric transformation methodology.

5.2.3 Implementation on a BDA Platform

Since the resulting application is iterative, we implemented it using the Apache Spark platform, which is currently a major representative of the BDA ecosystem [182], and similar in performance to other platforms [183].

Spark reuses a working set of data, known as resilient distributed dataset (RDD), through multiple parallel operations, built around an acyclic data flow model. It retains, however, the scalability, fault tolerance of map-reduce and its relevant data-locality features, in particular for MTC [184].

A particularity of the kernel binaries is that they are third-party pre-built black boxes. An effect of this is that they rely on hard-coded input paths for the intermediate files they handle. Nevertheless, in order to improve locality, our methodology approaches the simulation stages as a pipeline, so we can ensure the execution of HGS with the input generated by its corresponding execution of GROK. To achieve this we exploited Spark's partitioning mechanisms to ensure that each full realisation is computed in the same node in a pipelined manner.

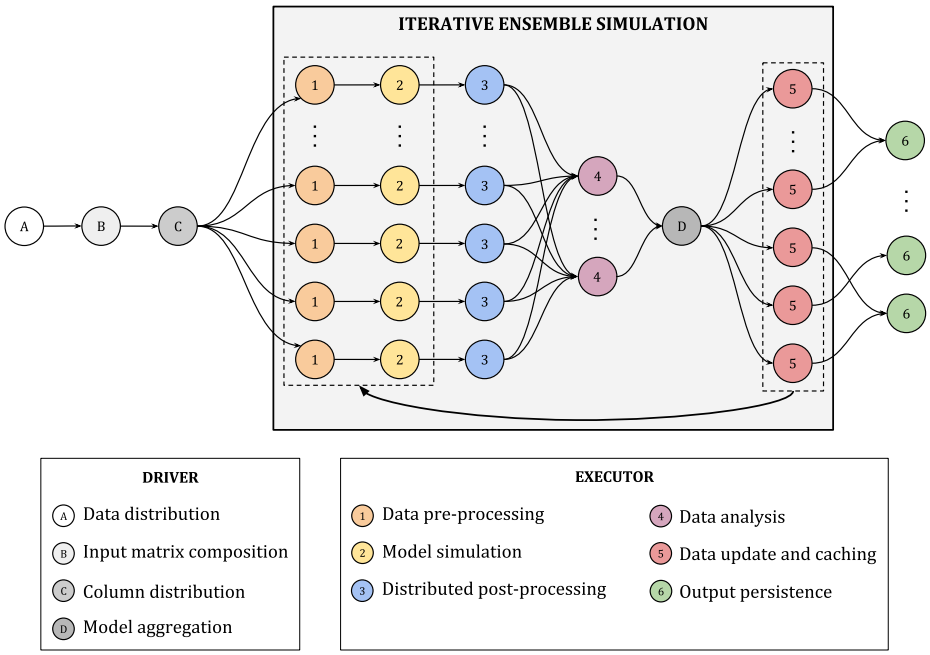


FIGURE 5.10: Final workflow of the transformed EnKF-HGS application, matching stages 1, 2 and 3 with Fig. 5.8. Dashed lines indicate that the iteration is executed over a distributed dataset.

Figure 5.10 depicts the stages that belong to the final implementation of the workflow in Spark. The procedures executed in the Spark driver process are identified using letters (from A to D), while numbers (1 to 6) refer to tasks that are computed distributively in the Spark executors. Similarly as in Fig. 5.8, the most relevant design and implementation details are described in the following paragraphs, along with the association of these procedures to the core stages in the workflow.

Ensemble Preparation Stage

A. Data distribution

The first step is to load the necessary auxiliary files that every executor will need to properly run its data partition. This includes, for instance,

the kernel binaries. Spark guarantees that these files will be available for the worker nodes in their current working directory.

B. Input matrix composition

Input data are read in the driver process in order to initialise the base model, composed of two main matrices, M_1 and M_2, in which each column $c_{1,i}$ and $c_{2,i}$ corresponds to an instantiation, r_i, of the model. Additional data structures are created and initialised, and the parameters of the simulation are obtained.

C. Column distribution

Both matrices are distributed by columns in order to build the realisation set, R. Each realisation r_i is composed of the corresponding columns from both matrices, $c_{1,i}$ and $c_{2,i}$. Figure 5.11 illustrates the realisation data distribution process, which yields the distributed dataset that will be transformed in the following stages and iterations. Additionally, we forced each partition to hold the data for a single realisation in order to induce fine-grained parallelism, as each task will only handle one realisation.

Iterative Ensemble Simulation

1. **Data pre-processing (GROK)**

   After realisations are distributed, the GROK kernel writes the realisation input data to a local file. HGS will read the realisations from these file in order to conduct the simulation of the model.

2. **Model simulation (HGS)**

   With the input files from GROK, HGS simulates the model and writes its output for subsequent analysis. In steps 1 and 2 we must ensure that both binaries will be executed in the same node to exploit data locality. To achieve this, we run GROK and HGS in the same map function, which

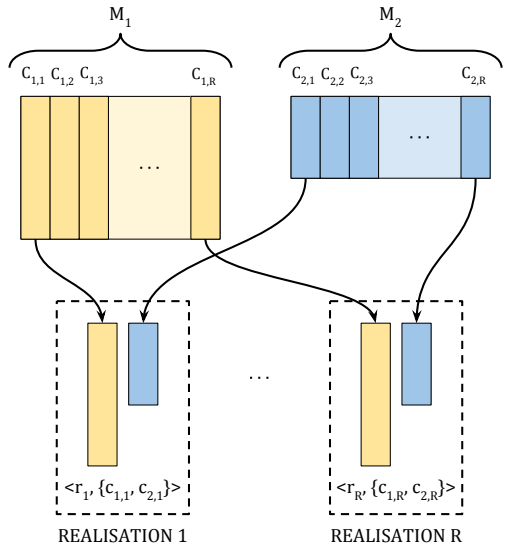


FIGURE 5.11: EnKF-HGS column distribution procedure. Both matrices are split column-wise, and realisations are built with the corresponding column from both matrices.

is an indivisible task in Spark. They thus act as an inner pipeline within the workflow.

3. Distributed post-processing

The post-processing stage is partially distributed. First, the output from each HGS execution is read in each executor in order to create an updated realisation set, R'. With this information we create auxiliary distributed matrices and conduct several distributed operations to avoid gathering the whole matrix in the driver.

4. Data analysis

Further operations with auxiliary matrices are executed in the driver in order to filter and randomise the input for the following iteration. The goal of this stage is to minimise the size of the dataset that needs to be collected in the driver prior the model update. Note that to achieve this, significant data shuffles must be executed.

D. Data aggregation

Unlike in the MPI design, not every stage of the analysis could be distributed in this implementation. There is a step in which we aggregate a final matrix that will be used to compute an update matrix.

5. Data update and caching

The update matrix is used to update every realisation. The resulting realisation set is persisted to the local storage of the nodes as a fault-tolerance mechanism, and the following iteration starts.

Output Management

6. Output persistence

After every iteration is executed, the output is stored to HDFS. Unlike the MPI version of the workflow, the new solution can execute I/O in parallel, as every partition is stored independently. Model aggregation can be conducted off-line if needed.

The following section presents a performance evaluation of the resulting application against its original implementation in order to analyse even further which elements must be taken into account to design a generalist architecture for HPC-BDA runtime convergence.

5.2.4 Evaluation

This evaluation focuses on absolute execution time and speed-up to analyse the effects of overheads in BDA environments. These may include communication overhead and increased memory consumption, given the deeper software stack and heavier resource usage of these platforms, and higher execution time if we consider the execution of the application in a BDA environment like a virtualised cloud setting.

TABLE 5.3: Technical specifications of the local cluster (testbed A) and the private cloud (testbed B) for the evaluation of EnKF-HGS.

Infrastructure	Cluster	OpenNebula
CPU	2 x Intel Xeon E5405 @2.00GHz	2 x Intel Xeon L5420 @2.50GHz
Total cores	8	8
Memory	8GB	8GB
OS	Linux Ubuntu 14.04.1 LTS	Linux Ubuntu LTS 14.0.4
Storage	2 x HD 1000GB + GlusterFS 3.6.9	HD 500GB
Network	1Gb/s Ethernet	1Gb/s Ethernet

Experimental Setup

In order to assess the performance and scalability of the application, we selected three different execution infrastructures: a cluster, a private cloud running OpenNebula, and a virtual cluster on the Amazon EC2 public cloud. The specifications and limitations of these testbeds are described as follows.

Testbed A: Local Cluster. This infrastructure comprised 11 worker nodes, with the specifications shown in Tab. 5.3. Each worker node holds 8GB of RAM and two Intel Xeon E5405 @2.00GHz processors, with four cores each. In addition, an auxiliary node was necessary to host the driver process of the Spark implementation, which required 7GB in the largest experiment we conducted. This means that the container in charge of running the driver would require 7GB plus a 10% memory overhead (as configured by default in the platform), 512MB extra memory for heap space, and other overhead sources like serialisation buffers. Since Spark adds significant memory overhead to drivers and executors, we had to add a larger node to bypass the memory constraints in the worker nodes. As a consequence, we added a node with an overall amount of 94GB of RAM and four Intel Xeon E7-4807 @1.87GHz processors with six cores each. The local cluster had already a pre-installed network file

system GlusterFS (a scalable and production-ready network file system, which was necessary for the MPI implementation execution.

Testbed B: Private Cloud. We relied on a virtual cluster running OpenNebula with the hardware described in Tab. 5.3. Notice that the main difference between this infrastructure and the cluster is the clock speed, which would benefit this testbed in the evaluation. To build the virtual cluster, we spawned 32 8-core virtual machines (VMs) with 7.5GB of RAM each, the maximum available memory per VM. Notice that this memory limitation is relevant, as there was no workaround to fit the driver safely in the largest experiments. For the network file system, we deployed the latest version of GlusterFS on three additional storage nodes, which were organised in a distributed volume with no data-replication in order to maximise the storage performance. On the computing nodes side, we exploited the FUSE-based Gluster Native Client for highly concurrent access to the file system.

Testbed C: Public Cloud. We selected *c4.2xlarge* instances from the Amazon EC2 catalogue to act as workers due to their balance between number of cores and amount of memory, and a *r3.xlarge* instance with larger memory to hold the driver. The virtual cluster for Spark was composed of 16 workers (128 cores in total), an *m3.medium* master, and the additional dedicated VM for the driver. MPI ran on a virtual cluster built with identical workers, plus three additional *c4.large* storage nodes running GlusterFS with the configuration similar to the previous testbed. Each storage node was provisioned with an 5GB general purpose SSD brick. Table 5.4 shows a summary of the selected instances and their assigned roles in both the Spark and MPI execution platforms. In addition, Tab. 5.5 describes the hardware characteristics published by the provider[2]. Given its public nature, this testbed could yield the largest differences

[2]Retrieved from https://aws.amazon.com/ec2/instance-types/

TABLE 5.4: Amazon EC2 instance selection for the virtual clusters running EnKF-HGS on testbed C.

Platform	**Spark**			**MPI**	
Node role	**master**	**driver**	**worker**	**compute**	**storage**
Type	m3.medium	r3.xlarge	c4.2xlarge	c4.2xlarge	c4.large
Amount	1	1	16	16	3

TABLE 5.5: Technical specifications of the selected public cloud instances for testbed C, as provided by the Amazon EC2 documentation.

Type	**Processor**(*)	**vCPU**	**Memory (GiB)**	**Storage (GB)**	**Network Performance**
m3.medium	Intel Xeon E5-2670 v2 @2.5GHz Intel Xeon E5-2670 @2.6GHz	1	3.75	SSD	Moderate
r3.xlarge	Intel Xeon E5-2670 v2 @2.5GHz	4	30.5	SSD	Moderate
c4.2xlarge	Intel Xeon E5-2666 v3 @2.9GHz	8	15	EBS	High
c4.large	Intel Xeon E5-2666 v3 @2.9GHz	2	3.75	EBS	Moderate

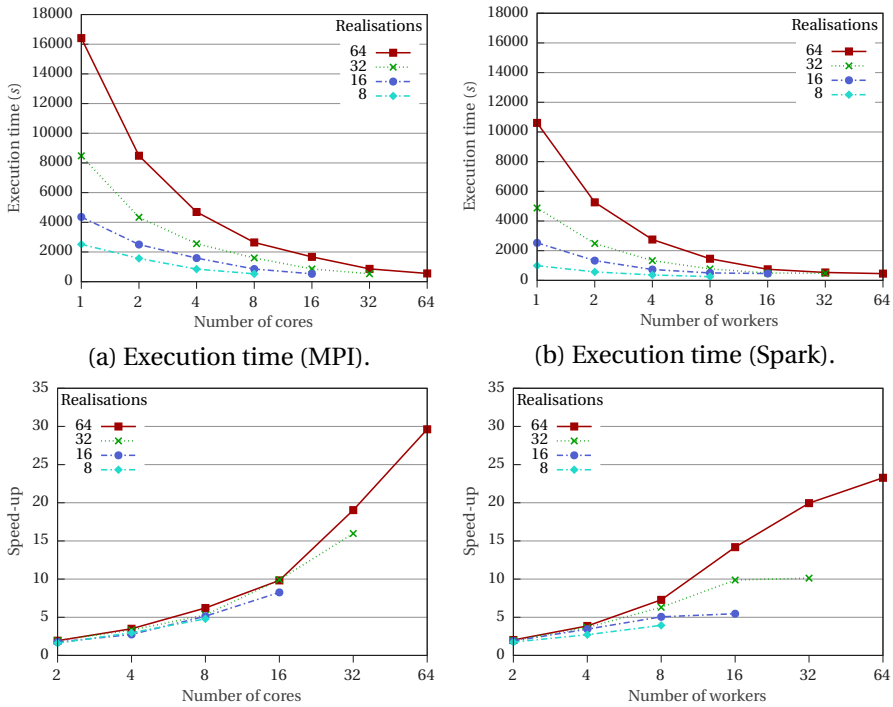
(*) More than one item is listed if VMs can be indistinctively launched on different physical processors.

in performance among independent executions. To assess this possibility, we conducted five runs of both applications for a small example comprising eight realisations and eight cores, and detected that the standard deviation is never higher than 5% of the average execution time.

Programming Models: Spark vs. MPI

The objective of these experiments is to detect the effects the execution models have on the performance of the workflow execution. We analysed the MPI implementation of EnKF-HGS, and its data-centric version built in Spark, both running in the local cluster formerly described.

We allocated Spark executors with one core in order to fairly compare scalability against single-core MPI processes. We experimented with increasing



(a) Execution time (MPI).

(b) Execution time (Spark).

(c) Speed-up (MPI).

(d) Speed-up (Spark).

FIGURE 5.12: Execution time and speed-up for the MPI and Spark implementations running on a local cluster.

realisation volumes and executor number, we measured the absolute execution time for a single execution (including the job launch time required by Spark), and we computed the speed-ups. The results for this execution on a local cluster are shown in Fig. 5.12, in which (a) and (c) correspond to MPI, and (b) and (d) correspond to Spark. Remarkably, Spark yields better execution times for every experiment, and its speed-ups are better the larger is the experiment for a given number of workers. This might be a result of the redesign process. However, the speed-up in Spark for the largest experiment (i.e. 64 realisations on 64 executors) is lower than in the MPI case. The problem in this case is that the 64 executors cannot be scheduled at once due to their large memory requirements.

The main conclusion from this experiment is that, while the BDA-inspired

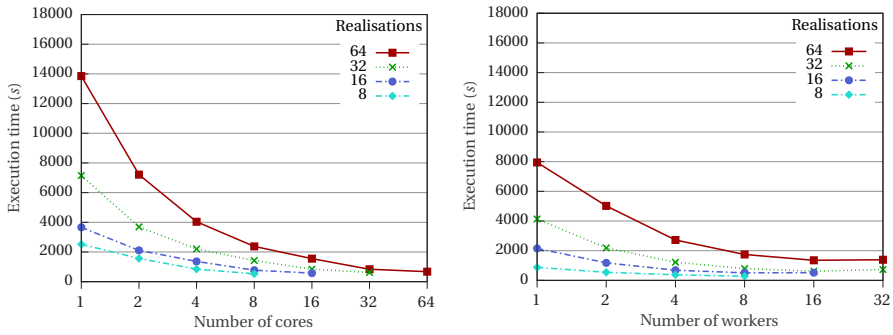
approach shows positive performance results, the memory overhead of the execution framework hurts scalability, as less parallel executors can be allocated in the same infrastructure. As a result, the slimmer MPI processes seem more suitable for large scale execution of this workflow. Another interesting aspect is related to the post-processing stage of the workflow and its effect on the overall execution time. At some point, with the growing number of the parallel executors, the post-processing computation becomes shorter in time than the data transferring time. As a result, the post-processing stage starts to affect the overall execution time more than with a fewer number of the parallel executors.

Computing Models: Cluster vs. Private Cloud

After analysing the programming models, we focused on the underlying computing models to assess their impact in performance. Hence, we conducted further experiments on the OpenNebula private cloud with the Spark and MPI implementations.

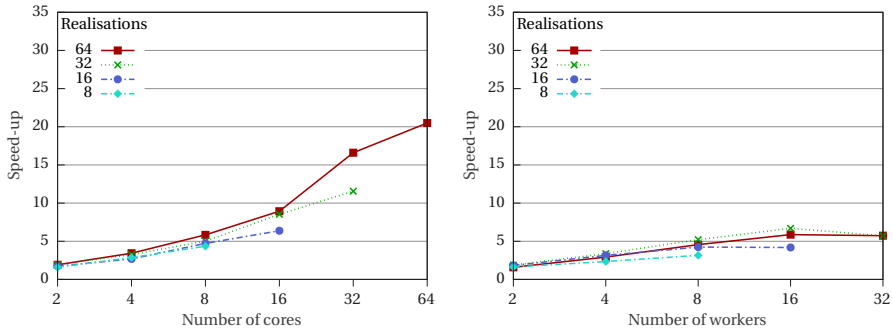
Figure 5.13 reflects analogous evaluation metrics, but for the experiments conducted in OpenNebula. Regarding the results for Spark, which is the approach that should benefit the most from BDA-oriented environments like clouds, while the overall evolution of speed-up seems similar in relative terms, we can clearly see that in OpenNebula the results are much more extreme, with even lower execution times, but also smaller speed-ups. Interestingly, MPI also shows a similar behaviour, with lower execution times and degraded speed-ups. These results could be related to the larger frequency in the physical processors of the virtual cluster, which yields faster executions, and the lower memory per core ratio, which hurts scalability.

With respect to the results for Spark, there are two remarkable exceptions. The first one is the 64 realisation execution with 64 workers, which failed as the container corresponding to the driver violates the system's memory limit, hence this result is not included. The other one is that the result for



(a) Execution time (MPI).

(b) Execution time (Spark).



(c) Speed-up (MPI).

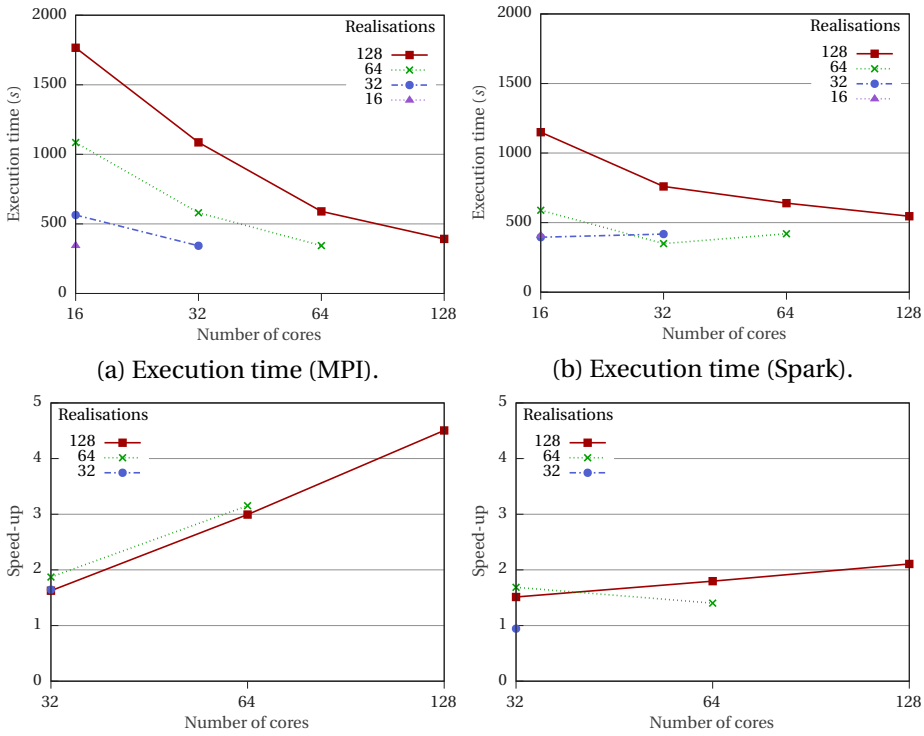
(d) Speed-up (Spark).

FIGURE 5.13: Execution time and speed-up for the MPI and Spark implementations running on a private OpenNebula cloud.

32 realisations with 32 workers shows the same result as the 16-worker one. We noticed that this is due to the lack of resources to launch a new container once the system is fully running tasks, which means the platform has to wait for an executor to finish, and then launch the former tasks. Considering that there is a synchronisation point after the concurrent computation of the realisations, the result is that the execution time is doubled, thus matching the time obtained for 16 executors.

Scaling-out to a Public Cloud

Given the resource limitations in our private infrastructures, we moved both execution paradigms to the Amazon EC2 public cloud to test further scaling. We incrementally increased the size of the virtual cluster and conducted



(a) Execution time (MPI).

(b) Execution time (Spark).

(c) Speed-up (MPI).

(d) Speed-up (Spark).

FIGURE 5.14: Execution time and speed-up for the MPI and Spark implementations running on a virtual cluster on the Amazon EC2 cloud.

experiments until one of the implementation showed significant scalability issues that made further increases in problem size or number of workers infeasible. Our goal was to determine which paradigm showed the most limitations to support large-scale executions.

We considered the set-up described in Sec. 5.2.4 enough to run 64 realisations smoothly in both MPI and Spark, and we attempted 128 realisations to check whether we could exploit all the cores in the system with the Spark implementation.

Figure 5.14 shows results coherent with the fast speed-up degradation shown in OpenNebula, since driver and executor memory sizes increase to the point in which all the realisations cannot be computed in parallel (i.e. memory per

core increased beyond 2GB). The major problem is not performance degradation, however, but the possibility to not being able to run the experiment: a job simply will not be launched if memory is not enough to host the driver or executor container, or their related overheads.

The former experiments, clearly indicate that the Spark implementation will not scale due to its memory requirements when running on the Spark stack. This is also problematic if cost is taken into consideration, as it would be required to select machines with larger memory per core, thus more expensive. On the other hand, the MPI-based workflow does not show an outstanding performance nor speed-up, but it is able to scale further with less resources, and in a stabler manner.

5.3 Discussion

The former evaluations show that BDA platforms –like Hadoop and Spark– and infrastructures –such as clouds– could be used to improve the efficiency and scalability of some types of scientific applications with minor modifications. More specifically, we detected that simulators relying on parameter-sweep and partitionable domains, and kernel-based workflows comprising many, loosely-coupled tasks could greatly benefit from the massive parallelism of BDA paradigms. This is the case for RPCS. Nevertheless, these results also indicate that these techniques are still far from disrupting the vast set of Monte Carlo workflows, in particular as the scale increases, as seen for EnKF-HGS.

We used Spark, a framework specially built for iterative processes, yet the iterative workflow we implemented was not able to scale properly, mainly due to memory overhead and bottlenecks introduced by the execution environment, as present in other Java-based map-reduce platforms [32] and Spark itself [35]. The deep component stack and its dependence on the JVM yield a significant memory consumption that also affects execution time due to

frequent garbage collection operations [33] and serialisation if bindings to other languages are used [34].

Another observation is the rough fitness of the simulation ensemble paradigm to the target use cases of the language, which yielded suboptimal performance as it is not possible to attain an implementation that exploits the full potential of both the Spark platform and the ensemble algorithm. This is mainly due to the limitations on matrix operations and data collection stages. However, the scalability issues we found in Spark for this use case are still valid, since they are tightly related to its overhead in the driver process.

There is another major problem we found with BDA-oriented frameworks, which is the problems they are built to solve. Some BDA-oriented frameworks show drawbacks in terms of generality and versatility. For example, our major development issue was dealing with operations involving the large matrices needed in the post-processing stage of the workflow. Although there are libraries to handle distributed matrices, like Spark's MLLib, the functionality is limited to the common operations needed for data analysis. Hence, we had to create ad-hoc workarounds to implement some matrix-dependant sections of the workflow, which degraded our development experience. Further parallelisations on the post-processing stage could be feasible, but they would require a full re-engineering of the workflow to the limitations on minimise the matrix operation limitations and to reduce the size of collected data. All of these aspects could hurt the behaviour of the BDA-based implementation against the solution built for MPI.

Despite the former, we experienced that building data-centric solutions for a BDA environment using Spark and Hadoop, and a consolidated cloud like Amazon EC2, definitely reduced development time given the abstract nature of data and objects in analytics. We were able to tailor the infrastructure as desired, without worrying about other users or software compatibility issues. We only had to consider the design of the workflow after applying the data-centric methodology, and some specific implementation particularities of

the language and the available functionalities. Moreover, we found that I/O management could become a bottleneck, in particular for RPCS, and being able to distribute data and access it locally was key for achieving a competitive workflow.

Another main benefit of having such BDA execution engines was their underlying resource manager and distributed file system, which had a major impact in easing data distribution and task management. However, it was necessary to conduct a very time-consuming tailoring of the configuration given its outstanding impact in the final performance and stability [185].

These observations provide further support for the motivation of having a HPC-BDA-capable platform that bridges the gap between process- and data-centric applications, since shifting HPC applications to a data-centric model is not sufficient to attain full convergence. Such platform must be based on a architecture that could blend the generalist nature and efficiency of process-centric runtimes, with the ability to reason about data processing without explicitly implementing data parallelism that BDA data-centric frameworks provide. The former features are highly desired by scientists who want to focus on their problem, rather than the computational elements of their work. As we have seen, they come at the cost of large amounts of memory overhead, which we could not fit in our private infrastructures. This would result in a highly productive and efficient mechanism to build and deploy both HPC and BDA applications.

5.4 Summary

This chapter presented and discussed our experience with the application of paradigms, platforms and infrastructures currently used in BDA, to two typical scientific HPC workflows through the data-centric transformation methodology described in Ch. 4. As a result, we provide the following contributions:

- The transformation of two HPC scientific applications, RPCS and EnKF-HGS, to a data-centric model.
- The implementations of the transformed applications on BDA platforms.
- An evaluation of these implementations taking into consideration their programming and computing models, with the goal of further understanding the BDA and HPC features that affect scalability and performance.
- An analysis of the strengths, weaknesses and limitations of the BDA and HPC paradigms derived from the former results, which support our analysis of the state of the literature and further motivate the development of the unified architecture.

This chapter includes content published in:

- S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, *"A cloudification methodology for multidimensional analysis: Implementation and application to a railway power simulator"* [137].
- S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, *"Efficient Design Assessment in the Railway Electric Infrastructure Domain using Cloud Computing"* [160].
- A. García, S. Caíno-Lores, F. García-Carballeira, and J. Carretero, *"A multi-objective simulator for optimal power dimensioning on electric railways using cloud computing"* [186].
- 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"* [158].
- S. Caíno-Lores, A. Lapin, P. Kropf, J. Carretero, *"Lessons Learned from Applying Big Data Paradigms to a Large Scale Scientific Workflow"* [123].

- S. Caíno-Lores, A. Lapin, P. Kropf, and J. Carretero, *"Methodological Approach to Data-Centric Cloudification of Scientific Iterative Workflows"* [159].

CHAPTER 6

GENERALIST INTEROPERABILITY ARCHITECTURE FOR HYBRID HPC-BDA APPLICATIONS

This chapter presents a generalist architecture to interoperate data- and process-centric runtimes, which allows building a platform suitable for applications with HPC and BDA needs. Following this design, we introduce an implementation and deployment using two specific platforms as building blocks, and present a series of optimisations to enhance the performance and functionality of said implementation.

6.1 Architecture Design

Our approach is to integrate the data-centric and process-centric runtimes 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 said runtimes:

1. BDA users can rely on process-centric runtimes 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 runtimes without any changes. Thus, interoperability must be enabled through a middle-ware 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 minimise 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 runtimes is one of its key features and must be enforced as much as possible. On the

other hand, the efficiency and scalability of process-centric runtimes should be exploited whenever possible to accelerate communication- and compute-intensive operations.

These design goals are embodied in Fig. 6.1, 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).
- A *runtime delegation system* capable of selecting the appropriate runtime for each step in the application (D1, D2, D5).

These elements are detailed in depth in the following sections, and refer directly to the overall thesis approach presented in Ch. 3.

6.1.1 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 SPMD manner. By distributing the overall data volume in chunks, data-centric platforms naturally obtain parallelism, workload balance, and 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. 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

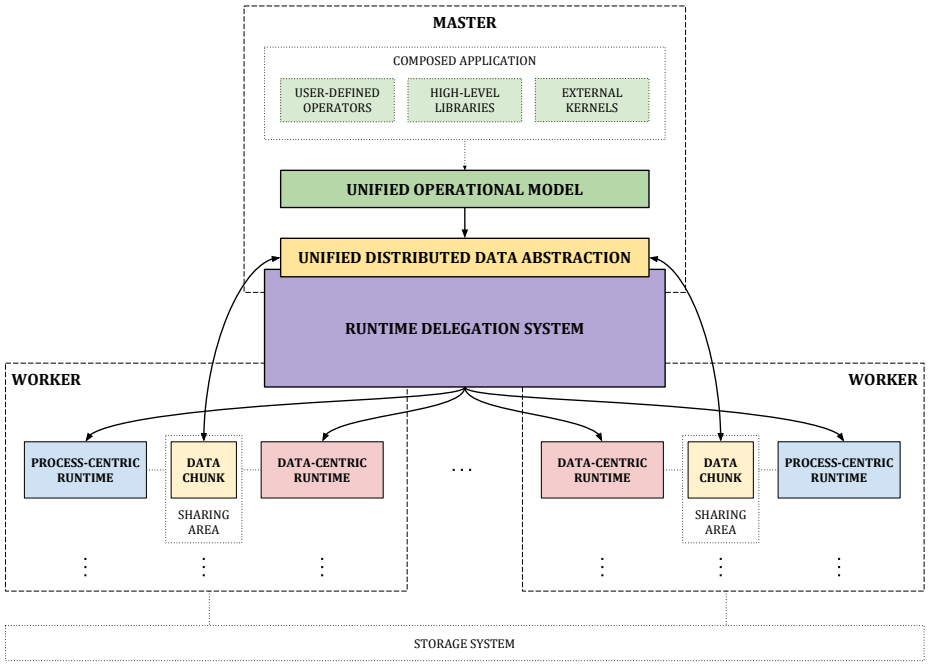


FIGURE 6.1: Overview of the abstract generalist architecture for HPC-BDA.

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 runtimes 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 Fig. 6.1, this abstraction represents a distributed collection of data organised in chunks, which can be locally accessible by both process- and data-centric computing units (D1).

Formally, a UDDA of type t, $\mathscr{U}_t$, can be defined as:

Definition 6.1. *Unified distributed data abstraction (UDDA)*
$\mathscr{U}_t = (A_t, s)$, where $s \in \{0,1\}$ indicates whether its state must be preserved, and $A_t = \{C_0, \cdots, C_{c-1}\}$ is a collection of $c \in \mathbb{N}^*$ data chunks.

A chunk $C_i \in A_t$, with $i \in \mathbb{N}^*$, $i \leq c$ is defined as:

Definition 6.2. *Data chunk*
$C_i = \{r : r \in T\}$, where T is the set of possible values limited by t.

Finally, the smallest discrete information unit in this abstraction is the data record:

Definition 6.3. *Data record*
Given a data type t, and the set of possible values limited by such data type, T, an element r is a data record in a UDDA of type t *iff* $r \in T$ and $\exists C_i \in A_t$ such that $r \in C_i$.

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

Property 6.1. *Generality of a UDDA data record*
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). Therefore

$$\exists\, \mathscr{U}_t \; \forall\, t$$

Property 6.2. *Type consistency of a UDDA*
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, thus

$$\{r \in C_i, C_i \in A_t\} \Rightarrow r \in T$$

Property 6.3. *Cardinality of a UDDA*
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. Consequently

$$\forall A_t \; \exists f(A_t, N) = A'_t, \text{ such that } |A'_t| = N$$

where N is the desired number of chunks specified by the user. Function f could be used to implement different rebalancing and redistribution mechanisms.

Property 6.4. *Locality of a UDDA data chunk*
Location of data chunks should be transparent to the user, and respected as much as possible by the execution runtimes. In addition, users should not be aware of the underlying topological relationships between data chunks, neither for interacting with individual records or defining new operations on the overall dataset (D3). Nevertheless, implementations of a UDDA will have to rely on locality information to track chunks, leading to the property that

$$\forall A_t \; \exists g(C_i) = l$$

where l is the location of chunk C_i. Function g could be used to implement different locality policies that adapt to the underlying infrastructure running the UDDAs.

Once the features of a unified dataset are defined, in the next section we proceed to elaborate on the operational model that derives from its definition and internal properties.

6.1.2 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, thus 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 they will not interfere with one another. These features made these paradigms very 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 inter-node 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 favoured by end users due to 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, it is clear that a unified architecture for BDA and HPC must support traditional data-centric operations and incorporate HPC-oriented functionality (D1). In addition, as shown in Fig. 6.1, 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).

To formalise the different operations we could conduct on a UDDA, we propose a unified operational model (UOM) that represents the function space derived from the definition and properties of the UDDA itself:

Definition 6.4. *Unified operational model (UOM) function space*
Given a UDDA, $\mathscr{U}_t = (A_t, s)$, there is a function space $(\mathscr{V}_u)^{\mathscr{U}_t}$ that maps $(A_t, s) \rightarrow (B_u, s')$, where $\mathscr{V}_u = (B_u, s')$ corresponds to a new UDDA of type u, $\mathscr{V}_u$. Notice that $t = u$ and $s = s'$ are not necessarily enforced.

It is important to highlight that the UOM does not represent an API on its own, since it is a formal definition of the function space inherent to the UDDA, which creates a theoretical frame to define operations in terms of their expected behaviour. Implementations of the UOM will generate an associated API by defining the particular functions supported, their properties, and their syntactic specification.

Among all the theoretical functions that can be defined in $(\mathscr{V}_u)^{\mathscr{U}_t}$, our analysis indicates that certain specialisations are completely necessary to expose a set of operations capable of meeting the requirements of composite applications:

Definition 6.5. *Stateless functions*
A function $f \in (\mathscr{V}_u)^{\mathscr{U}_t}$ is a *stateless function* if $f((A_t, s)) = (B_u, 0)$.

Definition 6.6. *Stateful functions*
A function $f \in (\mathscr{V}_u)^{\mathscr{U}_t}$ is a *stateful function* if $f((A_t, s)) = (B_u, 1)$.

Definition 6.7. *Type-constrained functions*
A function $f \in (\mathcal{V}_u)^{\mathcal{U}_t}$ is a *type-constrained function* if $u = t$.

Definition 6.8. *Cardinality-preserving functions*
A function $f \in (\mathcal{V}_u)^{\mathcal{U}_t}$ is a *cardinality-preserving function* if $|A_t| = |B_u|$, this is, the number of chunks remains constant.

Definition 6.9. *Tuple-based functions*
A function $f \in (\mathcal{V}_u)^{\mathcal{U}_t}$ is a *tuple-based function* if t defines an n_1-tuple set, T, and respectively, u defines an n_2-tuple set, U, with $n_1, n_2 \in \mathbb{N}^* \; n_1, n_2 > 1$.

These definitions can be used to characterise common operations in BDA and HPC environments and implement an application programming interface (API) suitable for the selected process- and data-centric runtimes. For example, a *reduce* from the traditional map-reduce paradigm is classified as a stateless type-constrained function, acting on tuples of one or more elements. Also, the UDDA resulting from a *reduce* operation will typically have less elements and chunks than the input UDDA, thus does not preserve cardinality. Another example is the *in-place all-reduce* operation, typical in HPC paradigms, which can classified as a stateful type-constrained function, since it preserves the nature and results of the dataset, including its cardinality. Implementations of this interface should make the addition of new operators as simple as possible, without losing the flexibility inherited by the UDDA and the function space it defines.

At this point, the theoretical frame defined by the UDDA and its associated UOM allows the formal definition of composite applications with interoperable steps. The next section describes how such stages can be matched to the appropriate runtime.

6.1.3 Runtime Delegation System

As depicted in Fig. 6.1, the proposed architecture follows a master-worker scheme. Using this structure, the definition of the application and the runtime-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 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 runtime delegation system (RDS), which will interpret the requested operations and select the appropriate runtime (D5). For increased adoption and simplicity to the end user, the RDS constitutes an entity that is independent of the underlying runtimes, and acts only as in intermediary without disrupting them (D2).

Formally, given a UOM function space, $(\mathcal{V}_u)^{\mathcal{U}_t}$, there is a set $\mathscr{D} \subseteq (\mathcal{V}_u)^{\mathcal{U}_t}$ that corresponds to the subset of functions that map to the data-centric runtime, and a set $\mathscr{P} \subseteq (\mathcal{V}_u)^{\mathcal{U}_t}$ that corresponds to the subset of functions that map to the process-centric runtime. The objective of the RDS is to enable the delegation of said operations to the appropriate runtime, further defining sets $\mathscr{D}$ and $\mathscr{P}$ in relation to the runtimes it must interact with.

Notice that we cannot impose $\mathscr{D} \cap \mathscr{P} = \emptyset$ because some operations may map by definition to either runtime. For instance, the *map* function from the map-reduce model can be implemented in either a data-centric (e.g. Hadoop) or process-centric runtime (e.g. MPI). Consequently, it is up to the RDS implementation to impose further constraints and limitations to these subsets. Typically, stateful functions will map to the process-centric runtime, and functions that are not type-constrained or cardinality-preserving could have a better fit in the data-centric runtime due to their increased flexibility, but this will ultimately depend on the selected underlying runtimes.

6.2 An 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*) [187]. 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 BDA tools.

Figure 6.2 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 behaviour of the system.

6.2.1 Selected Runtimes

All implementations of the generalist HPC-BDA architecture must build upon existing runtimes as building blocks. Their data abstractions, programming interfaces and execution models will impose technical limitations to the necessary interoperation mechanisms for each element in the architecture. Therefore, it is necessary to analyse in depth each runtime to find the key features that will enable the implementation of the architecture.

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

Data-Centric Runtime: 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

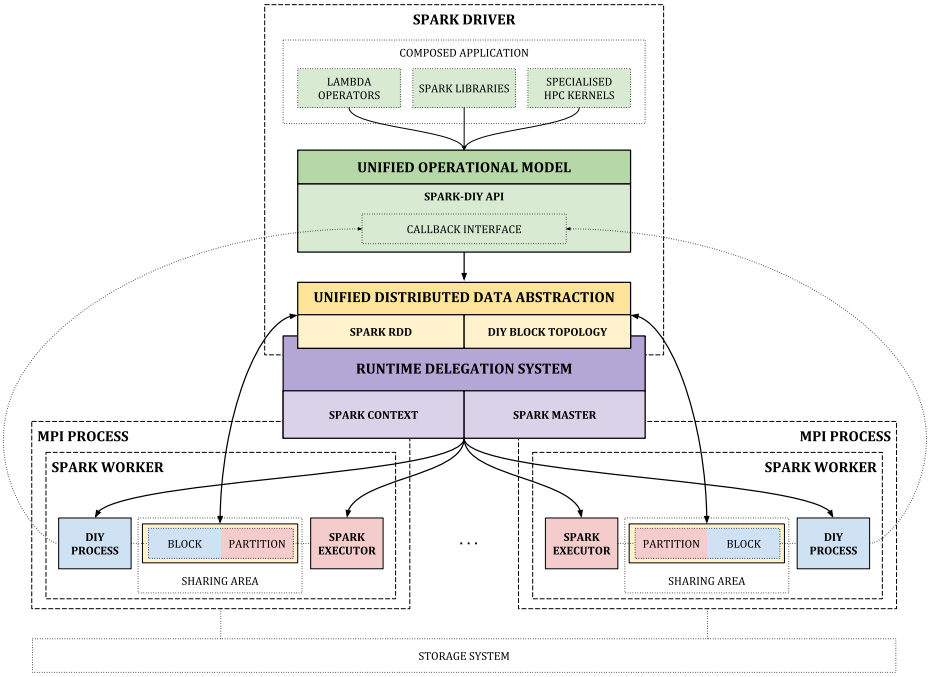


FIGURE 6.2: Implementation of the generalist architecture for HPC-BDA using Spark and DIY (Spark-DIY).

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)[31]. 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*.

Spark can be executed in standalone mode or on top of several resource managers such as YARN [188] and Mesos [189], and it allows the main driver process of a job to be placed inside one of its workers (*cluster mode*) or in the machine that submits the job (*client mode*). In any case, all Spark components run ultimately on top of a Java virtual machine (JVM).

Process-Centric Runtime: 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 neighbourhoods 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 neighbourhood, 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. Table 6.1 shows the contrast between using pure MPI and the DIY interface for a simple program.

Consequently, a problem can be decomposed into a large number of data-parallel sub-problems, and data can be efficiently exchanged among them 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 [190–194], and has demonstrated efficient scaling on leadership-class supercomputers. For example, benchmarks of strong and

TABLE 6.1: Transformation of a MPI program using DIY patterns.

MPI	DIY

`void ParallelAlgorithm() {`
`  ...`
`  MPI_Send();`
`  ...`
`  MPI_Recv();`
`  ...`
`  MPI_Barrier();`
`  ...`
`  MPI_File_write();`
`}`
	`void ParallelAlgorithm() {`
	`  ...`
	`  foreach(&LocalAlgorithm);`
	`  exchange();`
	`  reduce();`
	`  write_blocks();`
	`}`
	`void LocalAlgorithm() {`
	`  ...`
	`}`

weak scaling of parallel Delaunay tessellations [76], one of the libraries built on top of DIY, demonstrated parallel efficiency of over 90% on up to 128K MPI processes.

6.2.2 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 emphasise 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.
- Their execution models, to implement the RDS.

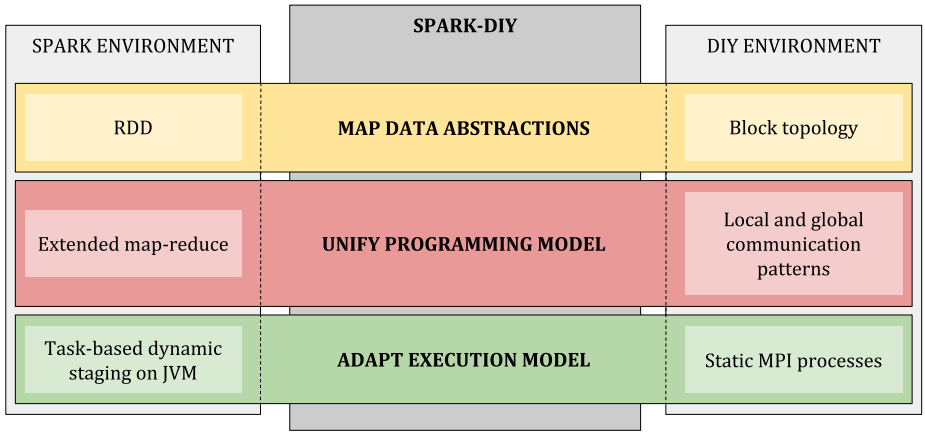


FIGURE 6.3: Interoperation mechanisms between Spark and DIY in the Spark-DIY platform.

The implementation details needed to connect each of these components between the two runtimes are summarised in Fig. 6.3 and detailed in the following sections.

Implementation of the Unified Distributed Data Abstraction

Both in the case of Spark and DIY, the way data are arranged determines the development of algorithms and the behaviour of the runtime. 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. In a nutshell, 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 topology, each data partition in a RDD maps directly to a data block in DIY. In this context, the RDD dataset is partitioned into independent

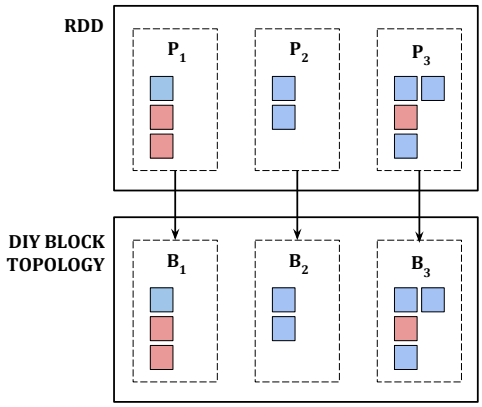


FIGURE 6.4: Mapping of RDD data partitions to DIY blocks.

DIY blocks, as shown in Fig. 6.4, 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, while adding topology information for the DIY-based communication patterns. 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.

In technical terms, many challenges appear when we attempt to interoperate DIY blocks with RDDs. In particular, partitions of RDDs live in the JVM as Java or Scala objects, while DIY blocks are specified using a native language like C/C++. Since UDDAs are required to be generic, data serialisation and type conversion will be necessary to generate appropriate bindings between both programming environments. The implementation of Spark-DIY relies heavily on JNI to manage memory buffers as transparently as possible, and collections of common data types (e.g. primitive types, tuples, strings) are offered in a catalogue for the users' convenience. Although user-defined data

types can be supported and there are several tools to assist in binding generation (e.g. the Simplified Wrapper and Interface Generator (SWIG) [195]), incorporating them efficiently requires tailoring the generated interface and significant knowledge of JNI, the JVM, and Spark-DIY internals. In any case, RDDs are sufficiently semantically rich for BDA usage, and it is unusual for HPC operations to depend on very complex data structures, thus we introduced important optimisations for collections of primitive data types, which are elaborated in Sec. 6.2.3.

There is an additional benefit of relying heavily on RDDs to implement the UDDA: 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. 6.2.3.

Implementation of the Unified Operational Model

Spark actions and transformations conform 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 and existing simulation kernels.

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 these new functionality. In particular, further features like interaction with native kernels and PFS support can be introduced by extending the Spark API with a similar interface. For example, we specifically define the *offload* operator to delegate computations to native simulation kernels.

With respect to the operations that already exist in Spark, Spark-DIY exposes the pure Spark version, and a similar operation that is translated to an algorithm built on top of DIY communication patterns. Since the mapping between RDDs and DIY block topologies is already established, this translation follows naturally because we are able to preserve the independence between partitions, and we can map data shuffles to underlying DIY algorithms. Consequently, these Spark-based operations on RDDs are internally expressed as algorithms that mimic the functionality expected from Spark. For example, the *map* and *filter* transformations in Spark can be translated to a *foreach* pattern in DIY, since both of them represent parallel and independent operations on the dataset; *reduceByKey* in Spark was translated to an algorithm based on the *swap-reduce* DIY pattern, which conducts several rounds of data exchanges between blocks, effectively shuffling data across the partitions; analogously, Spark's *reduce* corresponds to a *merge-reduce* pattern, similar to *swap-reduce* but merging the results in a single value.

Spark-DIY operations on partitions are triggered by the inner 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. Table 6.2 shows a sample of the Spark-DIY API and its relation with the callback definitions.

Implementation of the Runtime Delegation System

Figure 6.2 shows the deployment of Spark-DIY and the interaction of the Spark and DIY components that constitute this implementation of the RDS. 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

TABLE 6.2: Sample of the Spark-DIY API in contrast with the native Spark API, including the required callback definitions. The syntax is purely illustrative and does not reflect minor Scala-specific details.

Spark-DIY	Spark

`Callback extends DIYCallback {`
`  override unary(x) = {f(x)}`
`}`
`DIYmap(Callback())`
	`map(x => f(x))`

`Callback extends DIYCallback {`
`  override unary(x) = {f(x)}`
`}`
`DIYfilter(Callback())`
	`filter(x => f(x))`

`Callback extends DIYCallback {`
`  override binary(x,y) = {f(x,y)}`
`}`
`DIYreduce(Callback())`
	`reduce((x,y) => f(x,y))`

`Callback extends DIYCallback {`
`  override binary(x,y) = {f(x,y)}`
`}`
`DIYreduceByKey(Callback())`
	`reduceByKey((x,y) => f(x,y))`

`Callback extends DIYCallback {`
`  override kernel() = {f()}`
`}`
`DIYoffload(Callback())`
	`N/A`

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 RDS relies on the Spark context for data partitioning and the Spark master for task scheduling and serialisation. Moreover, there is a middle layer that 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, it is clear that pure Spark

operations will be delegated to the data-centric runtime, namely Spark executors. The remaining operations will be delegated to the DIY runtime, 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 serialisation, partitioning, and executor creation by wrapping the partition-block conversion in a function that is passed to a *mapPartitions* Spark operator. This creates executors that live in the MPI environment and contain the data of the corresponding partition, which enforces locality.

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. To achieve this, 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.

6.2.3 Optimisations and Enhancements

Given the state of the implementation at this point, we can observe that the specific features of each runtime 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 serialisation

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 optimisations and extensions we incorporated into this implementation to fully support the generalised architecture, and extend the functionality enabled by the selected runtimes.

We have conducted optimisations to solve two kinds of issues: limitations to the implementation the UDDA and UOM, and performance problems related to the way in which runtimes 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 even if the DIY block topology exists at certain times during the execution. For the second case, it must be acknowledged that the need for generality in the data abstractions adds significant overhead in terms of memory and execution time due to the need for additional serialisation. 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.

Figure 6.5 presents the Spark-DIY platform and new elements used to incorporate optimisations and enhancements that alleviate the former issues. The following paragraphs describe them in detail.

Shared Memory Regions for Stateful Operations

The implementation of Spark-DIY relies heavily on RDDs for data distribution and API support since they provide a straightforward mechanism to induce data locality and data-centric semantics into the DIY block topology. Nonetheless, RDDs are stateless, and the definition of UDDAs forces the inclusion of a state management mechanism.

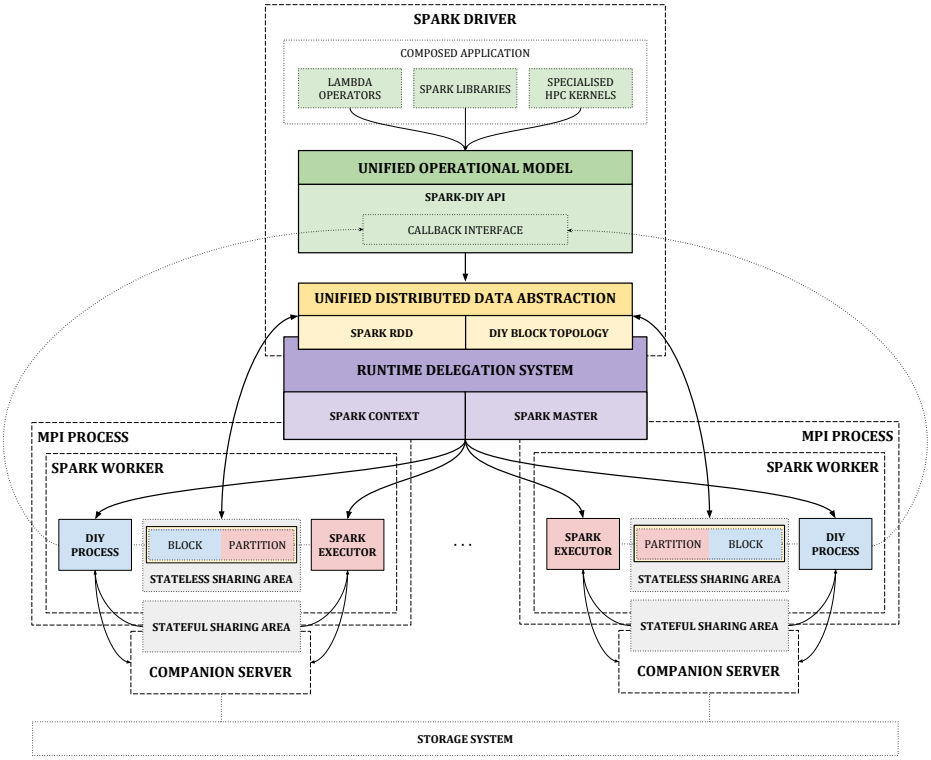


FIGURE 6.5: Optimised and enhanced implementation of Spark-DIY.

In order to coexist RDDs and stateful functions, we introduce a new architectonic element capable of preserving state after a RDD operation. This entity is depicted in Fig. 6.5 as a *companion server* that is intrinsically associated to a specific Spark worker, and can communicate with the executors spawned in it.

The companion server is responsible for managing a shared memory region that can hold a data chunk and maintain it even after the operation finishes and the 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 in 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 runtimes.

Data Serialisation Minimisation

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

Nevertheless, optimisations can be conducted if collections are limited to native data types (*byte*, *short*, *int*, *long*, *char*, *float*, *double*). Spark-DIY offers an interface for these types with reduced overhead, since serialisation between Spark and DIY is not required. This is especially useful for scientific tasks since most of their data are numeric.

To conduct this optimisation, we map data internally to on-heap buffers managed directly by the JNI interface and the DIY functors. Since the size of each record is fixed, we can conduct allocation optimisations and avoid the need for a user-defined serialisation method. In addition, for datasets containing primitive 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.

External Data Management via MPI I/O

Data-centric runtimes interact with storage through limited interfaces that assume the most common characteristics of input data used in BDA applications, mainly text files and data coming from non-relational databases. 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 exploit once again the companion server entity to act as 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 that is placed in the stateful shared data region for subsequent usage. This enables all the potential of MPI I/O to benefit from the highly optimised parallel I/O in HPC systems as an alternative to current storage systems like HDFS.

6.2.4 Usage

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

1. **Select the record data type.** The RDDs to be processed through DIY are collections of data records that we can convert to C++ data types through JNI. To ease this process, a catalogue is offered where users can select a pre-built data type that handles type conversion and memory management from and to the C++ code. Since users may want to use a custom data type not present in the catalogue, we have also developed the internals of Spark-DIY in a generic manner. New data types can be defined in a helper file later used by the JNI code generation utility of choice, which is SWIG in our particular case. New data types must define serialisation and deserialisation functions, since both RDD and DIY block elements need to be serialisable.

2. **Define the callback operators for the record.** Similarly as in Spark, the operations to be conducted on records must be defined. In order to access these operators from DIY, users must implement the proper method as an object that extends the Spark-DIY callback interface.

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

LISTING 6.1: Example of a word count application written with the Spark-DIY *DIYreduceByKey* operator.

3. **Delegate execution on a DIY dataset.** A DIY dataset contains an RDD and mimics the operations the user would normally run on the RDD, with additional functionality. Once an RDD is created along with its operators, we can run the Spark-equivalent transformations and actions implemented using the communication patterns of DIY, offload computations to native kernels, or execute read and write operations in the PFS via MPI. The result of this operation is a new RDD that can be further used in the driver with subsequent combinations of Spark functions or DIY algorithms.

Listing 6.1 shows a simple word count application written in Spark-DIY. Line 12 conducts a pure Spark map, but creates an RDD of the DIY data type *PairRecord*. This RDD is used as input to generate a unified DIY dataset in line 14, which is the input for the DIY reduction in line 15. Notice that the creation of such 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 the desired level of statefulness. 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.

6.3 Summary

This chapter highlighted the key design features that would interest both the HPC and BDA communities to build hybrid applications, and proposed an architecture to attain them. It included the formal definition of the core elements that constitute our proposed generic architecture for HPC-BDA runtime interoperability: the unified distributed data abstraction (UDDA), that can be used to share data between runtimes; the unified operational model (UOM), which defines the function space derived from such data collections; and the runtime delegation system (RDS), which maps each function in the UOM to the appropriate runtime.

This architecture can be implemented in different ways depending on the process- and data-centric runtimes of choice, and the mechanisms put in place to effectively meet the requirements of the architecture. We have explored the potential benefits of integrating a popular BDA platform like Apache Spark with HPC-oriented communication techniques represented by DIY block parallelism. We developed the Spark-DIY framework with these runtimes. Spark-DIY preserves the API and execution environment of Spark, thus making it compatible with any Spark-based application and tool, while providing efficient shuffle, collectives, and kernel offload by using DIY, a powerful library built on top of MPI. Finally, we remarked the optimisations and enhancements implemented in Spark-DIY to fully comply with the formal

architecture, improve performance, and extend the functionality to support HPC-oriented storage.

To summarise, this chapter presents the following contributions:

- The theoretical frame for the formal definitions of the UDDA and UOM of the generic HPC-BDA architecture, keeping into consideration a well-defined set of design goals.
- The construction of the generalist architecture to support workload delegation to the process- and data-centric runtimes.
- An implementation of the architecture, Spark-DIY, based on Apache Spark as data-centric runtime, and DIY as process-centric runtime.
- Additional optimisations and enhancements to ensure full coverage of the design goals and enable supplementary functionality.

This chapter includes content published in:

- 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"* [130].

CHAPTER 7

EVALUATION OF SPARK-DIY WITH HPC AND BDA APPLICATIONS

We selected two applications to analyse the behaviour of a prototype of the Spark-DIY platform: a typical tuple-based operator requiring heavy shuffling, and the EnKF-HGS HPC use case introduced in Sec. 5.2. While the first case aims to isolate potential communication and memory related bottlenecks, the latter exploits many features offered by the framework and showcases its potential against other traditional implementations like MPI or pure Spark.

We have evaluated these applications on 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 processor with 12 physical cores and 135GB of RAM each. Both the Spark and Spark-DIY clusters were configured with single-core workers to limit the number of executors in order to obtain a fair comparison against the MPI deployment. Therefore, each executor is mapped to one worker, and each worker is mapped to an MPI process.

The following sections present and analyse the results of these evaluations.

7.1 Evaluation with a Typical BDA Benchmark

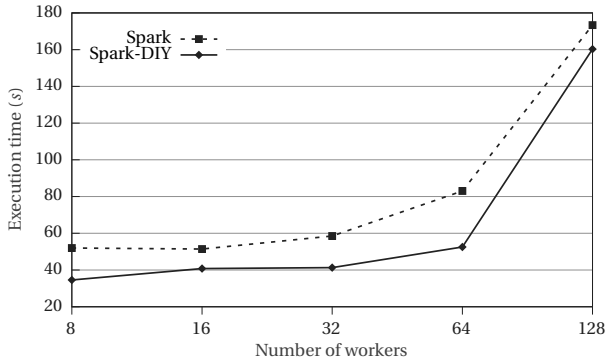
The literature indicates that communication-intensive operations generate most of the scalability issues. For example, EnKF-HGS makes extensive use of reductions in the post-processing stage, since the simulation results of each instantiation of the model need to be shared among them.

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

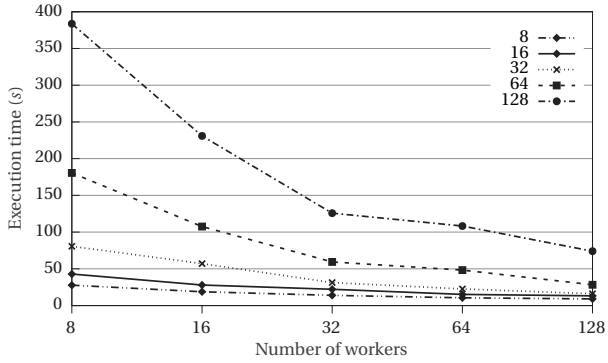
Weak scaling was tested on a dataset holding a constant problem-per-worker of two million records per partition. The objective is to determine how the behaviour of both frameworks evolves as communication for data distribution increases between workers. Additionally, strong scaling was analysed on datasets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

7.1.1 Performance with Generic Data Types

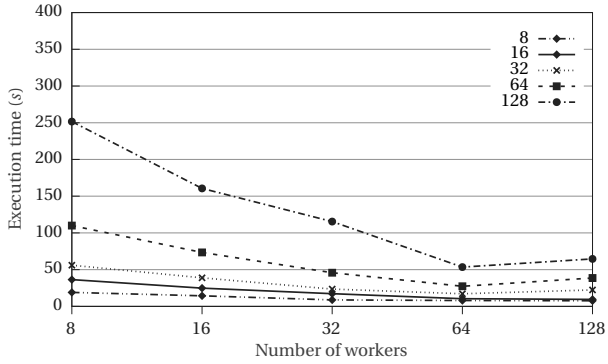
Figure 7.1 depicts the evaluation results for *reduceByKey* on *(string, integer)* pairs, thus showing the behaviour of the generic implementation of Spark-DIY against a Spark application that uses the same data interface. As indicated by Fig.7.1 (a), Spark-DIY offers competitive performance and a similar scaling trend against Spark, although they both fail to scale linearly as the problem size increases. Besides preserving the scaling trend of Spark, Spark-DIY reduces the execution time an average of 25.6%, but this improvement is reduced in the case of 128 workers and 256 millions of records. This shared trend and the reduction in the speed-up provided by Spark-DIY indicates an issue in the Spark platform, which is in charge of parallelisation and task generation in both cases, and this is the price we pay for keeping compatibility and native Spark and DIY frameworks unmodified.



(a) Weak scaling.



(b) Strong scaling (Spark).



(c) Strong scaling (Spark-DIY).

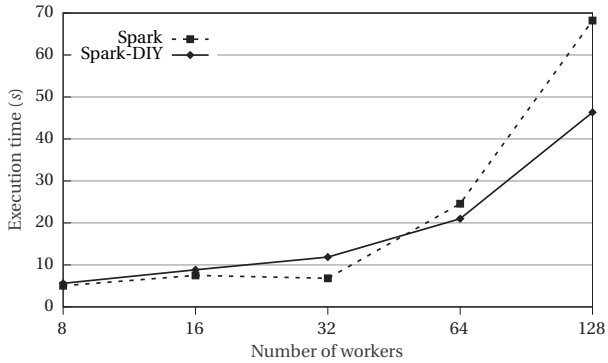
FIGURE 7.1: Evaluation results for *reduceByKey* on Spark and Spark-DIY in terms of weak scaling for a constant problem size of 2 millions of records (a); and strong scaling with variable dataset size of 8 to 128 millions of records (b and c). Records are collections of string-integer pairs.

Since we have shown the behaviour of the Spark-DIY *reduceByKey* is comparable to the Spark counterpart, we now focus on its scalability as the problem size increases for a fixed number of workers. Figure 7.1 shows execution times as the number of workers and the problem size increases for Spark (b) and Spark-DIY (c). The beneficial effects of DIY communication can be clearly appreciated in the figure, in comparison to the lower scale cases. As seen in the weak scaling results, data parallelisation and task management take a large portion of the overall execution time. Therefore, Spark-DIY operations are meaningful in those cases where there is communication involved, and it represents a significant portion of the problem. This effect is clearer as the dataset size increases, which again is a good feature of Spark-DIY, as it is intended for very large datasets. Although Spark-DIY delivers better execution time than Spark for the largest test case (128M records on 128 workers), the comparison against the behaviour of Spark-DIY for 64 workers indicates the existence of a scalability issue.

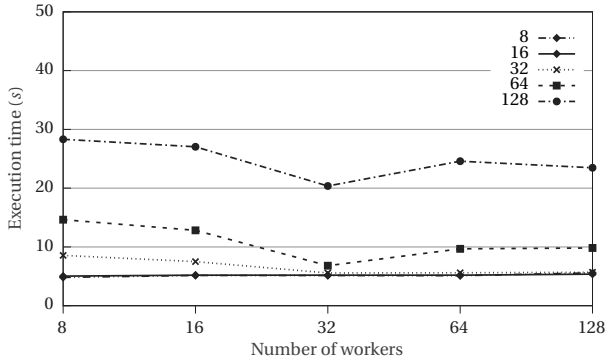
7.1.2 Performance with Primitive Data Types

Although the results in the previous section show promising performance, even considering the need for interoperability, a pure Spark application written with data types native to the selected programming language will deliver much better performance since less conversion and serialisation steps would be needed. With this in mind, and considering that our target use cases (namely applications from the scientific domain) typically rely on primitive data types, we now compare a native Spark implementation against a Spark-DIY implementation using the optimisations for primitive data types described in Sec. 6.2.3. These experiments using *reduceByKey* on *(integer, integer)* pairs are reflected on Fig. 7.2.

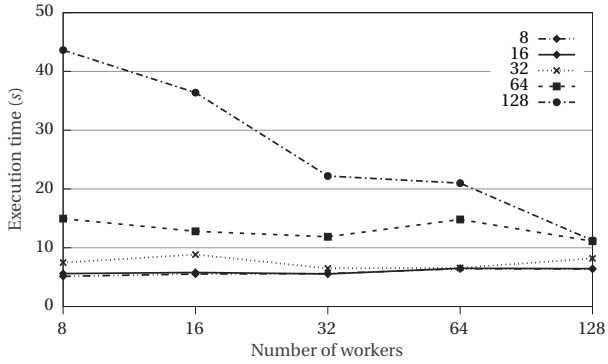
Interestingly, the scaling curves of both platforms do not indicate the same trend as it occurred in the generic case. Although at a smaller scale Spark performs better, Spark-DIY delivers 14.66% and 32% less execution time for



(a) Weak scaling.



(b) Strong scaling (Spark).



(c) Strong scaling (Spark-DIY).

FIGURE 7.2: Evaluation results for *reduceByKey* on Spark and Spark-DIY for primitive data types in terms of weak scaling for a constant problem size of 2 millions of records (a); and strong scaling with variable dataset size of 8 to 128 millions of records (b and c). Records are collections of 4-byte integers.

64 and 128 workers respectively. This is also supported by the strong scaling results portrayed in (b) and (c): Spark shows a flat curve, which contrasts with the rough slope in Spark-DIY for 128 millions of records. As a result, Spark-DIY is 52.1% and 14.6% faster than Spark using 128 and 64 workers respectively, but slower if the number of workers is less.

Consequently, there is a trade-off between the scale of the problem, the platform used, and the performance obtained. At scale, the performance of Spark-DIY for this use case is at least equivalent, and end users could exploit the flexibility and interoperability offered by Spark-DIY to enable the usage of higher-level Spark libraries and external HPC elements.

7.2 Evaluation with a Real-World Application

The use case presented in Sec. 5.2, EnKF-HGS, was originally designed as a HPC workflow to model the behaviour of hydrogeological systems, but the evolution of this scientific domain brought new challenges related to the opportunities found as decentralised infrastructures advanced. Besides specific models for pre-alpine valleys in the Swiss Emmental region, the tool can also incorporate real-time sensor data to refine its predictions. Figure 7.3 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.

This use case relies on cloud services for computation, data assimilation and storage. In addition, BDA computing platforms constitute a natural fit for EnKF-HGS because they provide facilities to collect data from streaming sources. On the other hand, this tool must handle many MPI simulations running in parallel, and high-performance is required as in any other scientific application. The combination of these requirements and features makes a

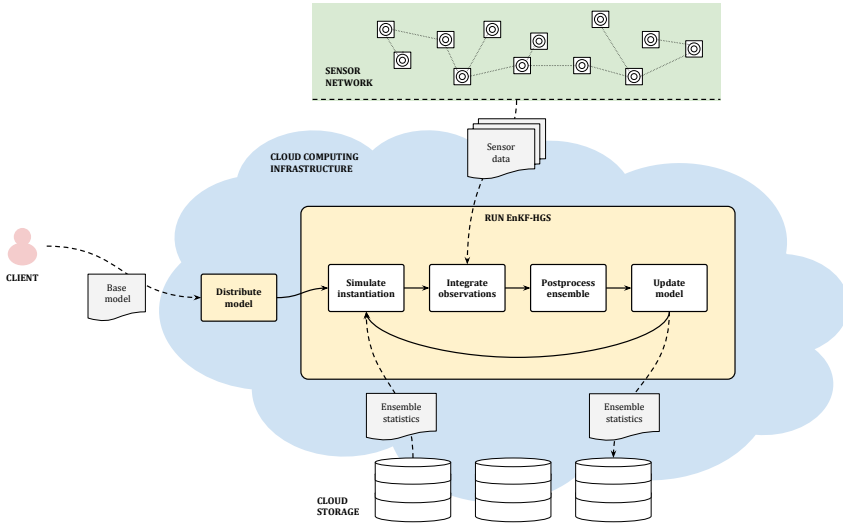


FIGURE 7.3: Interoperation of EnKF-HGS with the data assimilation sensor network and its supporting cloud infrastructure.

case for the need of scalable HPC-BDA convergence in EnKF-HGS. Moreover, other applications show similar needs to fuse sensor and simulation data, including weather forecasting [196], and carbon cycle [197] studies.

In Ch. 5 we reported our experience combining traditional HPC with BDA-inspired paradigms and platforms, in the context of scientific ensemble workflows like EnKF-HGS. Our goal was to provide a suitable environment that combined the HPC and BDA elements required by EnKF-HGS, so we integrated the simulation kernels with the Apache Spark framework, which also supports streaming, using the methodology in Ch. 4. We found that Spark was unable to scale due to 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 platform's overhead. Limitations of the shuffle phase have been reported by others [198, 199] as well. They can be traced back to multiple causes: high memory utilisation 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, and present the results of its optimisations as scale increases.

7.2.1 Building EnKF-HGS with Spark-DIY

The result of implementing the data-centric EnKF-HGS in Spark was an application with a parallel region in charge of executing the GROK and HGS kernels in Spark tasks, and a model update region that required all data to be collected in the driver process to reassemble the matrices of the model. As seen in its evaluation, this version of the application shows scalability issues do to the bottleneck found in the model update region.

Spark-DIY can help to mitigate this problem by delegating the analysis to DIY as an MPI kernel through the *offload* operator. This division in the parallel and model update regions is highlighted in Fig. 7.4.

This new implementation of EnKF-HGS combines the shared memory enhancements of Spark-DIY to enable statefulness in the model evaluation 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 in-place without needing any 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 analysis kernel implemented in MPI, which reduces drastically the development effort to obtain a full implementation of a BDA-capable

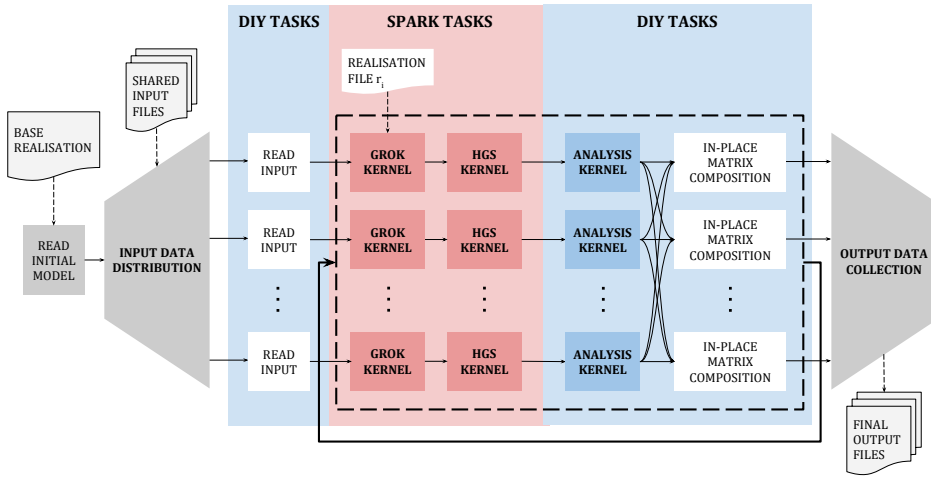


FIGURE 7.4: Implementation of data-centric EnKF-HGS on Spark-DIY. The parallel region is mapped to Spark tasks, and the model update region is delegated to DIY processes. Input data are read via MPI I/O through DIY tasks in collaboration with the companion servers.

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 absolutely necessary to eliminate the need to collect the matrices in the driver process. This is due to the analysis kernel being inherently stateful. Furthermore, these regions minimise the serialisations between iterations because they remain active after they are created by the companion servers. This has a positive effect in the reduction of data management overhead.

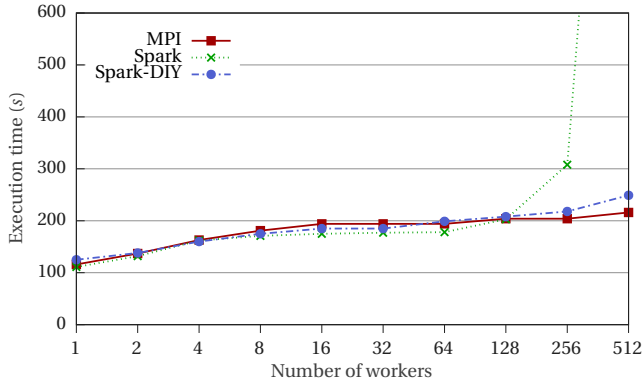
- The optimisations introduced in Spark-DIY tackling primitive data types also contribute to reduce the negative effects of serialisation in performance. Since EnKF-HGS relies on numeric algorithms, all of its data are handled in terms of floats or integers, and this simplifies the management of shared buffers through JNI.

- The MPI I/O external data management represents a major benefit versus a pure Spark implementation due to the way in which input data is organised. Input files can be read in parallel to retrieve the particular data required in each process, exactly in the way in which it is needed (i.e. in columns). This is advantageous because MPI I/O has good performance and scalability, and enables future optimisations 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 favour of storing the output directly through this mechanism.

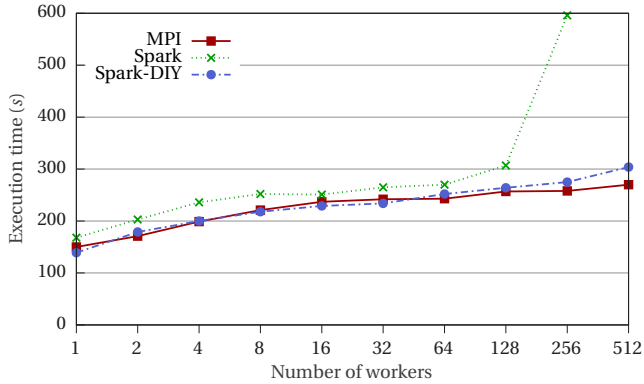
We now analyse the behaviour of EnKF-HGS implemented in Spark-DIY against the original MPI implementation and the version written in pure Spark resulting from Ch. 5.

7.2.2 Performance Results

Figure 7.5 shows the results of the evaluation of EnKF-HGS on Spark-DIY to compare its performance –measured in execution time– against the pure Spark implementation of EnKF-HGS introduced in Ch. 5, and its original MPI implementation taken as baseline. The three implementations are 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. We only report results for one and two iterations of the Kalman filter because succeeding iterations will incorporate further randomness that could lead to inadequate comparisons. In any case, both the parallel and model update regions are triggered with more than one iteration, so we cover the entire application workflow.



(a) Execution time for one iteration.



(b) Execution time for two iterations.

FIGURE 7.5: Evaluation results of EnKF-HGS on Spark-DIY measured in execution time (in seconds) for one (a) and two (b) iterations.

The results for one iteration show that Spark-DIY performs very 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. This reinforces the idea that data awareness can accelerate massively-parallel computations, as introduced in Ch. 5.

In the case of two 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 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, and are 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 realisations.

As a consequence of the I/O and shared memory management capabilities of Spark-DIY, the Spark-DIY implementation is consistently competitive against MPI as the number of realisations increases and achieves comparable scalability. The reason for this is the key effect of statefulness in the overall application: not only improves performance by eliminating overheads related to data being serialised and moved around as the procedure advances, but also eliminates the need of 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 trade-off for the flexibility of incorporating the elements of a whole new ecosystem into a HPC application, and further optimisations might alleviate this issue in future works.

7.3 Summary

This chapter presented the evaluation results of the implementation of the *reduceByKey* benchmark and the EnKF-HGS use case on the Spark-DIY platform, our implementation of the HPC-BDA architecture.

This platform showed good scalability and performance results against Spark and MPI. In particular, the evaluation of the EnKF-HGS use case indicated that Spark-DIY enables the integration of elements from both the BDA and HPC ecosystems for applications with diverse requirements without sacrificing scalability and performance. Spark-DIY introduced limited overheads in exchange for this flexibility, and allowed a level of scalability that would not be attainable by Spark on its own.

This chapter includes content published in:

- 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"* [130].

CHAPTER 8

CONCLUSIONS

Nowadays, there is a need for software solutions able to integrate the benefits of the BDA and HPC ecosystems for the development of applications that present requirements from both paradigms. This thesis presented advances in the context of BDA and HPC convergence, more specifically with respect to the interoperation of runtimes to support the composition and execution of hybrid applications.

The main goal of this thesis was **to research new approaches to facilitate the convergence of HPC and BDA paradigms by providing common abstractions and mechanisms for improving scalability, data locality exploitation, and execution adaptivity on large scale systems**, while preserving the most relevant features for their corresponding communities, in order to provide a system suitable for the composition of applications with BDA and HPC stages. This work achieved such goal by meeting its complementary objectives as follows:

O1 Analyse the key features that characterise HPC and BDA ecosystems. Chapter 2 introduced a deep analysis of the literature and current state of the HPC and BDA ecosystems, considering their traditional focus, and the most recent advances in terms of infrastructure, storage, programming model, execution model, and application domains. As a result, Ch. 3 elaborated on the challenges opened by the search for convergence and defined the characteristics that should be present in an ideal system for hybrid applications. With these in mind, we extracted a set of key design goals that guided the rest of our work.

O2 Provide a mechanism to reshape HPC-oriented workflows in order to adapt them to data-centric environments. The need for a technique to adapt HPC applications to a BDA-oriented environment was tackled in Ch. 4. This chapter presented a data-centric transformation methodology for iterative scientific applications, which constitute the majority of the applications in various scientific domains. Being able to incorporate the data-centric features of BDA platforms proved relevant to improve scalability and the possibility to interoperate such applications with higher-level analytics jobs, and new data sources like streamed data.

O3 Design and develop an architecture that offers runtime interoperability for hybrid HPC-BDA applications, incorporating unified operational and data models that support high-level analytics methods and high-performance kernels for composite applications. The ability to reason about data in such a natural manner made BDA-oriented programming models popular and easier to learn that the process-centric interfaces for HPC. With this into consideration, Ch. 6 introduced a new data abstraction that unifies the requirements of the distributed datasets found in BDA and HPC platforms, and a formal definition of an operational model suitable for HPC-BDA applications based on such abstraction. These elements are the core of the architecture introduced in Ch.6, which allows the composition of hybrid applications since data can be seamlessly fed to a data-centric or process-centric feature using the same interface. Their execution is managed by a runtime delegation entity that acts as bridge between both runtimes, without requiring changes in neither of them. Finally, we developed an implementation of this architecture based on Spark and an MPI-based library –DIY–, which we called Spark-DIY. Once the data model was unified, we were able to enrich the traditional approach to data-centric programming of the Spark API with operators that allow the

delegation of HPC operations to the process-centric runtime. For example, this includes the execution of legacy simulation kernels and communication-intensive stages for matrix operations. In addition, we incorporated optimisations to our Spark-DIY framework to support MPI I/O for further flexibility and support of traditional HPC storage systems. Since Spark-DIY runs with an unmodified version of Spark, higher-level libraries built on the RDD abstraction remain compatible with our implementation.

O4 Evaluate on meaningful use cases, representative of target hybrid applications. Chapter 5 presented EnKF-HGS, a real-world use case from the hydrogeology domain that embodies the nature of the applications our architecture aims to assist: it presents both HPC characteristics –as the need to preserve the state of the model between iterations, to execute compute-intensive simulations, and to update shared matrices in a all-to-all communication pattern–, and BDA features –like the need to support streamed data and run on cloud services–. After adapting this workflow to a data-centric approach, using the methods in Ch. 4, Ch. 7 presented its full implementation as a Spark-DIY application able to cope with the challenges of an evolving domain. Additional evaluation was also provided in said chapter for a *reduceByKey* benchmark.

This thesis yields the following contributions from the accomplishment of the former objectives:

C1 A data-centric enablement methodology aimed at reshaping HPC iterative scientific applications to match the data-centric paradigm of BDA platforms. This can be used to incorporate simulation stages to BDA applications, and update legacy HPC applications either by adapting them to new computing infrastructures like clouds, or incorporating BDA techniques that empower their flexibility such as data streaming and visualisation.

C2 **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. This data abstraction embodies a careful selection of the features required to interoperate BDA and HPC operations, and generates a theoretical frame that must be enforced by implementations of the architecture to preserve interoperability and formal correctness.

C3 **A generalist runtime interoperability architecture for HPC-BDA applications** based on the UDDA and UOM definitions. It includes a transparent runtime delegation system (RDS) that transfers each stage of the composite application to the appropriate runtime (process- or data-centric).

C4 **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 optimised 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.

C5 **An implementation of a real-world use case from the hydrogeology domain** enriched with features enabled by our architecture like cloud and streaming support for de-localisation and data assimilation, respectively.

8.1 Future Directions

At the moment, the data-centric transformation methodology involves steps that must be conducted by direct examination of the original application. As this process is critical, and it is intimately related with the structure and input

of the application, it should be conducted by an expert in the simulator to be transformed. Future work could simplify this stage by mean of automatic analysis and variable proposal, yet an expert would still be needed to assess the correctness of the suggestion. Furthermore, another future direction for this methodology is the analysis of the model update region of HPC applications, in order to assess which communication patterns could be transformed in a data-centric manner as well.

The architecture could be enhanced 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 multi-tenancy, fault-tolerance and elasticity. In addition, usability and productivity could be addressed with formal studies on the BDA and HPC user communities.

The implemented framework shows good performance and scalability for communication-intensive operations in comparison to Spark, and enables the integration of elements from both the BDA and HPC ecosystems for applications with diverse requirements. This is relevant for the BDA community since we offer improved performance and reduced latency for shuffle and other communication-intensive phases of Spark workflows. In addition, we expose the benefits of using supercomputing infrastructures without changing the Spark framework because we exploit MPI-based communication. Future work could improve these benefits by extending the Spark programming model to exploit other DIY communication patterns such as local neighbourhood block exchanges that are available in DIY, but have no Spark counterpart.

Finally, further real-world use cases could be built 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 are lacking in the HPC software stack, and Spark-DIY affords HPC practitioners of such characteristics that are commonplace in the BDA world. Demonstrators from the BDA side –like HPDA or IoT applications– would be particularly interesting, since they are not covered in this thesis.

8.2 Thesis Results and Achievements

Journals

1. 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", in *Future Generation Computer Systems*, April 2018. *Impact factor: 3.997, Q1.*

2. S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, "Efficient design assessment in the railway electric infrastructure domain using cloud computing", in *Integrated Computer-Aided Engineering*, vol. 24, pp. 57–72, December 2016. *Impact factor: 5.264, Q1.*

3. S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, "A cloudification methodology for multidimensional analysis: Implementation and application to a railway power simulator", in *Simulation Modelling Practice and Theory*, vol. 55, pp. 46–62, June 2015. *Impact factor: 1.482, Q1.*

International Conferences

1. 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 *5th IEEE/ACM International*

Conference on Big Data Computing, Applications and Technologies (BDCAT 2018), Zurich, Switzerland, December 2018. *Best-paper finalist*

2. S. Caíno-Lores, F. Isaila, and J. Carretero, "Data-Aware Support for Hybrid HPC and Big Data Applications", in *Doctoral Symposium at the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2017)*, Madrid, Spain, May 2017. *CORE A.*

3. S. Caíno-Lores, A. Lapin, P. Kropf, and J. Carretero, "Methodological Approach to Data-Centric Cloudification of Scientific Iterative Workflows", in *16th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2016)*, Granada, Spain, December 2016. *CORE B.*

4. S. Caíno-Lores, A. Lapin, P. Kropf, J. Carretero, "Lessons Learned from Applying Big Data Paradigms to a Large Scale Scientific Workflow", in *11th Workshop on Workflows in Support of Large-Scale Science (WORKS 2016)*, in conjunction with *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*, Salt Lake City, Utah, USA, November 2016.

5. S. Caíno-Lores, J. Carretero, "A Survey on Data-Centric and Data-Aware Techniques for Large Scale Infrastructures", in *18th International Conference on Computer and Information Sciences (ICCIS 2016)*, Dubai, UAE, March 2016.

6. A. García, S. Caíno-Lores, F. García-Carballeira, and J. Carretero, "A multi-objective simulator for optimal power dimensioning on electric railways using cloud computing", in *5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2015)*, Kolmar, France, July 2015. *Best-paper finalist*

7. S. Caíno-Lores, A. Garcia, F. Garcia-Carballeira, and J. Carretero, "A cloudification methodology for numerical simulations", in *Euro-Par 2014: Parallel Processing Workshops*, Porto, Portugal, August 2014.

National Conferences

1. S. Caíno-Lores, A. Lapin, P. Kropf, and, J. Carretero "Cloudification of a Legacy Hydrological Simulator using Apache Spark", in *XXVII Jornadas de Paralelismo*, Salamanca, Spain, September 2016.

2. S. Caíno-Lores, A. García, F. García-Carballeira, and J. Carretero, "Breaking data dependences in numerical simulations using Map-Reduce", in *XXV Jornadas de Paralelismo*, Valladolid, Spain, September 2014.

Research stays

1. ANL Graduate Research Aide: "Big Data and HPC convergence for data analytics and visualisation", Argonne National Laboratory, USA, August-December, 2017. Supervisor: Dr. Tom Peterka.

2. COST Action STSM: "Future directions on the optimisation of hydrogeological simulations", Université de Nechâtel, Switzerland, January, 2017. Supervisor: Prof. Peter Kropf.

3. COST Action STSM: "Data-centric cloudification methodologies for scientific computing", Université de Nechâtel, Switzerland, February, 2016. Supervisor: Prof. Peter Kropf.

Grants and Awards

1. Graduate Research Aide Appointment, 2017, Argonne National Laboratory, USA.

2. PPI-B-17 Research Stay Grant, 2017, Carlos III University of Madrid, Spain.

3. FPU15 Research Stay and Exchange Program, 2017, Spanish Ministry of Education, Spain.

4. IC1305 NESUS COST Action Short-Term Scientific Mission Grant, 2017, European Cooperation in Science and Technology (COST), European Union.
5. FPU15 Research Training Program for Academic Staff Fellowship, 2016 Spanish Ministry of Education, Spain.
6. International IBM PhD Fellowship Award. 2016, IBM, USA.
7. IC1305 NESUS COST Action Short-Term Scientific Mission Grant, 2016, European Cooperation in Science and Technology (COST), European Union.
8. PIF03-1516 Research Training Program Grant, 2015, Carlos III University of Madrid, Spain.

Projects

1. Spanish Ministry of Economics and Competitiveness, TIN2016-79637-P, "Towards Unification Of HPC And Big Data Paradigms" (BIGHPC).
2. European Union, COST Action IC1305, "Network for Sustainable Ultrascale Computing Platforms" (NESUS).
3. European Union, INEA/CEF/ICT/A2016/1278042, "Multiple Access to eDElivery" (MADE).
4. European Union, INEA/CEF/ICT/A2016/1271635, "Integrating the eIdentification in European cloud platforms according to the eIDAS Regulation" (eID@Cloud).
5. European Union, ICT-09-2014, "Refactoring Parallel Heterogeneous Resource-Aware Applications" (RePhrase).
6. Spanish Ministry of Economics and Competitiveness, TIN2013-41350-P, Técnicas de gestión escalable de datos para high-end computing systems.

7. Spanish Ministry of Economics and Competitiveness, TIN2011-15734-E, Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H).

8. Administrator of Railway Infrastructures (ADIF), Proyecto para la Investigación sobre la aplicación de las TIC a la innovación de las diferentes infraestructuras correspondientes a las instalaciones de electrificación y suministro de energía (SIRTE), JM/RS 3.9/1500.0009/0-00000.

BIBLIOGRAPHY

[1] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In *2014 IEEE International Congress on Big Data*, pages 645–652, June 2014. doi: 10.1109/BigData.Congress.2014.137.

[2] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493, 2015. ISSN 1572-9184.

[3] Daniel A Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[4] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META group research note*, 6(70):1, 2001.

[5] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[6] Philip Russom et al. Big data analytics. *TDWI best practices report, fourth quarter*, 19(4):1–34, 2011.

[7] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.

[8] Big Data Value Association. European big data value strategic research and innovation agenda. Technical report, October 2017.

[9] Chun-Wei Tsai, Chin-Feng Lai, Han-Chieh Chao, and Athanasios V. Vasilakos. Big data analytics: a survey. *Journal of Big Data*, 2(1):21, October 2015. ISSN 2196-1115. doi: 10.1186/s40537-015-0030-3. URL `https://doi.org/10.1186/s40537-015-0030-3`.

[10] Domenico Talia, Paolo Trunfio, and Fabrizio Marozzo. *Data analysis in the cloud: models, techniques and applications*. Elsevier, 2015.

[11] Peter Mell and Tim Grance. Effectively and securely using the cloud computing paradigm. *NIST, Information Technology Laboratory*, pages 304–311, 2009.

[12] 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 *2013 IEEE International Congress on Big Data*, pages 62–69, June 2013. doi: 10.1109/BigData.Congress.2013.18.

[13] Pethuru Raj, Anupama Raman, Dhivya Nagaraj, and Siddhartha Duggirala. *High-Performance Big-Data Analytics: Computing Systems and Approaches*. Springer, 2015.

[14] Eric E. Schadt, Michael D. Linderman, Jon Sorenson, Lawrence Lee, and Garry P. Nolan. Computational solutions to large-scale data management and analysis. *Nature Reviews Genetics*, 11(9):647–657, September 2010. ISSN 1471-0064. doi: 10.1038/nrg2857. URL `https://www.nature.com/articles/nrg2857`.

[15] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3 (5):637–646, 2016.

[17] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber,

and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[18] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Joerg Fritsch and Coral Walker. The problem with data. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 708–713. IEEE, 2014.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. 37(5):29–43, 2003.

[22] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[23] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010. doi: 10.1109/MSST.2010.5496972.

[24] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Mapreduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[25] Radu Tudoran, Alexandru Costan, and Gabriel Antoniu. Mapiterativereduce: a framework for reduction-intensive data processing on azure clouds. In *Proceedings of third international workshop on MapReduce and its Applications Date*, pages 9–16. ACM, 2012.

[26] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[27] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[28] Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, and Judy Qiu. Scalable parallel computing on clouds using twister4azure iterative mapreduce. *Future Generation Computer Systems*, 29(4):1035–1048, 2013.

[29] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851593. URL `http://doi.acm.org/10.1145/1851476.1851593`.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010.

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[32] 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 Transactions on Parallel and Distributed Systems*, 26(8):2300–2315, Aug 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2345068.

[33] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Manuel Costa, Derek Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. *Young*, 4:8, 2015.

[34] Luca Salucci, Daniele Bonetta, and Walter Binder. Lightweight multi-language bindings for apache spark. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, pages 281–292, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-319-43658-6. doi: 10.1007/978-3-319-43659-3_21. URL `http://dx.doi.org/10.1007/978-3-319-43659-3_21`.

[35] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. *How Data Volume Affects Spark Based Data Analytics on a Scale-up Server*, pages 81–92. Springer International Publishing, Cham, 2016. ISBN 978-3-319-29006-5. doi: 10.1007/978-3-319-29006-5_7. URL `http://dx.doi.org/10.1007/978-3-319-29006-5_7`.

[36] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2188-4. doi: 10.1145/2484425.2484427.

[37] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[38] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[39] Svilen R Mihaylov, Zachary G Ives, and Sudipto Guha. Rex: recursive, delta-based data-centric computation. *Proceedings of the VLDB Endowment*, 5(11):1280–1291, 2012.

[40] Ciprian Dobre and Fatos Xhafa. Parallel programming paradigms and frameworks in big data era. *International Journal of Parallel Programming*, 42(5):710–738, 2014.

[41] Fan Zhang, Ciprian Docan, Manish Parashar, Scott Klasky, Norbert Podhorszki, and Hasan Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1352–1363. IEEE, 2012.

[42] Fan Zhang, Qutaibah M Malluhi, Tamer Elsayed, Samee U Khan, Keqin Li, and Albert Y Zomaya. Cloudflow: A data-aware programming model for cloud workflow applications on modern hpc systems. *Future Generation Computer Systems*, 2014.

[43] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[44] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd workshop on many-task computing on grids and supercomputers*, page 8. ACM, 2009.

[45] A. Al-Badarneh, H. Najadat, M. Al-Soud, and R. Mosaid. Phoenix: A mapreduce implementation with new enhancements. In *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, pages 1–5, July 2016. doi: 10.1109/CSIT.2016.7549451.

[46] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213934.

[47] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[48] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14, 2010.

[49] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367513. URL `http://dx.doi.org/10.14778/2367502.2367513`.

[50] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767921. URL `http://dx.doi.org/10.1109/ICDE.2011.5767921`.

[51] Ladislav Hluchý, Martin Bobák, Henning Müller, Mara Graziani, Jason Maassen, Hanno Spreeuw, Matti Heikkurinen, Jörg Pancake-Steeg, Stefan Spahr, Nils Otto vor dem Gentschen Felde, Maximilian Höb, Jan Schmidt, Adam S. Z. Belloum, Reginald Cushing, Piotr Nowakowski, Jan Meizner, Katarzyna Rycerz, Bartosz Wilk, Marian Bubak, Ondrej Habala, Martin Šeleng, Štefan Dlugolinský, Viet Tran, and Giang Nguyen. *Heterogeneous Exascale Computing*. Springer International Publishing, Cham, 2020. ISBN 978-3-030-14350-3. doi: 10.1007/978-3-030-14350-3_5. URL https://doi.org/10.1007/978-3-030-14350-3_5.

[52] Katherine Yelick, Susan Coghlan, Brent Draney, Richard Shane Canon, et al. The magellan report on cloud computing for science. *US Department of Energy, Washington DC, USA, Tech. Rep*, 2011.

[53] Ioan Raicu, Ian T Foster, and Pete Beckman. Making a case for distributed file systems at exascale. In *Proceedings of the third international workshop on Large-scale system and application performance*, pages 11–18. ACM, 2011.

[54] Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project*, page 53, 2008.

[55] Shawn M. Strande, Pietro Cicotti, Robert S. Sinkovits, William S. Young, Rick Wagner, Mahidhar Tatineni, Eva Hocks, Allan Snavely, and Mike Norman. Gordon: Design, performance, and experiences deploying and supporting a data intensive supercomputer. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*, XSEDE '12, pages 3:1–3:8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1602-6. doi: 10.1145/2335755.2335789. URL http://doi.acm.org/10.1145/2335755.2335789.

[56] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[57] Thomas Willhalm and Nicolae Popovici. Putting intel®threading building blocks to work. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, pages 3–4, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-031-9. doi: 10.1145/1370082.1370085. URL http://doi.acm.org/10.1145/1370082.1370085.

[58] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, Redmond, WA, USA, 1st edition, 2011. ISBN 0735651752, 9780735651753.

[59] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[60] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Feb 2009. doi: 10.1109/PDP.2009.43.

[61] Pablo D. Mininni, Duane Rosenberg, Raghu Reddy, and Annick Pouquet. A hybrid mpi–openmp scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Computing*, 37(6):316 – 326, 2011. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2011.05.004. URL http://www.sciencedirect.com/science/article/pii/S0167819111000512.

[62] Suchuan Dong and George Em Karniadakis. Dual-level parallelism for high-order cfd methods. *Parallel Computing*, 30(1):1 – 20, 2004. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2003.05.

020. URL http://www.sciencedirect.com/science/article/pii/S016781910300173X.

[63] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[64] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[65] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016.

[66] J. Guan, S. Yan, and J. Jin. An openmp-cuda implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-gpu computing systems. *IEEE Transactions on Antennas and Propagation*, 61(7):3607–3616, July 2013. ISSN 0018-926X. doi: 10.1109/TAP.2013.2258882.

[67] P.S. Rakić, D.D. Milašinović, Ž. Živanov, Z. Suvajdžin, M. Nikolić, and M. Hajduković. Mpi–cuda parallelization of a finite-strip program for geometric nonlinear analysis: A hybrid approach. *Advances in Engineering Software*, 42(5):273 – 285, 2011. ISSN 0965-9978. doi: https://doi.org/10.1016/j.advengsoft.2010.10.008. URL http://www.sciencedirect.com/science/article/pii/S0965997810001286. PARENG 2009.

[68] 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.*, 38(4):23–29, March 2011. ISSN 0163-5999. doi: 10.1145/1964218.1964223. URL http://doi.acm.org/10.1145/1964218.1964223.

[69] M. U. Ashraf, F. Alburaei Eassa, A. Ahmad Albeshri, and A. Algarni. Performance and power efficient massive parallel computational model

for hpc heterogeneous exascale systems. *IEEE Access*, 6:23095–23107, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2823299.

[70] Steven J. Plimpton and Karen D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, September 2011. ISSN 0167-8191. doi: 10.1016/j.parco.2011.02.004.

[71] Yi Wang, Gagan Agrawal, Tekin Bicer, and Wei Jiang. Smart: A mapreduce-like framework for in-situ scientific analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 51. ACM, 2015.

[72] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer. Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1098–1108, May 2017. doi: 10.1109/IPDPS.2017.31.

[73] Yanfei Guo, Wesley Bland, Pavan Balaji, and Xiaobo Zhou. Fault tolerant mapreduce-mpi for hpc clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 34:1–34:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807617. URL `http://doi.acm.org/10.1145/2807591.2807617`.

[74] Fan Liang and Xiaoyi Lu. Accelerating iterative big data computing through mpi. *Journal of Computer Science and Technology*, 30(2):283–294, 2015.

[75] Tom Peterka, Robert Ross, Attila Gyulassy, Valerio Pascucci, Wesley Kendall, Han-Wei Shen, Teng-Yok Lee, and Abon Chaudhuri. Scalable parallel building blocks for custom data analysis. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 105–112. IEEE, 2011.

[76] Tom Peterka, Dmitriy Morozov, and Carolyn Phillips. High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 997–1007. IEEE Press, 2014.

[77] 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 Transactions on Parallel and Distributed Systems*, 26(11):3066–3078, Nov 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2365784.

[78] Francisco J. Clemente-Castelló, Bogdan Nicolae, Rafael Mayo, Juan Carlos Fernández, and M. Mustafa Rafique. On exploiting data locality for iterative mapreduce applications in hybrid clouds. In *BDCAT'16: 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 118–122, Shanghai, China, 2016.

[79] Francisco J. Clemente-Castelló, Bogdan Nicolae, Rafael Mayo, and Juan Carlos Fernández. Performance model of mapreduce iterative applications for hybrid cloud bursting. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1794–1807, 2018.

[80] IBM. GPU Enabler for Spark, 2017. URL `https://github.com/IBMSparkGPU/GPUEnabler`.

[81] Tekin Bicer, Doğa Gürsoy, Vincent De Andrade, Rajkumar Kettimuthu, William Scullin, Francesco De Carlo, and Ian T. Foster. Trace: a high-throughput tomographic reconstruction engine for large-scale datasets. *Advanced Structural and Chemical Imaging*, 3(1):6, Jan 2017. ISSN 2198-0926. doi: 10.1186/s40679-017-0040-7. URL `https://doi.org/10.1186/s40679-017-0040-7`.

[82] James Fox, Yiming Zou, and Judy Qiu. Software frameworks for deep learning at scale. *Internal Indiana University Technical Report*, 2016.

[83] 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 *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 291–300, May 2015. doi: 10.1109/IPDPS.2015.83.

[84] Y. Wang, R. Goldstone, W. Yu, and T. Wang. Characterization and optimization of memory-resident mapreduce on hpc systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 799–808, May 2014. doi: 10.1109/IPDPS.2014.87.

[85] Carlos Maltzahn, Esteban Molina-Estolano, Amandeep Khurana, Alex J Nelson, Scott A Brandt, and Sage Weil. Ceph as a scalable alternative to the hadoop distributed file system. *login: The USENIX Magazine*, 35: 38–49, 2010.

[86] Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics: Do we really need to reinvent the storage stack? In *HotCloud*, 2009.

[87] Pengfei Xuan, Jeffrey Denton, Pradip K. Srimani, Rong Ge, and Feng Luo. Big data analytics on traditional hpc infrastructure using two-level storage. In *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems*, DISCS '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3993-3. doi: 10.1145/2831244.2831253. URL `http://doi.acm.org/10.1145/2831244.2831253`.

[88] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda. Accelerating spark with rdma for big data processing: Early experiences. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, pages 9–16, Aug 2014. doi: 10.1109/HOTI.2014.15.

[89] Isaac Lopez. Idc talks convergence in high performance data analysis, 2013. *URL: http://www.datanami.com/2013/06/19/idc_talks_convergence_in_high_performance_data_analysis/(visited on 02/14/2016)*, 2013.

[90] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 277–284, Dec 2008. doi: 10.1109/eScience.2008.59.

[91] Satish Narayana Srirama, Pelle Jakovits, and Eero Vainikko. Adapting scientific computing problems to clouds using mapreduce. *Future Generation Computer Systems*, 28(1):184 – 192, 2012. ISSN 0167-739X.

[92] Sangwon Seo, E.J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726, Nov 2010.

[93] Pengfei Xuan, Yueli Zheng, S. Sarupria, and A. Apon. Sciflow: A dataflow-driven model architecture for scientific computing using hadoop. In *Big Data, 2013 IEEE International Conference on*, pages 36–44, Oct 2013.

[94] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, and Lavanya Ramakrishnan. Riding the elephant: Managing ensembles with hadoop. In *Proceedings of the 2011 ACM International Workshop on Many Task Computing on Grids and Supercomputers*, MTAGS '11, pages 49–58, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1145-8. doi: 10.1145/2132876.2132888.

[95] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann,

V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. Root — a c++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180(12):2499–2512, 2009. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2009.08.005. URL http://www.sciencedirect.com/science/article/pii/S0010465511000701.

[96] 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 Transactions on Parallel and Distributed Systems*, 24(1):158–169, Jan 2013. ISSN 1045-9219. doi: 10.1109/TPDS.2012.88.

[97] 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 *Big Data (Big Data), 2015 IEEE International Conference on*, pages 918–927, Oct 2015. doi: 10.1109/BigData.2015.7363840.

[98] Youssef SG Nashed, David J Vine, Tom Peterka, Junjing Deng, Rob Ross, and Chris Jacobsen. Parallel ptychographic reconstruction. *Optics express*, 22(26):32082–32097, 2014.

[99] Andre Luckow, Pradeep Mantha, and Shantenu Jha. Pilot-abstraction: A valid abstraction for data-intensive applications on hpc, hadoop and cloud infrastructures? *arXiv preprint arXiv:1501.05041*, 2015.

[100] Saba Sehrish, Jim Kowalkowski, and Marc Paterno. Spark and hpc for high energy physics data analyses. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1048–1057. IEEE, 2017.

[101] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Scientific computing meets big data technology: An astronomy use case. In *2015 IEEE International*

Conference on Big Data (Big Data), pages 918–927, Oct 2015. doi: 10.1109/BigData.2015.7363840.

[102] Saba Sehrish, Jim Kowalkowski, and Marc Paterno. Exploring the performance of spark for a scientific use case. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1653–1659. IEEE, 2016.

[103] Zhuoyao Zhang. Processing data-intensive work ows in the cloud, 2012.

[104] K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman. Rethinking data management for big data scientific workflows. In *Big Data, 2013 IEEE International Conference on*, pages 27–35, Oct 2013. doi: 10.1109/BigData.2013.6691724.

[105] V. Nuthula and N. R. Challa. Cloudifying apps - a study of design and architectural considerations for developing cloudenabled applications with case study. In *Cloud Computing in Emerging Markets (CCEM), 2014 IEEE International Conference on*, pages 1–7, Oct 2014. doi: 10.1109/CCEM.2014.7015487.

[106] Satish Narayana Srirama and Jaagup Viil. Migrating scientific workflows to the cloud: Through graph-partitioning, scheduling and peer-to-peer data sharing. In *High Performance Computing and Communications, 2014 IEEE Intl Conf on*, pages 1105–1112. IEEE, 2014.

[107] Pierre Matri, Yevhen Alforov, Alvaro Brandon, Michael Kuhn, Philip Carns, and Thomas Ludwig. Could blobs fuel storage-based convergence between hpc and big data? In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 81–86. IEEE, 2017.

[108] Pierre Matri, Yevhen Alforov, Álvaro Brandon, María S. Pérez, Alexandru Costan, Gabriel Antoniu, Michael Kuhn, Philip Carns, and Thomas Ludwig. Mission possible: Unify hpc and big data stacks towards

application-defined blobs at the storage layer. *Future Generation Computer Systems*, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.07.035. URL http://www.sciencedirect.com/science/article/pii/S0167739X17330583.

[109] Y. Zhao, X. Fei, I. Raicu, and S. Lu. Opportunities and challenges in running scientific workflows on the cloud. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 455–462, Oct 2011. doi: 10.1109/CyberC.2011.80.

[110] G. Lin, B. Han, J. Yin, and I. Gorton. Exploring cloud computing for large-scale scientific applications. In *2013 IEEE Ninth World Congress on Services*, pages 37–43, June 2013. doi: 10.1109/SERVICES.2013.13.

[111] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: The montage example. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[112] 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 *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 640–645, Dec 2008.

[113] G. Bruce Berriman, Ewa Deelman, Gideon Juve, Mats Rynge, and Jens-S. Vöckler. The application of cloud computing to scientific workflows: a study of cost and performance. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1983), 2012. ISSN 1364-503X. doi: 10.1098/rsta.2012.0066.

[114] T. Gunarathne, Tak-Lon Wu, J. Qiu, and G. Fox. Mapreduce in the clouds for science. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 565–572, Nov 2010. doi: 10.1109/CloudCom.2010.107.

[115] Constantinos Evangelinos and C Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2. *ratio*, 2(2.40):2–34, 2008.

[116] A. Gupta and D. Milojicic. Evaluation of hpc applications on cloud. In *2011 Sixth Open Cirrus Summit*, pages 22–26, Oct 2011. doi: 10.1109/OCS.2011.10.

[117] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66, Dec 2009. doi: 10.1109/ESCIW.2009.5408002.

[118] Z. Hill and M. Humphrey. A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster? In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 26–33, Oct 2009. doi: 10.1109/GRID.2009.5353067.

[119] A. Iosup, S. Ostermann, M.N. Yigitbasi, R. Prodan, T. Fahringer, and D.H.J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, June 2011. ISSN 1045-9219.

[120] G. D'Angelo. Parallel and distributed simulation from many cores to the public cloud. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 14–23, July 2011.

[121] Dunhui Yu, Jian Wang, Bo Hu, Jianxiao Liu, Xiuwei Zhang, Keqing He, and Liang-Jie Zhang. A practical architecture of cloudification of legacy applications. In *Services (services), 2011 ieee world congress on*, pages 17–24. IEEE, 2011.

[122] S.N. Srirama, V. Ivanistsev, P. Jakovits, and C. Willmore. Direct migration of scientific computing experiments to the cloud. In *High Performance*

Computing and Simulation (HPCS), 2013 International Conference on, pages 27–34, July 2013. doi: 10.1109/HPCSim.2013.6641389.

[123] Silvina Caíno-Lores, Andrei Lapin, Peter G Kropf, and Jesús Carretero. Lessons learned from applying big data paradigms to large scale scientific workflows. In *WORKS@ SC*, pages 54–58, 2016.

[124] G. M. Slota, S. Rajamanickam, and K. Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 293–302, May 2016. doi: 10.1109/IPDPS.2016.93.

[125] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow. Hpc-abds high performance computing enhanced apache big data stack. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1057–1066, May 2015. doi: 10.1109/CCGrid.2015.122.

[126] 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 *2016 IEEE International Conference on Big Data (Big Data)*, pages 204–213, Dec 2016. doi: 10.1109/BigData.2016.7840606.

[127] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L Willke. Bridging the gap between hpc and big data frameworks. *Proceedings of the VLDB Endowment*, 10(8):901–912, 2017.

[128] N. Malitsky, A. Chaudhary, S. Jourdain, M. Cowan, P. O'Leary, M. Hanwell, and K. K. Van Dam. Building near-real-time processing pipelines with the spark-mpi platform. In *2017 New York Scientific Data Summit (NYSDS)*, pages 1–8, Aug 2017.

[129] Alex Gittens, Kai Rothauge, Shusen Wang, Michael Mahoney, Lisa Gerhardt, Prabhat, Jey Kottalam, Michael Ringenburg, and Kristyn Maschhoff. Accelerating large-scale data analysis by offloading to high-performance computing libraries using alchemist. In *SIGKDD'18: 24th ACM International Conference on Knowledge Discovery and Data Mining*, London, UK, 2018.

[130] 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 *2018 IEEE/ACM 5th International Conference on Big Data Computing Applications and Technologies (BDCAT)*, pages 1–10, Dec 2018. doi: 10.1109/BDCAT.2018.00010.

[131] Silvina Caıno-Lores and Jesús Carretero. A survey on data-centric and data-aware techniques for large scale infrastructures. *Int. J. Comput. Electr. Autom. Control Inf. Eng*, 10(3):517–523, 2016.

[132] M Asch, T Moore, R Badia, M Beck, P Beckman, T Bidot, F Bodin, F Cappello, A Choudhary, B de Supinski, et al. Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry. *The International Journal of High Performance Computing Applications*, 32(4): 435–479, 2018.

[133] Big Data Value Association (BDVA) and European Technology Platform for High-Performance Computing (ETP4HPC). The technology stacks of high performance computing and big data computing: What they can learn from each other. Technical report, November 2018.

[134] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, and Geoffrey C Fox. Anatomy of machine learning algorithm implementations in mpi, spark, and flink. *The International Journal of High Performance Computing Applications*, 32(1):61–73, 2018.

[135] Awais Ahmad, Anand Paul, Sadia Din, M. Mazhar Rathore, Gyu Sang Choi, and Gwanggil Jeon. Multilevel data processing using parallel algorithms for analyzing big data in high-performance computing. *International Journal of Parallel Programming*, 46(3):508–527, Jun 2018. ISSN 1573-7640. doi: 10.1007/s10766-017-0498-x. URL https://doi.org/10.1007/s10766-017-0498-x.

[136] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z Ibrahim, and Jay Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM, 2016.

[137] Silvina Caíno-Lores, Alberto García Fernandez, Felix Garcia-Carballeira, and Jesus Carretero. A cloudification methodology for multidimensional analysis: Implementation and application to a railway power simulator. *Simulation Modelling Practice and Theory*, 55(0):46 – 62, 2015. ISSN 1569-190X. doi: http://dx.doi.org/10.1016/j.simpat.2015.04.002. URL http://www.sciencedirect.com/science/article/pii/S1569190X15000611.

[138] Orcun Yildiz and Shadi Ibrahim. On the performance of spark on hpc systems: Towards a complete picture. In *Asian Conference on Supercomputing Frontiers*, pages 70–89. Springer, 2018.

[139] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. Dash: a recipe for a flash-based data intensive supercomputer. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010. doi: 10.1109/SC.2010.16.

[140] Jorge Salazar. *Conquering Big Data Through the Usage of the Wrangler Supercomputer*, pages 321–329. Springer International Publishing, Cham, 2016. ISBN 978-3-319-33742-5. doi: 10.

1007/978-3-319-33742-5_16. URL https://doi.org/10.1007/978-3-319-33742-5_16.

[141] P. A. Dhuldhule, J. Lakshmi, and S. K. Nandy. High performance computing cloud – a platform-as-a-service perspective. In *2015 International Conference on Cloud Computing and Big Data (CCBD)*, pages 21–28, Nov 2015. doi: 10.1109/CCBD.2015.56.

[142] European Technology Platform for High-Performance Computing (ETP4HPC). Strategic research agenda 2015 update. Technical report, 2015. URL http://www.etp4hpc.eu/pujades/files/ETP4HPC%20SRA%202%20Single%20Page.pdf.

[143] Transition to exascale computing (h2020-fethpc-2016-2017), April 2017. URL http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/fethpc-02-2017.html.

[144] Angelos Bilas, Toni Cortes, Domenico Talia, María S. Perez, Javier Garcia-Blas, Pilar González-Férez, André Brinkmann, Stergios Anastasiadis, Malcom Muggeridge, Carmela Comito, Sai Narasimhamurthy, Anna Queralt, and Florin Isaila. Data storage for big data in the exascale era: Challenges and prospects. Technical report, September 2015. URL https://www.dropbox.com/s/ws58kxm26j3o20a/nesus_report_WG4_sep2015.pdf.

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

[146] S. Caíno-Lores, F. Isaila, and J. Carretero. Data-aware support for hybrid hpc and big data applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 719–722, May 2017. doi: 10.1109/CCGRID.2017.55.

[147] J. Carretero S. Caino-Lores, A. Lapin and P. Kropf. Applying big data paradigms to a large scale scientific workflow: Lessons learned and future directions. *Future Generation Computer Systems*, (April), 2018. doi: https://doi.org/10.1016/j.future.2018.04.014.

[148] Claudia Szabo, Quan Z. Sheng, Trent Kroeger, Yihong Zhang, and Jian Yu. Science in the cloud: Allocation and execution of data-intensive scientific workflows. *Journal of Grid Computing*, 12(2):245–264, 2014. ISSN 1572-9184. doi: 10.1007/s10723-013-9282-3. URL `http://dx.doi.org/10.1007/s10723-013-9282-3`.

[149] Y. Zhao, I. Raicu, and I. Foster. Scientific workflow systems for 21st century, new bottle or new wine? In *2008 IEEE Congress on Services - Part I*, pages 467–471, July 2008. doi: 10.1109/SERVICES-1.2008.79.

[150] Ryan Mork, Paul Martin, and Zhiming Zhao. Contemporary challenges for data-intensive scientific workflow management systems. In *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science*, WORKS '15, pages 4:1–4:11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3989-6. doi: 10.1145/2822332.2822336.

[151] F. Duro, J. García, F. Isaila, J. Carretero, J Wozniak, and Ross R. Flexible data-aware scheduling for workflows over an in-memory object store. In *Proceedings of IEEE/ACM CCGrid 2016*, 2016.

[152] D. de Oliveira, E. Ogasawara, F. Baião, and M. Mattoso. Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 378–385, July 2010. doi: 10.1109/CLOUD.2010.64.

[153] I. Raicu, I.T. Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11, Nov 2008. doi: 10.1109/MTAGS.2008.4777912.

[154] Edward N Zalta et al. Stanford encyclopedia of philosophy, 2003.

[155] Georgiy V Bobashev, D Michael Goedecke, Feng Yu, and Joshua M Epstein. A hybrid epidemic model: combining the advantages of agent-based and equation-based approaches. In *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pages 1532–1537. IEEE Press, 2007.

[156] H Van Dyke Parunak, Robert Savit, and Rick L Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 10–25. Springer, 1998.

[157] Chris Swinerd and Ken R McNaught. Design classes for hybrid simulations involving agent-based and system dynamics models. *Simulation Modelling Practice and Theory*, 25:118–133, 2012.

[158] 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 Generation Computer Systems*, 2018.

[159] Silvina Caíno-Lores, Andrei Lapin, Peter Kropf, and Jesús Carretero. Methodological approach to data-centric cloudification of scientific iterative workflows. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 469–482. Springer, 2016.

[160] Silvina Caíno-Lores, Alberto García, Félix García-Carballeira, and Jesús Carretero. Efficient design assessment in the railway electric infrastructure domain using cloud computing. *Integrated Computer-Aided Engineering*, 24(1):57–72, 2017.

[161] Wolfgang Kurtz, Harrie-Jan Hendricks Franssen, Hans-Peter Kaiser, and Harry Vereecken. Joint assimilation of piezometric heads and groundwater temperatures for improved modeling of river-aquifer interactions. *Water Resources Research*, 50(2):1665–1688, 2014. ISSN 1944-7973.

[162] Gerrit Burgers, Peter-Jan van Leeuwen, and Geir Evensen. Analysis scheme in the ensemble Kalman filter. *Monthly Weather Review*, 126 (6):1719–1724, 1998.

[163] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proc. of the First Workshop on Hot Topics in Cloud Computing*, page 118, 2009.

[164] Jesús Carretero, José M Pérez, Félix García-Carballeira, Alejandro Calderón, Javier Fernández, Jose D García, Antonio Lozano, Luis Cardona, Norberto Cotaina, and Pierre Prete. Applying rcm in large scale systems: a case study with railway networks. *Reliability Engineering & System Safety*, 82(3):257–273, 2003.

[165] Sifei Lu, Reuben Mingguang Li, William Chandra Tjhi, Kee Khoon Lee, Long Wang, Xiarong Li, and Di Ma. A framework for cloud-based large-scale data analytics and visualization: Case study on multiscale climate data. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 618–622. IEEE, 2011.

[166] Chaowei Yang, Michael Goodchild, Qunying Huang, Doug Nebert, Robert Raskin, Yan Xu, Myra Bambacus, and Daniel Fay. Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing? *International Journal of Digital Earth*, 4(4):305–329, 2011.

[167] Gen-Tao Chiang, Martin T Dove, C Isabella Bovolo, and John Ewen. Implementing a grid/cloud escience infrastructure for hydrological sciences. In *Guide to e-Science*, pages 3–28. Springer, 2011.

[168] G. Bauser, Harrie-Jan Hendricks Franssen, Stauffer Fritz, Hans-Peter Kaiser, U. Kuhlmann, and W. Kinzelbach. A comparison study of two different control criteria for the real-time management of urban groundwater works. *Journal of Environmental Management*, 105:21 – 29, 2012. ISSN 0301-4797.

[169] Michael P McGuire, Martin C Roberge, and Jie Lian. Hydrocloud: A cloud-based system for hydrologic data integration and analysis. In *Computing for Geospatial Research and Application (COM. Geo), 2014 Fifth International Conference on*, pages 9–16. IEEE, 2014.

[170] 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 *High Performance Computing and Communications, 2014 IEEE Intl Conf on*, pages 959–965, 2014. doi: 10.1109/HPCC.2014.154.

[171] D Ceperley, GV Chester, and MH Kalos. Monte carlo simulation of a many-fermion study. *Physical Review B*, 16(7):3081, 1977.

[172] Zhenqin Li and Harold A Scheraga. Monte carlo-minimization approach to the multiple-minima problem in protein folding. *Proceedings of the National Academy of Sciences*, 84(19):6611–6615, 1987.

[173] Lihong Wang and Steven L Jacques. Monte carlo modeling of light transport in multi-layered tissues in standard c. *The University of Texas, MD Anderson Cancer Center, Houston*, pages 4–11, 1992.

[174] Diego Perez, Philipp Rohlfshagen, and Simon M Lucas. Monte-carlo tree search for the physical travelling salesman problem. In *European Conference on the Applications of Evolutionary Computation*, pages 255–264. Springer, 2012.

[175] Geir Evensen. Sequential data assimilation with a nonlinear quasi-geostrophic model using monte carlo methods to forecast error statistics. *Journal of Geophysical Research: Oceans*, 99(C5):10143–10162, 1994. ISSN 2156-2202.

[176] R Therrien, R McLaren, E Sudicky, and S Panday. A Three-dimensional Numerical Model Describing Fully-integrated Subsurface and Surface Flow and Solute Transport, 2010.

[177] D. Partington, P. Brunner, S. Frei, C. T. Simmons, A. D. Werner, R. Therrien, H. R. Maier, G. C. Dandy, and J. H. Fleckenstein. Interpreting streamflow generation mechanisms from integrated surface-subsurface flow models of a riparian wetland and catchment. *Water Resources Research*, 49(9):5501–5519, 2013. ISSN 1944-7973. doi: 10.1002/wrcr.20405.

[178] O.S. Schilling, J. Doherty, W. Kinzelbach, H. Wang, P.N. Yang, and P. Brunner. Using tree ring data as a proxy for transpiration to reduce predictive uncertainty of a model simulating groundwater–surface water–vegetation interactions. *Journal of Hydrology*, 519, Part B:2258 – 2271, 2014. ISSN 0022-1694. doi: http://dx.doi.org/10.1016/j.jhydrol.2014.08.063.

[179] Philip Brunner and Craig T. Simmons. Hydrogeosphere: A fully integrated, physically based hydrological model. *Ground Water*, 50(2): 170–176, 2012. ISSN 1745-6584.

[180] Reed M. Maxwell, Mario Putti, Steven Meyerhoff, Jens-Olaf Delfs, Ian M. Ferguson, Valeriy Ivanov, Jongho Kim, Olaf Kolditz, Stefan J. Kollet, Mukesh Kumar, Sonya Lopez, Jie Niu, Claudio Paniconi, Young-Jin Park, Mantha S. Phanikumar, Chaopeng Shen, Edward A. Sudicky, and Mauro Sulis. Surface-subsurface model intercomparison: A first set of benchmark results to diagnose integrated hydrology and feedbacks. *Water Resources Research*, 50(2):1531–1549, 2014. ISSN 1944-7973. doi: 10.1002/2013WR013725.

[181] Mikko Ilmari Jyrkama. *A methodology for estimating groundwater recharge*. 2004.

[182] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015. ISSN 1041-4347. doi: 10.1109/TKDE.2015.2427795.

[183] O. C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández. Spark versus flink: Understanding performance in big data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–442, Sept 2016. doi: 10.1109/CLUSTER.2016.22.

[184] Zhao Zhang, Kyle Barbary, Frank Austin Nothaft, Evan Sparks, Oliver Zahn, Michael J Franklin, David A Patterson, and Saul Perlmutter. Scientific computing meets big data technology: An astronomy use case. *arXiv preprint arXiv:1507.03325*, 2015.

[185] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 97–110, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4314-5. doi: 10.1145/2907294.2907310. URL `http://doi.acm.org/10.1145/2907294.2907310`.

[186] F. García-Carballeira A. García, S. Caíno-Lores and J. Carretero. A multiobjective simulator for optimal power dimensioning on electric railways using cloud computing. In *Proceedings of the 5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 428–438, 2015. ISBN 978-989-758-120-5. doi: 10.5220/0005573404280438.

[187] D. Morozov and T. Peterka. Block-parallel data analysis with diy2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 29–36, Oct 2016. doi: 10.1109/LDAV.2016.7874307.

[188] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[189] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[190] Kewei Lu, Han-Wei Shen, and Tom Peterka. Scalable computation of stream surfaces on large scale vector fields. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1008–1019. IEEE, 2014.

[191] Youssef SG Nashed, David J Vine, Tom Peterka, Junjing Deng, Rob Ross, and Chris Jacobsen. Parallel ptychographic reconstruction. *Optics express*, 22(26):32082–32097, 2014.

[192] Christopher Sewell, Jeremy Meredith, Kenneth Moreland, Tom Peterka, Dave DeMarle, Li-ta Lo, James Ahrens, Robert Maynard, and Berk Geveci. The sdav software frameworks for visualization and analysis on next-generation multi-core and many-core architectures. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 206–214. IEEE, 2012.

[193] Boonthanome Nouanesengsy, Teng-Yok Lee, Kewei Lu, Han-Wei Shen, and Tom Peterka. Parallel particle advection and ftle computation for time-varying flow fields. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 61. IEEE Computer Society Press, 2012.

[194] Tom Peterka, Robert Ross, Boonthanome Nouanesengsy, Teng-Yok Lee, Han-Wei Shen, Wesley Kendall, and Jian Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 580–591. IEEE, 2011.

[195] Dave Beazley, SWIG Team, et al. Simplified wrapper and interface generator. *See http://swig. sourceforge. net*, 1995.

[196] Keiichi Kondo, Koji Terasaki, and Takemasa Miyoshi. Assimilating satellite radiances without vertical localization using the local ensemble transform kalman filter with up to 1280 ensemble members. In *EGU General Assembly Conference Abstracts*, volume 19, page 2170, 2017.

[197] Mathew Williams, Paul A Schwarz, Beverly E Law, James Irvine, and Meredith R Kurpius. An improved analysis of forest carbon dynamics using data assimilation. *Global change biology*, 11(1):89–105, 2005.

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

[199] B. Nicolae, C. H. A. Costa, C. Misale, K. Katrinis, and Y. Park. Leveraging adaptive i/o to optimize collective data shuffling patterns for big data analytics. *IEEE Transactions on Parallel and Distributed Systems*, 28 (6):1663–1674, June 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2627558.